Versant ReVind Reference Manual

Release 7.0.1.4

VERSANT

Versant History, Innovating for Excellence

In 1988, Versant's visionaries began building solutions based on a highly scalable and distributed object-oriented architecture and a patented caching algorithm that proved to be prescient.

Versant's initial flagship product, the Versant Object Database Management System (ODBMS), was viewed by the industry as the one truly enterprise-scalable object database.

Leading telecommunications, financial services, defense and transportation companies have all depended on Versant to solve some of the most complex data management applications in the world. Applications such as fraud detection, risk analysis, simulation, yield management and real-time data collection and analysis have benefited from Versant's unique object-oriented architecture.

For more Information please visit **www.versant.com**

Versant US

Versant Corporation

255 Shoreline Drive, Suite 450, Redwood City, CA 94065

Ph +1 650-232-2400, Fx +1 650-232-2401

Versant Europe

Versant GmbH

Wiesenkamp 22b, 22359 Hamburg, Germany

Ph +49.40.60990-0, Fx +49.40.60990-113

Table of Contents

CHAPTER 1

SQL-to-Object Mapping

This Chapter provides an overview and details of the architecture of SQL to Object mapping.

The Chapter covers the following in detail:

- Overview
- Architecture
- Class to Table Mapping
- Mapping of Types

OVERVIEW

The Versant ReVind(Versant/SQL or VSQL) suite of software modules permits you to use conventional Structured Query Language (SQL) semantics to access data that resides in a Versant object database. The resulting application architecture can offer the strengths of both the relational and the object database models, such as the openness and inter connectivity of relational tables along with the expressiveness and performance of object collections.

Fluid database definition and natural propagation of data into the database is facilitated by the way that classes and object instances help create the database without the usual constraints of table columns and table rows. For these reasons, if not purely for performance reasons, object disciplined storage lays the foundation on which information will sit far into the future.

The Versant ReVind(VSQL) product suite addresses the need to obtain additional connectivity and openness for this information through an SQL-compliant form of object database access. Interactive SQL commands can be composed as demand dictates, empowering you to access and present data just the way you need it to be, if only for the duration of a fleeting moment.

The schema of a Versant ReVind environment is maintained transparently on your behalf. Versant ReVind automatically transforms complex object models into two-dimensional tables. You do not need—nor do you have—access to the DDL portions of SQL in order to maintain database schemas. Instead, as a Versant database definition changes, the schema honored by Versant ReVind automatically tracks those changes upon startup of the Versant ReVind daemon. Serving as online data dictionary, an object meta-schema is maintained and cached as a set of relational tables, which normal SQL queries (optionally through ODBC) can refine into reports. Once authenticated as the dba, you are only allowed to make minor, fine-tuning customizations to the automatically mapped schema for a database.

The objects and object schema stored in Versant Object Database Management System (ODBMS) are transparently accessed when you use the Versant database API through an object-oriented programming language. The Versant ReVind suite of modules frees you to move beyond the exclusive use of an object-oriented programming language and associated edit-compile-load cycles through which to create, update, and query objects. With the SQL suite, you (or your users) can also take advantage of stable standardized language and the inter connectivity that arises from its widespread application.

SQL data access can extend the reach of the already considerable data processing power found in a custom application that takes advantage of the Versant database API. This is evidenced particularly in terms of increased inter connectivity with other data processing tools that normally require a relational database instead of an object database.

The option of using ODBC-compliant tools for Versant database connectively is a similarly spirited option which is addressed by the Versant/ODBC module. For ODBC users, additional

documentation is provided, primarily in the form of the "Versant ODBC Manual." For additional information about these and other products, see `http://www.versant.com`.

Features

With the freedom to use SQL semantics, objects are visible through a relational database lens. However, SQL is not expressive enough to present a complex object model transparently; the Versant ReVind Suite uses some transformation rules to help resolve the treatment of certain difficult to represent object provisions. These rules are referred to as mode mapping transformations, or "model mappings" in short. These model mappings permit the object model to be treated entirely as an ER (Entity-Relationship) model. The details of this mapping are described later in this chapter.

Versant ReVind Suite provides the following features:

Dynamic Mapping

Versant ReVind is able to map the object schema in any Versant ODBMS database and allow SQL and ODBC access from components of the Versant ReVind Suite with minimal user input.

Query Performance

Versant ReVind uses navigational access based on the associations defined in the object schema whenever joins are attempted between tables based on the object-id columns. This implies that join performance would be within a reasonable range of transparent navigational access for an equivalent query made in C++.

Standard SQL-92 with no extensions

Instead of extending SQL-92 with object-oriented extensions, Versant ReVind Suite uses standard SQL-92 to present an ER representation of the object schema. This has the benefit of allowing ODBC compliant tools to access Versant databases using the same mechanisms as a user of the Interactive Versant Query Tool.

Scaleable

Versant ReVind uses cursors to access the Versant database allowing efficient use of resources and enabling a quick response to the query. The object cache is maintained by Versant ReVind engine and the main memory storage system cache size is configurable. In case a sort is requested where the data cannot be accommodated in the main memory storage system, local swap space would be used to satisfy the memory requirements.

Security

Versant ReVind provides full SQL-92 security over and above the user access list security provided by the Versant Object Database Management System.

A "special user" installs and setups Versant ReVind. This user will have full access rights over all tables and users. This user could grant and revoke select, insert, update, delete rights for different users. Typically, insert, update, and delete rights should be reserved for a very restricted group of users.

Software Modules

The Versant ReVind(VSQL) Suite includes the following software modules:

Versant ReVind Mapper

Versant ReVind Mapper is the software module that maps the Versant Object Database Management System object model to an SQL-92, relational model that can be presented to the user and accessed using SQL.

The object model that is mapped to SQL using Versant ReVind Mapper could have been created using any object definition language: C++ or C. The mapper has database compile-time as well as database run-time roles.

Versant ReVind

Versant ReVind is the software module that provides SQL access to Versant databases through the run-time portion of the Versant ReVind Mapper.

Versant ReVind and Versant ReVind Mapper are linked together as a single process. This process is a client of the Versant Object Database Management System.

Versant/ISQL

Interactive Versant/SQL is the software module that provides command-line SQL access to Versant databases.

Versant/ODBC

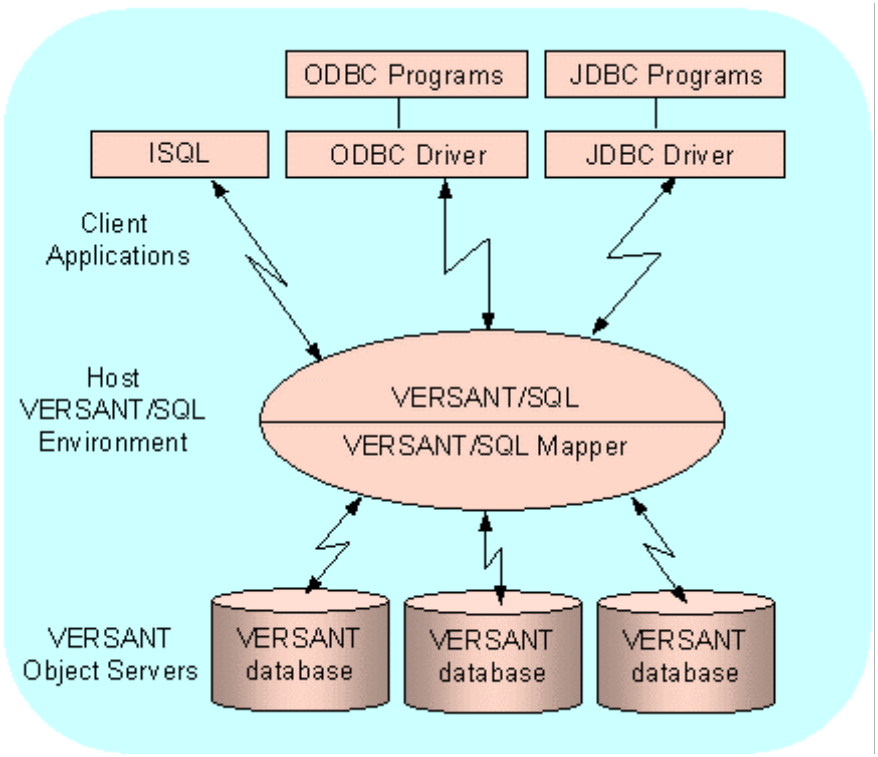
Available as a Versant ReVind suite option, the Versant/ODBC module allows any ODBC 3.0 compliant commercial off-the-shelf tool to access Versant databases.

Versant/ODBC is built as a DLL (dynamically linked library) and executes on the client machine. Any ODBC compliant tool can access a Versant database using Versant/ODBC, which then translates the ODBC requests to Versant ReVind requests and forwards them to a local or remote Versant ReVind process.

The ODBC API conformance level is core, level 1 and most of level 2. This support includes outer joins, positioned updates, union operations, ODBC-compatible scalar functions and extended data types. The SQL language conformance is extended SQL grammar.

Versant/ODBC has been tested with tools such as CrystalReports, Microsoft Visual Basic, PowerBuilder and Microsoft Access.

ARCHITECTURE

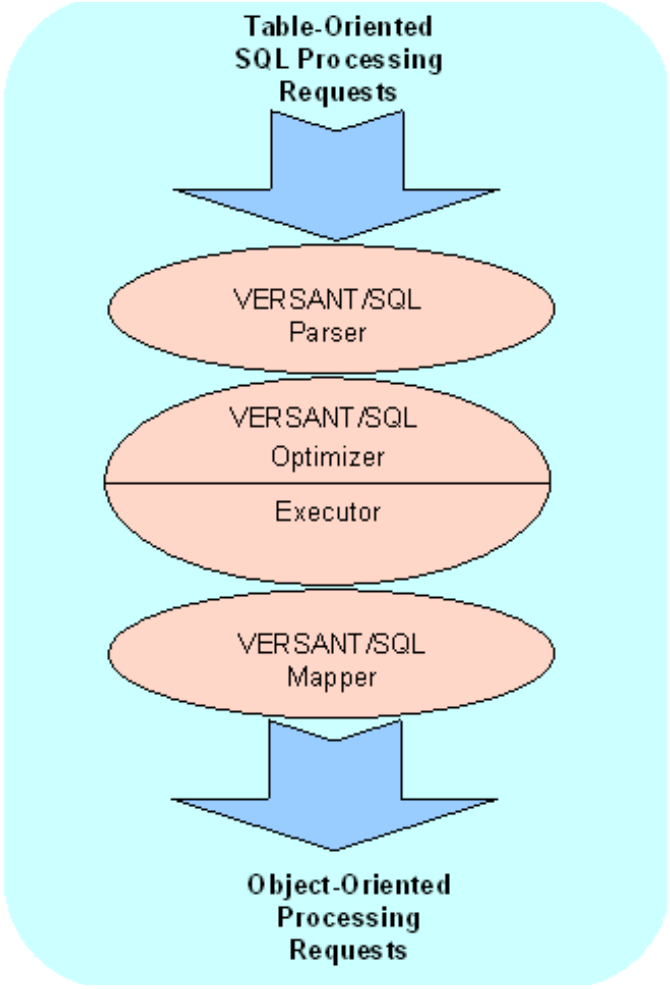


At the heart of Versant ReVind Suite is the Versant ReVind engine. Versant ReVind appears as a client to one or more Versant database servers.

Versant/ODBC uses standard TCP/IP to communicate to the Versant ReVind process. The Versant ReVind host process is called `dhserver`. It can create ("fork") a process or create a thread to access a database whenever a connection to a known data source is requested from Versant/ODBC.

Versant ReVind parses, optimizes and executes a SQL statement by making calls to the Versant ReVind Mapper. The Versant ReVind Mapper uses the Versant interface to access and update data from Versant databases.

Versant ReVind can either satisfy a processing request based on data in its cache or it can forward the request to the Versant ReVind Mapper, which either processes the request using the data in its object cache or accesses a Versant database as necessary.



The Versant ReVind(Versant/SQL) Parser converts a SQL statement to a relational algebraic tree.

The Versant ReVind(Versant/SQL) Optimizer converts the relational algebraic tree to the most efficient form for execution.

The Versant ReVind(Versant/SQL) Executor traverses the tree and executes the nodes by making calls to the Versant ReVind Mapper.

CLASS TO TABLE MAPPING

Mapping Issues

There are several issues that need to be handled in order to map an object model to a relational model. In the degenerate case, a class can be mapped to a table, an attribute can be mapped to a column, and an instance can be mapped to a row. The issues that make the mapping complex are class inheritance, multi-valued attributes, arrays of objects and embedded objects. Also, from access point of view, the main issues that need to be tackled are "joins" based on navigational access.

SQL users expect to be able to access "data-dictionaries"; tables that represent meta-model information. In Versant ReVind these tables are called base tables.

Versant ReVind maintains descriptive information about a database in three different types of tables:

- base tables (data dictionaries)
- extended tables that keep track of authorization and statistics information
- utility tables that can readjust the dynamic mapping in limited ways

The Versant ReVind Mapper uses an in-memory representation of persistent classes and attributes to perform the mapping. When a Versant database is accessed using Versant ReVind either using Interactive Versant Query Tool or using ODBC. The Versant ReVind Mapper queries and loads the schema information from the database and uses the mapping rules to map the object schema to an in-memory representation of tables, columns and base tables. This mapping also maps the Versant attribute types to the corresponding SQL types.

A mapping repository object stores information about the mapping of the schema from one or more Versant databases to the global entity-relationship schema presented to the user.

Base Tables

Versant ReVind(VSQL) maintains meta level information about the relational schema presented to the user in base tables. There are three base tables that can be queried by the user: `systables, syscolumns` and `sysindexes`.

Versant ReVind Mapper loads these tables upon startup based on the schema information stored in any Versant database. Once these tables are loaded, they can be queried using SQL. Not all information presented in these tables can be directly mapped to persistent

schema information. The Versant ReVind Mapper itself adds entries to these tables based on the mapping rules.

Following is additional information about these base tables:

`systables`

Every row in the table `systables` represents a table. Tables are of different types: user tables, system tables and views. Every table has a unique id and other properties such as percent-free that are assigned values during startup.

`syscolumns`

Entries in the `syscolumns` table represent columns and their properties such as type, width, precision, etc. The column types represent the SQL types that are mapped from the corresponding class attribute types.

`sysindexes`

The table `sysindexes` tracks the indexes in Versant databases. There is one entry in `sysindexes` for every component of an index (currently Versant Object Database Management System supports only single component indexes).

In addition to the database indexes, there are also entries for "navigational indexes". As described in the following section "Simple Mapping", the `[SelfOID]` column is used to uniquely identify a row in a table; correspondingly, there is an entry in `sysindexes` for a hash index on the `[SelfOID]` column for every table.

Extended Tables

In addition to the base tables listed above, Versant ReVind uses extended tables to maintain security and statistics information. Unlike the base tables that are transient and populated upon startup, the extended tables are loaded from persistent classes that are stored in the database. Information, such as user authorization granted by the database administrator, needs to be persistent. Only the database administrator has authority to insert entries in these tables.

Utility Tables

Versant ReVind also maintains database configuration information in these utility tables:

`VQNeededClasses`

Is used to trim the effective database schema to a subset of the tables/classes that would normally be available, possibly decreasing the startup delay for the Versant ReVind daemon.

`VQColumnMaxLength`

Is used to configure the maximum permissible string length of columns that are being mapped to the `VARCHAR` data type.

`VQPrefixString`

Is used to help create shorter table names corresponding to long class names.

`VQDuplicatedTables`

Is used to help resolve to unique names any table names that are otherwise duplicated.

Table and Index Scans

When a query is performed on a table, Versant ReVind looks up the `sysindexes` table to check if the columns that are being accessed have an index defined on them (prior to this Versant ReVind would have checked the authorization and ensured that the user has access rights).

For more details on authorization, please refer “Security” on page 77 in "Chapter 2 - Initiating Versant Revind".

If one or more index entries are found for the column(s) being queried, based on the type of scan operator, an appropriate index will be selected for the query. A database lookup based on an index is called an "index scan". Not all index scans are implemented as database queries. As explained in the section on "Navigational Joins", when the index column is an object-id, the index scan maps to a simple de reference of the association.

If the column being scanned does not have an index entry in the `sysindexes` table, Versant ReVind uses a table scan to fetch the rows from the table. For tables that directly represent classes in Versant databases, the table scan is implemented using cursors that are directly supported by the Versant Object Database Management System. For other tables such as base tables the table scan is handcrafted (an iterator is used to iterate over entries maintained in the Versant ReVind Mapper mapping repository). For tables that represent multi-valued attributes, a table scan corresponds to using an iterator to iterate over elements in the multi-valued attribute.

For more information, please refer “Mapping of a Multi-Valued Association (LinkVstr<type>)” on page 28.

Simple Mapping

Every table that represents a database class has a virtual column named [SelfOID] to represent the unique object-id of every row. In most cases the column [SelfOID] will map to the logical object-id (LOID) of the object. (Exceptions are discussed in sub-section on multi-valued attributes.)

See “Mapping of a Multi-Valued Association (LinkVstr<type>)” on page 28.

The [SelfOID] column is an important notion for the mapping. As objects in the Versant Object Database Management System are not distinguishable by attribute values only, the LOID of the object is the only attribute that can be considered a "primary key" column for a table that represents a persistent class. The [SelfOID] column is a dot delimited string representation of the LOID of the object (such as '5.0.1056'). The real need for the [SelfOID] column becomes apparent in the section on "Navigational Joins".

See “Navigational Joins” on page 27.

When inserting values into a table, any arbitrary string can be passed as the [SelfOID] column value. Versant ReVind Mapper ignores this string as the Versant database assigns a unique LOID to the new persistent object.

Updates to the [SelfOID] column will not be allowed by the Versant ReVind Mapper.

Example

Suppose the following definition of class Employee. Also suppose that a btree index has been set on the attribute name.

Class definitions

```
class Employee
{
   Pstring  name;   // BTREE index
   int      soc;
   float    salary;
};
```

The object schema for this Versant database is the following.

```
Object schema: Class Employee
attribute     attribute                    index
name          type          occurrences    type
```

```
-----------------------------------------------------
name          vstr<char>     1              btree
soc           o_4b           1
salary        o_float        1
```

The table schema that is mapped to this object schema is the following:

```
Mapped schema: Table Employee
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
soc           INTEGER
salary        REAL
```

Mapping of Associations

An association from a class `A` to a class `B` is represented as a column in the table that represents class `A`. The values in this column would represent the logical object identifier `(LOID)` of the referenced object. This allows the user to "navigate" the association by performing a standard SQL join operation. There is no guarantee that the object being referred to exists (as it may have been deleted).

Example

Suppose the following definition of class `Employee`. Also suppose that a btree index has been set on the attribute `name` in both class `Employee` and class `Department`.

Class definitions

```
class Employee
{
   PString   name;   // BTREE index
   Link<Department>  department;
};
class Department
{
   PString name;     // BTREE index
};
```

The object and mapped schemas would be the following:

Object schema*: *Class Employee

```
attribute     attribute                      index
name          type           occurrences     type
-----------------------------------------------------
name          vstr<char>     1               btree
department    (Department)   1
```

Mapped schema*: *Table Employee

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
department    VARCHAR                       foreign key references
                                            [Department.SelfOID]
```

Object schema*: *Class Department

```
attribute     attribute                      index
name          type           occurrences     type
-----------------------------------------------------
name          vstr<char>     1               btree
```

Mapped schema*: *Table Department

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
```

Navigational Joins

In order for this mapping to satisfy the performance requirements, a join in SQL should map to navigation in the Versant ReVind Mapper. Therefore, a query such as:

```
select Department.name
from Employee, Department
where Employee.name = ‘Joe Smith’
   and Department.SelfOID=Employee.department
```

would result in an index scan on table `[Employee]` on the indexed column `name`. For every `[Employee]` tuple returned, the `[departmentOID]` column value would be used to perform an index scan on table `[Department]`. The "join" in this case would really not be performed, as the `[departmentOID]` value is sufficient to retrieve the `(Department)` object.

Mapping of a Multi-Valued Association (LinkVstr<type>)

Because SQL-92 does not support multi-valued attributes, any multi-valued class attribute is mapped to a separate table.

The "virtual" table would be named `"TableName_AttributeName"`. The table would have three columns:

`SelfOID`

A column that represents the object-id of the object that has the multi-valued attribute.

`RowNumber`

A column that represents the position of the element in the multi-valued attribute.

`Element`

A column for the element itself.

To access the values of a multi-valued attribute of an object, the user can perform a "join" between the regular table (that represents the class) and the table that represents a multi-valued attribute using the `[SelfOID]` column values in both tables.

Example

Class definitions

```
class Employee
{
   PString          name;        // BTREE index
   Link<Department> department;
```

```
};
class Department
{
   PString            name;      // BTREE index
   LinkVstr<Employee> employees;
};
```

Object schema*: *Class Employee

```
attribute     attribute                      index
name          type           occurrences     type
-----------------------------------------------------
name          vstr<char>     1               btree
department    (Department)   1
```

Mapped schema*: *Table Employee

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
department    VARCHAR                       foreign key references
                                            [Department.SelfOID]
```

Object schema*: *Class Department

```
attribute     attribute                      index
name          type           occurrences     type
-----------------------------------------------------
name          vstr<char>     1               btree
employees     (Employee)     VECTOR
```

Mapped schema*: *Table Department

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
```

Mapped schema*: *Table Department_employees

```
column          column          index           key
name            type            type            type
-------------------------------------------------------
SelfOID         VARCHAR         HASH            foreign key references
                                                [Department.SelfOID]
employees       VARCHAR         BTREE           foreign key references
                                                [Employee.SelfOID]
RowNumber       INTEGER         HASH
```

In order for this mapping to satisfy the performance requirements, a join in SQL should map to navigation in the Versant ReVind Mapper. Therefore, a query such as:

```
select Employee.name
from Employee, Department, Department_employees
where Department.name = 'Engineering'
   and Department.SelfOID=Department_employees.SelfOID
   and Department_employees.employees = Employee.selfOID
```

would result in an index scan on table `Department` on the indexed column `[name]`. As the join between the `Department` table and the `Department_employees` table is based on the `[SelfOID]` column, for every `Department` row returned, the `[SelfOID]` column value would be used to perform a (hash) index scan on table `Department_employees`. The "join" in this case would really not be performed as the `[SelfOID]` value in table `Department_employees` is sufficient to locate the `Department` object from the database.

The index scan on the table `Department_employees` really corresponds to an iteration over the `employees` attribute of the `Department` object. For every call to fetch the next row from the index scan, Versant ReVind Mapper would fetch the next element from the `employees` attribute and return it is the column value `[employees]` of the table `Department_employees`.

The "join" from the `Department_employees` table to the `Employee table` is also implemented using navigation, as the `[employees]` column values are used to perform the index scan on table `Employee`. In order to perform the `Employee` table projection, the Versant ReVind Mapper fetches the `Employee` objects by de referencing the object-ids `(LOID)` used to perform the index scan and returns the attribute values requested by the projection.

The `[RowNumber]` column allows queries on a subset of the elements in the multi-valued attribute. For example, to return the first ten employees that work in a department, you could use the `[RowNumber]` column:

```
select Employee.name
```

```
from Employee, Department, Department_employees
where Department.name = 'Engineering'
   and Department.SelfOID=Department_employees.SelfOID
   and Department_employees.employees = Employee.SelfOID
   and Department_employees.RowNumber < 10
```

The only difference in the result of the execution (from the previous query) is that only the first ten employees in the link vstr `employees` attribute would be considered during the index scan (assuming that the index starts from `0`).

NOTE:- For three tables, at the most a two-way join would be performed. In other words, in the above query, the table `[Employee]` would not be joined back to the `[Department]` table. Also, in the above schema, there are really two "paths" to join objects of class `Employee` with objects of class `Department`. Whenever you want to "navigate" from `Employees` to their associated `Departments`, there is no need to include the `Department_employees` table in the query.

Mapping of a Single Embedded Object

The Versant ReVind Mapper would map an embedded object of type `Embedded` in class `TopLevelClass` to its flattened elementary attributes. In the mapping, there would not be any table `Embedded`, except when type `Embedded` is also a persistent capable class `Embedded`. In this case, there would also be a table `Embedded`; but table `TopLevelClass` would use the schema described below.

Example

Class definitions

```
class Employee
{
   PString   name;   // BTREE index
   Location  home;
};
class Location
{
   PString city;
   PString country;
};
```

Object schema*: *Class Employee

```
attribute     attribute                      index
name          type           occurrences     type
-----------------------------------------------------
name          vstr<char>     1               btree
home          Location       1
```

Mapped schema*: *Table Employee

```
column        column         index           key
name          type           type            type
-----------------------------------------------------
SelfOID       VARCHAR        HASH            primary
name          VARCHAR        BTREE
home_city     VARCHAR
home_country  VARCHAR
```

A sample query would be:

```
select Employee.home_country, Employee.home_country
from Employee
where Employee.name = ‘Joe Smith’
```

It would be handled just like a simple query.

Mapping of an Array of Embedded Objects

An array of embedded objects is mapped using the scheme described in the subsection “Mapping Multi-Valued Associations”.

For more information, please refer “Mapping of a Multi-Valued Association (LinkVstr<type>)” on page 28.

An array of embedded objects of type `Embedded` in class `TopLevelClass` is mapped to a separate table `TopLevelClass_Embedded` flattened out to its elementary attributes. You would be able to perform a "navigational join" from the table `TopLevelClass` to the table `TopLevelClass_Embedded` using the `[SelfOID]` column in both tables. In this mapping also, the `[RowNumber]` column represents an index into the table of embedded objects.

Example

Class definitions

```
class Employee
{
   PString   name;   // BTREE index
   Location  homes[2];
};
class Location
{
   PString   city;
   PString   country;
};
```

Object schema*: *Class Employee

```
attribute     attribute                    index
name          type          occurrences    type
-----------------------------------------------------
name          vstr<char>    1              btree
homes         Location      2
```

Mapped schema*: *Table Employee

```
column        column        index          key
name          type          type           type
-----------------------------------------------------
SelfOID       VARCHAR       HASH           primary
name          VARCHAR       BTREE
```

Mapped schema*: *Table Employee_homes

```
column        column        index          key
name          type          type           type
-----------------------------------------------------
SelfOID       VARCHAR       HASH           foreign key references
                                           [Employee.selfOID]
city          VARCHAR
country       VARCHAR
RowNumber     INTEGER       HASH
```

A sample query would be:

```
select Employee_homes.city, Employee_homes.country
from Employee, Employee_home
where Employee.name = 'Joe Smith'
   and Employee.SelfOID = Employee_home.SelfOID
```

It would be handled just like a simple query. An insert into the table `[Employee_homes]` would be constrained by the number of occurrences allowed for that attribute in the class `(Employee)`. In the example above, every employee object can have at the most two homes.

Mapping of Vstr<elemental type>

An attribute of type `Vstr<char>` would be mapped to a column type `VARCHAR`.

All other `Vstr<element>` types `(Vstr<o_1b>, Vstr<o_u1b>, Vstr<o_2b>, Vstr<o_u2b>, Vstr<o_4b>, Vstr<o_u4b>, Vstr<o_bool>)` will be handled just like multi-valued associations.

Mapping of Inheritance

A class `DerivedClass` derived from class `BaseClass` would be mapped to table `DerivedClass` with columns representing attributes of class `Derived` as well as all attributes from all superclasses of class `Derived` (such as attributes from class `BaseClass`).

Table `BaseClass` which is mapped from class `BaseClass` would not have any columns that represent attributes from class `DerivedClass` that are not present in class `BaseClass`. All Versant database queries that implement the SQL `SELECT` are recursive: they return objects of the queried class and all its subclasses.

Example

Class definitions

```
class Employee
{
   PString  name;   // BTREE index
   float    salary;
```

```
};
class TemporaryEmployee : public Employee
{
   int      hoursPerDay;
};
```

Object schema*: *Class Employee

```
attribute     attribute                    index
name          type          occurrences    type
-----------------------------------------------------
name          vstr<char>    1              btree
homes         o_float       1
```

Mapped schema*: *Table Employee

```
column        column        index          key
name          type          type           type
-----------------------------------------------------
SelfOID       VARCHAR       HASH           primary
name          VARCHAR       BTREE
salary        FLOAT
```

Object schema*: *Class TemporaryEmployee

```
attribute     attribute                    index
name          type          occurrences    type
-----------------------------------------------------
hoursPerDay   o_4b
```

Mapped schema*: *Table TemporaryEmployee

```
column        column        index          key
name          type          type           type
-----------------------------------------------------
SelfOID       VARCHAR       HASH           primary
name          VARCHAR       BTREE
salary        FLOAT
hoursPerDay   INTEGER
```

A sample query would be:

```
select Employee.name, Employee.salary
from Employee
where Employee.salary > 100000
```

Versant ReVind would recursively select instances of class `Employee` and its subclasses (which includes class `TemporaryEmployee`). In this query, the user can only query attributes that are common to all employees.

To query a subclass of the class `(Employee)`, a sample query would be:

```
select TemporaryEmployee.name, TemporaryEmployee.hourPerDay
from TemporaryEmployee
where TemporaryEmployee.hoursPerDay > 9
```

To execute this query, Versant ReVind Mapper would perform a table scan on table `TemporaryEmployee` and recursively select all objects of class `TemporaryEmployee`.

An insert into the table `Employee` can only end up creating an object of class `Employee`; similarly, an insert into the table `TemporaryEmployee` can only map to an instantiation of an `TemporaryEmployee` object.

An update to a row of table `Employee` would be allowed for only those columns that are visible in the `Employee` table. The actual object mapped to an `Employee` row may be an instance of a subclass of class `Employee`.

A navigational join between the table `Employee` and table `TemporaryEmployee` would be permitted though it is possible that the same attribute ends up being displayed as two different columns.

Mapping of C++/Versant Collection classes

Mapping C++/Versant collection classes such as `V??Dictionary<key,value>` and `V?Set<type>` needs special consideration as these collections are implemented using multiple vstrs that have values that are useful primarily in the context of the behavior associated with the collections.

For example, `VVSet<type>` is implemented using three `vstrs: members, table, and links` where `members` holds the actual objects whereas the other two vstrs are used by the member functions defined on the `VVSet<type>` class.

As all C++/Versant collections are implemented using one or more vstrs, the collection would be mapped just like a class that has `Vstr<type>` attributes. The only constraint placed on C++/Versant collections is that not all vstrs are mapped, as some vstrs would be considered private to the C++/Versant collection and mapping them would not be useful when accessed through SQL.

For example, in a `VVSet<type>`, only the `members` vstr is mapped to a table. The `links` and `table` vstrs are hidden and not mapped to any table or column. The mapping rules used by the Versant ReVind Mapper dictate which class attributes are considered "hidden".

Inserts, updates, deletes will not be allowed for elements in a C++/Versant collection using the SQL interface. This will ensure that the integrity of the collection is maintained.

Example

Class definitions

```
class Employee
{
   PString   name;   // BTREE index
};
class Department
{
   PString         name;
   VISet<Employee> employees;
};
```

Object schema*: *Class Employee

```
attribute     attribute                      index
name          type           occurrences     type
-----------------------------------------------------
name          vstr<char>     1               BTREE
```

Mapped schema*: *Table Employee

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
```

Object schema*: *Class Department

```
attribute     attribute                      index
name          type           occurrences     type
-------------------------------------------------------
name          Vstr<char>     1               BTREE
employees     Vstr<PObject> members,
              Vstr<o_u4b> table,
              Vstr<o_4b> links
```

Mapped schema*: *Table Department

```
column        column         index          key
name          type           type           type
-------------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR
```

Mapped schema*: *Table Department_employees

```
column        column         index          key
name          type           type           type
-------------------------------------------------------
SelfOID       VARCHAR        HASH           foreign key references
                                            [Department.SelfOID]
member        VARCHAR                       foreign key references
                                            [Employee.SelfOID]
```

A sample query that accesses the set of employees for a department would be:

```
select Employee.name
from Employee, Department, Department_employees
where Department.name = 'Engineering'
   and Department.selfOID = Department_employees.selfOID
   and Department_employees.memberOID = Employee.selfOID
```

For a `V??Dictionary<key,value>` a sample mapping would be as follows:

Class definition

```
class Department
{
   PString name;
   VVIDictionary<Employee, Employee> employeeManager;
};
```

Object schema*: *Class Department

```
attribute     attribute                    index
name          type          occurrences    type
-----------------------------------------------------
name          vstr<char>    1              BTREE
employees     Vstr<PVirtual> keys,
              Vstr<PObject> values,
              Vstr<o_4b> table,
              Vstr<o_4b> links
```

Mapped schema*: *Table Department

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           primary
name          VARCHAR        BTREE
```

Mapped schema*: *Table Department_employeeManager

```
column        column         index          key
name          type           type           type
-----------------------------------------------------
SelfOID       VARCHAR        HASH           foreign key references
                                            [Department.SelfOID]
keys          VARCHAR                       foreign key references
                                            [Employee.SelfOID]
keys          VARCHAR                       foreign key references
                                            [Employee.SelfOID]
```

A sample query that returns the employee name and the manager name for an employee would be:

```
select Employee.name, managername
from Employee e, Department_employeeManager d_em
where Employee.name = ‘Joe Smith’
and Department_employeeManager.keys = Employee.SelfOID
and managername =
   (select Employee.name
    from Employee
    where d_em.values = Employee.SelfOID)
```

Application-Specific Customization of Mappings

The mappings of class names to table names, maximum string lengths to values other than 128, and the suppression of the mappings of certain classes to table equivalents can be affected to suit application-specific needs. In general, the meta-schema tables known as utility tables serve this purpose.

These tables and the mapping customization tasks that are commonly associated with them are described in the Chapter 2 “Initiating Versant Revind” on page 43.

MAPPING OF TYPES

The attribute types in a Versant database schema are mapped to the corresponding SQL types.

In most cases the mapping of types is straightforward (such as `o_4b` being mapped to SQL type `INTEGER`). In cases where the type mapping is not direct, Versant ReVind Mapper maps the types and provides data conversion when accessing mapping data from persistent objects of a class to rows of a table.

Attribute Type	Repetition Factor	SQL Type	Comments
`char`	-1	`VARCHAR`	`Vstr<char>, Vstr<o_u1b>` in C++
`char`	> -1	`CHARACTER`	char or array of char
`o_object`		`VARCHAR`	
`o_1b`		`TINYINT`	
`o_2b`		`SMALLINT`	
`o_4b`		`INTEGER`	
`o_u1b`		`SMALLINT`	
`o_u2b`		`INTEGER`	
`o_u4b`		`INTEGER`	
`o_float`		`REAL`	
`o_double`		`FLOAT`	
`o_bool`		`BIT`	
`PString`		`VARCHAR`	
`VString`		`VARCHAR`	
`VDate`		`DATE`	data conversion by SQL Mapper
`VTime`		`TIMESTAMP`	data conversion by SQL Mapper
`o_stptr`		`VARCHAR`	LOID, small integer, character
`Link<> or LinkAny`		`VARCHAR`	LOID of object

In this release, Versant ReVind does not support DDL (data definition language).

Mapping of SQL types to the corresponding Versant types.

SQL Type	Repetition Factor	Versant Type
char		o_char
Number	[5]	o_4b
short		o_2b
long		o_4b
float		o_float
double		o_double
date		o_4b
money	[5]	o_4b
time	[9]	o_1b
timestamp	[9]	o_1b
tinyint		o_1b
binary		Vstr<o_u1b>
bit		o_u1b
Ivarchar		not supported
Ivarbinary		not supported
bigint	[2]	o_4b

CHAPTER 2

Initiating Versant Revind

This Chapter describes the several steps that need to be performed to start using SQL against a Versant database.

The Chapter covers the following in detail:

- Getting Started
- Preparing a Versant Database for SQL Access
- Achieving Faster Start up Times
- Avoiding the Truncation of Long Character Strings
- Maintaining String Lengths Less Than 128
- Table Names
- Opening and Closing Database Connections
- Limitations for SQL Queries
- Retrieving Meta-Schema Information
- Mapping Information In Utility Tables
- Setting the System Memory Cache
- Setting the Default Date Format
- Switching Activity Logging On and Off
- Exporting Data from a Versant SQL Database

- Security

GETTING STARTED

A pre-requisite for using SQL against Versant database is to have Versant ReVind installed for at least one server and at least one client that is properly networked to the server. The client and server Versant ReVind systems can also be on the same system.

For specific instructions, see the installation information in the "Versant ReVind for *Platform* Release Package" written for your particular platform.

To start using SQL against a Versant database, several steps must be performed:

- You must request the generation of table schema for an existing Versant database.

  For more information, please refer “Preparing a Versant Database for SQL Access” on page 47.

- You can customize the size of string values to a length other than 128 for selected columns of a Versant ReVind table.

  For more information, please refer “Preparing a Versant Database for SQL Access” on page 47 and also “Maintaining String Lengths Less Than 128” on page 51.

- You can customize the naming convention for tables names mapped from class names.

  For more information, please refer “Customizing the Mapped Table Names” on page 69.

- You can reduce the scope of the persistent classes to be mapped to SQL by the Versant ReVind Mapper in order to shorten the time required for startup.

  For more information, please refer “Achieving Faster Startup Times” on page 49.

- You can learn about the various ways to open Versant databases so that they can be accessed through SQL statements.

  For more information, please refer “Opening and Closing Database Connections” on page 55.

As the final item suggests, this chapter also describes some of the ways that operations in Versant ReVind diverge from customary SQL operations. The divergence sometimes permits more options. Other times, you may be made aware of a constraint.

- Due to Versant ReVind support for distributed databases, the way you refer to a database within an open request may differ.

For more information, please refer “Opening and Closing Database Connections” on page 55.

- Due to the character case insensitivity of Versant ReVind, you may need to pay more attention to the commands you enter to refer to tables.

  For more information, please refer “Table Names” on page 52.

- Due to the extensive metadata information Versant ReVind maintains in system tables, you can identify the types of data that can be found in a given source database by using special queries.

  For more information, please refer “Retrieving Meta-schema Information ” on page 60.

- Versant ReVind may constrain your queries.

  For more information, please refer “Limitations For SQL Queries” on page 59.

The remaining tasks in this chapter are discretionary. To satisfy the needs of a particular site or operating platform, you should perhaps learn:

- Changing memory cache settings and the default date format.

  For more information, please refer “Setting the System Memory Cache” on page 70 and also “Setting the Default Date Format” on page 72.

- Maintaining an audit trail of user activity.

  For more information, please refer “Switching Activity Logging On and Off” on page 74.

- Repurposing data.

  For more information, please refer “Exporting Data from a Versant ReVind Database” on page 76.

- Securing access to information.

  For more information, please refer “Security” on page 77.

PREPARING A VERSANT DATABASE FOR SQL ACCESS

To prepare a database for SQL access, use the Versant ReVind `schload` utility. This utility inserts into the Versant database supporting information needed to handle the kinds of processing requests generated by the SQL Mapper module. Once such support is added to one or more Versant databases, both `misql` (the interactive SQL engine and co-resident SQL Mapper) and the standalone SQL Server, `dhserver` will be able to create successful connections to the prepared Versant databases.

Although this book does not describe the running of a standalone SQL Server process through the `dhserver` daemon, such a preparation will be necessary if you wish to use SQL Suite extensions such as Versant/ODBC. You will also need the support of the standalone daemon if you wish to use the standalone interactive SQL engine, `isql`, either locally where the `dhserver` is already running, or remotely from one or more client systems that are networked to the system that is running the `dhserver` daemon. Those client systems can be any mixture of Windows and UNIX platforms for which there are client `isql` applications, or ODBC client applications.

Following is the general syntax for invoking `schload`. Note that you must be logged in as the dba of the Versant database you specify as the argument. For a database residing on a remote host, enter:

```
schload database_name@dbhost
```

For a database residing on a local host, enter:

```
schload database_name
```

Besides loading the schema tables, the `schload` utility will also grant query `(SELECT)` privileges to all current and future users of the specified database, by virtue of granting `SELECT` to `PUBLIC` on all the currently mapped tables. If such default access settings are either overly permissive or overly strict;

See “Security” on page 77.

You need not run `schload` more than once for a database. (And sometimes you do not need to even run it once, providing you only connect to it indirectly by opening other databases first as described in the "Opening Multiple Databases" section.)

See “Opening Multiple Database Connections” on page 56.

After you have run the `schload` utility to generate the SQL schema for a Versant database, the schema information is captured in various system tables. Some of this information is modifiable, as the next sequence of sections describe.

ACHIEVING FASTER STARTUP TIMES

The amount of processing time required to obtain a prompt for the first SQL request you submit on a new database connection will be influenced by the size of the schema of the database that you are trying to access. You may be able to shorten this delay if you are able to limit access to some of the data in the underlying Versant database. However, you will need to know exactly those classes (tables) for which access is unnecessary.

Mapping a Subset of the Object Schema

By default, upon connection to a database, Versant ReVind loads all the object schema from a Versant database. Depending on the size of the schema, this could be a fairly lengthy startup operation. You may only need access to a small subset of the full database schema using SQL. To support a mode of operation with reduced scope of access, Versant ReVind Mapper checks the contents of the class `VQNeededClasses` to determine whether it should map a more limited subset of the schema. If there are no instances of this class (which corresponds to the absence of rows in the `VQNeededClasses` table when viewed from Versant ReVind), then the Versant ReVind Mapper maps all class objects to the generated SQL table schema.

As database administrator (dba), you can use SQL to insert entries in the table, `VQNeededClasses` to select just those classes you want to be visible within the Versant ReVind schema as tables. For example, to limit the generated SQL schema to include only the `Employee` table, you could enter:

```
ISQL> INSERT INTO "VQNeededClasses" (className) VALUES ('Employee')
ISQL> COMMIT WORK;
```

AVOIDING THE TRUNCATION OF LONG CHARACTER STRINGS

If you know that your underlying Versant database has strings of length greater than 128 characters, and you want full access to them from Versant ReVind, you should reset each of their maximum lengths.

By inserting values greater than 128 for the value of the `maxLength` column of a new row in the `VQColumnMaxLength` system table, you will be able to read and write strings longer than 128 characters for those items that received explicitly defined maximum lengths. Note that the `VARCHAR` SQL data type, which is the variable-length string type that is automatically mapped for any Versant database character strings, cannot exceed the upper limit described in the "Versant ReVind Common Language Elements" chapter.

For example, to set a maximum length of 256 for the `name` column of the `Employee` table:

```
ISQL> INSERT INTO "VQColumnMaxLength" (tableName, columnName,
maxLength) VALUES (’Employee’, ‘name’, 256);
ISQL> COMMIT WORK;
```

For more information, please refer Chapter 5 “Versant ReVind Common Language Elements” on page 131.

MAINTAINING STRING LENGTHS LESS THAN 128

Because by default Versant ReVind generates relational schema where strings are permitted to be as long as 128 bytes, you may wish to add a constraint that does not allow the entry of strings that long.

For example, to keep the string value for `name` in class `Employee` under 50 characters, you could enter the following SQL command after logging in as the dba for your database:

```
ISQL>  INSERT INTO "VQColumnMaxLength" (tableName, columnName, maxL
ength)
   1>  VALUES ('Employee', 'name', 50);
ISQL>  COMMIT WORK;
```

TABLE NAMES

In Versant ReVind, table names are treated in case-insensitive fashion. Like string literals, they need to be enclosed in double-quote characters, hence the syntactic moniker "Delimited Identifier."

In certain cases, the surrounding delimiter characters are necessary for other reasons: you may need to refer to a table that corresponds to a class that was named with a leading underscore character, or other character that is normally disallowed as a component of an SQL identifier.

For example, if `jill` is the owner of the oddly-named table `_Author`, you can nevertheless access it with the following syntax:

```
SELECT * FROM "jill"."Author";
```

To avoid the need to always use a “Delimited Identifier” to refer to a table, you can use a synonym. To help you create a set of synonyms with which to access each table in one or more databases, you can run the `sqlutil` utility. The `sqlutil` utility is located in the `bin` directory underneath the installation directory. You should add this directory to your command search paths in your shell’s `PATH` environment variable.

NOTE:- It will even make synonyms for tables for which you are the owner. That way, you can universally refer to tables without making reference to any owner, and without double-quote delimiters.

The form of `sqlutil` command that you should use is:

```
sqlutil -S <database>
```

After running the `sqlutil` utility this way, you can refer to tables in any of the specified database in case-insensitive fashion, using only the table name.

For example, to create a synonym for table `Author` in `db1`, run the following:

```
sqlutil -S db1
```

Afterwards, you can access table `Author` in any of the following ways:

```
isql db1
ISQL> SELECT * FROM Author;
ISQL> SELECT * FROM author;
ISQL> SELECT * FROM aUthor;
```

To create synonyms, the `sqlutil` utility directly modifies the extended (system) table: `syssynonyms`. By querying this table before and after you run the `sqlutil` utility, you should be able to see exactly which synonyms were created.

Refer to the section, “Retrieving Meta-schema Information ” on page 60 to learn about querying system tables.

Also note that you can use the normal SQL `CREATE SYNONYM` command from within `isql` to create a synonym.

For example:

```
isql db1
ISQL> CREATE SYNONYM author FOR "Author";
ISQL> COMMIT WORK;
ISQL> SELECT * FROM author;
```

Table Name Syntax

Following is a list of the syntax that are to be used for Versant ReVind table names:

- Use double-quotes around table names.
- Table names in SQL are case-insensitive, whereas class names in a Versant Database are case-sensitive.
- If a table does not have a synonym, it should be accessed with double-quotes around the table name.

  For example:

  ```
  ISQL> select * from "Author";
  ```
- If a table has a synonym, the synonym need not be double- quoted.

  For example:

  ```
  ISQL> create synonym author for "Author";
  ISQL> select * from author;
  ISQL> commit work;
  ```
- Non-owners must prefix a table name with the owner name.

  If you are not the owner of a table, you must prefix a table name with the owner name

  Every user table presented by Versant ReVind Mapper is owned by dba of Versant ODBMS. In SQL-92, if the user that is accessing a table is not the owner of the table, the table name should be prefixed by the owner name.

For example if the dba is "jill":

```
ISQL> select * from "jill"."Author";
```

However, if the user has created a synonym for that table, then the user is the owner of the synonym; in this case the user need not prefix the owner name when accessing the table using the synonym.

For example:

```
ISQL> select * from author;
```

OPENING AND CLOSING DATABASE CONNECTIONS

How you refer to a database that you want to open is subject to varied requirements depending on how the database is hosted with respect to Versant and Versant ReVind servers.

How you refer to a database may also be subject to constraints depending on its preparation for use as a served database. (The `schload` utility must be used to generate a table schema for it directly, otherwise it cannot be opened other than from a multiple-database open request. See the preceding section "Preparing a Versant Database for SQL Access" as well as the section "Opening Multiple Databases.") In general you use the `isql` and `misql` commands to open database connections.

Referring to Remotely and Locally Hosted Databases

There are potentially three different hosts:

Versant ReVind client host

The host on which the Versant ReVind utilities (`isql, misql`) are executed.

Versant ReVind server host

The host on which the Versant ReVind server process (`dhserver`) is installed and is running.

Versant database host

The host on which the Versant database resides.

Database Name Syntax

Following is a discussion of the syntax to use for each case:

For local Versant database access, use the simple database name.

To access a local Versant database with Versant ReVind, simply use the database name when invoking the Versant ReVind utilities `misql` or `isql` or when specifying the database name to Versant/ODBC. (All the options except `misql` require the services of a `dhserver` daemon.)

For example:

```
isql mydatabase
```

For remote Versant database access, use `database@host` syntax.

To access a Versant database on a host that is remote to the Versant ReVind server host, use the fully qualified Versant database name when invoking the Versant ReVind utilities `isql` or `misql` or when specifying the database name to Versant/ODBC. (All the options except `misql` require the services of a `dhserver` daemon.)

For example:

```
isql mydatabase@remotedatabasehost
```

For remote Versant ReVind access, use `versant:T:host:db` syntax.

To access Versant ReVind on a remote host, use the following general syntax:

```
versant:T:remoteSqlHost:dbname
```

In the above, substitute the name of the Versant ReVind server host for `remoteSQLHost` and the database name for `dbname`.

For example:

```
isql versant:T:sqlserver:db1
```

Opening Multiple Database Connections

Versant ReVind can connect to one or more Versant databases. When you are connected to multiple databases, you appear to get a single logical database that includes classes from all the databases.

Class Definitions

So that classes can be presented coherently, the definitions of the classes you will access with Versant ReVind should have the same definition in all connected databases.

Tables that represent the same class in two or more databases will have a pseudo-column named `DBNumber`. The first database that is specified in the connection statement will have a `DBNumber` of `0`, the second a `DBNumber` of `1`, and so on. You can then use the column `DBNumber` to restrict a query to a specific database. By default, unless a restriction on the column `DBNumber` is specified, a query is distributed to all connected databases that have a class that maps to the table being queried.

Connection Syntax

You can connect to two or more Versant databases by concatenating the database names with the "+" symbol.

For example, the following command will start the interactive query tool `misql` and connect to the databases named `database1, database2, and database3`. There should be no spaces between the database names and the + symbol.

```
misql database1+database2+database3
```

The first database that is specified must have the schema for the base and extended tables. Accordingly, the Versant dba for that database should have previously run the `schload` initialization and configuration utility for that database. In case of `isql`, `schload` utility must be run on all of the participating databases while in case of `misql` only the first database should have previously run the `schload` utility.

For example, consider a class named `Employee` that is located in the databases `database1 and database2`:

```
class Employee (database1)
------------------------
  Name          Salary
------------------------
  John Smith    25000
  Jill Barns    45000
------------------------
class Employee (database2)
------------------------
  Name          Salary
------------------------
  Tom Rich      125000
  Jane Money    145000
------------------------
```

The following session in `misql` illustrates a multi-database query over class `Employee` in `database1 and database2`:

```
misql database1+database2
ISQL>  select * from "Employee";
ISQL> ----------------------------------
ISQL>  Name         Salary      DbNumber
ISQL> ----------------------------------
ISQL>  John Smith    25000        0
ISQL>  Jill Barns    45000        0
ISQL>  Tom Rich      125000       1
ISQL>  Jane Money    145000       1
ISQL>
ISQL>  select * from "Employee" where dbnumber = 0;
ISQL> ----------------------------------
ISQL>  Name         Salary      DbNumber
ISQL> ----------------------------------
ISQL>  John Smith    25000        0
ISQL>  Jill Barns    45000        0
```

LIMITATIONS FOR SQL QUERIES

Following are the maximum sizes for various attributes of a Versant ReVind environment and queries:

SQL Element	Maximum Size
Maximum length of an SQL statement	50,000 (For MISQL) 35,000 (For ISQL)
Maximum length of a local database name in connection string. When referring to a remote database, you must append a site name using the `database@node` syntax. The site name node must be no longer than 223 characters.	1024
Maximum length for a user-name in a connect string	128
Maximum number of database connections	10
Maximum number of nesting levels in an SQL statement	25
Maximum number of table references in an SQL statement	250
Maximum size of input parameters for an SQL statement	512
Maximum number of outer references in an SQL statement	25
Maximum nesting level for view references	25

RETRIEVING META-SCHEMA INFORMATION

System Tables

For each database that you open, you have access to the metadata that Versant ReVind maintains in a set of system tables. These system tables store information about table spaces, tables, columns, indexes, constraints, and privileges.

Any `GRANT` and `REVOKE` statements that you enter cause Versant ReVind to update certain of these system tables. Users must have read access to the system catalog tables. The database administrator has update access to these tables, but should only effect changes to them through SQL `GRANT` and `REVOKE` commands.

There are two types of tables in the system catalog: base tables and extended tables. Base tables store the information on the table spaces, tables, columns, and indexes that make up the database. The extended tables contain information on constraints, privileges, and statistical information.

The owner of the system tables is `vsqldba`. If you connect to a Versant ReVind environment with a User ID other than `vsqldba`, you need to qualify references to the tables in SQL queries. For example:

```
SELECT * FROM vsqldba.systables
```

The following shows details of the columns in selected system tables. Here is the SQL query that generated the data for this table. You can modify it to generate a similar list that includes user-created tables by omitting the line `and st.tbltype = 'S'`.

```
ISQL> select sc.tbl 'Table', sc.col 'Column', sc.coltype 'Data Type',
   1> sc.width 'Size'
   2> from vsqldba.syscolumns sc, vsqldba.systables st
   3> where sc.tbl = st.tbl and st.tbltype = 'S'
   4> order by sc.tbl, sc.col
```

sys_keycol_usage

This extended table contains one entry for each column on which a primary or foreign key is specified.

```
sys_keycol_usage
Columns          Data Type     Size
owner            varchar       32
colposition      integer       4
colname          varchar       32
cnstrname        varchar       32
tblname          varchar       32
```

sys_ref_constrs

This extended table contains one entry for each referential constraint specified on a user table.

```
sys_ref_constrs
Columns          Data Type    Size
refcnstrname     varchar      32
deleterule       varchar      1
owner            varchar      32
tblname          varchar      32
cnstrname        varchar      32
refowner         varchar      32
reftblname       varchar      32
```

sysattachtbls

This base table contains one entry for each table link.

```
sysattachtbls
Columns          Data Type     Size
db_link          varchar       32
linkowner        varchar       32
remtbl           varchar       32
remowner         varchar       32
tbl              varchar       32
owner            varchar       32
```

syscolauth

This extended table contains the update privileges held by users on individual columns of tables in the database.

```
syscolauth
Columns          Data Type     Size
ref              varchar       1
upd              varchar       1
col              varchar       32
tbl              varchar       32
tblowner         varchar       32
grantee          varchar       32
grantor          varchar       32
```

syscolumns

This base table contains one row for each column of every table in the database.

```
syscolumns
Columns          Data Type     Size
dflt_value       varchar       250
nullflag         varchar       1
scale            integer       4
width            integer       4
coltype          varchar       10
owner            varchar       32
id               integer       4
col              varchar       32
tbl              varchar       32
```

sysdatatypes

This extended table contains information on each data type supported by the database.

```
sysdatatypes
```

Columns	Data Type	Size
localtypename	character	1
autoincr	smallint	2
odbcmoney	smallint	2
unsignedattr	smallint	2
searchable	smallint	2
casesensitive	smallint	2

nullable	smallint	2
createparams	varchar	128
dhtypename	varchar	128
datatype	smallint	2
typeprecision	integer	4
literalprefix	character	1
literalsuffix	character	1
minimumscale	smallint	2
maximumscale	smallint	2
sqldatatype	smallint	2
sqldatetimesub	smallint	2
numprecradix	integer	4
intervalprecision	smallint	2

sysdbauth

This extended table contains the database-wide privileges held by users.

sysdbauth

Columns	Data Type	Size
res_acc	varchar	1
grantee	varchar	32
dba_acc	varchar	1

sysdblinks

This extended table contains one entry for each data source link.

sysdblinks

Columns	Data Type	Size
host	varchar	128
password	varchar	30
username	varchar	32
db_link	varchar	32
owner	varchar	32

sysidxstat

This extended table contains index statistics for each index contained by the database.

```
sysidxstat
Columns          Data Type     Size
nlevels          smallint      2
recsz            integer       4
nleaf            integer       4
idxid            integer       4
tblid            integer       4
```

sysindexes

This base table contains one row for each component of an index in the database. For an index with `n` components, there will be `n rows` in this table.

```
sysindexes
Columns          Data Type     Size
idxseq           integer       4
idxorder         varchar       1
idxtype          varchar       1
colname          varchar       32
tblowner         varchar       32
tbl              varchar       32
idxowner         varchar       32
idxname          varchar       32
idxsegid         integer       4
id               integer       4
idxcompress      varchar       1
idxmethod        varchar       1
```

syssynonyms

This extended table contains one entry for each synonym contained by the database.

```
syssynonyms
Columns          Data Type     Size
sremdb           varchar       32
ispublic         smallint      2
stblowner        varchar       32
stbl             varchar       32
```

```
screator          varchar       32
sowner            varchar       32
sname             varchar       32
```

systabauth

This extended table contains the table privileges held by users.

```
systabauth
Columns           Data Type     Size
grantor           varchar       32
grantee           varchar       32
tblowner          varchar       32
tbl               varchar       32
ins               varchar       1
del               varchar       1
upd               varchar       1
ref               varchar       1
alt               varchar       1
ndx               varchar       1
sel               varchar       1
```

systables

This base table contains one entry for each table contained by the database.

```
systables
Columns           Data Type     Size
has_ucnstrs       varchar       1
has_ccnstrs       varchar       1
has_fcnstrs       varchar       1
has_pcnstrs       varchar       1
segid             integer       4
tblpctfree        integer       4
tbltype           varchar       1
owner             varchar       32
creator           varchar       32
tbl               varchar       32
id                integer       4
rssid             integer       4
tbl_status        varchar       1
```

systblspaces

This base table contains one entry for each table space in the database.

```
systblspaces
Columns          Data Type     Size
tsname           varchar       32
id               integer       4
```

systblstat

This extended table contains table statistics for each user table.

```
systblstat
Columns          Data Type     Size
tblid            integer       4
card             integer       4
pagesz           integer       4
recsz            integer       4
npages           integer       4
```

sysviews

This extended table contains information on each view contained by the database.

```
sysviews
Columns          Data Type     Size
seq              integer       4
viewtext         varchar       900
viewname         varchar       32
creator          varchar       32
owner            varchar       32
```

MAPPING INFORMATION IN UTILITY TABLES

Overview

The tables that are known as utility tables are initially created empty. These are the only metadata tables that you may alter directly, with the exception of `VQDuplicatedTables`. Through these tables, you can affect how classes will be mapped to relational database tables. The utility tables all begin with the letters `VQ`.

Several topics elsewhere in this chapter show the usefulness of the utility tables.

See "Customizing the Mapped Table Names" on page 69 below for information about how to obtain generated table names that are shorter (or otherwise different) than their corresponding class names.

See "Achieving Faster Startup Times" on page 49, "Avoiding the Truncation of Long Character Strings" on page 50 and "Maintaining String Lengths Less Than 128" on page 51 for descriptions of other tasks involving utility tables.

Definitions for Utility Tables

A definition for each of the utility tables follows:

VQColumnMaxLength

This utility table can contain one record for each string-valued column for which you wish to establish a maximum-length setting other than the default of 128 using `maxLength`.

```
VQColumnMaxLength
Columns          Data Type     Size
SelfOID          char          19
tableName        varchar       128
columnName       varchar       128
maxLength        int           4
```

VQNeededClasses

This utility table can contain one record for each class that must be mapped to a corresponding SQL table for access through Versant ReVind in future sessions. If this table is left empty, all of the available classes will continue to be mapped to SQL tables. If this table is altered, its effect will not be noticed until all connections to the database are dropped and reestablished.

```
VQNeededClasses
Columns          Data Type     Size
SelfOID          char          19
className        varchar       128
```

VQPrefixString

This utility table can contain records that help specify alternative names for tables rather than the default table names that Versant ReVind normally chooses to correspond exactly to class names. The base name of the class is kept but the leading portion of the class name matching the `classPrefix` column is dropped and replaced with the value in the `replaceWith` column.

```
VQPrefixString
Columns          Data Type     Size
SelfOID          char          19
classPrefix      varchar       128
replaceWith      varchar       128
```

VQDuplicatedTables

This utility table can contain records that assign distinct names for tables that would otherwise be assigned duplicate names when names if they were developed solely in accordance with the `VQPrefixString` table described preceding.

```
VQDuplicatedTables
Columns          Data Type     Size
className        char          128
tableName        char          128
```

Customizing the Mapped Table Names

One way that you can customize the object-to-SQL mappings is in terms of the naming of the tables that correspond to classes. C++ class names can be shortened during the mapping process to shorter table names in the final SQL schema.

The `VQPrefixString` exists solely for this purpose. It is created with no entries initially, which leaves long class names intact for use as table names. Consider the following modification to map the prefix `com.versant` to an empty string during the generation of schema names for tables:

```
ISQL> INSERT INTO "VQPrefixString" (classPrefix, repaceWith)
   1> VALUES ('com.versant','');
ISQL> COMMIT WORK;
```

Suppose also that your Versant database includes classes named `versant.com.ListOfObject` as well as a class named `ListOfObject`. With the `VQPrefixString` table set as previously shown, two classes will be staged to receive the same table name. To resolve the ambiguity, the mapping engine automatically enters records for each duplicated table in `VQDuplicatedTables`. Each of these records reveals a mapping to a unique table name from a given class name. The query you should enter to discover any explicitly resolved mappings, is:

```
ISQL> SELECT className, tableName FROM "VQDuplicatedTables")
CLASSNAME                      TABLENAME
---------                      ---------
com.versant.ListOfObject       List.Of.Object
ListOfObject                   List.Of.Object_1
```

To discover those classes that will require conflict-resolving entries to be inserted into the `VQDuplicatedTables` upon startup without first having to close and reopen a database, you can run the `mapper` command after you commit each of a series of changes to the `VQPrefixString` table.

SETTING THE SYSTEM MEMORY CACHE

Main-memory storage system characteristics are established for all clients of a Versant ReVind server using the environment variables of the server system. The main-memory storage system provides a mechanism for Versant ReVind server to cache data in memory instead of on disk.

This cache is not the same as the Versant object cache, which is used for persistent objects. By using the main-memory storage system for volatile data such as temporary tables and dynamic indexes, you can improve performance of many queries, such as joins.

Depending on the amount of memory available on the Versant ReVind server system, certain queries may create temporary tables too large to be stored in memory. In such cases, the main-memory storage system swaps blocks of data to a disk file as necessary.

Depending on the platform you are using, the method you use to establish these settings differs.

For UNIX systems, you typically establish environmental settings in the shell startup files for your preferred shell, such as `.profile (or .shrc)` for the Bourne and Korn shells.

For example, if you use the `csh` command interpreter of UNIX systems to set a 2 megabyte cache, you could rely upon the following command line:

```
setenv TPE_MM_CACHESIZE 2000
```

To achieve the same effect on Windows, use the `Environment` tab within the `System` control panel available through the `Settings` command in the `Start Menu`.

The following environment variables control the characteristics of how the main-memory storage system uses memory to create temporary tables.

`TPE_MM_CACHESIZE`

Specifies the size, in kilobytes, of the main memory cache used by the flat file system for temporary tables.

The default value is `1000 KB` of memory.

The main-memory storage system uses the cache for storing temporary tables for sorting and creating dynamic indexes during processing. Increasing the `TPE_MM_CACHESIZE` value should improve performance (for certain queries).

`TPE_MM_BLOCKSIZE`

Limits the maximum size of a row the main-memory storage system uses in internal temporary tables.

The size is specified in kilobytes. The default is `4 KB`.

Increase this value only if complex queries generate result sets whose rows are greater than `4 KB`. Complex queries include those that join multiple tables or include an `ORDER BY` clause in `SELECT` statements.

`TPE_MM_SWAPSIZE`

Specifies the size, in kilobytes, of the swap file the main-memory storage system uses when it writes to disk from its main memory cache.

The default value is `40,000 KB`.

This value is typically not changed from the default.

SETTING THE DEFAULT DATE FORMAT

The default date format is established for all clients of a Versant ReVind server using the environment variable `TPE_DFLT_DATE`.

Changing the default date format affects:

- The default output format of date values.
- The interpretation by Versant ReVind of date literals in queries and `INSERT` statements.

If `TPE_DFLT_DATE` is not set, the default format is `US`.

Following are values and formats that can be specified with `TPE_DFLT_DATE`:

```
Value    Default Format
US         mm/dd/yyyy
UK         dd/mm/yyyy
ISO        yyyy/mm/dd
```

For a new value to take effect, the value of `TPE_DFLT_DATE` must be changed before starting the `dhserver` daemon or before running `misql`.

Controlling Interpretation of Years in Date Literals With DH_Y2K_CUTOFF

By default, SQL generates an Invalid date string error if the year component of date literals is specified as anything but 4 digits. The `DH_Y2K_CUTOFF` runtime variable changes this default behavior to allow 1- and 2-digit year specifications and control how SQL interprets them. (3-digit year specifications always generate an error.)

The following table lists the different values for `DH_Y2K_CUTOFF` and how SQL interprets them:

TABLE 1. Table: Values of DH_Y2K_CUTOFF Runtime Variable

Value	Interpretation of 1- or 2-Digit Year in Date Literals
Not set	
Set to no value	1- or 2-digit years not allowed
Less than zero	
Greater than 100	
0	20th century: Adds 1900 to value (for instance, 99denotes 1999).
100	21st century: Adds 2000 to value (for instance, 99denotes 2099).
Greater that zero and less than 100	Depends on value of literal:· If value is less than `DH_Y2K_CUTOFF`, 21st century · If value is greater than or equal to `DH_Y2K_CUTOFF`, 20th century

SWITCHING ACTIVITY LOGGING ON AND OFF

By setting the environment variable TPESQLDBG at the Versant ReVind server system, you can enable or disable various types of activity logging.

For a new setting to take effect, the Versant ReVind daemon, dhserver must be stopped first. The required sequence is: stop the daemon, reset TPESQLDBG, then restart the daemon.

The Versant ReVind log file is name sql_server.log and is in the directory referenced by the environment variable TPEDATADIR.

The log file output is useful when debugging connectivity problems between the Versant ReVind clients and the Versant ReVind server. It also serves to understand the performance characteristics of queries.

The TPEDATADIR variable has the following format:

```
TPESQLDBG "ABCD"
```

For each character position in this string of four characters, you can specify either Y or N to enable (Y) or disable (N) each of several categories of logging.

For example, to use the csh command interpreter of UNIX systems to enable the first and fourth categories of logging, enter the following command line:

```
setenv TPESQLDBG "YNNY"
```

Y at Position 1 (Statement Activity Logging)

Causes the Versant ReVind server to log details of how the Versant ReVind server processes SQL statements passed to it by applications. Details include:

- Original SQL statement passed by the application
- Decomposition of the statement by the Versant ReVind parser
- Optimization strategy chosen by the Versant ReVind optimizer

Y at Position 2 (Tree Cache Size Logging)

Causes the Versant ReVind server to log the cache size of the binary trees created during processing. Generally, you will want to set this position N (disabled).

Y at Position 3 (Dictionary Activity Logging)

Causes the Versant ReVind server to log internal information about the actions of the data dictionary manager. Generally, you will want to set this position `N` (disabled).

Y at Position `4` (Run Optimization Activity Logging)

Causes the Versant ReVind server to log details of runtime operations performed by the Versant ReVind optimizer.

Generally, when you enable logging, you set `TPESQLDBG` to `"YNNY"`. For best performance, however, and to limit the size of the log file, do not set `TPESQLDBG`.

EXPORTING DATA FROM A VERSANT REVIND DATABASE

The `dbdump` utility exports records from a database to a file.

For more information refer to “dbdump” on page 108, in "Chapter 4 - Versant ReVind Utilities Reference".

You can use `dbdump` to export variable or fixed length records. Data files can use multiple-character record delimiters.

SECURITY

Versant ReVind provides two levels of security:

- Versant security for database connections -

  Only users that have been allowed access to a Versant database by the dba are permitted to use Versant ReVind to access the database.

  This authentication is based on the user-id (on Unix) or user-name (on Windows) of the user that attempts to connect to Versant ReVind (either using the Versant/Interactive Query Tool or the Versant/ODBC driver). A database connection will be rejected if the user is not allowed access to the database involved.

  User access to a Versant database is controlled with the Versant `dbuser` utility.

- SQL security for table accesses -

  This authentication takes place differently for the two different table types. For system tables, exclusive access is accorded to the underlying Versant database dba. For other tables, authentication can take place on a per-table basis for those users who have been given database connection access.

Securing Database Connections

Authentication of database connections takes place on a per-user basis. So the significant events that will cause renewed consideration of database connection security is either the addition of a new database, or a new user, or both. Such events occur even for Versant servers without an associated Versant ReVind server. Therefore, the administration of these security settings is the same as that for any Versant system:

- New users -

  Newly added users will not have database connection access until the dba (owner) for the database explicitly gives them access using the `dbuser` command.

- New databases -

  Neither new nor existing users will have access to a newly deployed database (only the owner of the database will have access to it at the outset). In this case, the `dbuser` command must be run for both new and existing users.

  When a database is initially prepared for Versant ReVind access with the `schload` utility, `schload` grants dba privilege to the underlying Versant dba.

- Changed user roles -

When one or more users are transferred or become otherwise employed, you may need to remove or otherwise change their access privileges. Again, the command used is no different for Versant ReVind server than for a Versant server.

See your Versant *Database Administration Manual* for additional information about the dbuser utility.

Securing Table Access

Securing table access will be a renewed consideration when certain milestones (or events) are encountered. When a Versant database is first prepared to also function as a Versant ReVind database is one important event. The addition of new accounts for existing or new Versant ReVind client systems is another time to reconsider table access settings. Also, the departure of users may warrant renewed consideration of table access.

The addition of new tables (due to the addition of new classes in a Versant database that is also set up as a Versant ReVind access) also warrants the reconsideration of table access settings.

- Consideration of newly added databases -

  You caused the initial system tables for authentication to be built when you ran the `schload` utility to prepare a Versant database for SQL access. This process granted `SELECT` privileges to `PUBLIC`, which effectively grants all existing and any new users read access to all tables—if they can get a database connection at all (as described in the preceding section). If you need to restrict access not just by who is allowed to connect to new databases with read access, then you need to establish additional table-level security settings.

- Consideration of newly added users -

  New users will automatically have `SELECT` permissions on all tables constructed from the database, because `schload` previously granted `SELECT` to `PUBLIC`. So, for each database within which you need to restrict read access to specific tables and for which you have granted one more new users connection access, you will most likely revoke the `SELECT` table access privilege that was automatically given to new users into tables that you want to continue treating as confidential. Similarly, for each database for which you want one or more new users to have write access to one or more tables, you will need to grant one or more of the `INSERT, DELETE`, and `UPDATE` privileges for each table and each new user. Of course, for privileges that were granted by way of `PUBLIC`, the exercise of granting write privileges to new users for each table can be avoided. Similarly, the use of the table wildcard `ALL` can considerably streamline the number of explicit `grant` or `revoke`

commands you need to enter when you have more than one new user and more than one table in a database.

- Consideration of newly added classes (equivalent to newly added tables) -

  The process of adding new classes effectively adds tables in terms of Versant ReVind access.

  To at least permit read access to all users for any new and all existing tables, you should use the Versant ReVind utility `sqlutil`.

  For example, to grant `SELECT` to `PUBLIC` on database `db1`:

  ```
  sqlutil -G db1
  ```

  Additionally, privileges other than `SELECT` should be considered for each user with respect to each new table.

To manipulate access permissions that relate to operations on data for one or all tables and for one or all users, you use the SQL `grant` and `revoke` commands. These commands can make reference to the following operations: `SELECT, INSERT, UPDATE`, and `DELETE`.

For example, the following is a sample session that gives to user `john` the ability to `INSERT` data into table `Author` in database `db1`. The following also gives all database users `INSERT` privileges on table `Book`:

```
misql  db1
ISQL> grant INSERT on "Author" to ‘john’;
ISQL> grant INSERT on "Book" to PUBLIC;
ISQL> commit work;
ISQL> quit;
```

CHAPTER 3

Versant Interactive SQL Tool

This Chapter provides the Versant Interactive SQL tool usage notes & command descriptions.

The Chapter covers the following in detail:

- Versant Interactive SQL Tool Usage Notes
- Versant Interactive SQL Commands

VERSANT INTERACTIVE SQL TOOL USAGE NOTES

Overview of Versant Interactive SQL

Versant/Interactive SQL lets you issue SQL commands directly from a terminal and see results displayed at the terminal. You can use interactive SQL to:

- Learn how SQL statements work
- Test and prototype SQL statements to be embedded in programs
- Modify an existing database with data definition statements
- Perform ad-hoc queries and generate formatted reports with special ISQL formatting commands

With few exceptions, you can issue any SQL statement in interactive SQL that can be embedded in a program, including `CREATE`, `SELECT`, and `GRANT` statements. Interactive SQL includes an online help facility with syntax and descriptions of the supported statements.

To use Versant Interactive SQL:

1. Start the Versant Interactive SQL Tool.
2. Issue Versant ReVind commands at the `ISQL>` prompt and terminate them with a semicolon. Commands can be entered using either uppercase or lowercase characters.

   You can continue commands on multiple lines. `ISQL` automatically prompts for continuation lines until you terminate the command with a semicolon.

   To execute host operating system commands from the `ISQL` prompt, type `HOST` followed by the operating system command. After completion of the host command, the `ISQL>` prompt returns. To execute SQL scripts from `ISQL`, type `@` followed by the name of the file containing SQL commands.

Types of Versant Interactive SQL Commands

Versant Interactive SQL accepts four types of commands:

- Standard SQL statements
- Report generation commands
- Transaction commands

- History management commands

Generally, standard SQL statements have the same syntax in `ISQL` as in embedded SQL, with the following exceptions:

- Statements do not begin with `EXEC SQL`
- Statements cannot refer to host variables (for example, the `SELECT INTO` statement is not supported)

Starting and Stopping the Versant Interactive SQL Tool

Before Starting the Versant Interactive SQL Tool

Before invoking the Versant Interactive SQL Tool do the following:

1. Make sure that you have set your Versant ReVind environment as per the installation instructions in your release notes.
2. Make sure that the Versant ReVind daemon is running.

Starting the Versant Interactive SQL Tool

The Versant Interactive SQL Tool can be run in either of two modes: merged or network. Which mode is used depends on how you invoke it.

Merged mode-

If you invoke the Versant Interactive SQL Tool with the `misql` command, it will be run in a single process with Versant ReVind and Versant ReVind Mapper.
You can use `misql` to connect with either a local or remote database, but it can be invoked only on a machine on which Versant ReVind software has been installed.

Network mode-

If you invoke the Versant Interactive SQL Tool with the `isql` command, it will run as a process and communicate using network services with another process running Versant ReVind and Versant ReVind Mapper.
You can use `isql` to connect with either a local or remote database. You do not have to run `isql` on the machine on which Versant ReVind software has been installed.

Stopping the Versant Interactive SQL Tool

To exit from interactive SQL, type `EXIT` or `QUIT`.

Versant Interactive SQL Tool Syntax

`misql` and `isql`

The general syntax for `misql` is:

```
misql [options] database_name
```

The general syntax for `isql` is:

```
isql [options] database_name
```

Issue the `isql` or `misql` command at the shell prompt. You will then see the `ISQL` or `MISQL` prompt. Once you have started the Versant Interactive SQL Tool, you can issue `ISQL` commands at the prompt.

For information on how to specify a database name, please refer “Database Name Syntax” on page 55 in "Chapter 2 - Initiating Versant Revind".

You can supply optional switches and arguments to the `isql` and `misql` commands. Options are:

`-s script_file`

The name of an SQL script file that Versant ReVind executes when it invokes `ISQL`.

`-u user_name`

The user name Versant ReVind uses to connect to the database specified in the `connect_string`. Versant ReVind verifies the user name against a corresponding password before it connects to the database. If omitted, the default value depends on the environment. (On UNIX, the value of the `DH_USER` environment variable specifies the default user name. If `DH_USER` is not set, the value of the `USER` environment variable specifies the default user name.)

`-a password`

The password Versant ReVind uses to connect to the database specified in the `connect_string`. Versant ReVind verifies the password against a corresponding user name

before it connects to the database. If you use -u option and not the -a option, then it will prompt you to enter the password.

`-n`

The `-n` option is used to disable the `ISQL>` prompt, that is the prompt will not be displayed.

`connect_string`

A string that specifies which database to connect to. The connect string can be a simple database name or a complete connect string. If omitted, the default value depends on the environment (on UNIX, the value of the `DB_NAME` environment variable specifies the default connect string).

Formatting and Printing Reports

Formatting of database query results makes the output of a database query more presentable and understandable. The formatted output of an `ISQL` database query can be either displayed on the screen or spooled to a printer to produce a hardcopy of the report.
`ISQL` supported commands allow the specification of column headings, page/screen titles and allow the computation and printing of additional information at specified points in the query results.

The following subsections explain how this functionality can be used to format the display output of a database query:

Formatting Columns

When the results of a column are displayed the database column name is used as the default heading. Since column names can be cryptic the `COLUMN` command can be used to specify a more understandable heading as shown below:

```
COLUMN cust_name HEADING 'Customer Name';
```

Instead of the cryptic `cust_name` the more readable `Customer Name` is printed as the heading for the column.
The `COLUMN's FORMAT` option can be used to specify the display format of data printed for a column. The various format options provided by `ISQL` are given in the reference section for the `COLUMN` statement.

The following example illustrates the use of the `COLUMN's FORMAT` option:

```
COLUMN base_price FORMAT "$999.99";
```

Specifying Titles

ISQL provides functionality for specifying a title which will be printed on each page of the report. The title can be optionally printed on either the top or the bottom of each page. The title can also be horizontally positioned by specifying the keywords LEFT, CENTER, RIGHT or by specifying the actual column number corresponding to the required positioning of the title. The following example illustrates the use of the TITLE command:

```
TITLE TOP LEFT "Orders Summary" RIGHT "Sep 9, 1988" ;
```

Using the BREAK Command

The BREAK command allows the grouping of rows produced by a query and to take a specified action for a group such as skipping lines in the output or printing some computed information related to the group.
First the specification of the BREAK command is shown and then the possible actions which can be executed when the break occurs is examined. A BREAK can be specified on a column as follows:

```
BREAK ON column_name;
```

The above command specifies a break whenever the column value of the specified column changes. The use of a break on the specified column makes sense if the database query displays information with the rows ordered by the column values of the column specified in the BREAK command.

A BREAK can be specified on multiple columns as shown below:

```
BREAK ON column_name, column_name ...;
```

A BREAK can be specified on every row returned by the query, on each page of the query results or at the end of the report. The form for each of the above cases is shown below:

```
BREAK ON ROW;
BREAK ON PAGE;
BREAK ON REPORT;
```

One of the actions on the occurrence of a break can be for formatting like skipping lines or a page as shown below:

```
BREAK ON  ...  SKIP;
BREAK ON  ...  SKIP number_of_lines;
BREAK ON  ...  PAGE;
```

The other action which can be specified on a break is computing a value related to the group and displaying it in the report.
Computing the sum of a column for a group can be performed by using the following statement:

```
COMPUTE SUM OF column_name ON break_column;
```

ISQL will compute the sum of the specified column for the group formed by the break on the break_column and display it. The column_name on which the sum is being computed might or might not be the same as the break_column. More than one column_name can be specified in the COMPUTE command as illustrated in the following example:

```
COMPUTE SUM OF column_name, column_name, ... ON break_column;
```

This will compute and display the sum of each individual column specified in the COMPUTE command. SUM is one of the functions supported in the COMPUTE statement.

The following list specifies the functions supported by the COMPUTE statement:

AVG

Average value of a column in a group

MIN

Minimum value of a column in a group

MAX

Maximum value of a column in a group

SUM

Sum of the values of a column in a group

COUNT

Number of rows in the group

Online Help

ISQL supports an online help facility that can be invoked by using the HELP command. Typing HELP at the ISQL prompt will display a help file which will list the options accepted by the HELP command. The various forms of the HELP command are listed below:

HELP

Displays the options that can be specified for HELP.

HELP COMMANDS

Displays all the commands that ISQL accepts.

HELP command_name

Displays help file corresponding to the specified command.

TABLE is an ISQL command that displays all the tables present in the database including any system tables. TABLE can be used also to display the description of a single table by explicitly giving the table name. Both forms of the TABLE command are shown below:

```
TABLE;
TABLE table_name;
```

Transactions

A transaction is started with the execution of the first SQL statement. A transaction is committed using the COMMIT WORK command and rolled back using the ROLLBACK WORK command.
If the AUTOCOMMIT option is set to ON, then ISQL treats each SQL statement as a single transaction. This prevents the user from holding locks on the database for an extended period of time. This is very critical when the user is querying an online database on which a transaction processing application is executing in real time.
A set of SQL statements can be executed as part of a transaction and committed using the COMMIT WORK command. This is shown below:

```
<SQL statement>
<SQL statement>
<SQL statement>
COMMIT WORK;
```

Instead, a transaction can also be rolled back using the ROLLBACK WORK command as shown:

```
<SQL statement>
<SQL statement>
<SQL statement>
ROLLBACK WORK ;
```

An SQL statement starting immediately after a `COMMIT WORK` or `ROLLBACK WORK` command starts a new transaction.

VERSANT INTERACTIVE SQL COMMANDS

This section does not include descriptions of the standard SQL statements.

@(Execute)

```
@script_file
```

Executes the SQL commands stored in the specified SQL script file. The commands specified in the file are not added to the history buffer.
The following SQL commands may be stored in a file called `"cmdfile"`:

```
connect to
set echo on ;
create table stores (item_no integer, item_name char(20));
insert into stores values (1001,"chassis");
insert into stores values (1002,"chips");
select * from stores where item_no > 1001;
set echo off ;
```

To execute the above commands stored in "`cmdfile`", enter:

```
ISQL> @ cmdfile
```

BREAK

```
BREAK ON  {column_name[, column_name]| ROW | PAGE | REPORT }
          [SKIP n];
BREAK;
```

The `BREAK` command presents summary information on a portion of the data selected by a query. A `BREAK` can be specified on any of the following events:

- Change in the value of a column
- Selection of each row
- End of a page
- End of a report

The `BREAK` command without any clauses displays the `BREAK` that is currently in effect.

The BREAK command with one or more clauses specifies the events that cause an SQL query to be interrupted and the execution of the associated COMPUTE and DISPLAY statements.
Only one BREAK command might be in effect at a time. When a new BREAK command is entered, it replaces the previous BREAK command. The BREAK command can specify one or more columns on which the break can occur.
ON column_name causes a break when the value of the column specified by the column name changes.
ON ROW causes a break on every row selected by a SELECT statement.
ON PAGE causes a break at the end of each page. The end of a page is dependent on the page size. The page size can be set by the SET command.
ON REPORT causes a break at the end of a report or query.
The optional SKIP clause can be used to skip the specified number of lines whenever the specified break occurs.
The following set of commands compute the number of orders placed by each customer and displays the message Number of orders and the count each time the break on cust_id occurs:

```
ISQL> BREAK ON cust_id;
ISQL> COMPUTE COUNT OF order_id in n_ord ON cust_id;
ISQL> DISPLAY COL 10 "Number of orders = ", n_ord ON cust_id;
ISQL> SELECT cust_id, order_id, date_ordered
      FROM order_tbl
      ORDER BY cust_id;
```

CLEAR

```
CLEAR option;
```

The CLEAR command clears parameters set by other ISQL commands. The option specifies the parameter to clear. The CLEAR command options are as follows:

- CLEAR BREAK clears the break set by the BREAK command.
- CLEAR HISTORY clears the ISQL command history buffer.
- CLEAR COLUMN clears all the options set by the COLUMN command.
- CLEAR COMPUTE clears all the options set by the COMPUTE command.
- CLEAR DISPLAY clears the displays set by the DISPLAY command.
- CLEAR TITLE clears the titles set by the TITLE command.

To clear the break, enter:

```
ISQL> CLEAR BREAK;
```

To clear the titles, enter:

```
ISQL> CLEAR TITLE;
```

COLUMN

```
COLUMN column_name FORMAT format_string [HEADING text];
COLUMN;
```

The COLUMN command sets a format and heading specification for the column specified by column_name. The FORMAT clause specifies the format in which the data is to be displayed. The HEADING clause specifies the text of the column's heading.
The COLUMN command without any arguments displays the current column specifications.
The following are the valid combinations of format strings:

- If the column is of type CHAR, the format can be specified by a string which consists of the letter 'A' followed by the column width.
- If the column is of any of the numeric types, then the format string can consist of the following characters:

```
CHAR    EXAMPLE    DESCRIPTION
9       99999      Number of 9's specifies width.
0       09999      Display leading zeroes.
$       $9999      Value will be prefixed by '$'.
B       B9999      Display blanks if the value is zero.
,       99,999     Display a comma at position specified by the comma.
MI      99999MI    Display '-' after a negative value.
PR      99999PR    Display the negative value between '<' and '>'.
```

- If the column is of type DATE, the format string might contain any of the characters m, d, or y. Any other character in the format string is displayed literally.

  The meaning of m, d, y or dy can be suppressed by embedding them with double quotes and putting the whole FORMAT string in single quotes.

The following command specifies the format of the column base_price to be $999.99 and the heading to be Retail Price:

```
ISQL> COLUMN base_price FORMAT "$999.99"
HEADING "Retail Price";
```

The FORMAT string specified as,

```
ISQL> COLUMN order_date FORMAT 'dd, mm "order date"'
```

will display `order_date` as it is.

COMPUTE

```
COMPUTE [AVG | MAX | MIN | SUM | COUNT] OF column_name
     IN variable_name
     ON [column_name | ROW | PAGE | REPORT] ;
COMPUTE ;
```

The `COMPUTE` command performs computations on a set of selected rows when a break associated with that `COMPUTE` command occurs.

- The functions `AVG`, `MAX`, `MIN`, and `SUM` can be used only when the column type is of a numeric type. The function `COUNT` can be used for any column type.
- The `OF` clause specifies the column whose value is to be computed. The column specified in the `OF` clause must also be specified in the `SELECT` command.
- `variable_name` specifies the name of the variable where the computed value is to be stored.
- The `ON` clause specifies the break associated with the `COMPUTE`.
- `COMPUTE` without any arguments displays the currently defined `COMPUTE` specifications.

The following set of commands compute the average quantity of items ordered for the items having the same `item_id` and displays the computed value each time a break on `item_id` occurs:

```
ISQL> BREAK ON item_id;
ISQL> COMPUTE AVG OF qty IN avg_qty ON item_id;
ISQL> DISPLAY COL 10 "Average Order Quantity = ",
      avg_qty on item_id;
ISQL> SELECT * from order_detail ORDER by item_id;
```

DEFINE

```
DEFINE variable_name = value;
DEFINE ;
```

The `DEFINE` command defines a variable and assigns the specified ASCII value to it. The `DEFINE` command without any arguments displays the type and value of all the defined variables. The

defined variable can be referred to in a DISPLAY command to print the value contained by the variable.

- variable_name specifies the name by which the variable can be referenced.
- value is the ASCII string that is assigned to the variable.

The following command defines a variable called nestate and assigns the value NH to it:

```
ISQL> DEFINE nestate = "NH";
```

DISPLAY

```
DISPLAY
   { [COL n | @column_name] {"text"|variable|column}
   [,[COL n | @column_name] {"text"|variable|column},...] }
   ON { columnname | ROW | PAGE | REPORT };
```

The DISPLAY command displays the specified text, variable value and/or a column value when a break associated with that DISPLAY occurs.

- COL n specifies the column number at which the text/variable/column is to be displayed.
- @column_name displays the text/variable/column at the display position of the column specified by column_name.
- text is a character string to be displayed.
- variable is the name of the variable whose value is to be displayed.
- column is the name of the column whose value is to be displayed.
- The ON clause specifies the break associated with the DISPLAY.

The following set of commands compute the number of orders placed by each customer and displays the message "Number of orders Placed by", followed by the customer name and the count of orders:

```
ISQL> BREAK ON cust_id;
ISQL> COMPUTE COUNT OF order_id IN n_ord ON cust_id;
ISQL> DISPLAY COL 5 "Number of orders placed by ",
      cust_name, " = ", n_ord ON cust_id;
ISQL> SELECT cust_name, order_tbl.cust_id, order_id
      FROM customer, order_tbl
      WHERE customer.cust_id = order_tbl.cust_id
      ORDER BY order_tbl.cust_id;
```

EDIT

```
E[DIT] [cmdnum];
```

The EDIT command allows editing of the specified command from the command history buffer. If the command number is not specified, the last command in the history buffer is edited.
The edited command is appended in the history buffer to become the new previous command.
The environment variable EDITOR can be set to the editor of choice.
By default the vi editor is invoked by ISQL for the EDIT command on UNIX. Currently, there is no default editor on DOS. So the environment variable EDITOR must be explicitly set to be able to edit ISQL commands on DOS.
The following command allows editing of the 5th command in the history buffer:

```
ISQL> EDIT 5;
```

The following command edits the last command in the history buffer:

```
ISQL> EDIT;
```

EXIT

```
EXIT
```

The EXIT command terminates the ISQL session.
Related Commands — QUIT

GET

```
G[ET] filename;
```

The GET command reads the contents of the specified file into the current command buffer until a semicolon is encountered.
The command read by GET can be executed by invoking the RUN command.
The following example loads the contents of the file named STOCK into the current command buffer and then executes it:

```
ISQL> GET STOCK;
ISQL> RUN;
```

HELP

```
HE[LP] {COMMANDS|CLAUSES};
HE[LP] ;
```

The HELP command displays the help information for the specified command or clause.

- HELP COMMANDS displays a list of the commands that are supported by help text.
- HELP CLAUSES display a list of clauses for which help text is available.
- HELP command with no clauses display the help text for the HELP command.

The following HELP command will give a brief description of the SELECT statement.

```
ISQL> HELP SELECT;
```

HISTORY

```
HI[STORY];
```

The HISTORY command lists the most recent commands maintained in the command history buffer. A command number, associated with each command in the history buffer is also listed.
The RUN command lets you run a previously entered command by just typing this number.
The SET HISTORY command sets the size of the history buffer.
The commands LIST, EDIT, HISTORY and RUN are not added to the history buffer.
The following example lists the commands in the history buffer:

```
ISQL> HI ;
```

HOST

```
{ HOST | SH | ! } [host_command];
```

The HOST command executes a host operating system command without terminating the current ISQL session.

- If the host_command is not specified ISQL invokes the host command line interpreter and allows the user to execute any number of operating system commands before returning to ISQL.
- An exclamation mark in front of the host command can save you from typing HOST.

Either one of the following commands executes the UNIX command to list files (ls) in the current working directory:

```
UNIX:
ISQL> HOST ls ;
ISQL> !ls ;
```

LIST

```
L[IST] [cmdnum];
```

The LIST command lists the command with the specified command number from the command history buffer and makes it the current command by adding it to the end of the history list.
If the command number is not specified, the last command in the history buffer is listed.
The following command lists the 5th command in the history buffer:

```
ISQL> LIST 5;
```

QUIT

```
Q[UIT]
```

The QUIT command terminates the current ISQL session.
Related Commands — EXIT

RUN

```
R[UN] [cmdnum];
```

The RUN command runs the command with the specified command number from the command history buffer (See HISTORY.) If command number is not specified, the last command in the history buffer is rerun.
The following command runs the 5th command in the history buffer:

```
ISQL> RUN 5;
```

SAVE

```
S[AVE] filename;
```

The SAVE command saves the last command in the history buffer in the specified file. The GET command can be used to read a command from a file.

```
ISQL> SAVE /tmp/tmp1;
```

SET

```
SET PAGESIZE n;
SET LINESIZE n;
SET HISTORY n;
SET COMMAND LINES n;
SET REPORT { ON | OFF };
SET PAUSE { ON | OFF };
SET TIME { ON | OFF };
SET AUTOCOMMIT { ON | OFF };
SET ECHO { ON | OFF };
SET DISPLAY COST { ON | OFF };
SET TRANSACTION ISOLATION LEVEL { 0 | 1 | 2 | 3 };
```

The SET command changes various characteristics of an interactive SQL session. If you want your settings to be used as default, you will have to open the file sql conf from TPEROOT/lib directory and modify the required parameter with appropriate value.

Arguments:

PAGESIZE

Sets the number of lines per page. The default is 72 lines.

LINESIZE

Sets the number of characters per line. The default is 78 characters.

HISTORY

Sets the size of the history buffer. The default, and maximum, is 250 commands.

COMMAND LINES

Sets the number of command lines to be displayed. The default is 20.

REPORT ON | OFF

SET REPORT ON copies only SQL command results, not the SQL command itself, into the file opened by the SPOOL filename ON command. SET REPORT OFF copies both the SQL command and the results to the file. SET REPORT OFF is the default.

PAUSE ON | OFF

SET PAUSE ON prompts the user after displaying one page of results on the screen. This is the default mode.

TIME ON | OFF

SET TIME ON displays the time taken for executing a database query command. SET TIME OFF disables the display and is the default.

ECHO ON | OFF

SET ECHO ON option displays commands as they are executed from a command file. SET ECHO OFF option suppresses the display and is the default.

AUTOCOMMIT ON | OFF

SET AUTOCOMMIT ON commits changes and starts a new transaction immediately after each SQL command is executed. SET AUTOCOMMIT ON is the default. SET AUTOCOMMIT OFF requires that you end transactions explicitly with a COMMIT or ROLLBACK WORK command.

SET DISPLAY COST ON | OFF

SET DISPLAY COST ON displays the values the Versant ReVind optimizer uses to calculate the least-costly query strategy for a particular SQL command:

- Relative cost of execution
- Number of rows in the table (the cardinality)
- Size in bytes of the relational algebraic tree used for the query

The UPDATE STATISTICS command updates the values displayed by SET DISPLAY COST ON. SET DISPLAY COST OFF suppresses the display and is the default.

SET TRANSACTION ISOLATION LEVEL 0 | 1 | 2 | 3

Specifies the isolation level explicitly. The default is 0. The following table shows how the isolation level integers correspond to ANSI/ISO standard isolation level keywords:

Number	Isolation Level	Transactional Behavior
0	UNCOMMITTED READ	Cursor batch objects are unstable, and non-repeatable reads and phantom updates may occur.
1	COMMITTED READ	Cursor batch objects are stable, but non-repeatable reads and phantom updates may occur.
2	REPEATABLE READ	Cursor batch objects are stable, and reads are repeatable but phantom updates may occur.
3	SERIALIZABLE	Cursor batch objects are stable, reads are repeatable and phantom updates may not occur.

NOTE:- As the isolation level increases, objects and reads become more stable, but concurrency is reduced.

REPEATABLE READ is the highest isolation level and UNCOMMITTED READ is the lowest isolation level.
(Some of the behavior of an interactive SQL session is influenced through other settings as well. For example, the EDIT and SPOOL commands can be affected by environment variables.) The following example illustrates some SET commands. SET REPORT ON disables writing of SQL commands to a SPOOL output file. SPOOL outfile opens an output file called outfile. SET TIME ON enables display of the time taken for subsequent commands:

```
ISQL> SET REPORT ON;
ISQL> SPOOL outfile ON;
ISQL> SET TIME ON;
```

The following example turns off the automatic commit setting for an interactive SQL session, then commits changes from a transaction manually:

```
ISQL> SET AUTOCOMMIT OFF;
ISQL> DELETE FROM cust_acct WHERE name = "James";
ISQL> COMMIT WORK;
```

The following example sets the transaction isolation level to REPEATABLE READ:

```
ISQL> SET TRANSACTION ISOLATION LEVEL 2;
```

SHOW

```
SHOW [ PAGESIZE | LINESIZE | HISTORY | COMMAND LINES |
       REPORT | PAUSE | TIME | AUTOCOMMIT | ECHO | SPOOL |
       CONNECTION | TRANSACTION ISOLATION LEVEL ] ;
```

The SHOW command displays the values of the various options supported by the ISQL tool.

Options:

PAGESIZE

The PAGESIZE option displays the number of lines per page.

LINESIZE

The LINESIZE option displays the number of characters per line.

HISTORY

The HISTORY option displays the size of the history buffer.

COMMAND LINES

The COMMAND LINES option shows the number of lines of a command that will be displayed.

REPORT

The REPORT option displays its current value (ON/OFF).

SPOOL

The SPOOL option displays its current value (ON/OFF). If the spool option is "ON", the name of the spool file is also displayed.

PAUSE

The PAUSE option displays its current value (ON/OFF).

TIME

The TIME option displays its current value (ON/OFF).

AUTOCOMMIT

The AUTOCOMMIT option displays its current value (ON/OFF).

ECHO

The ECHO option displays its current value (ON/OFF).

TRANSACTION ISOLATION LEVEL

The TRANSACTION ISOLATION LEVEL displays the current isolation level (0|1|2|3).

CONNECTION

The CONNECTION option is used to display the status of all the databases that are started. SHOW CONNECTION displays the following for each database:

- connection name
- whether the connection is default or not
- whether the connection is the current connection or not

- The CONNECT TO command must be used to connect to a database.
- If no connection name has been associated with a connection, the database name itself becomes the connection name.
- The SET CONNECTION command must be used to make a connected database current.

```
ISQL> SHOW
                    ISQL  ENVIRONMENT
                 -------------------------

EDITOR ................... : vi
HISTORY buffer size ...... : 50    PAUSE ................... : ON
COMMAND LINES ............ : 10    TIMEing command execution : OFF

SPOOLing ..................: ON    LINESIZE .................: 78
REPORTing Facility ........: ON    PAGESIZE .................: 72
Spool File ................: spool_file

AUTOCOMMIT ................: OFF    ECHO commands ...........: ON
TRANSACTION ISOLATION LEVEL: 0 (Snapshot)

                    DATABASE CONNECTIONS
                 -------------------------

DATABASE      CONNECTION NAME      IS DEFAULT ?     IS CURRENT ?
-----   ------------   ---------  ------------
```

```
salesdb       conn_1              No              Yes
```

SPOOL

```
SPOOL filename [ON] ;
SPOOL OFF ;
SPOOL OUT ;
```

The SPOOL command writes output from interactive SQL statements to the specified file.

Arguments:

`filename ON ;`

Opens the file specified by `filename` and writes the displayed output into that file. The filename cannot include punctuation marks such as a period (`.`) or comma (`,`).

`OFF;`

Closes the file opened by the `SPOOL ON` command.

`OUT;`

Closes the file opened by the `SPOOL ON` command and prints the file. The `SPOOL OUT` command passes the file to the system utility command `pr` and the output is piped to `lpr`.
To record the displayed output into the file called `STK`, enter:

```
ISQL> SPOOL STK ON ;
ISQL> SELECT * FROM customer ;
ISQL> SPOOL OFF ;
```

START

```
ST[ART] file [argument argument ... ];
```

The `START` command executes the SQL command stored in the specified file.
If any arguments are specified then they are used for substituting any parameter references in the command file for this invocation. The first argument replaces all occurrences of `&1`, the second option replaces all occurrences of `&2`, and so on.
Unlike the `GET` command, arguments can be passed to the command stored in the file. `START` executes the command whereas the `GET` command only fetches the command to the history buffer.

A file called /tmp/cust might contain the SQL command:

```
SELECT * from customer where city = "&1" ;
```

To execute this command file with an option Oakland, enter:

```
ISQL> START /tmp/cust Oakland;
```

TABLE

```
T[ABLE] [tablename];
```

The TABLE command displays a brief description of the specified table. The description contains the following information about each column:

- The name of the table column
- Whether the column value is permitted to be Null.
  - The column's data type
  - The maximum allowed string length when the column's data type is string (VARCHAR)

If the tablename is not specified, a list of all the tables in the database owned by the current user is displayed.

To get a brief description of the table named customer, enter:

```
ISQL> TABLE customer;
```

TITLE

```
TITLE [TOP | BOTTOM] [SKIP n]
   { LEFT | CENTER | RIGHT | COL n } text;
```

The TITLE command specifies the title text to be printed together and the page position where it should be printed.

Arguments:

```
TOP | BOTTOM
```

The first item in the position specification is the optional TOP or BOTTOM keyword. This keyword specifies whether the title is to be printed on the top or bottom of the page. The default title position is the top of the page.

SKIP

SKIP can be used to skip the specified number of lines before the title is printed.

LEFT

LEFT aligns the text to the left margin. If neither CENTER nor RIGHT is specified, LEFT is the default treatment

CENTER

CENTER centers the text.

RIGHT

RIGHT pushes the text to the right margin.

COL

COL indents text to the specified column.
Entering the TITLE command with no arguments displays any previously set title.
The following command specifies a title at the top of the page for subsequent queries:

```
ISQL> TITLE TOP LEFT "Orders Summary" RIGHT "Sep 9, 1988" ;
```

CHAPTER 4

Versant ReVind Utilities Reference

This Chapter covers the details of Versant ReVind utilities.

The Chapter explains the following:

- Versant ReVind Utilities
- dbdump
- dbload
- schload
- sqlutil

VERSANT REVIND UTILITIES

dbdump

```
dbdump -f commands_file [-n] database_name
```

Export records from a database to a file.

Command elements

`commands_file`

The name of a command file.

`-n`

Parse the commands file and display errors, if any, without exporting data.
If the parsing is successful a message, "`No errors in the commands file`" will be seen on `stdout.`

`database`

The database that contains the records to export.

For information on how to specify a database name, please refer “Database Name Syntax” on page 55 in "Chapter 2 - Initiating Versant Revind".

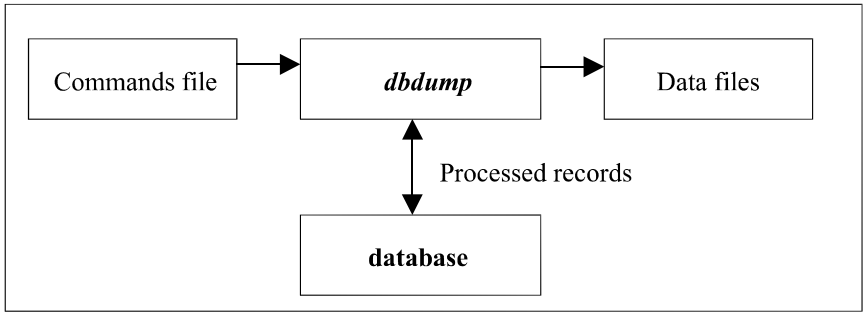
Overview

The `dbdump` utility exports records from a database to a file. On the command line, you must specify a command file. The command file specifies:

- The name of an export file.
- `DEFINE RECORD` statements that specify the format of the exported records.
- The names of the database tables and columns that contain the records.

You can use `dbdump` to export variable or fixed length records. Data files can use multiple-character record delimiters.

Execution Flow



For example, to export the contents of a database named customers per the specifications in a command file named `orders.cmd`:

```
dbdump -f orders.cmd customers
```

Prerequisites

Before running `dbdump`, you need:

- A valid, readable commands file
- `SELECT` privileges on the tables named in the commands file.
- Access privileges to the database named on the command line.

Data File Formats

Data files must be in one of the following record formats.

Variable Length Records-

For variable length records, the fields in the data file can be of varying length.
Unless the keyword `FIXED` is used in the commands file, `dbdump` assumes that you want variable length records.

Fixed Length Records-

For fixed length records, the fields in the data file must be of fixed length.
You can use fixed length records for either ASCII or binary data. You can use variable length records only for ASCII data.

Commands File

The commands file supplies instructions for `dbdump`.
There is no file naming convention for the commands file. For example, the commands file name could be `orders.cmd`.
The commands file must contain:

- A `DEFINE RECORD` statement to define the format and contents of the output file
- A `FOR RECORD` statement to specify the name of the data file, the source table and column names, and a query which is to be used to find data to export.

For example:

```
DEFINE RECORD ord_rec
   AS ( ord_no, item_name, date, item_qty )
   FIELD DELIMITER ' ';
FOR RECORD ord_rec
   DUMP INTO "ord.dat"
   USING SELECT order_no, product, order_date, qty
   FROM items;
```

The `DEFINE RECORD` Statement

The `DEFINE RECORD` statement defines the records to be placed in the output file.
The general syntax of the `DEFINE RECORD` statement is:

```
DEFINE RECORD record_name
{  OF FIXED LENGTH record_length
   AS ( field_name
        POSITION ( start_position : end_position )
        { CHAR | SHORT | LONG | FLOAT | DOUBLE }
        [ ,... ]
      )
|  AS ( field_name [ , ... ]
        [ FIELD DELIMITER 'delimiter_char' ]
        [ RECORD DELIMITER 'delimiter_string' ];
}
```

Record clause

```
DEFINE RECORD record_name
```

`DEFINE RECORD`

The keywords that begin the statement.

`record_name`

The name of the record to dump.
The record name can be any arbitrary name, as long as it is used consistently in the `FOR RECORD` statement.

Field clause for fixed length fields

```
OF FIXED LENGTH record_length
AS ( field_name
     POSITION ( start_position : end_position )
     { CHAR | SHORT | LONG | FLOAT | DOUBLE }
     [ ,... ]
    )
```

Elements are:

`OF FIXED LENGTH`

The keywords that specify that you are using fixed length records.

`record_length`

The number of characters, including any delimiters, in each record.
The length of the record must be the same for all records, and field and record delimiters will be ignored. The data files that contain fixed length records can either be ASCII or binary files.

AS

The keyword for the clause that describes the one or many fields that are contained in each record.
Each field is described with a name, a start and end position, and a data type.

`field_name`

An arbitrary field name.
Field names are used in the `VALUES` clause of the `FOR RECORD` statement.

`POSITION`

The keyword that starts the clause that defines the placement within a record of each field.

```
( start_position : end_position )
```

For each record, the start position and end positions of a field.
The first position of each record is 1 and not 0.
The start and end parameters must be enclosed in parentheses, be separated with a colon, and should be unsigned integers.

```
CHAR | SHORT | LONG | FLOAT | DOUBLE
```

The data type.

Valid types are `CHAR, SHORT, LONG, FLOAT,` and `DOUBLE.`
If date and time types are to be dumped, they can be specified as characters in the data file and the type specification can be `CHAR`.
The following is an example of a `DEFINE RECORD` statement for fixed length records:

```
DEFINE RECORD rec_one
OF FIXED LENGTH 20
AS ( fld1  POSITION (1:4)   SHORT,
     fld2  POSITION (5:15)  CHAR,
     fld3  POSITION (16:20) CHAR  );
```

Field clause for variable length fields

```
AS ( field_name [ , ... ]
     [ FIELD DELIMITER 'delimiter_char' ]
     [ RECORD DELIMITER 'delimiter_string' ];
```

Elements are:

AS

The keyword for the clause that describes the one or many fields that is contained in each record.

`field_name`

An arbitrary field name.
Field names are used in the `VALUES` clause of the `FOR EACH RECORD` statement.

FIELD DELIMITER

Optional specification of a field delimiter character.
Specify the field delimiter as a literal. The default is a comma.

RECORD DELIMITER

Optional specification of a record delimiter string.
Specify the record delimiter as a literal. The default is the new line string, /n.

The FOR RECORD Statement

The FOR RECORD statement writes each valid record into the data file after selecting the record from the database.
The general syntax for the FOR RECORD statement is:

```
FOR RECORD record_name
   DUMP INTO data_file_name
   USING select_statement;
```

Elements are:

FOR RECORD record_name

Specify the name of the record to be dumped in record_name.
The record name must be the same as the name used in the DEFINE RECORD statement.

DUMP INTO data_file_name

Specify the data file name (the file to receive the records) in data_file_name.
If the specified file already exists, its existing contents will be overwritten with the new records.
USING select_statement
Specify any valid SELECT statement in select_statement.

For a reference to SELECT statements, please refer “SELECT” on page 219 in "Chapter 6 - Versant ReVind Statements".

Examples

The following commands file writes records from the dept table. The output data file name is deptrecs.out which is an ASCII file in the variable length record format.

```
DEFINE RECORD dept_rec
   AS ( dept_no, dept_name, location )
```

```
   FIELD DELIMITER ' ';
FOR RECORD dept_rec
   DUMP INTO "deptrecs.out"
   USING SELECT no, name, loc
   FROM vsqldba.dept;
```

The following commands file writes records from the `customer` table. The output data file is `cust.out`. It is a binary file of fixed length record format.

```
DEFINE RECORD cust_rec OF FIXED LENGTH 37
AS ( cust_no POSITION (1:4) LONG,
     cust_name POSITION (5:15) CHAR,
     cust_street POSITION (16:28) CHAR,
     cust_city POSITION (29:34) CHAR,
     cust_state POSITION (35:36) CHAR
   );
FOR  RECORD cust_rec
     DUMP INTO "cust.out"
     USING SELECT no, name, city, street, state
     FROM vsqldba.customer;
```

The following commands file dumps records from the `orders` table. The output data file is `orders.out` which is a binary file in fixed length record format.

```
DEFINE RECORD orders_rec
OF FIXED LENGTH 31
AS ( order_no   POSITION (1:4)   LONG,
     order_date POSITION (6:16)  CHAR,
     product    POSITION (18:25) CHAR,
     qty        POSITION (27:30) LONG );
FOR RECORD orders_rec
   DUMP INTO "orders.out"
   USING SELECT no, date, prod, units
   FROM vsqldba.orders;
```

dbload

```
dbload -f commands_file [options] database_name
```

Load records from a data file to a database.

Command elements

`commands_file`

The name of a command file.

`database_name`

The name of the database to load records into.

For information on how to specify a database name, please refer “Database Name Syntax” on page 55 in "Chapter 2 - Initiating Versant Revind".

The command file contains:

- The name of a data file.
- `DEFINE RECORD` statements that specify the format of the records in the data file.
- The names of the database tables and columns for the records.

Command options

`-l logfile`

A log file. The default is `stderr`.
Errors and statistics are written to the log file. The following statistics are written:

- Number of records read
- Number of records skipped
- Number of records loaded
- Number of records rejected

`-b badfile`

A name of the file to write the not loaded row into.
By default, bad records are written to the file `badfile` in the current directory.

`-c commit_frequency`

The number of records written between transaction commits. The default frequency is 100 records.

`-e maxerrs`

The maximum number of tolerable errors. The default is `50` errors.
The execution process will stop if the number of tolerable errors is exceeded.

`-s skipcount`

The number of records to skip in the first data file. The default is zero rows.
If multiple files are specified, rows are skipped only in the first file.

`-m maxrows`

The maximum number of rows to write.
After `maxrows`, dbload will stop writing rows to the database.

`-n`

Cause `dbload` to parse the commands file and display errors, if any, without doing the database load.
If the parsing is successful a message, `"No errors in the commands file"` is seen on `stdout`.

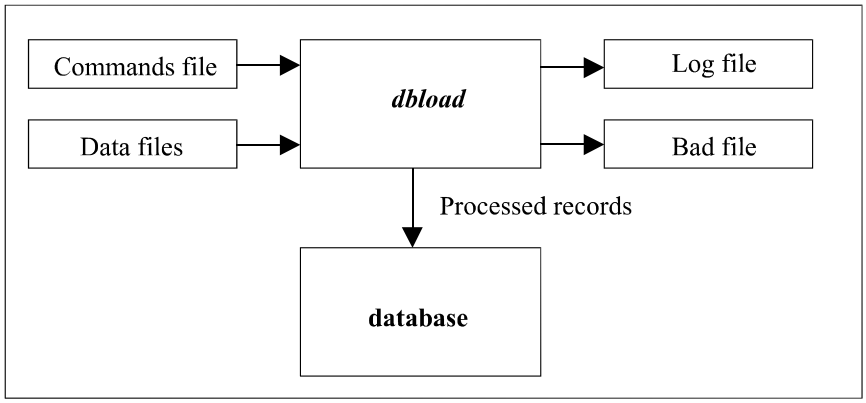
Overview

The `dbload` utility loads records from an input data file into tables of a database.
On the command line, you must specify a command file. The command file specifies:

- The name, location, and format of the data file.
- The format of the data records (specified with `DEFINE RECORD` statements.)
- The destination tables and columns.

You can use `dbload` to load variable or fixed length records. Data files can use multiple-character record delimiters.
You can also control other characteristics, such as error handling and logging, with command line parameters.

Execution Flow



For example, to load a database named `customers` using the specifications in a command file named `orders.cmd`:

```
dbload -f orders.cmd customers
```

Prerequisites

Before running `dbload`, you need:

- A valid, readable commands file
- `INSERT` privileges on the tables named in the commands file.
- Access privileges to the database named on the command line.

Data File Formats

Data files must be in one of the following record formats:

Variable Length Records-

For variable length records, the fields in the data file can be of varying length.
Unless the keyword `FIXED` is used in the commands file, `dbload` assumes that the records are variable length records.

Fixed Length Records-

For fixed length records, the fields in the data file must be of fixed length.

You can use fixed length records for either ASCII or binary data. You can use variable length records only for ASCII data.

Commands File

The commands file supplies instructions for `dbload`.
There is no file naming convention for the commands file. For example, the commands file name to load the orders table could be `orders.cmd`.
The commands file must contain:

- A `DEFINE RECORD` statement to define the format and contents of the data file
- A `FOR EACH` statement to specify the name of the data file and the target table and column names.

For example:

```
DEFINE RECORD ord_rec
   AS ( ord_no, item_name, date, item_qty )
   FIELD DELIMITER ' ';
FOR EACH RECORD ord_rec
   FROM "ord.in"
   INSERT INTO vsqldba."orders" (order_no, product, order_date, qty)
   VALUES (ord_no, item_name, date, item_qty);
NEXT RECORD
```

The above commands specification instructs `dbload` to load records into the `orders` table. The fields in the data file, `ord.in`, appear in the order listed in the `DEFINE RECORD` statement.

The `DEFINE RECORD` Statement

The `DEFINE RECORD` statement describes the data found in the data file.
The general syntax of the `DEFINE RECORD` statement is:

```
DEFINE RECORD record_name
{   OF FIXED LENGTH record_length
    AS ( field_name
         POSITION ( start_position : end_position )
         { CHAR | SHORT | LONG | FLOAT | DOUBLE }
         [ ,.... ]
```

```
        )
 |  AS ( field_name [, ...] )
         [ FIELD DELIMITER  'delimiter_char' ]
         [ RECORD DELIMITER 'delimiter_string' ];
}
```

Record clause

```
DEFINE RECORD record_name
DEFINE RECORD
```

The keywords that begin the statement.

`record_name`

The name of the record to be loaded.
The record name can be any arbitrary name, as long as it is used consistently in the `FOR EACH RECORD` statement.

Field clause for fixed length fields

```
OF FIXED LENGTH record_length
AS ( field_name
     POSITION ( start_position : end_position )
     { CHAR | SHORT | LONG | FLOAT | DOUBLE }
     [ ,.... ]
    )
```

Elements are:

`OF FIXED LENGTH`

The keywords that specify that you are using fixed length records.

`record_length`

The number of characters, including any delimiters, in each record.
The length of the record must be the same for all records, and field and record delimiters will be ignored. The data files that contain fixed length records can either be ASCII or binary files.

AS

The keyword for the clause that describes the one or many fields that are contained in each record.
Each field is described with a name, a start and end position, and a data type.

`field_name`

An arbitrary field name.
Field names are used in the `VALUES` clause of the `FOR EACH RECORD` statement.

`POSITION`

The keyword that starts the clause that defines the placement within a record of each field.

`( start_position : end_position )`

For each record, the start position and end positions of a field.
The first position of each record is 1 and not 0.
The start and end parameters must be enclosed in parentheses, be separated with a colon, and be unsigned integers.

`CHAR | SHORT | LONG | FLOAT | DOUBLE`

The data type.
Valid types are `CHAR, SHORT, LONG, FLOAT,` and `DOUBLE`.
If date and time types are to be inserted they can be specified as characters in the data file and the type specification can be `CHAR`.

The following is an example of a `DEFINE RECORD` statement for fixed length records:

```
DEFINE RECORD rec_one
OF FIXED LENGTH 20
AS ( fld1  POSITION (1:4)   SHORT,
     fld2  POSITION (5:15)  CHAR,
     fld3  POSITION (16:20) CHAR  );
```

Field clause for variable length fields

```
AS ( field_name [, ...] )
     [ FIELD DELIMITER  'delimiter_char' ]
     [ RECORD DELIMITER 'delimiter_string' ];
```

Elements are:

AS

The keyword for the clause that describes the one or many fields that are contained in each record.

`field_name`

An arbitrary field name.
Field names are used in the `VALUES` clause of the `FOR EACH RECORD` statement.

`FIELD DELIMITER`

Optional specification of a field delimiter character.
Specify the field delimiter as a literal. The default is a comma.

`RECORD DELIMITER`

Optional specification of a record delimiter string.
Specify the record delimiter as a literal. The default is the new line string, `/n`.

The `FOR EACH` Statement

The `FOR EACH` statement scans for each valid record in the data file and inserts the record into the database.
The general syntax of the `FOR EACH` statement is:

```
FOR EACH RECORD record_name
     FROM data_file_name, ...
     INSERT INTO table_name [ (field_name, ...) ]
     VALUES (value, ...);
NEXT RECORD
```

Elements are:

`FOR EACH RECORD ... NEXT RECORD`

The keywords indicating that the flow of control will loop until all records found in the data file have been processed.

`record_name`

The name of the record to be loaded.
The record name must be the same as the name used in the DEFINE RECORD statement.

FROM data_file_name, ...

Specify the data file name (the file containing the records) in data_file_name.

INSERT INTO *table_name*

Specify the name of the table to receive the data in table_name.

For information on how to specify a table name, please refer “Table Names” on page 52 in "Chapter 2 - Initiating Versant Revind".

The table must already exist, you must have access privileges to the database containing the table, and you must have INSERT privileges on the table.

(field_name, ...)

Optional specification of one or many names of columns in the target table.
This field_name parameter is not the same as the field_name parameter in the DEFINE RECORD statement. In DEFINE RECORD, field names are arbitrary names that refer to values parsed from the data file. In the FOR EACH RECORD statement, field names must be column names.
If field names are not specified, dbload assumes that values are supplied for columns in the same order as they were created in the target table.

VALUES *(value, ...)*

Values to be inserted.
To specify a value, you can use a field name defined for the data file in the DEFINE RECORD statement, a numeric or character constant, or NULL.
The specified values must either correspond one to one with the list of column names specified in the field_name parameter of the FOR EACH RECORD statement or else correspond to the table columns in the same order as they were created.
A values list does not have to use field names in the same order as they were specified in the DEFINE RECORD statement.
The following example shows the values list interchanged with respect to the list in the DEFINE RECORD statement. Note that the following does specify field names in the FOR EACH RECORD statement and that the field names are actual column names.

DEFINE RECORD dept_rec

```
   AS ( dept_no, dept_name, location )
   FIELD DELIMITER ' ';
FOR EACH RECORD dept_rec
   FROM "dept.in"
   INSERT INTO vsqldba.department (loc, no, name)
   VALUES (location, dept_no, dept_name);
NEXT RECORD
```

Examples

The following commands file will load records into a `dept` table. The input data file name is `deptrecs.in`, which is an ASCII file in variable length record format.

```
DEFINE RECORD dept_rec
   AS ( dept_no, dept_name, location )
   FIELD DELIMITER ' ';
FOR EACH RECORD dept_rec
   FROM "deptrecs.in"
   INSERT INTO vsqldba.dept (no, name, loc)
   VALUES (dept_no, dept_name, location);
NEXT RECORD
```

The following commands file will load records into a `customer` table. The input data file is `cust.in`, which is a binary file in fixed length record format.

```
DEFINE RECORD cust_rec
   OF FIXED LENGTH 36
   AS ( cust_no POSITION (1:4) LONG,
        cust_name POSITION (5:15) CHAR,
        cust_street POSITION (16:28) CHAR,
        cust_city POSITION (29:34) CHAR,
        cust_state POSITION (35:36) CHAR  );
FOR EACH RECORD cust_rec
   FROM "cust.in"
   INSERT INTO vsqldba.customer (no, name, city, street, state)
   VALUES (cust_no, cust_name, cust_city, cust_street, 'CA');
NEXT RECORD
```

The following commands file will load records into an `orders` table. The input data file is `orders.in`, which is a binary file in fixed length record format.

```
DEFINE RECORD orders_rec
   OF FIXED LENGTH 30
   AS ( order_no POSITION (1:4) LONG,
        order_date POSITION (6:16) CHAR,
        product POSITION (18:25) CHAR,
        qty POSITION (27:30) LONG  );
FOR EACH RECORD orders_rec
   FROM "orders.in"
   INSERT INTO systpe.orders (no, date, prod, units)
   VALUES (order_no, order_date, product, qty);
NEXT RECORD
```

Errors

Invalid records-

Invalid records that are encountered during the processing of records from the data files are flagged as bad records and are written to the bad file.

Data file errors-

Any error in the input data file is messaged in the log file (if specified in the command line option) along with execution statistics.

Command file errors-

Following are possible error messages related to the commands file.

`Record name redefined.`

The record name in the `DEFINE RECORD` statement is multiply defined. The record name must be unique.

`Error in record definition.`

An unspecified error was found in the `DEFINE RECORD` statement.

`Too many fields in record definition.`

The number of fields used in the record definition is more than the maximum allowed.

`Position not specified for fixed length record.`

The start and end positions were not specified.

`Position for SHORT not specified correctly.`

The size of the field (start position to end position) must be equal to the size of `SHORT`.

`Position for LONG not specified correctly.`

The size of the field (start position to end position) must be equal to the size of `LONG`.

`Position for FLOAT not specified correctly`

The size of the field (start position to end position) must be equal to the size of `FLOAT`.

`Position for DOUBLE not specified correctly.`

The size of the field (start position to end position) must be equal to the size of `DOUBLE`.

```
Field delimiter must be a single character.
Invalid record delimiter.
Record not defined.
```

The `FOR EACH` statement is used with a record name that is not defined.

`Mismatch in value list.`

The number of values specified in the `VALUES` list does not match with that specified in the `DEFINE RECORD` list.

`Too many data files specified.`

Currently, the maximum number of data files that can be specified in a `FOR EACH` statement is 10.

`Column not found in record definition.`

Fatal Errors-

The following are non-recoverable errors:

`No memory.`

```
Table not found.
No columns in the table.
Column not found.
Too many fields.
```

More than the maximum number of fields allowed, is specified in the table list of the `FOR EACH` statement.

```
Cannot open <bad file name>.
Cannot open <data file name>.
Cannot open log file <log file name>.
```

schload

```
schload [options] databaseName
```

Initialize a database for use with Versant ReVind.

For information on how to specify a database name, please refer “Database Name Syntax” on page 55 in "Chapter 2 - Initiating Versant Revind".

The `schload` utility will load Versant ReVind Mapper schema and authorization tables into the specified database and grant query privileges to all valid users of the database.

This utility is normally used during the installation process to initialize one or more databases. If you add a database after installation of Versant ReVind and want to make single connections to it, then you must use `schload` to initialize it.

Options:

`-p`
Password corresponding to the user name entered in `-u`.
For normal usage, users can just pass in the `-u` option and `schload` will prompt them for the password. Running with just the `-p` option without the `-u` option will result in an error.

`-u`
User name to access the DB.
This user has to be a valid user of the DB. You'll be prompted for a password if no password is entered using the `-p` option.

If you will be making multiple connections, then only the first database in your multiple connection needs to be loaded. The `schload` utility accepts the same format for multiple-database specifications as do the `misql` and `isql` commands.

For example, if you will be connecting with:

```
misql db1+db2
```

then enter

```
schload db1+db2
```

You can also load `db1` with:

```
schload db1
```

If you will be connecting, in separate sessions, with the commands:

```
misql db1
```

and later with

```
misql db2
```

then load `db1` and `db2` in separate `schload` operations.

See also `sqlutil`**.**

sqlutil

```
sqlutil -option dbname
```

Grant access privileges to the users of a database.

For information on how to specify a database name, please refer “Database Name Syntax” on page 55 in "Chapter 2 - Initiating Versant Revind".

Following are the alternatives for the `option` parameter:

```
-G
```

Grant query privileges to all valid users of the database.
The `-G` option can be used only by the owner of the database (the Versant dba.)
Using this option is the same as using the SQL `GRANT` command to grant `SELECT` on `ALL` to `PUBLIC`.
For example, the following are equivalent ways of doing the same thing:

```
sqlutil -G db1
```

is the same as:

```
isql db1
ISQL> grant select on all to public;
ISQL> commit work;
-N username
-A password
```

The options `–N` and `–A` should be used only with `-S` option.

`-S`

Allows the current user to refer to tables with a synonym rather than using the fully qualified name in double quotes.
Any valid user of the database to create persistent synonyms for tables can run this option. These synonyms can then be used by the user who invoked `sqlutil -S`.
This option is the same as creating individual synonyms for each table, except that you do not have to individually specify each table.
-
For example, the following command:

```
sqlutil -S db1
```

is the same as creating synonyms for each individual table with a series of commands such as:

```
isql db1
ISQL> create synonym author for "Author";
ISQL> create synonym book for "Book";
ISQL> create synonym chapter for "Chapter";
....
ISQL> commit work;
```

For example, before running `sqlutil` with `-S`, if you do not own table `Author`, you must refer to it with the following syntax:

```
ISQL> select * from vsqldba."Author";
```

After running `sqlutil` with `-S`, you can use the following syntax:

```
ISQL> select * from author;
```

`-P`

Allows user to create public synonyms for all tables. All existing as well as new users can use these synonyms. This option should be used alongwith `–S` option.
To create public synonym for specific table, `CREATE PUBLIC SYNONYM` command should be fired explicitly from within MISQL/ISQL.

For more information, please refer to Chapter 6 “Versant ReVind Statements” on page 185.

NOTE:- After a schema is added in Versant database and before synonyms are created, it is essential to stop `DHSERVER` daemon if running.

The `sqlutil` utility is located in the Versant ReVind bin directory. The Versant ReVind administrator can verify that the utility was successfully executed by querying the extended table: `systabauth`.

CHAPTER 5

Versant ReVind Common Language Elements

This Chapter describes the language elements that are supported by Versant ReVind.

The Chapter describes the following in detail:

- SQL Language Elements
- SQL Reserved Words
- Identifiers
- Data Types
- Query Expressions
- Search Conditions
- Expressions
- Literals
- Date-Time Format strings

SQL LANGUAGE ELEMENTS

Following is a brief description of language elements that are common to many SQL statements. Versant ReVind supports a subset of the SQL language vocabulary.

Identifiers are user-supplied names for elements such as tables, views, cursors, and columns. SQL statements use those names to refer to the elements.

Data types control how SQL stores column values.

Query expressions retrieve values from tables. Query expressions form the basis of other SQL statements and syntax elements.

Search conditions specify a condition that is true or false about a given row or group of rows. Query expressions and `UPDATE` statements specify search conditions to restrict the number of rows in the result table.

Expressions are a symbol or string of symbols used to represent or calculate a single value in a SQL statement. When SQL encounters an expression, it retrieves or calculates the value represented by the expression and uses that value when it executes the statement.

Literals are a type of SQL expression that specify a constant value. Some SQL constructs allow literals but prohibit other forms of expressions.

Date-time format strings control the output of date and time values. Versant ReVind interprets format strings and replaces them with formatted values.

Functions are a type of SQL expression that generate a derivative value. Each of the available functions is listed alphabetical in a subsequent chapter.

SQL RESERVED WORDS

Reserved words are SQL keywords you can use as identifiers if you delimit them with double quotation marks. If you use SQL keywords without delimiting them, the statement generates one of the following errors:

```
error(-20003): Syntax error
error(-20049): Keyword used for a name
```

The following table lists all the reserved words in Versant ReVind. The list is alphabetic and reads left to right.

A	**ABS**	**ACOS**	**ADD**
ADD_MONTHS	ALL	ALTER	AN
AND	ANY	ARRAY	AS
ASC	ASCII	ASIN	ATAN
ATAN2	AVG	BEGIN	BETWEEN
BIGINT	BINARY	BIND	BINDING
BIT	BY	CALL	CASCADE
CASE	CAST	CEILING	CHAR
CHARACTER	CHARTOROWID	CHECK	CHR
CLEANUP	CLOSE	COALESCE	COALESCE
COLGROUP	COMMIT	COMPLEX	COMPRESS
CONCAT	CONNECT	CONSTRAINT	CONTINUE
CONVERT	COS	COUNT	CREATE
CURRENT	CURSOR	CVAR	DATABASE
DATABASE	DATABASE	DATAPAGES	DATE
DAYOFMONTH	DAYOFWEEK	DAYOFYEAR	DB_NAME
DBA	DEC	DECIMAL	DECLARATION
DECLARE	DECODE	DEFAULT	DEFINITION
DEGREES	DELETE	DESC	DESCRIBE
DESCRIPTOR	DHTYPE	DISTINCT	DOUBLE
DOUBLE	DROP	ELSE	END

END	ESCAPE	EXCLUSIVE	EXEC
EXECUTE	EXISTS	EXIT	EXP
EXPLICIT	FETCH	FIELD	FILE
FLOAT	FLOOR	FOR	FOREIGN
FOUND	FROM	GO	GOTO
GRANT	GREATEST	GROUP	HASH
HAVING	HOUR	IDENTIFIED	IFNULL
IMMEDIATE	IN	INDEX	INDEXPAGES
INDICATOR	INITCAP	INPUT	INSERT
INSTR	INT	INTEGER	INTERFACE
INTERSECT	INTO	IS	JOIN
KEY	LAST_DAY	LEAST	LEFT
LENGTH	LIKE	LINK	LIST
LOCATE	LOCK	LOG	LOG
LOG10	LONG	LONG	LOWER
LPAD	LTRIM	LVARBINARY	LVARCHAR
MAIN	MAX	MIN	MINUS
MINUTE	MOD	MODE	MODIFY
MONEY	MONTH	MONTHS_BETWEEN	NEXT_DAY
NOCOMPRESS	NOT	NOWAIT	NULL
NULLIF	NULLIF	NULLVALUE	NUMBER
NUMERIC	NVL	OBJECT_ID	ODBC_CONVERT
ODBCINFO	OF	ON	OPEN
OPTION	OR	ORDER	OUTER
OUTPUT	PCTFREE	PI	POWER
PRECISION	PREFIX	PREPARE	PRIMARY
PRIVILEGES	PUBLIC	QUARTER	RADIANS
RAND	RAW	REAL	RECORD
REFERENCES	RENAME	RESOURCE	RESTRICT
RETURN	REVOKE	ROLLBACK	ROWID

ROWIDTO-CHAR	ROWNUM	RPAD	RTRIM
SEARCHED_CASE	SECOND	SECTION	SELECT
SERVICE	SET	SHARE	SHORT
SIGN	SIMPLE_CASE	SIN	SIZE
SMALLINT	SOME	SOUNDEX	SPACE
SQL	SQL_BIGINT	SQL_BINARY	SQL_BIT
SQL_CHAR	SQL_DATE	SQL_DECIMAL	SQL_DOUBLE
SQL_FLOAT	SQL_INTEGER	SQL_LONGVARBINARY	SQL_LONGVARCHAR
SQL_REAL	SQL_SMALLINT	SQL_TIME	SQL_TIMESTAMP
SQL_TINYINT	SQL_VARBINARY	SQL_VARCHAR	SQLERROR
SQLWARNING	SQRT	START	STATISTICS
STOP	STORAGE_ATTRIBUTES	STORAGE_MANAGER	STORE_IN_DHARMA
SUBSTR	SUFFIX	SUM	SUSER_NAME
SYNONYM	SYSDATE	SYSDATE	SYSTIME
SYSTIME	SYSTIMESTAMP	SYSTIMESTAMP	TABLE
TAN	THEN	TIME	TIMEOUT
TIMESTAMP	TINYINT	TO	TO_CHAR
TO_DATE	TO_NUMBER	TO_TIME	TO_TIMESTAMP
TPE	TRANSACTION	TRANSLATE	UID
UID	UNION	UNIQUE	UNSIGNED
UPDATE	UPPER	USER	USER_ID
USER_NAME	USING	USING	UUID
VALUES	VARBINARY	VARCHAR	VARIABLES
VERSION	VIEW	WEEK	WHEN
WHENEVER	WHERE	WITH	WORK
YEAR			

IDENTIFIERS

SQL syntax requires users to supply names for elements such as tables, views, cursors, and columns when they define them. SQL statements must use those names to refer to the table, view, or other element. In syntax diagrams, SQL identifiers are shown in lowercase type.

The maximum length for SQL identifiers is 32 characters.

There are two types of SQL identifiers:

- Conventional identifiers.
- Delimited identifiers enclosed in double quotation marks.

Conventional Identifiers

Unless they are delimited identifiers (see "Delimited Identifiers"), SQL identifiers must:

- Begin with an upper case or lower case letter.
- Contain only letters, digits, or the underscore character (_).
- Not be reserved words.

Except for delimited identifiers, SQL does not distinguish between uppercase and lower case letters in SQL identifiers. It converts all names to lower case, but statements can refer to the names in mixed case.

Delimited Identifiers

Delimited identifiers are SQL identifiers enclosed in double quotation marks ("). Enclosing a name in double quotation marks preserves the case of the name and allows it to be a reserved word and special characters. (Special characters are any characters other than letters, digits, or the underscore character.) Subsequent references to a delimited identifier must also use enclosing double quotation marks. To include a double-quotation mark character in a delimited identifier, precede it with another double-quotation mark.

DATA TYPES

Data Types Overview

This section describes the data types SQL supports for table columns. For completeness, data types that are not supported by Versant ReVind, such as MONEY, are nevertheless described in this chapter. The table following identifies all of the data types that may actually be found in a Versant ReVind database, as indicated by "Yes" entries.

SQL Type	Support Status
CHARACTER	Yes
VARCHAR	Yes
CHARACTER VARYING	
CHAR VARYING	
LVARCHARLONG VAR-CHAR	No
TINYINT	Yes
SMALLINT	Yes
INTEGER	Yes
BIGINT	No
NUMERIC	No
DECIMAL	No
MONEY	No
REAL	Yes
DOUBLE PRECISION	No
FLOAT	Yes
DATE	Yes
TIME	Yes
TIMESTAMP	Yes
BIT	Yes

BINARY	No
VARBINARY	No
LVARBINARY	No
LONG BINARY	No

For the schema that is present when Versant ReVind starts up, you will have access to all the supported operations and functions for the recorded data type of each column of each table. You can query for the SQL data types of each of the columns in every tables through the `syscolumns` table as described elsewhere.

You can read and write the supported data types within the constraints of the data type, as described here. Constraints are likely to be either a maximum length or an inability to copy the value of one column to another due to its having a different — albeit related — data type. There are several broad categories of SQL data types:

- Character
- Exact numeric
- Approximate numeric
- Date-time
- Bit String

All the data types can store null values. A null value indicates that the value is not known and is distinct from all non-null values.

Each data type has two corresponding representations:

- In the Versant ReVind engine, for internal manipulation and for passing to and from the underlying storage manager
- In the host language that contains embedded SQL statements

The two representations are the same for most of the data types. Where they are not, Versant ReVind automatically converts from one form to the other, and application programs do not have to handle differences.

Character Data Types

Following are character data types:

See “Character String Literals” on page 176 for details on specifying values to be stored in character columns.

Syntax

```
char_data_type ::
   { CHARACTER | CHAR } [(length)] [ CHARACTER SET charset-name ]
|   { CHARACTER VARYING | CHAR VARYING | VARCHAR } [(length)]
      [ CHARACTER SET charset-name ]
|   LVARCHAR | LONG VARCHAR
```

Syntax Elements

`{ CHARACTER | CHAR } [(length)] [ CHARACTER SET charset-name ]`

Type `CHARACTER` (abbreviated as `CHAR`) corresponds to a null terminated character string with the maximum length specified. The default length is `1`. The maximum length is `2000`.

The optional `CHARACTER SET` clause specifies an alternative character set supported by the underlying storage system. Following section "Specifying the Character Set for Character Data Types" describes general considerations for using this clause. See the documentation for the underlying storage system for details on valid values for `charset-name`, if any.

`{ CHARACTER VARYING | CHAR VARYING | VARCHAR } [(length)]`

`[ CHARACTER SET charset-name ]`

Type `CHARACTER VARYING` corresponds to a variable-length character string with the maximum length specified. The default length is `1`. The maximum length is `2000`.

The optional `CHARACTER SET` clause specifies an alternative character set supported by the underlying storage system. Following section "Specifying the Character Set for Character Data Types" describes general considerations for using this clause. See the documentation for the underlying storage system for details on valid values for `charset-name`, if any.

`LVARCHAR | LONG VARCHAR`

Type `LONG VARCHAR` corresponds to an arbitrarily-long character string with a maximum length limited by the specific storage system.

The arbitrary size and unstructured nature of LONG data types restrict where they can be used.

- LONG columns are allowed in select lists of query expressions and in INSERT statements.
- INSERT statements can store data from columns of any type into a LONG VARCHAR column, but LONG VARCHAR data cannot be stored in any other type.
- CONTAINS predicates are the only predicates that allow LONG columns (and then only if the underlying storage system explicitly supports CONTAINS predicates).
- Conditional expressions, arithmetic expressions, and functions cannot specify LONG columns.
- UPDATE statements cannot specify LONG columns.

Specifying the Character Set for Character Data Types

SQL allows column definitions of type CHARACTER and CHARACTER VARYING to specify an alternate character set. If you omit the CHARACTER SET clause in a column definition, the default character set is the standard 7-bit ASCII character set.

The character set associated with a table column defines which set of characters can be stored in that column, how those characters are represented in the underlying storage system, and how character strings using the character set compare with each other:

- The set of characters allowed in a character set is called the "repertoire" of the character set. The default ASCII character set has a repertoire of 128 characters, shown in the table below. Other character sets, such as Unicode, specify much larger repertoires and include characters for many languages other than English.
- The storage representation for a character set is called the "form of use" of the character set. The form of use for the default ASCII character set is a single byte (or "octet") containing a number designating a particular ASCII character. Other character sets, such as Unicode, use two or more bytes (or a varying number of bytes, depending on the character) for each character.
- The rules used to control how character strings compare with each other is called the "collation" of a character set. Each character set specifies a collating sequence that defines relative values of each character for comparing, merging and sorting character strings. Storage systems may also define additional collations that override the default for a character set. SQL statements specify such collations with the COLLATE clause in character column definitions, basic predicates, the GROUP BY clause of query expressions, and the ORDER BY clause of SELECT statements.

The following table shows the characters in the default ASCII character set and the decimal values that designate each character. (This is the default representation on UNIX; other operating systems may have slight differences in their definitions of the default ASCII character set.) The values also define the collating sequence for the character set. For instance, this collating sequence specifies that a lower case letter is always a larger value than an upper case letter.

Val	Char	Val	Char	Val	Char	Val	Char
0	NUL	17	DC1	34	"	60	<
1	SOH	18	DC2	35	#	61	=
2	STX	19	DC3	36	$	62	>
3	ETX	20	DC4	37	%	63	?
4	EOT	21	NAK	38	&	64	@
5	ENQ	22	SYN	39	'	65-90	A-Z
6	ACK	23	ETB	40	(	91	[
7	BEL	24	CAN	41	)	92	\
8	BS	25	EM	42	*	93	]
9	HT	26	SUB	43	+	94	^
10	NL	27	ESC	44	,	95	_
11	VT	28	FS	45	-	96	`
12	NP	29	GS	46	.	97-122	a-z
13	CR	30	RS	47	/	123	{
14	SO	31	US	48-57	0-9	124	|
15	SI	32	SP	58	:	125	}
16	DLE	33	!	59	;	126	~
127	DEL						

Support for any character sets and collations other than the default ASCII character set is completely dependent on the underlying storage system. When statements refer to a character set or collation name that is not supported by the underlying storage system, SQL generates an error.

Exact Numeric Data Types

Following are exact numeric data types:

See “Numeric Literals” on page 175 for details on specifying values to be stored in numeric columns.

Syntax

```
exact_numeric_data_type ::
   TINYINT
|   SMALLINT
|   INTEGER
|   BIGINT
|   NUMERIC | NUMBER [ ( precision [ , scale ] ) ]
|   DECIMAL [(precision, scale)]
|   MONEY [(precision)]
```

Syntax Elements

`TINYINT`

Type `TINYINT` corresponds to an integer value stored in one byte.

The range of `TINYINT` is `-128` to `127`.

`SMALLINT`

Type `SMALLINT` corresponds to an integer value of length 2 bytes.

The range of `SMALLINT` is `-32768` to `+32767`.

`INTEGER`

Type `INTEGER` corresponds to an integer of length 4 bytes.

The range of values for `INTEGER` columns is `-2 ** 31` to `2 ** 31 -1`.

`BIGINT`

Type `BIGINT` corresponds to an integer of length 8 bytes.

The range of values for `BIGINT` columns is `-2 ** 63` to `2 ** 63 -1`.

`NUMERIC | NUMBER [ ( precision [ , scale ] ) ]`

Type NUMERIC corresponds to a number with the given precision (maximum number of digits) and scale (the number of digits to the right of the decimal point). By default, NUMERIC columns have a precision of 32 and scale of 0. If NUMERIC columns omit the scale, the default scale is 0.

The range of values for a NUMERIC type column is -n to +n where n is the largest number that can be represented with the specified precision and scale. If a value exceeds the precision of a NUMERIC column, SQL generates an overflow error. If a value exceeds the scale of a NUMERIC column, SQL rounds the value.

NUMERIC type columns cannot specify a negative scale or specify a scale larger than the precision.

The following example shows what values will fit in a column created with a precision of 3 and scale of 2:

```
insert into t4 values(33.33);
error(-20052): Overflow error
insert into t4 values(33.9);
error(-20052): Overflow error
insert into t4 values(3.3);
1 record inserted.
insert into t4 values(33);
error(-20052): Overflow error
insert into t4 values(3.33);
1 record inserted.
insert into t4 values(3.33333);
1 record inserted.
insert into t4 values(3.3555);
1 record inserted.
select * from t4;
    C1
    --
  3.30
  3.33
  3.33
  3.36
4 records selected
```

```
DECIMAL [(precision, scale)]
```

Type DECIMAL is equivalent to type NUMERIC.

```
MONEY [(precision)]
```

Type MONEY is equivalent to type NUMERIC with a fixed scale of 2.

Approximate Numeric Data Types

Syntax

```
approx_numeric_data_type ::
   REAL
|   DOUBLE PRECISION
|   FLOAT [ (precision) ]
```

Syntax Elements

REAL

Type REAL corresponds to a single precision floating point number equivalent to the C language float type.

DOUBLE PRECISION

Type DOUBLE PRECISION corresponds to a double precision floating point number equivalent to the C language double type.

FLOAT [(precision)]

Type FLOAT corresponds to a double precision floating point number of the given precision. By default, FLOAT columns have a precision of 8.

See “Numeric Literals” on page 175 for details on specifying values to be stored in numeric columns.

Date-Time Data Types

Syntax

```
date_time_data_type ::
   DATE
|   TIME
|   TIMESTAMP
```

```
|   INTERVAL
```

Syntax Elements

`DATE`

Type `DATE` stores a date value as three parts: year, month, and day. The range for the parts is:

- Year: `1` to `9999`.
- Month: `1` to `12`.
- Day: Lower limit is `1`; the upper limit depends on the month and the year.

`TIME`

Type `TIME` stores a time value as four parts: hours, minutes, seconds, and milliseconds. The range for the parts is:

- Hours: `0` to `23`.
- Minutes: `0` to `59`.
- Seconds: `0` to `59`.
- Milliseconds: `0` to `999`.

`TIMESTAMP`

Type `TIMESTAMP` combines the parts of `DATE` and `TIME`.

`INTERVAL`

TBS

See “Date-Time Literals” on page 176 for details on specifying values to be stored in date-time columns and “Date-Time Format Strings” on page 181 for details on using format strings to specify the output format of date-time columns.

Bit String Data Types

Syntax

```
bit_string_data_type ::
   BIT
|   BINARY [(length)]
```

```
|   VARBINARY (length)]
|   LVARBINARY | LONG VARBINARY
```

Syntax Elements

`BIT`

Type `BIT` corresponds to a single bit value of `0` or `1`.

SQL statements can assign and compare values in `BIT` columns to and from columns of types `CHAR, VARCHAR, BINARY, VARBINARY, TINYINT, SMALLINT,` and `INTEGER.` However, in assignments from `BINARY, VARBINARY,` and `LONG VARBINARY,` the value of the first four bits must be `0001` or `0000`.

No arithmetic operations are allowed on `BIT` columns.

`BINARY [(length)]`

Type `BINARY` corresponds to a bit field of the specified length of bytes. The default length is 1 byte. The maximum length is `2000 bytes`.

In interactive SQL, `INSERT` statements must use a special format to store values in `BINARY` columns. They can specify the binary values as a bit string, hexadecimal string, or character string. `INSERT` statements must enclose binary values in single-quote marks, preceded by `b` for a bit string and `x` for a hexadecimal string:

	Prefix	Suffix	Example (for same 2 byte data)
bit string	b'	'	b'1010110100010000'
hex string	x'	'	x'ad10'
char string	'	'	'ad10'

SQL interprets a character string as the character representation of a hexadecimal string.

If the data inserted into a `BINARY` column is less than the length specified, SQL pads it with zeroes.

`BINARY` data can be assigned and compared to and from columns of type `BIT`, `CHAR`, and `VARBINARY` types. No arithmetic operations are allowed.

`VARBINARY [(length)]`

Type VARBINARY corresponds to a variable-length bit field with the maximum length specified. The default length is 1. Otherwise, VARBINARY columns have the same characteristics as BINARY.

LVARBINARY | LONG VARBINARY

Type LONG VARBINARY corresponds to an arbitrarily-long bit field with the maximum length defined by the underlying storage system.

The arbitrary size and unstructured nature of LONG data types restrict where they can be used.

- LONG columns are allowed in select lists of query expressions and in INSERT statements.
- INSERT statements can store data from columns of any type into a LONG VARCHAR column, but LONG VARCHAR data cannot be stored in any other type.
- CONTAINS predicates are the only predicates that allow LONG columns (and then only if the underlying storage system explicitly supports CONTAINS predicates).
- Conditional expressions, arithmetic expressions, and functions cannot specify LONG columns.
- UPDATE statements cannot specify LONG columns.

QUERY EXPRESSIONS

Description

A query expression selects the specified column values from one or more rows contained in one or more tables specified in the FROM clause. The selection of rows is restricted by a search condition in the WHERE clause. The temporary table derived through the clauses of a select statement is called a result table.

Query expressions form the basis of other SQL statements and syntax elements:

- SELECT statements are query expressions with optional ORDER BY and FOR UPDATE clauses.
- CREATE VIEW statements specify their result table as a query expression.
- INSERT statements can specify a query expression to add the rows of the result table to a table.
- UPDATE statements can specify a query expression that returns a single row to modify columns of a row.
- Some search conditions can specify query expressions. Basic predicates can specify query expressions, but the result table can contain only a single value. Quantified and IN predicates can specify query expressions, but the result table can contain only a single column.
- The FROM clause of a query expression can itself specify a query expression, called a derived table.

General Syntax

```
query_expression ::
    query_specification
|   query_expression set_operator query_expression
|   ( query_expression )
set_operator ::
     {  UNION [ ALL ]  |  INTERSECT  |  MINUS }
query_specification ::
SELECT [ALL | DISTINCT]
       {
           *
    | { table_name | alias } . *[, { table_name | alias } . *]...
```

```
    | expr [ [ AS ] [ ' ] column_title [ ' ] ]
[, expr [ [ AS ] [ ' ] column_title [ ' ] ] ] ...
        }
FROM table_ref [ { versant ORDERED } ]
[ , table_ref [ { versant ORDERED } ] …
[ WHERE search_condition ]
[ GROUP BY [table.]column_name [ COLLATE collation-name ]
           [, [table.]column_name [ COLLATE collation-name ] ] ...
[ HAVING search_condition ]
table_ref ::
          table_name [ AS ] [ alias [ ( column_alias [ , … ] ) ] ]
                 |  ( query_expression ) [ AS ] alias
[ ( column_alias [ , … ] ) ] |
[ ( ] joined_table [ ) ]
joined_table ::
           table_ref CROSS JOIN table_ref |  table_ref [ INNER |
 LEFT [ OUTER ] ] JOIN table_ref ON search_condition
```

SELECT Syntax

```
SELECT [ ALL | DISTINCT ]
```

`DISTINCT` specifies that the result table omits duplicate rows. `ALL` is the default, and specifies that the result table includes all rows.

```
SELECT * | { table_name | alias } . *
```

Specifies that the result table includes all columns from all tables named in the `FROM` clause. For instance, the following examples both specify all the columns in the `customers` table:

```
SELECT * FROM customers;
SELECT customers.* FROM customers;
```

The `tablename.*` syntax is useful when the select list refers to columns in multiple tables, and you want to specify all the columns in one of those tables:

```
SELECT CUSTOMERS.CUSTOMER_ID, CUSTOMERS.CUSTOMER_NAME, ORDERS.*
    FROM CUSTOMERS, ORDERS …
SELECT expr [ [ AS ] [ ' ] column_title [ ' ] ]
```

Specifies a list of expressions, called a select list, whose results will form columns of the result table. Typically, the expression is a column name from a table named in the FROM clause. The expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.

The optional column_title argument specifies a new heading for the associated column in the result table. Enclose the new title in single or double quotation marks if it contains spaces or other special characters:

```
SELECT order_value, order_value * .2 AS 'order "markup"' FROM orders;
ORDER_VALUE                ORDER "MARKUP"
-----------                --------------
5000000.00                   1000000.00
 110000.00                     22000.00
3300000.00                    660000.00
```

You can qualify column names with the name of the table they belong to:

```
SELECT CUSTOMER.CUSTOMER_ID FROM CUSTOMERS
```

You must qualify a column name if it occurs in more than one table specified in the FROM clause:

```
SELECT CUSTOMERS.CUSTOMER_ID
    FROM CUSTOMERS, ORDERS
```

Qualified column names are always allowed even when they are not required.

FROM Syntax

```
FROM table_ref …
```

The FROM clause specifies one or more table references. Each table reference resolves to one table (either a table stored in the database or a virtual table resulting from processing the table reference) whose rows the query expression uses to create the result table.

There are three forms of table references:

- A direct reference to a table, view or synonym
- A "derived table" specified by a query expression in the FROM clause
- A joined table that combines rows and columns from multiple tables

The usage notes specific to each form of table reference follow:

If there are multiple table references, SQL joins the tables to form an intermediate result table that is used as the basis for evaluating all other clauses in the query expression. That intermediate result table is the Cartesian product of rows in the tables in the FROM clause, formed by concatenating every row of every table with all other rows in all tables.

```
FROM table_name [ AS ] [ alias [ ( column_alias [ , … ] ) ] ]
```

Explicitly names a table. The name listed in the FROM clause can be a table name, a view name, or a synonym.

The alias is a name you use to qualify column names in other parts of the query expression. Aliases are also called "correlation names".

If you specify an alias, you must use it, and not the table name, to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

For example, the following query expression joins the table customer with itself. It uses the aliases x and y and returns information on customers in the same city as customer SMITH:

```
SELECT y.cust_no, y.name
     FROM customer x, customer y
     WHERE  x.name = 'SMITH'
        AND y.city = x.city ;
```

Similar to table aliases, the column_alias provides an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in table_name. Also, if you specify column aliases in the FROM clause, you must use them — not the column names — in references to the columns.

```
FROM ( query_expression ) [ AS ] alias [ ( column_alias [ , … ] ) ]
```

Specifies a derived table through a query expression. With derived tables, you must specify an alias to identify the derived table.

Derived tables can also specify column aliases. Column aliases provide an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in the result table of the query expression. Also, if you specify column aliases in the FROM clause, you must use them, and not the column names, in references to the columns.

```
FROM [ ( ] joined_table [ ) ]
```

Combines data from two table references by specifying a join condition. The syntax currently allowed in the FROM clause supports only a subset of possible join conditions:

- CROSS JOIN specifies a Cartesian product of rows in the two tables.
- INNER JOIN specifies an inner join using the supplied search condition.
- LEFT OUTER JOIN specifies a left outer join using the supplied search condition.

You can also specify these and other join conditions in the WHERE clause of a query expression.

See “Inner Joins” on page 155 and “Outer Joins” on page 158 for more detail on both ways of specifying joins.

Other Syntax

```
{ versant ORDERED }
```

Directs the SQL engine optimizer to join the tables in the order specified. Use this clause when you want to override the SQL engine's join-order optimization. This is useful for special cases when you know that a particular join order will result in the best performance from the underlying storage system. Since this clause bypasses join-order optimization, carefully test queries that use it; to make sure the specified join order is faster than relying on the optimizer.

NOTE:- The braces ({ and }) are part of the required syntax and not syntax conventions.

```
SELECT sc.tbl 'Table', sc.col 'Column',
  sc.coltype 'Data Type', sc.width 'Size'
FROM systpe.syscolumns sc, systpe.systables st
   { versant ORDERED }
WHERE sc.tbl = st.tbl AND st.tbltype = 'S'
ORDER BY sc.tbl, sc.col;
```

```
WHERE search_condition
```

The WHERE clause specifies a search_condition that applies conditions to restrict the number of rows in the result table. If the query expression does not specify a WHERE clause, the result table includes all the rows of the specified table reference in the FROM clause.

The search_condition is applied to each row of the result table set of the FROM clause. Only rows that satisfy the conditions become part of the result table. If the result of the search_condition is NULL for a row, the row is not selected.

Search conditions can specify different conditions for joining two or more tables.

See “Inner Joins” on page 155 and “Outer Joins” on page 158 for more details.

See “Search Conditions ” on page 161 for details on the different kinds of search conditions.

```
SELECT *
     FROM customer
     WHERE city = 'BURLINGTON' AND state = 'MA' ;
     SELECT *
     FROM  customer
     WHERE  city IN (
          SELECT city
          FROM customer
          WHERE name = 'SMITH') ;
```

```
GROUP BY column_name ...
```

Specifies grouping of rows in the result table:

- For the first column specified in the `GROUP BY` clause, SQL arranges rows of the result table into groups whose rows all have the same values for the specified column.
- If a second `GROUP BY` column is specified, SQL then groups rows in each main group by values of the second column.
- SQL groups rows for values in additional `GROUP BY` columns in a similar fashion.

All columns named in the `GROUP BY` clause must also be in the select list of the query expression. Conversely, columns in the select list must also be in the `GROUP BY` clause or be part of an aggregate function.

```
HAVING search_condition
```

The `HAVING` clause allows conditions to be set on the groups returned by the `GROUP BY` clause. If the `HAVING` clause is used without the `GROUP BY` clause, the implicit group against which the search condition is evaluated is all the rows returned by the `WHERE` clause.

A condition of the `HAVING` clause can compare one aggregate function value with another aggregate function value or a constant.

```
-- select customer number and number of orders for all
-- customers who had more than 10 orders prior to
-- March 31st, 1991.
SELECT cust_no, count(*)
    FROM  orders
```

```
    WHERE order_date < to_date ('3/31/1991')
    GROUP BY cust_no
    HAVING count (*) > 10 ;
UNION [ALL]
```

Appends the result table from one query expression to the result table from another.

The two query expressions must have the same number of columns in their result table, and those columns must have the same or compatible data types.

The final result table contains the rows from the second query expression appended to the rows from the first. By default, the result table does not contain any duplicate rows from the second query expression. Specify `UNION ALL` to include duplicate rows in the result table.

```
-- Get a merged list of customers and suppliers.
    SELECT name, street, state, zip
    FROM customer
    UNION
    SELECT name, street, state, zip
    FROM supplier ;
-- Get a list of customers and suppliers
-- with duplicate entries for those customers who are
-- also suppliers.
    SELECT name, street, state, zip
    FROM customer
    UNION ALL
    SELECT name, street, state, zip
    FROM supplier ;
```

`INTERSECT`

Limits rows in the final result table to those that exist in the result tables from both query expressions.

The two query expressions must have the same number of columns in their result table, and those columns must have the same or compatible data types.

```
-- Get a list of customers who are also suppliers.
    SELECT name, street, state, zip
    FROM customer
    INTERSECT
    SELECT name, street, state, zip
    FROM supplier ;
```

MINUS

Limits rows in the final result table to those that exist in the result table from the first query expression minus those that exist in the second. In other words, the MINUS operator returns rows that exist in the result table from the first query expression but that do not exist in the second.

The two query expressions must have the same number of columns in their result table, and those columns must have the same or compatible data types.

```
-- Get a list of suppliers who are not customers.
    SELECT name, street, state, zip
    FROM supplier ;
    MINUS
    SELECT name, street, state, zip
    FROM customer;
```

Authorization

The user executing a query expression must have any of the following privileges:

- DBA privilege.
- SELECT permission on all the tables/views referred to in the query_expression.

SQL Compliance	SQL-92. Extensions: { versant ORDERED }clause, MINUS set operator
Environment	Interactive SQL, ODBC applications
Related Statements	CREATE TABLE, CREATE VIEW, INSERT, Search Conditions, SELECT, UPDATE

Inner Joins

Inner joins specify how the rows from one table reference are to be joined with the rows of another table reference. Inner joins usually specify a search condition that limits the number of rows from each table reference that become part of the result table generated by the inner join operation.

If an inner join does not specify a search condition, the result table from the join operation is the Cartesian product of rows in the tables, formed by concatenating every row of one table with every row of the other table. Cartesian products (also called cross products or cross joins) are not practically useful, but SQL logically processes all join operations by first forming the Cartesian products of rows from tables participating in the join.

If specified, the search condition is applied to the Cartesian product of rows from the two tables. Only rows that satisfy the search condition become part of the result table generated by the join.

A query expression can specify inner joins in either its FROM clause or in its WHERE clause. For each formulation in the FROM clause, there is an equivalent syntax formulation in the WHERE clause. Currently, not all syntax specified by the SQL-92 standard is allowed in the FROM clause.

Syntax

```
from_clause_inner_join ::
        |  FROM table_ref CROSS JOIN table_ref
       |  FROM table_ref [ INNER ] JOIN table_ref ON search_condition
where_clause_inner_join ::
           FROM table_ref, table_ref WHERE search_condition
```

FROM Syntax

```
FROM table_ref CROSS JOIN table_ref
```

Explicitly specifies that the join generates the Cartesian product of rows in the two table references. This syntax is equivalent to omitting the WHERE clause and a search condition. The following queries illustrate the results of a simple CROSS JOIN operation and an equivalent formulation that does not use the CROSS JOIN syntax:

```
SELECT * FROM T1; -- Contents of T1
          C1          C2
          --          --
          10          15
          20          25
2 records selected
SELECT * FROM T2;  -- Contents of T2
          C3 C4
          -- --
          10 BB
          15 DD
2 records selected
SELECT * FROM T1 CROSS JOIN T2; -- Cartesian product
          C1          C2          C3 C4
          --          --          -- --
          10          15          10 BB
```

```
          10           15           15 DD
          20           25           10 BB
          20           25           15 DD
4 records selected
SELECT * FROM T1, T2; -- Different formulation, same results
          C1           C2           C3 C4
          --           --           -- --
          10           15           10 BB
          10           15           15 DD
          20           25           10 BB
          20           25           15 DD
4 records selected
FROM table_ref [ INNER ] JOIN table_ref ON search_condition
FROM table_ref, table_ref WHERE search_condition
```

These two equivalent syntax constructions both specify `search_condition` for restricting rows that will be in the result table generated by the join. In the first format, `INNER` is optional and has no effect. There is no difference between the `WHERE` form of inner joins and the `JOIN ON` form.

Equi-joins Syntax

An equi-join specifies that values in one table equal some corresponding column's values in the other:

```
-- For customers with orders, get their name and order info, :
SELECT customer.cust_no, customer.name,
     orders.order_no, orders.order_date
     FROM customers INNER JOIN orders
     ON customer.cust_no = orders.cust_no ;
-- Different formulation, same results:
SELECT customer.cust_no, customer.name,
     orders.order_no, orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no ;
```

Self-joins Syntax

A self join, or auto join, joins a table with itself. If a `WHERE` clause specifies a self join, the `FROM` clause must use aliases to have two different references to the same table:

```
-- Get all the customers who are from the same city as customer SMITH:
     SELECT y.cust_no, y.name
     FROM  customer AS x INNER JOIN customer AS y
```

```
     ON x.name = 'SMITH' AND y.city = x.city ;
-- Different formulation, same results:
     SELECT y.cust_no, y.name
     FROM  customer x, customer y
     WHERE  x.name = 'SMITH' AND y.city = x.city ;
```

Outer Joins

An outer join between two tables returns more information than a corresponding inner join. An outer join returns a result table that contains all the rows from one of the tables even if there is no row in the other table that satisfies the join condition.

In a left outer join, the information from the table on the left is preserved: the result table contains all rows from the left table even if some rows do not have matching rows in the right table. Where there are no matching rows in the left table, SQL generates null values.

In a right outer join, the information from the table on the right is preserved: the result table contains all rows from the right table even if some rows do not have matching rows in the left table. Where there are no matching rows in the right table, SQL generates null values.

SQL supports two forms of syntax to support outer joins:

- In the `WHERE` clause of a query expression, specify the outer join operator (+) after the column name of the table for which rows will *not* be preserved in the result table. Both sides of an outer-join search condition in a `WHERE` clause must be simple column references. This syntax is similar to Oracle's SQL syntax, and allows both left and right outer joins.
- For left outer joins only, in the `FROM` clause, specify the `LEFT OUTER JOIN` clause between two table names, followed by a search condition. The search condition can contain only the join condition between the specified tables.

Versant's SQL implementation does not support full (two-sided) outer joins.

Syntax

```
from_clause_inner_join ::
          FROM table_ref LEFT OUTER JOIN table_ref ON search_condition
where_clause_inner_join ::
           WHERE [table_name.]column (+) = [table_name.]column
         | WHERE [table_name.]column = [table_name.]column (+)
```

Examples

The following example shows a left outer join. It displays all the customers with their orders. Even if there is not a corresponding row in the orders table for each row in the customer table, NULL values are displayed for the order.order_no and order.order_date columns.

```
SELECT customer.cust_no, customer.name, orders.order_no,
            orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no (+) ;
```

The following series of examples illustrates the outer join syntax:

```
SELECT * FROM T1; -- Contents of T1
C1   C2
--   --
10   15
20   25
2 records selected
SELECT * FROM T2; -- Contents of T2
C3   C4
--   --
10   BB
15   DD
2 records selected
-- Left outer join
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C3;
C1   C2   C3   C4
--   --   --   --
10   15   10   BB
20   25
2 records selected
 -- Left outer join: different formulation, same results
 SELECT * FROM T1, T2 WHERE T1.C1 = T2.C3 (+);
C1   C2   C3   C4
--   --   --   --
10   15   10   BB
20   25
2 records selected
 -- Right outer join
 SELECT * FROM T1, T2 WHERE T1.C1 (+) =  T2.C3;
C1   C2   C3   C4
--   --   --   --
10   15   10   BB
```

```
           15   DD
2 records selected
```

SEARCH CONDITIONS

Description

A search condition specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement.

Search conditions contain one or more predicates. The predicates that can be part of a search condition are described in the following subsections:

Syntax

```
search_condition ::
        [NOT] predicate
        [ { AND | OR } { predicate | ( search_condition ) } ]
predicate ::
         basic_predicate
|        quantified_predicate
|        between_predicate
|        null_predicate
|        like_predicate
|        contains_predicate
|        exists_predicate
|        in_predicate
|        outer_join_predicate
```

Logical Operators: OR, AND, NOT

Logical operators combine multiple search conditions. SQL evaluates multiple search conditions in this order:

1. Search conditions enclosed in parentheses. If there are nested search conditions in parentheses, SQL evaluates the innermost search condition first.
2. Search conditions following NOT.
3. Search conditions combined by AND.
4. Search conditions combined by OR.

Examples

```
SELECT *
    FROM customer
    WHERE name = 'LEVIEN' OR name = 'SMITH' ;
SELECT *
    FROM customer
    WHERE city = 'PRINCETON' AND state = 'NJ' ;
SELECT *
    FROM customer
    WHERE  NOT (name = 'LEVIEN' OR name = 'SMITH') ;
```

Relational Operators

Relational operators specify how SQL compares expressions in basic and quantified predicates.

Syntax

```
relop ::
     =
   | <> | != | ^=
   | <
   | <=
   | >
   | >=
```

Relational Operator	Predicate is:
=	True if the two expressions are equal.
<> | != | ^=	True if the two expressions are not equal. The operators != and ^= are equivalent to <>.
<	True if the first expression is less than the second expression.
<=	True if the first expression is less than or equal to the second expression.

>	True if the first expression is greater than the second expression.
>=	True if the first expression is greater than or equal to the second expression.

See "Basic Predicate" and "Quantified Predicate" for more information.

Basic Predicate

A basic predicate compares two values using a relational operator (see "Relational Operators"). If a basic predicate specifies a query expression, then the query expression must return a single value. Basic predicates often specify an inner join. See "Inner Joins" for more detail.

If the value of any expression is null or the query_expression does not return any value, then the result of the predicate is set to false.

Basic predicates that compare two character expressions can include an optional `COLLATE` clause. The `COLLATE` clause specifies a collation sequence supported by the underlying storage system.

See "Specifying the Character Set for Character Data Types" on page 140 for notes on character sets and collations.

(See the documentation for your underlying storage system for details on any supported collations.)

Syntax

```
basic_predicate ::
    expr relop { expr | (query_expression) } [ COLLATE collation_name ]
```

Quantified Predicate

The quantified predicate compares a value with a collection of values using a relational operator. A quantified predicate has the same form as a basic predicate with the `query_expression` being preceded by `ALL, ANY` or `SOME` keyword. The result table returned by `query_expression` can contain only a single column.

When ALL is specified the predicate evaluates to true if the query_expression returns no values or the specified relationship is true for all the values returned.

When SOME or ANY is specified the predicate evaluates to true if the specified relationship is true for at least one value returned by the query_expression. There is no difference between the SOME and ANY keywords. The predicate evaluates to false if the query_expression returns no values or the specified relationship is false for all the values returned.

See "Relational Operators" on page 162.

Syntax

```
quantified_predicate ::
    expr relop { ALL | ANY | SOME } (query_expression)
```

Example

```
10 <  ANY ( SELECT COUNT(*)
                FROM order_tbl
                GROUP BY custid
              )
```

BETWEEN Predicate

The BETWEEN predicate can be used to determine if a value is within a specified value range or not. The first expression specifies the lower bound of the range and the second expression specifies the upper bound of the range.

The predicate evaluates to true if the value is greater than or equal to the lower bound of the range, or less than or equal to the upper bound of the range.

Syntax

```
between_predicate ::
    expr [ NOT ] BETWEEN expr AND expr
```

Example

```
salary BETWEEN 2000.00 AND 10000.00
```

NULL Predicate

The NULL predicate can be used for testing null values of database table columns.

Syntax

```
null_predicate ::
    column_name IS [ NOT ] NULL
```

Example

```
contact_name IS NOT NULL
```

CONTAINS Predicate

The SQL CONTAINS predicate is an extension to the SQL standard that allows storage systems to provide search capabilities on character and binary data. See the documentation for the underlying storage system for details on support, if any, for the CONTAINS predicate.

Syntax

```
column_name [ NOT ] CONTAINS 'string'
```

NOTES:-

- column_name must be one of the following data types: CHARACTER, VARCHAR, LONG VARCHAR, BINARY, VARBINARY, or LONG VARBINARY.
- There must be an index defined for column_name, and the CREATE INDEX statement for the column must include the TYPE clause, and specify an index type that indicates to the underlying storage system that this index supports CONTAINS predicates. See the documentation for your storage system for the correct TYPE clause for indexes that support CONTAINS predicates.
- The format of the quoted string argument and the semantics of the CONTAINS predicate are defined by the underlying storage system.

LIKE Predicate

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified after the LIKE keyword in a string constant. The pattern can be specified by a string in which the underscore (_) and percent sign (%) characters have special semantics.

The ESCAPE clause can be used to disable the special semantics given to characters ' _ ' and ' % '. The escape character specified must precede the special characters in order to disable their special semantics.

Syntax

```
like_predicate ::
    column_name [ NOT ] LIKE string_constant
    [ ESCAPE escape-character ]
```

NOTES:-

- The column name specified in the LIKE predicate must refer to a character string column.
- A percent sign in the pattern matches zero or more characters of the column string.
- An underscore character in the pattern matches any single character of the column string.

Examples

```
cust_name LIKE '%Computer%'
cust_name LIKE '___'
item_name LIKE '%\_%' ESCAPE '\'
```

In the first example, for all strings with the substring `Computer`, the predicate will evaluate to true. In the second example, for all strings which are exactly three characters long, the predicate will evaluate to true. In the third example, the backslash character `' \ '` has been specified as the escape character, which means that the special interpretation given to the character `'_ '` is disabled. The pattern will evaluate to `TRUE` if the column `item_name` has embedded underscore characters.

EXISTS Predicate

The `EXISTS` predicate can be used to check for the existence of specific rows. The `query_expression` returns rows rather than values. The predicate evaluates to true if the number of rows returned by the `query_expression` is non-zero.

Syntax

```
exists_predicate ::
    EXISTS ( query_expression )
```

Example

```
EXISTS (SELECT * FROM order_tbl
               WHERE order_tbl.custid = :custid)
```

In this example, the predicate will evaluate to true if the specified customer has any orders.

IN Predicate

The `IN` predicate can be used to compare a value with a set of values. If an `IN` predicate specifies a query expression, then the result table it returns can contain only a single column.

Syntax

```
in_predicate ::
    expr [ NOT ] IN { ( query_expression ) |
                            ( constant , constant [ , ... ] ) }
```

Example

```
address.state IN ('MA', 'NH')
```

Outer Join Predicate

An outer join predicate specifies two tables and returns a result table that contains all of the rows from one of the tables, even if there is no matching row in the other table. See "Outer Joins" for more information.

Syntax

```
outer_join_predicate ::
    [ table_name. ] column = [ table_name. ] column (+)
    |  [table_name. ] column (+) = [ table_name. ] column
```

EXPRESSIONS

Expressions in General

An expression is a symbol or string of symbols used to represent or calculate a single value in a SQL statement. When you specify an expression in a statement, SQL retrieves or calculates the value represented by the expression and uses that value when it executes the statement.

Expressions are also called scalar expressions or value expressions.

Syntax

```
expr ::
   [ { table_name | alias } . ] column-name
|   character-literal
|   numeric-literal
|   date-time-literal
|   aggregate-function
|   scalar-function
| concatenated-char-expr
|   numeric-arith-expr
|   date-arith-expr
|   conditional-expr
|   ( expr )
```

Table-Column Syntax

```
[ { table_name | alias } . ] column-name
```

A column in a table.

You can qualify column names with the name of the table they belong to:

```
SELECT CUSTOMER.CUSTOMER_ID FROM CUSTOMERS
```

You must qualify a column name if it occurs in more than one table specified in the `FROM` clause:

```
SELECT CUSTOMER.CUSTOMER_ID
    FROM CUSTOMERS, ORDERS
```

Qualified column names are always allowed even when they are not required.

You can also qualify column names with an "alias." Aliases are also called "correlation names."

The `FROM` clause of a query expression can specify an optional alias after the table name. If you specify an alias, you must use it — not the table name — to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

See “Query Expressions” on page 148 for details on query expressions.

The following example shows a query expression that joins the table `customer` with itself. It uses the aliases *x* and *y* and returns information on customers in the same city as customer *SMITH*:

```
SELECT y.cust_no, y.name
     FROM  customer x, customer y
     WHERE  x.name = 'SMITH'
        AND y.city = x.city ;
```

Syntax Elements

`character-literal | numeric-literal | date-time-literal`

Literals that specify a constant value. See "Literals" for details on specifying literals.

`aggregate-function | scalar function`

An SQL function. See "Functions."

`concatenated-char-expr`

An expression that concatenates multiple character expressions into a single character string.

See “Concatenated Character Expressions” on page 170.

`numeric-arith-expr`

An expression that computes a value from numeric values.

See “Numeric Arithmetic Expressions” on page 171.

`date-arith-expr`

An expression that computes a value from date-time values.

See “Date Arithmetic Expressions” on page 172.

`conditional-expr`

An expression that evaluates a search condition or expression and returns one of multiple possible results depending on that evaluation.

See “Conditional Expressions” on page 173.

`( expr )`

An expression enclosed in parentheses. SQL evaluates expressions in parentheses first.

Concatenated Character Expressions

Description

The `||` concatenation operator (two vertical bars) concatenates the two character expressions it separates.

The concatenation operator is similar to the `CONCAT` scalar function. However, the concatenation operator allows easy concatenation of more than two character expressions, while the `CONCAT` scalar function requires nesting.

Syntax

```
concatenated-char-expr ::
   { character-literal | character-expr }
|| { character-literal | character-expr }
[  { character-literal | character-expr }
|| { character-literal | character-expr } ]
[ … ]
```

Syntax Elements

`character-literal`

A character literal.

See “Character String Literals” on page 176 for details on specifying character literals.

`character-expr`

Any expression that evaluates to a character string, including column names and scalar functions that return a character string.

See “Data Types” on page 137 for details of character data types.

Examples

```
ISQL> SELECT 'Today''s date is ' || TO_CHAR(SYSDATE) FROM SYSCALC
TABLE;
TODAY'S DATE IS 08/17/1998
--------------------------
Today's date is 08/17/1998
1 record selected
```

Numeric Arithmetic Expressions

Numeric arithmetic expressions compute a value using addition, subtraction, multiplication, and division operations on numeric literals and expressions that evaluate to any numeric data type.

Syntax

```
numeric_arith_expr ::
[ + | - ]
{ numeric_literal | numeric_expr }
[ { + | - | * | / } numeric_arith_expr ]
```

Syntax Elements

```
 [ + | - ]
```

Unary plus or minus operator

`numeric_literal`

A numeric literal.

See “Numeric Literals” on page 175 for details on specifying numeric literals.

`numeric_expr`

Any expression that evaluates to a numeric data type, including:

- Column names.
- Subqueries that return a single value.
- Aggregate functions.
- CAST or CONVERT operations to numeric data types.
- Other scalar functions that return a numeric data type.

See “Data Types” on page 137 for details of numeric data types.

```
{ + | - | * | \ }
```

Addition, subtraction, multiplication, or subtraction operator. SQL evaluates numeric arithmetic expressions in the following order:

- Unary plus or minus.
- Expressions in parentheses.
- Multiplication and division, from left to right.
- Addition and subtraction, from left to right.

Date Arithmetic Expressions

Date arithmetic expressions compute the difference between date-time expressions in terms of days or milliseconds. SQL supports these forms of date arithmetic:

- Addition and subtraction of integers to and from date-time expressions.
- Subtraction of a date-time expression from another.

Syntax

```
date_arith_expr ::
     date_time_expr { + | - } int_expr
|    date_time_expr - date_time_expr
```

Syntax Elements

```
date_time_expr
```

An expression that returns a value of type DATE or TIME or TIMESTAMP. A single date-time expression cannot mix data types. All elements of the expression must be the same data type.

Date-time expressions can contain date-time literals, but they must be converted to DATE or TIME using the CAST, CONVERT, or TO_DATE functions (see the following examples, and "CAST" and "CONVERT function (extension)").

int_expr

An expression that returns an integer value. SQL interprets the integer differently depending on the data type of the date-time expression:

- For DATE expressions, integers represent days.
- For TIME expressions, integers represent milliseconds.
- For TIMESTAMP expressions, integers represent milliseconds.

Examples

The following example manipulates DATE values using date arithmetic. SQL interprets integers as days and returns date differences in units of days:

```
SELECT C1, C2, C1-C2 FROM DTEST
c1   c2         c1-c2
1956-05-07   1952-09-29   1316
select sysdate,
   sysdate - 3 ,
   sysdate - cast ('9/29/52' as date)
from dtest;
sysdate   sysdate-3   sysdate-convert(date,9/29/52)
1995-03-24   1995-03-21   15516
```

The following example manipulates TIME values using date arithmetic. SQL interprets integers as milliseconds and returns time differences in milliseconds:

```
select systime,
       systime - 3000,
       systime - cast ('15:28:01' as time)
from dtest;
systime   systime-3000   systime-convert(time,15:28:01)
15:28:09   15:28:06      8000
```

Conditional Expressions

Conditional expressions are a subset of scalar functions that generate different results depending on the value of their arguments. They provide some of the flexibility of traditional programming

constructs to allow expressions to return alternate results depending on the value of their arguments.

The following scalar functions provide support for conditional expressions.

See for detail discussion of each function in “Scalar Functions” on page 234 in "Chapter 7 - Versant ReVind Function Reference".

CASE

CASE is the most general conditional expression. It specifies a series of search conditions and associated expressions. SQL returns the value specified by the first expression whose associated search condition evaluates as true. If none of the expressions evaluate as true, the CASE expression returns a null value (or the value of some other default expression if the CASE expression includes the ELSE clause).

All the other conditional expressions can also be expressed as CASE expressions.

DECODE

DECODE provides a subset of the functionality of CASE that is compatible with Oracle SQL syntax. DECODE is not SQL-92 compatible.

NULLIF

NULLIF substitutes a null value for an expression if it is equal to a second expression.

COALESCE

COALESCE specifies a series of expressions. SQL returns the first expression whose value is not null. If all the expressions evaluate as null, COALESCE returns a null value.

IFNULL

IFNULL substitutes a specified value if an expression evaluates as null. If the expression is not null, IFNULL returns the value of the expression.

LITERALS

Overview

Literals are a type of expression that specify a constant value (they are also called constants). You can specify literals wherever SQL syntax allows expressions. Some SQL constructs allow literals but prohibit other forms of expressions.

There are three types of literals:

- Numeric
- Character string
- Date-time

The following sections discuss each type of literal:

Numeric Literals

A numeric literal is a string of digits that SQL interprets as a decimal number. SQL allows the string to be in a variety of formats, including scientific notation.

Syntax

```
[+|-]{[0-9][0-9]...}[.[0-9][0-9]...][[E|e][+|-][0-9]{[0-9]}]
```

Examples

The following are all valid numeric strings:

```
123
123.456
-123.456
12.34E-04
```

Character String Literals

A character string literal is a string of characters enclosed in single quotation marks ('). To include a single quotation mark in a character-string literal, precede it with an additional single quotation mark.

The following SQL examples show embedding quotation marks in character-string literals:

```
insert into quote values('unquoted literal');
insert into quote values('''single-quoted literal''');
insert into quote values('"double-quoted literal"');
insert into quote values('O''Hare');
select * from quote;
c1
unquoted literal
'single-quoted literal'
"double-quoted literal"
O'Hare
```

Date-Time Literals

SQL supports special formats for literals to be used in conjunction with date-time data types. Basic predicates and the `VALUES` clause of `INSERT` statements can specify date literals directly for comparison and insertion into tables. In other cases, you need to convert date literals to the appropriate date-time data type with the `CAST`, `CONVERT`, or `TO_DATE` scalar functions.

Enclose date-time literals in single quotation marks.

Date Literals

Date literals specify a day, month, and year, using any one of the following formats, enclosed in single quotation marks ('):

Syntax

```
date-literal ::
  {d 'yyyy-mm-dd'}
|  mm-dd-[[[y]y]y]y
|  mm/dd/[[[y]y]y]y
|  yyyy-mm-dd
```

```
|  yyyy/mm/dd
|  dd-mon-[[[y]y]y]y
|  dd/mon/[[[y]y]y]y
```

Syntax Elements

`{d 'yyyy-mm-dd'}`

A date literal enclosed in an escape clause compatible with ODBC. Precede the literal string with an open brace (`{`) and a lower case `d`. End the literal with a close brace. For example:

```
INSERT INTO DTEST VALUES ({d '1994-05-07'})
```

If you use the ODBC escape clause, you must specify the date using the format `yyyy-mm-dd`.

`dd`

The day of month as a 1- or 2-digit number (in the range `01-31`), assuming the `US` setting for the `TPE_DFLT_DATE` environment variable.

`mm`

The month value as a 1- or 2-digit number (in the range `01-12`).

`mon`

The first 3 characters of the name of the month (in the range `'JAN'` to `'DEC'`).

`[[[y]y]y]y`

The year as a 1- to 4-digit number. SQL supplies the following default values if you omit one or more digits:

Format	Default	Example	Result
`yyy`	`0`	`insert into datetest values ('5/7/956');`	05/07/0956
`yy`	`19`	`insert into datetest values ('5/7/56');`	05/07/1956
`y`	`190`	`insert into datetest values ('5/7/6');`	05/07/1906

yyyy

The year as a 4-digit number.

Examples

The following SQL examples show some of the supported formats for date literals:

```
CREATE TABLE T2 (C1 DATE, C2 TIME);
INSERT INTO T2 (C1) VALUES('5/7/56');
INSERT INTO T2 (C1) VALUES('7/MAY/1956');
INSERT INTO T2 (C1) VALUES('1956/05/07');
INSERT INTO T2 (C1) VALUES({d '1956-05-07'});
INSERT INTO T2 (C1) VALUES('29-sEP-1952');
SELECT C1 FROM T2;
c1
1956-05-07
1956-05-07
1956-05-07
1956-05-07
1952-09-29
```

Time Literals

Time literals specify an hour, minute, second, and millisecond, using the following format, enclosed in single quotation marks ('):

Syntax

```
time-literal ::
   {t 'hh:mi:ss'}
|   hh:mi:ss[:mls]
```

Syntax Elements

```
{t 'hh:mi:ss'}
```

A time literal enclosed in an escape clause compatible with ODBC. Precede the literal string with an open brace (`{`) and a lower case `t`. End the literal with a close brace. For example:

```
INSERT INTO TTEST VALUES ({t '23:22:12'})
```

If you use the ODBC escape clause, you must specify the time using the format `hh:mi:ss`.

`hh`

The hour value as a 1- or 2-digit number (in the range `00` to `23`).

`mi`

The minute value as a 1- or 2-digit number (in the range `00` to `59`).

`ss`

The seconds value as a 1- or 2-digit number (in the range `00` to `59`).

`mls`

The milliseconds value as a 1- to 3-digit number (in the range `000` to `999`).

Examples

The following SQL examples show some of the formats SQL will and will not accept for time literals:

```
INSERT INTO T2 (C2) VALUES('3');
error(-20234): Invalid time string
INSERT INTO T2 (C2) VALUES('8:30');
error(-20234): Invalid time string
INSERT INTO T2 (C2) VALUES('8:30:1');
INSERT INTO T2 (C2) VALUES('8:30:');
error(-20234): Invalid time string
INSERT INTO T2 (C2) VALUES('8:30:00');
INSERT INTO T2 (C2) VALUES('8:30:1:1');
INSERT INTO T2 (C2) VALUES({t'8:30:1:1'});
SELECT C2 FROM T2;
c2
08:30:01
08:30:00
08:30:01
08:30:01
```

Timestamp Literals

Timestamp literals specify a date and a time separated by a space, enclosed in single quotation marks ('):

Syntax

```
   {ts 'yyyy-mm-dd hh:mi:ss'}
|   ' date-literal time-literal '
```

Syntax Elements

`{ts 'yyyy-mm-dd hh:mi:ss'}`

A timestamp literal enclosed in an escape clause compatible with ODBC. Precede the literal string with an open brace (`{`) and a lowercase `ts`. End the literal with a close brace. For example:

```
INSERT INTO DTEST
VALUES ({ts '1956-05-07 10:41:37'})
```

If you use the ODBC escape clause, you must specify the timestamp using the format `yyyy-mm-dd hh:mi:ss`.

`date-literal`

A date literal.

`time-literal`

A time literal.

Example

```
SELECT * FROM DTEST WHERE C1 = {ts '1956-05-07 10:41:37'}
```

DATE-TIME FORMAT STRINGS

Overview

The TO_CHAR scalar function supports a variety of format strings to control the output of date and time values. The format strings consist of keywords that SQL interprets and replaces with formatted values.

The format strings are case sensitive. For instance, SQL replaces `'DAY'` with all upper case letters, but follows the case of `'Day'`.

Supply the format strings, enclosed in single quotation marks, as the second argument to the `TO_CHAR` function. For example:

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
---------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

For details of the `TO_CHAR` function, see "`TO_CHAR`".

Date Format Strings

A date format string can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals.

`CC`	The century as a 2-digit number.
`YYYY`	The year as a 4-digit number.
`YYY`	The last 3 digits of the year.

YY	The last 2 digits of the year.
Y	The last digit of the year.
Y,YYY	The year as a 4-digit number with a comma after the first digit.
Q	The quarter of the year as 1-digit number (with values 1, 2, 3, or 4).
MM	The month value as 2-digit number (in the range 01-12).
MONTH	The name of the month as a string of 9 characters ('JANUARY' to 'DECEMBER ').
MON	The first 3 characters of the name of the month (in the range 'JAN' to 'DEC').
WW	The week of year as a 2-digit number (in the range 01-52).
W	The week of month as a 1-digit number (in the range 1-5).
DDD	The day of year as a 3-digit number (in the range 001-365).
DD	The day of month as a 2-digit number (in the range 01-31).
D	The day of week as a 1-digit number (in the range 1-7, 1 for Sunday and 7 for Saturday).
DAY	The day of week as a 9 character string (in the range 'SUNDAY' to 'SATURDAY '.
DY	The day of week as a 3 character string (in the range 'SUN' to 'SAT').
J	The Julian day (number of days since DEC 31, 1899) as an 8 digit number.
TH	When added to a format keyword that results in a number, this format keyword ('TH') is replaced by the string 'ST', 'ND', 'RD' or 'TH' depending on the last digit of the number.

Example

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
```

```
--------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

Time Format Strings

A time format string can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals.

AM PM	The string 'AM' or 'PM' depending on whether time corresponds to forenoon or afternoon.
A.M. P.M.	The string 'A.M.' or 'P.M.' depending on whether time corresponds to forenoon or afternoon.
HH12	The hour value as a 2-digit number (in the range 00 to 11).
HH	The hour value as a 2-digit number (in the range 00 to 23).
HH24	
MI	The minute value as a 2-digit number (in the range 00 to 59).
SS	The seconds value as a 2-digit number (in the range 00 to 59).
SSSSS	The seconds from midnight as a 5-digit number (in the range 00000 to 86399).
MLS	The milliseconds value as a 3-digit number (in the range 000 to 999).

Example

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
--------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

CHAPTER 6

Versant ReVind Statements

This Chapter provides description of the Versant ReVind statements

The Chapter covers details of the following:

- Versant ReVind Statement Reference

VERSANT REVIND STATEMENT REFERENCE

COMMIT WORK

```
COMMIT [ WORK ] ;
```

Used to commit a transaction explicitly after executing one or more SQL statements. Committing a transaction makes permanent any changes made by the SQL statements.

The set of SQL statements executed prior to executing `COMMIT WORK` statement are executed as one atomic transaction that is recoverable, serializable and durable.

On a system failure and/or the execution of the `ROLLBACK WORK`, the transaction is marked for abortion and the database is rolled back to its initial state.

A commit operation makes any database modifications made by that transaction permanent. Once a commit operation is executed the database modifications cannot be rolled back.

Once a commit operation is executed the transaction modifications are guaranteed to be durable irrespective of any transient system failures.

The atomicity applies only to the database modification and not to any direct I/O performed to devices such as the terminal, printer and OS files by the application code.

A commit operation releases any locks implicitly or explicitly acquired by the transaction on the database.

Under certain circumstances, Versant ReVind marks a transaction for abort but does not actually roll it back. Without an explicit rollback, any subsequent updates will not take effect, since a `COMMIT` statement cause Versant ReVind to recognize the transaction as marked for abort, and instead implicitly rolls back the transaction. Versant ReVind marks a transaction for abort under these conditions:

- Hardware or software system failures
- Lock timeout errors

SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	`ROLLBACK WORK`

CONNECT

```
CONNECT TO connect_string
      [AS connection_name]
      [USER user_name]
      [USING password]
connect_string::
      { DEFAULT
      | db_name
      | db_type:net_protocol:host_name:db_name }
net_protocol::
      { P | T }
```

Establishes a connection to a database. Optionally, the CONNECT statement can also specify a name for the connection and a user-name/password for authentication.

Arguments to CONNECT must either be string literals enclosed in quotation marks or character-string host variables.

Arguments:

connect_string

The string that specifies the database to connect to. If the CONNECT statement specifies DEFAULT, SQL tries to connect to the environment-defined database, if any. (How you define the default database varies between operating systems. On UNIX, the value of the DB_NAME environment variable specifies the default connect string.)

The connect string can be a simple database name or a complete connect string. A complete connect string has the following components:

db_type	Type of database. The only currently-supported database type is 'versant'.
net_protocol	Network protocol. Specify P for a local connection, T for a remote network connection. P directs SQL to use the local pipe interprocess communication (IPC) protocol. T directs SQL to use the TCP/IP protocol.
host_name	Name of the system where the database resides.
db_name	Name of the database.

connection_name

The name for the connection for use in DISCONNECT and SET CONNECTION statements. If the CONNECT statement omits a connection name, the default is the name of the database. Connection names must be unique.

user_name

User name for authentication of the connection. SQL verifies the user name against a corresponding password before it connects to the database. If omitted, the default value depends on the environment. (On UNIX, the value of the DH_USER environment variable specifies the default user name. If DH_USER is not set, the value of the USER environment variable specifies the default user name.)

password

Password for authentication of the connection. SQL verifies the password against a corresponding user name before it connects to the database. If omitted, the default value depends on the environment.

(On UNIX, the value of the DH_PASSWD environment variable specifies the default password.)

Arguments to CONNECT must either be string literals enclosed in quotation marks or character-string host variables.

An application can connect to more than one database at a time, with a maximum of 10 connections. However, the application can actually gain access to only one database at a time. The database name specified in the CONNECT statement becomes the active one.

If an application executes a SQL statement before connecting to a database, SQL tries to connect to the database specified through the DB_NAME environment variable, if that environment variable is defined. If the connection is successful, the SQL statement executes on that database.

The following examples illustrate the CONNECT statement:

- The first statement shown connects to the salesdb database on the local system.
- The second statement connects to the custdb database on the local system using a network protocol.
- The third statement connects to the custdb database on the local mode, using the local pipe IPC protocol.
- The fourth statement connects to the environment-defined database by default (if any).

```
CONNECT TO "salesdb" AS "sales_conn";
CONNECT TO "versant:T:localhost:custdb" AS "cust_conn";
CONNECT TO "versant:P:localhost:custdb" AS "cust_conn";
CONNECT TO DEFAULT;
```

Authorization	None
SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	`DISCONNECT, SET CONNECTION`

CREATE INDEX

```
CREATE [ UNIQUE ] INDEX index_name
       ON table_name
       ({column_name [ASC | DESC]} [, ...])
       [ PCTFREE number]
       [ STORAGE_ATTRIBUTES 'attributes' ]
       [ TYPE 'ix_type' ];
```

Creates an index on the specified table using the specified columns of the table. An index improves the performance of SQL operations whose predicates are based on the indexed column. However, an index slows performance of `INSERT, DELETE` and `UPDATE` operations.

Arguments are:

`UNIQUE`

A `UNIQUE` index will not allow the table to contain any rows with duplicate column values for the set of columns specified for that index.

`index_name`

The name of the index has to be unique within the local database.

`table_name`

The name of the table on which the index is being built.

`column_name [, ...]`

The columns on which searches and retrievals will be ordered. These columns are called the index key. When more than one column is specified in the `CREATE INDEX` statement a concatenated index is created.

`ASC | DESC`

The index can be ordered as either ascending (ASC) or descending (DESC) on each column of the concatenated index. The default is ASC.

PCTFREE number

Specifies the desired percentage of free space for a index. The PCTFREE clause indicates to the storage system how much of the space allocated to an index should be left free to accommodate growth. However, the actual behavior of the PCTFREE clause depends entirely on the underlying storage system. The SQL engine passes the PCTFREE value to the storage system, which may ignore it or interpret it. If the CREATE statement does not include a PCTFREE clause, the default is 20. See the documentation for your storage system for details.

STORAGE_ATTRIBUTES 'attributes'

A quoted string that specifies index attributes that are specific to a particular storage system. The SQL engine passes this string to the storage system, and its effects are defined by the storage manager. See the documentation for your storage system for details.

TYPE 'ix_type'

A single-character that specifies the type of index. The valid values for the TYPE argument and their meanings are specific to the underlying storage system. See the documentation for your storage system for details.

CLUSTERED

Specifies that the index keys indicate the physical placement of records on storage media. During query optimization, SQL gives preference to indexes defined with the CLUSTERED argument over any other indexes. There can be at most one clustered index defined for a table.

Example:

```
CREATE UNIQUE INDEX custindex ON customer (cust_no) ;
```

Authorization	The user executing this statement must have any of the following privileges: • DBA privilege • Ownership of the index. • INDEX privilege on the table.
SQL Compliance	ODBC Core SQL grammar. Extensions: PCTFREE, STORAGE_ATTRIBUTES, and TYPE

Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE TABLE, DROP INDEX

CREATE SYNONYM

```
CREATE [PUBLIC] SYNONYM synonym
          FOR [owner_name.] { table_name | view_name | synonym } ;
```

Creates a synonym for the table, view or synonym specified. A synonym is an alias that SQL statements can use instead of the name specified when the table, view, or synonym was created.

`PUBLIC`

Specifies that the synonym will be public: all users can refer to the name without qualifying it. By default, the synonym is private: other users must qualify the synonym by preceding it with the user name of the user who created it.

Users must have the DBA privilege to create public synonyms.

`SYNONYM synonym`

Name for the synonym.

`FOR [owner_name.] { table_name | view_name | synonym }`

Table, view, or synonym for which SQL creates the new synonym.

Example:

```
CREATE SYNONYM customer FOR smith.customer ;
CREATE PUBLIC SYNONYM public_suppliers FOR smith.suppliers ;
```

Authorization	Users executing CREATE SYNONYM must have the DBA privilege or RESOURCE privilege. Users executing CREATE PUBLIC SYNONYM statement must have the DBA privilege.
SQL Compliance	Extension
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	DROP SYNONYM

CREATE TABLE

```
CREATE TABLE [ owner_name. ] table_name
  ( column_definition
      [ , { column_definition } ] ... )
  [ TABLE SPACE table_space_name ]
CREATE TABLE [ owner_name. ] table_name
  [ ( column_name [NULL | NOT NULL], ...) ]
  AS query_expression ;
column_definition ::
  column_name data_type
  [ DEFAULT {
        literal
      | USER
      | NULL
      | UID
      | SYSDATE
      | SYSTIME
      | SYSTIMESTAMP } ]
  [ column_constraint [ column_constraint ... ] ]
```

Creates a table definition.

A table definition consists of a list of column definitions that make up a table row.

There are two forms of the `CREATE TABLE` statement. The first form explicitly specifies column definitions. The second form, with the `AS query_expression` clause, implicitly defines the columns using the columns in the query expression.

Arguments are:

`owner_name`

Specifies the owner of the table. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

`table_name`

Names the table definition. SQL defines the table in the database named in the last `CONNECT` statement.

`column_name data_type`

Names a column and associates a data type with it. The column names specified must be different than other column names in the table definition. The `data_type` must be a supported data type.

This release does not support 'national character'.

`DEFAULT`

Specifies an explicit default value for a column. The column takes on the value if an `INSERT` statement does not include a value for the column. If a column definition omits the `DEFAULT` clause, the default value is `NULL`.

The `DEFAULT` clause accepts the following arguments:

`literal`	An integer, numeric or string constant.
`USER`	The name of the user issuing the `INSERT` or `UPDATE` statement on the table. Valid only for columns defined with character data types.
`NULL`	A null value.
`UID`	The user id of the user executing the `INSERT` or `UPDATE` statement on the table.
`SYSDATE`	The current date. Valid only for columns defined with `DATE` data types.
`SYSTIME`	The current time. Valid only for columns defined with `TIME` data types.
`SYSTIMESTAMP`	The current date and time. Valid only for columns defined with `TIMESTAMP` data types.

`AS query_expression`

Specifies a query expression to use for the data types and contents of the columns for the table. The types and lengths of the columns of the query expression result become the types and lengths of the respective columns in the table created. The rows in the resultant set of the query expression are inserted into the table after creating the table. In this form of the `CREATE TABLE` statement, column names are optional. If omitted, the names for the table columns are also derived from the query expression.

If user wants to select specific columns in a query expression, then the user also has to choose `SELFOID` column, else choose all the columns.

In the following example, the user issuing the CREATE TABLE statement must have REFERENCES privilege on the column itemno of the table john.item:

```
CREATE TABLE supplier_item (
  supp_no     INTEGER NOT NULL PRIMARY KEY,
  item_no     INTEGER NOT NULL REFERENCES john.item (itemno),
  qty         INTEGER
) ;
```

The following CREATE TABLE statement explicitly specifies a table owner, systpe:

```
CREATE TABLE systpe.account (
  account  integer,
  balance  money (12),
  info     char (84)
) ;
```

The following example shows the AS query_expression form of CREATE TABLE to create and load a table with a subset of the data in the customer table:

```
CREATE TABLE systpe.dealer (name, street, city, state)
  AS
    SELECT SELFOID,name, street, city, state
    FROM customer
    WHERE customer.state IN ('CA','NY', 'TX') ;
```

The following example includes a NOT NULL column constraint and DEFAULT clauses for column definitions:

```
CREATE TABLE emp (
  empno  integer NOT NULL,
  deptno  integer DEFAULT 10,
  join_date date DEFAULT NULL
) ;
```

Column Constraints

```
column_constraint ::
    NOT NULL [ UNIQUE KEY ]
  | REFERENCES [ owner_name. ] table_name [ ( column_name ) ]
  | CHECK ( search_condition )
```

Specifies a constraint for a column that restricts the values that the column can store so that `INSERT, UPDATE,` or `DELETE` statements that violate the constraint fail. The error will be `Constraint violation` with `SQLCODE` of `-20116`.

Column constraints are similar to table constraints but their definitions are associated with a single column.

Arguments are:

`NOT NULL`

Restricts values in the column to values that are not null.

`NOT NULL UNIQUE`

Defines the column as a unique key that cannot contain null or duplicate values.

Columns with `NOT NULL UNIQUE` constraints defined for them are also called "candidate keys."

We always refer to the `SELFOID` of the reference table. Hence it is not true that other tables can name unique keys in their `REFERENCES` clauses. If they do, SQL restricts operations on the table containing the unique key:

- A `DROP TABLE` statement that deletes the table will fail.
- Any `DELETE` and `UPDATE` statements that modify values in the column that match a foreign key's value will also fail.

The following example creates a `NOT NULL UNIQUE` constraint to define the column `ss_no` as a unique key for the table `employee`:

```
CREATE TABLE employee (
  empno        INTEGER NOT NULL PRIMARY KEY,
  ss_no        INTEGER NOT NULL UNIQUE,
  ename        CHAR (19),
  sal          NUMERIC (10, 2),
  deptno       INTEGER NOT NULL
) ;
REFERENCES table_name [ (column_name) ]
```

Defines the column as a foreign key and specifies a matching primary or unique key in another table.

The REFERENCES clause names the matching primary or unique key.

A foreign key and its matching primary or unique key specify a "referential constraint", a value stored in the foreign key must either be null or be equal to some value in the matching unique or primary key.

You can omit the column_name argument if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

The following example defines order_item.orditem_order_no as a foreign key that references the primary key orders.order_no:

```
CREATE TABLE orders (
  order_no INTEGER NOT NULL PRIMARY KEY,
  order_date DATE
) ;
CREATE TABLE order_item (
  orditem_order_no INTEGER REFERENCES orders ( order_no ),
  orditem_quantity INTEGER
) ;
```

NOTE:- The second CREATE TABLE statement in the previous example could have omitted the column name order_no in the REFERENCES clause, since it refers to the primary key of table orders.

```
CHECK (search_condition)
```

Specifies a column-level check constraint. SQL restricts the form of the search condition.

The search condition must not:

- Refer to any column other than the one with which it is defined.
- Contain aggregate functions, subqueries, or parameter references.

The following example creates a check constraint:

```
CREATE TABLE supplier (
  supp_no     INTEGER NOT NULL,
  name        CHAR (30),
  status      SMALLINT,
  city        CHAR (20) CHECK (supplier.city <> 'MOSCOW')
```

```
) ;
FOREIGN KEY ... REFERENCES
```

Defines the first column list as a foreign key and, in the REFERENCES clause, specifies a matching primary key in another table.

A foreign key and its matching primary key specify a "referential constraint", the combination of values stored in the columns that make up a foreign key must either:

- Have at least one of the column values be null.
- Be equal to some corresponding combination of values in the matching unique or primary key.

You can omit the column list in the REFERENCES clause if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

The following example defines the combination of columns `student_courses.teacher` and `student_courses.course_title` as a foreign key that references the primary key of the table courses:

NOTE:- The REFERENCES clause does not specify column names because the foreign key refers to the primary key of the `courses` table.

```
CREATE TABLE courses (
  teacher       CHAR (20) NOT NULL,
  course_title  CHAR (30) NOT NULL,
  PRIMARY KEY   (teacher, course_title)
) ;
CREATE TABLE student_courses (
  student_id    INTEGER,
  teacher       CHAR (20),
  course_title  CHAR (30),
  FOREIGN KEY   (teacher, course_title) REFERENCES courses
) ;
```

SQL evaluates the referential constraint to see if it satisfies the following search condition:

```
(student_courses.teacher IS NULL
```

OR

```
student_courses.course_title IS NULL)
```

OR

```
EXISTS (
  SELECT *
    FROM student_courses
    WHERE (student_courses.teacher = courses.teacher
    AND   student_courses.course_title =
            courses.course_title)
)
```

Any `INSERT, UPDATE` or `DELETE` statements that cause the search condition to be false violate the constraint will fail, and generate an error.

`CHECK (search_condition)`

Specifies a table-level check constraint.

The syntax for table-level and column level check constraints is identical. Table-level check constraints must be separated by commas from surrounding column definitions.

SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than columns that precede it in the table definition.
- Contain aggregate functions, subqueries, or parameter references

The following example creates a table with two column-level check constraints and one table-level check constraint:

```
CREATE TABLE supplier (
  supp_no   INTEGER NOT NULL,
  name      CHAR (30),
  status    SMALLINT
            CHECK (supplier.status BETWEEN 1 AND 100 ),
  city      CHAR (20)
            CHECK (supplier.city IN
                    ('NEW YORK', 'BOSTON', 'CHICAGO')),
            CHECK (supplier.city <> 'CHICAGO'
                   OR supplier.status = 20)
) ;
```

Authorization	The user executing this statement must have either DBA or RESOURCE privilege. If the CREATE TABLE statement specifies a foreign key that references a table owned by a different user, the user must have the REFERENCES privilege on the corresponding columns of the referenced table. The AS query_expression form of CREATE TABLE requires the user to have select privilege on all the tables and views named in the query expression.
SQL Compliance	SQL-92, ODBC Minimum SQL grammar. Extensions: TABLE SPACE, PCTFREE, STORAGE_MANAGER, and AS query_expression
Environment	Embedded SQL, Interactive SQL, ODBC applications
Related Statements	DROP TABLE, Query Expressions

CREATE VIEW

```
CREATE VIEW [ owner_name. ] view_name
   [ ( column_name, column_name,... ) ]
   AS [ ( ] query_expression [ ) ]
   [ WITH CHECK OPTION ] ;
```

Creates a view with the specified name on existing tables and/or views. The owner_name is made the owner of the created view.

The column names specified for the view are optional and provide an alias for the columns selected by the query specification. If the column names are not specified then the view will be created with the same column names as the tables and/or views it is based on.

A view is deletable if deleting rows from that view is allowed. For a view to be deletable, the view definition has to satisfy the following conditions:

- The first FROM clause contains only one table reference or one view reference.
- There are no aggregate functions, DISTINCT clause, GROUP BY or HAVING clause in the view definition.
- If the first FROM clause contains a view reference, then the view referred to is deletable.

A view is updateable if updating rows from that view is allowed. For a view to be updateable, the view has to satisfy the following conditions:

- The view is deletable (That is, it satisfies all the conditions specified above for deletability).
- All the select expressions in the first `SELECT` clause of the view definition are simple column references.
- If the first `FROM` clause contains a view reference, then the view referred to is updateable.

A view is insertable if inserting rows into that view is allowed. For a view to be insertable, the view has to satisfy the following conditions:

- The view is updateable (That is, it satisfies all the conditions specified above for updatability).
- If the first `FROM` clause contains a table reference, then all `NOT NULL` columns of the table are selected in the first `SELECT` clause of the view definition.
- If the first `FROM` clause contains a view reference, then the view referred to is insertable.

The `WITH CHECK OPTION` clause can be specified only if the view is updateable.

If `WITH CHECK OPTION` clause is specified when defining a view, then during any update or insert of a row on this view, it is checked that the updated/inserted row satisfies the view definition (That is, the row is selectable using the view).

Example:

```
CREATE VIEW ne_customers AS
     SELECT cust_no, name, street, city, state, zip
     FROM customer
     WHERE state IN ('NH', 'MA', 'NY', 'VT')
     WITH CHECK OPTION ;
CREATE VIEW order_count (cust_number, norders) AS
     SELECT cust_no, COUNT(*)
     FROM orders
     GROUP BY cust_no;
```

Authorization	The user executing this statement must have the following privileges: • DBA or RESOURCE privilege. • SELECT privilege on all the tables/views referred to in the view definition. If owner_name is specified and is different from the name of the user executing the statement, then the user must have DBA privilege.
SQL Compliance	SQL-92, ODBC Core SQL grammar
Environment	Interactive SQL, ODBC applications
Related Statements	Query Expressions, DROP VIEW

DELETE

```
DELETE FROM [owner_name.] { table_name | view_name }
          [ WHERE search_condition ];
```

Deletes zero, one or more rows from the specified table that satisfy the search condition specified in the WHERE clause. If the optional WHERE clause is not specified, then the DELETE statement deletes all rows of the specified table.

The FROM clause of a subselect statement in the search condition, if any, can refer only to the table being deleted.

If the table has primary/candidate keys, and if there exists references from other tables to the rows to be deleted, the statement is rejected.

Example:

```
DELETE FROM customer
               WHERE customer_name = 'RALPH' ;
```

Authorization	The user executing this statement must have the following privileges: • DBA privilege. • Ownership of the table. • DELETE permission on the table. If the target is a view, then the DELETE privilege is required on the target base table referred to in the view definition.
SQL Compliance	SQL-92, ODBC Extended SQL grammar
Environment	Embedded SQL, Interactive SQL, ODBC applications
Related Statements	Search Conditions

DISCONNECT

```
DISCONNECT
   { connection_name | ALL | CURRENT | DEFAULT }
```

where:

```
connection_name::
      { character_literal | host_variable }
```

Terminates the connection between an application and a database environment.

If a connection name is specified it should identify a dormant or current connection. If the connection_name specified is the current connection, the connection to the dormant DEFAULT database, if any, (this connection should have been previously achieved through CONNECT TO DEFAULT) is made the current connection; else no current connection exists.

If ALL is specified, all established connections are disconnected. After the execution of this statement, a current connection does not exist.

If CURRENT is specified, the current connection, if any, is disconnected. Here too, the connection to the dormant DEFAULT database, if any, (this connection should have been previously achieved through CONNECT TO DEFAULT) is made the current connection; else no current connection exists.

If DEFAULT is specified, the DEFAULT connection, if any, is disconnected. If this connection happens to be the current connection, no current connection exists after the execution of this statement.

Example:

```
DISCONNECT "conn_1";
DISCONNECT CURRENT;
DISCONNECT ALL;
DISCONNECT DEFAULT;
```

Authorization	None
SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	`DISCONNECT, SET CONNECTION`

DROP INDEX

```
DROP INDEX [index_owner_name.]index_name
          [ON [table_owner_name.]table_name]
```

Deletes an index on the specified table.

Arguments are:

`index_owner_name`

If `index_owner_name` is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

`table_name`

The `table_name` argument is optional. If specified, the `index_name` is verified to correspond to the table.

Example:

```
DROP INDEX custindex ON customer ;
```

Authorization	The user executing this statement must have any of the following privileges: • DBA privilege • Ownership of the index
SQL Compliance	ODBC Core SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	`CREATE INDEX`

DROP SYNONYM

```
DROP [PUBLIC] SYNONYM synonym ;
```

Drops the specified synonym.

Arguments are:

`PUBLIC`

Specifies that the synonym was created with the `PUBLIC` argument.

SQL generates the `Base table not found` error if `DROP SYNONYM` specifies `PUBLIC` and the synonym was not a public synonym. Conversely, the same error message occurs if `DROP SYNONYM` does not specify public and the synonym was created with the `PUBLIC` argument.

To drop a public synonym, you must have the DBA privilege.

`SYNONYM synonym`

Name for the synonym.

For example:

```
DROP SYNONYM customer ;
DROP PUBLIC SYNONYM public_suppliers ;
```

Authorization	Users executing DROP SYNONYM must have either the DBA privilege or be the owner of the synonym. Users executing DROP PUBLIC SYNONYM must have the DBA privilege
SQL Compliance	Extension
Environment	Interactive SQL, ODBC applications
Related Statements	CREATE SYNONYM

DROP TABLE

```
DROP TABLE  [owner_name.]table_name ;
```

Deletes the specified table.

If owner_name is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

When a table is dropped, the indexes on the table and the privileges associated with the table are dropped automatically.

Views dependent on the dropped table are not automatically dropped, but become invalid.

If the table is part of another table's referential constraint (if the table is named in another table's REFERENCES clause), the DROP TABLE statement fails. Use the ALTER TABLE statement to delete any referential constraints that refer to the table before issuing the DROP TABLE statement.

For example:

```
DROP TABLE customer ;
```

Authorization	The user executing this statement must have any of the following privileges: DBA privilege, ownership of the table.
SQL Compliance	SQL-92, ODBC Minimum SQL grammar
Environment	Embedded SQL, Interactive SQL, ODBC applications
Related Statements	CREATE TABLE

DROP VIEW

```
DROP VIEW [owner_name.]view_name ;
```

Deletes the view from the database.

If `owner_name` is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

When a view is dropped, other views which are dependent on this view are not dropped. The dependent views become invalid.

Example:

```
DROP VIEW newcustomers ;
```

Authorization	The user executing this statement must have any of the following privileges: DBA privilege, ownership of the view.
SQL Compliance	SQL-92, ODBC Core SQL grammar
Environment	Embedded SQL, Interactive SQL, ODBC applications
Related Statements	`CREATE VIEW`

GET DIAGNOSTICS

```
GET DIAGNOSTICS
   :param = header_info_item [ , :param = header_info_item ] …
GET DIAGNOSTICS EXCEPTION number
   :param = detail_info_item [ , :param = detail_info_item ] …
header_info_item ::
   { NUMBER
   | MORE
   | COMMAND_FUNCTION
   | DYNAMIC_FUNCTION
   | ROW_COUNT }
detail_info_item ::
   { CONDITION_NUMBER
   | RETURNED_SQLSTATE
   | CLASS_ORIGIN
   | SUBCLASS_ORIGIN
   | ENVIRONMENT_NAME
   | CONNECTION_NAME
   | CONSTRAINT_CATALOG
```

```
| CONSTRAINT_SCHEMA
| CONSTRAINT_NAME
| CATALOG_NAME
| SCHEMA_NAME
| TABLE_NAME
| COLUMN_NAME
| CURSOR_NAME
| MESSAGE_TEXT
| MESSAGE_LENGTH
| MESSAGE_OCTET_LENGTH }
```

Retrieves information about the execution of the previous SQL statement. `GET DIAGNOSTICS` extracts information from the SQL diagnostics area, a data structure that contains information about the execution status of the most recent SQL statement.

There are two components to the diagnostics area:

- The header contains overall information about the last SQL statement as a whole
- The detail area contains information for a particular condition (an error, warning, or success condition) associated with execution of the last SQL statement. The diagnostics area can potentially contain multiple detail areas corresponding to multiple conditions generated by the SQL statement described by the header.

NOTE:- The SQL diagnostics area currently supports only one detail area.

There are two forms of the `GET DIAGNOSTICS` statement, one that extracts header information (`GET DIAGNOSTICS`), and one that extracts detail information (`GET DIAGNOSTICS EXCEPTION number`).

The `GET DIAGNOSTICS` statement itself does not affect the contents of the diagnostics area. This means applications can issues multiple `GET DIAGNOSTICS` statements to retrieve different items of information about the same SQL statement.

Arguments are:

`:parameter`

A host-language variable to receive the information returned by the `GET DIAGNOSTICS` statement. The host-language program must declare `parameter` to be compatible with the SQL data type of the information item.

`header_info_item`

One of the following keywords, which returns associated information about the diagnostics area or the SQL statement:

`NUMBER`

The number of detail areas in the diagnostics area. Currently, `NUMBER` is always `1`. `NUMBER` is type `NUMERIC` with a scale of `0`.

`MORE`

Whether the diagnostics area contains information on all the conditions resulting from the statement. `MORE` is a one-character string with a value of `Y` (all conditions are detailed in the diagnostics area) or `N` (all conditions are not detailed).

`COMMAND_FUNCTION`

If the statement was a static SQL statement, contains the character-string code for the statement (as specified in the SQL-92 standard). If the statement was a dynamic statement, contains either the character string `'EXECUTE'` or `'EXECUTE IMMEDIATE'`.

`DYNAMIC_FUNCTION`

For dynamic SQL statements only (as indicated by `'EXECUTE'` or `'EXECUTE IMMEDIATE'` in the `COMMAND_FUNCTION` item), contains the character-string code for the statement (as specified in the SQL-92 standard).

`ROW_COUNT`

The number of rows affected by the SQL statement.

`EXCEPTION number`

Specifies that `GET DIAGNOSTICS` extracts detail information. `number` specifies which of multiple detail areas `GET DIAGNOSTICS` extracts. Currently, `number` must be the integer `1`.

`detail_info_item`

One of the following keywords, which returns associated information about the particular error condition:

`CONDITION_NUMBER`

The sequence of this detail area in the diagnostics area. Currently, `CONDITION_NUMBER` is always `1`. The sequence of this detail area in the diagnostics area. Currently, `CONDITION_NUMBER` is always `1`.

`RETURNED_SQLSTATE`

The SQLSTATE value that corresponds to the condition.

CLASS_ORIGIN

Whether the SQLSTATE class code is defined by the SQL standard (indicated by the character string 'ISO 9075') or by Versant ReVind. Whether the SQLSTATE class code is defined by the SQL standard (indicated by the character string 'ISO 9075') or by Versant ReVind.

SUBCLASS_ORIGIN

Whether the SQLSTATE subclass code is defined by the SQL standard (indicated by the character string 'ISO 9075') or by Versant ReVind.

ENVIRONMENT_NAME

Not supported currently.

CONNECTION_NAME

Not supported currently.

CONSTRAINT_CATALOG

Not supported currently.

CONSTRAINT_SCHEMA

Not supported currently.

CONSTRAINT_NAME

Not supported currently.

CATALOG_NAME

Not supported currently.

SCHEMA_NAME

Not supported currently.

TABLE_NAME

If the error condition involves a table, the name of the table.

COLUMN_NAME

If the error condition involves a column, the name of the affected columns.

`CURSOR_NAME`

Not supported currently.

`MESSAGE_TEXT`

The associated message text for the error condition.

`MESSAGE_LENGTH`

The length in characters of the message in the `MESSAGE_LENGTH` item.

`MESSAGE_OCTET_LENGTH`

Not supported currently.

Example:

```
GET DIAGNOSTICS :num = NUMBER, :cmdfunc = COMMAND_FUNCTION
GET DIAGNOSTICS EXCEPTION :num
   :sstate = RETURNED_SQLSTATE, :msgtxt = MESSAGE_TEXT
```

SQL Compliance	SQL-92
Environment	Embedded SQL
Related Statements	`WHENEVER`

GRANT

```
GRANT  { RESOURCE, DBA }
    TO user_name [ , user_name ] ... ;
```

Grant database-wide privileges, either system administration (DBA) or general creation (`RESOURCE`), to specified users.

```
GRANT  { privilege [ , privilege ] ... | ALL [ PRIVILEGES ] }
    ON table_name
    TO { user_name [ , user_name ] ... | PUBLIC }
   [WITH GRANT OPTION] ;
```

Grant privileges on specific tables and views to specified users.

```
GRANT EXECUTE ON procedure_name
    TO { user_name [ , user_name ] ... | PUBLIC } ;
```

Grant privilege to execute the specified stored procedure.

```
privilege ::
    {   SELECT
      | INSERT
      | DELETE
      | ALTER
      | INDEX
      | UPDATE [ (column, column, ... ) ]
      | REFERENCES [ (column, column, ... ) ] }
```

Arguments are:

DBA

Allows the specified users to create, access, modify, or delete any database object, and to grant other users any privileges.

RESOURCE

Allows the specified users to issue CREATE statements. The RESOURCE privilege does not allow users to issue DROP statements on database objects. Only the owner of the object and users with the DBA privilege can drop database objects.

ALTER

Allows the specified users to modify the table or view

DELETE

Allows the specified users to delete rows in the table or view

INDEX

Allows the specified users to create an index on the table or view.

INSERT

Allows the specified users to add new rows to the table or view.

`SELECT`

Allows the specified users to read data in the table or view.

`UPDATE [ (column, column, ... ) ]`

Allows the specified users to modify existing rows in the table or view. If followed by a column list, the users can modify values only in the columns named.

`REFERENCES [ (column, column, ... ) ]`

Allows the specified users to refer to the table from other tables' constraint definitions. If followed by a column list, constraint definitions can refer only to the columns named.

For more detail on constraint definitions, see "Table Constraints" and "Column Constraints".

`ALL`

Grants all privileges for the table/view.

`ON table_name`

The table or view for which SQL grants the specified privileges.

`EXECUTE ON procedure_name`

Allows execution of the specified stored procedure.

`TO user_name [ , user_name ] ...`

The list of users for which privileges are granted.

`TO PUBLIC`

Grants the specified privileges on the table/view to any user with access to the system.

`WITH GRANT OPTION`

Allows the specified users to grant their access rights or a subset of their rights to other users.

Example:

```
GRANT ALTER ON cust_view TO dbuser1 ;
GRANT SELECT ON newcustomers TO dbuser2;
GRANT EXECUTE ON sample_proc TO dbuser2;
```

Authorization	The user granting DBA or RESOURCE privileges must have the DBA privilege. The user granting privileges on a table must have any of the following privileges: • DBA privilege • Ownership of the table • All the specified privileges on the table, granted with the WITH GRANT OPTION clause
SQL Compliance	SQL-92, ODBC Core SQL grammar. Extensions: ALTER, INDEX, RESOURCE, DBA privileges
Environment	Interactive SQL, ODBC applications
Related Statements	REVOKE

INSERT

```
INSERT INTO [owner_name.] { table_name | view_name }
          [ (column_name, column_name, ...) ]
          { VALUES (value, value, ...) | query_expression };
```

Inserts new rows into the specified table/view that will contain either the explicitly specified values or the values returned by the query expression.

If the optional list of column names is specified, then only the values for those columns need be supplied. The rest of the columns of the inserted row will contain NULL values, provided the table definition allows NULL values and there is no DEFAULT clause for the columns. If a DEFAULT clause is specified for a column and the column name is not present in the optional column list, then the column takes the default value.

If the optional list is not specified then all the column values have to be either explicitly specified or returned by the query expression. The order of the values should be the same as the order in which the columns have been declared in the declaration of the table/view.

Explicit specification of the column values provides for insertion of only one row at a time. The query expression option allows for insertion of multiple rows at a time.

If the table contains a foreign key, and there does not exist a corresponding primary key that matches the values of the foreign key in the record being inserted, the insert operation is rejected.

Examples:

```
INSERT INTO customer (cust_no, name, street, city, state)
       VALUES
         (1001, 'RALPH', '#10 Columbia Street', 'New York', 'NY') ;

     INSERT INTO neworders (order_no, product, qty)
          SELECT order_no, product, qty
          FROM orders
          WHERE order_date = SYSDATE ;
```

Authorization	The user executing this statement must have any of the following privileges: • DBA privilege. • Ownership of the table. • `INSERT` privilege on the table. If a `query_expression` is specified, then the user must have any of the following privileges: • DBA privilege. • `SELECT` privilege on all the tables/views referred to in the `query_expression`.
SQL Compliance	SQL-92, ODBC Core SQL grammar
Environment	Embedded SQL, Interactive SQL, ODBC applications
Related Statements	Query Expressions

LOCK TABLE

```
LOCK TABLE table_name [ , ... ]
IN { SHARE | EXCLUSIVE } MODE
[ NOWAIT ] ;
```

Explicitly locks the specified tables for shared or exclusive access.

Versant ReVind does not control the details of locking and data consistency within transactions. See the documentation for your storage manager for details.

Explicit locking can be used to improve the performance of a single transaction at the cost of decreasing the concurrency of the system and potentially blocking other transactions. It is more efficient to explicitly lock a table if you know ahead of time that the transaction would be updating a substantial part of a table. The efficiency is gained by decreased overhead of the implicit locking mechanism and any potential waits for acquiring page level locks for the table.

Explicit locking can be used to minimize potential deadlocks in situations where a substantial part of a table is being modified by a transaction. The benefits of table locking should always be compared with the disadvantages of loosing concurrency before a choice is made between explicit and implicit locking.

The `SHARE` mode allows other transactions to read the table but does not allow modifications on the table.

The `EXCLUSIVE` mode does not allow any other transactions to read and/or modify the table.

If the lock request cannot be honored by the system (due to a conflict lock held by another transaction) then in the normal case the transaction is suspended until the specified lock can be acquired. The `NOWAIT` option provides an immediate return of control if the lock cannot be acquired.

Locks that are acquired explicitly and/or implicitly are released only when the transaction is ended using either the `COMMIT` or the `ROLLBACK WORK` statement.

Example:

```
LOCK TABLE customer
   IN EXCLUSIVE MODE ;
```

Authorization	The user executing this statement must have any of the following privileges: • DBA privilege. • `SELECT` privilege on the table.
SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	`SELECT, INSERT, DELETE`

RENAME

```
RENAME [owner_name.] oldname TO [owner_name.] newname ;
```

Renames the specified view name or synonym to the new name specified

Arguments are:

`[owner_name.]`

Optional owner-name qualifier for the name. If the owner name is not the same as that of the current user, the current user must have the DBA privilege.

If specified, the owner name must be the same for `oldname` and `newname`. In other words, you cannot change the owner of a table, view, or synonym with `RENAME`.

`oldname`

Current name of the view or synonym.

`newname`

New name for the view or synonym.

Example:

```
RENAME sitem TO supplier_item ;
```

Authorization	The user executing this statement must have any of the following privileges: • DBA privilege • Ownership of the view/synonym • `ALTER` privilege on the view
SQL Compliance	Extension
Environment	Interactive SQL, ODBC applications
Related Statements	`CREATE VIEW, CREATE SYNONYM`

REVOKE

```
REVOKE { RESOURCE | DBA }
  FROM { user_name [ , user_name ] ... } [RESTRICT|CASCADE];
```

Revoke database-wide privileges, either system administration (`DBA`) or general creation (`RESOURCE`) for the specified users.

```
REVOKE [ GRANT OPTION FOR ]
       { privilege [ , privilege, ] ... | ALL [ PRIVILEGES ] }
```

```
  ON     table_name
  FROM { user_name [ , user_name ] ... | PUBLIC }
       [ RESTRICT | CASCADE ] ;
```

Revoke specific privileges on specific tables and views for specific users.

```
REVOKE [ GRANT OPTION FOR ] EXECUTE ON procedure_name
  FROM { user_name [ , user_name ] ... | PUBLIC }
       [ RESTRICT | CASCADE ] ;
```

Revoke the privilege to execute the specified stored procedure for the specified users.

```
privilege ::
       {   SELECT
         | INSERT
         | DELETE
         | ALTER
         | INDEX
         | UPDATE [ (column, column, ... ) ]
         | REFERENCES [ (column, column, ... ) ]
       }
```

Arguments are:

`GRANT OPTION FOR`

Revokes the grant option for the privilege from the specified users. The actual privilege itself is not revoked. If specified with `RESTRICT`, and the privilege was passed on to other users, the `REVOKE` statement fails and generates an error. Otherwise, `GRANT OPTION FOR` implicitly revokes any rights the user may have in turn given to other users.

`{ privilege [ , privilege, ] ... | ALL [ PRIVILEGES ] }`

List of privileges to be revoked. See the description in the `GRANT` statement for details on specific privileges. Revoking `RESOURCE` and `DBA` rights can only be done by the administrator or a user with DBA rights.

If a user has been granted access to a table by more than one user then all the users have to perform a revoke for the user to lose his access to the table.

Using the keyword `ALL` revokes all the rights granted on the table/view.

`ON table_name`

The table or view for which SQL revokes the specified privileges.

`EXECUTE ON procedure_name`

Revokes the right to execute the specified stored procedure.

`FROM user_name [ , user_name ] ...`

Revokes the specified rights on the table or view from the specified list of users.

`FROM PUBLIC`

Revokes the specified rights on the table or view from any user with access to the system.

`RESTRICT | CASCADE`

If the `REVOKE` statement specifies `RESTRICT`, SQL checks to see if the privilege being revoked was passed on to other users (possible only if the original privilege included the `WITH GRANT OPTION` clause). If so, the `REVOKE` statement fails and generates an error. If the privilege was not passed on, the `REVOKE` statement succeeds.

If the `REVOKE` statement specifies `CASCADE`, revoking the access right of a user also revokes the rights from all users who received the privilege as a result of that user giving the privilege to others.

If the `REVOKE` statement specifies neither `RESTRICT nor CASCADE`, the behavior is the same as for `CASCADE`.

Example:

```
REVOKE INSERT ON customer FROM dbuser1 ;
REVOKE ALTER ON cust_view FROM dbuser2 ;
```

Authorization	The user revoking DBA or `RESOURCE` privileges must have the DBA privilege. The user revoking privileges on a table must have any of the following privileges: • DBA privilege • Ownership of the table • All the specified privileges on the table, granted with the `WITH GRANT OPTION` clause
SQL Compliance	SQL-92, ODBC Core SQL grammar. Extensions: `ALTER, INDEX, RESOURCE,` DBA privileges

Environment	Interactive SQL, ODBC applications
Related Statements	`GRANT`

ROLLBACK WORK

```
ROLLBACK [ WORK ] ;
```

Ends the current transaction and undoes any database changes performed during the transaction.

Under certain circumstances, Versant ReVind marks a transaction for abort but does not actually roll it back. Without an explicit rollback, any subsequent updates will not take effect, since a `COMMIT` statement cause Versant ReVind to recognize the transaction as marked for abort, and instead implicitly rolls back the transaction. Versant ReVind marks a transaction for abort under these conditions:

- Hardware or software system failures
- Lock timeout errors

Authorization	None
SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	`COMMIT WORK`

SELECT

```
select_statement ::
  query_expression
    ORDER BY
      { expr | posn }
      [ COLLATE collation_name ]
      [ ASC | DESC ]
      [ , { expr | posn }
          [ COLLATE collation_name ]
          [ASC | DESC] ,... ]
    FOR UPDATE
      [ OF [table].column_name, ... ]
      [ NOWAIT ] ;
```

```
query_expression ::
    query_specification
  | query_expression set_operator query_expression
  | ( query_expression )

set_operator ::
  {   UNION [ ALL ]
    | INTERSECT
    | MINUS }

query_specification ::
  SELECT
    [ALL | DISTINCT]
    {   *
      | { table_name | alias } . *
        [, { table_name | alias } . * ]
      | expr
        [ [ AS ]
        [ ' ] column_title [ ' ] ]
          [, expr [ [ AS ]
          [ ' ] column_title [ ' ] ] ] ...
    }
  FROM
      table_ref
    [ {versant ORDERED} ]
      [, table_ref [ { versant ORDERED } ] …
  [ WHERE search_condition ]
  [ GROUP BY
      [table.]column_name [COLLATE collation-name]
        [, [table.]column_name [COLLATE collation-name] ] ...
  [ HAVING search_condition ]

table_ref ::
  table_name
      [ AS ] [ alias [ ( column_alias [ , … ] ) ] ]
    | ( query_expression )
        [ AS ] alias [ ( column_alias [ , … ] ) ]
```

```
    | [ ( ] joined_table [ ) ]

joined_table ::
  table_ref
      CROSS JOIN table_ref
    | table_ref [ INNER | LEFT [ OUTER ] ]
        JOIN table_ref ON search_condition

joined_table ::
     table_ref CROSS JOIN table_ref
  |  table_ref [ INNER | { LEFT | RIGHT | FULL } [ OUTER ] ]
     JOIN table_ref
         { ON search_condition }
       | { USING ( column_name [ , … ] ) }
       |  table_ref NATURAL
            [ INNER | { LEFT | RIGHT | FULL } [ OUTER ] ]
            JOIN table_ref
       |  table_ref UNION JOIN table_ref
```

Selects the specified column values from one or more rows contained in the table(s) specified in the `FROM` clause. The selection of rows is restricted by the `WHERE` clause. The temporary table derived through the clauses of a select statement is called a result table.

The general form of the `SELECT` statement is a query expression with optional `ORDER BY` and `FOR UPDATE` clauses.

For more detail on query expressions, see “Query Expressions” on page 148 in "Chapter 5 - Versant ReVind Common Language Elements".

Arguments for `query_expression` are:

`ORDER BY` clause

```
ORDER BY
  {expr | posn}
  [ COLLATE collation_name ]
  [ASC | DESC]
    [, {expr | posn}
       [ COLLATE collation_name ]
       [ASC | DESC], ... ]
```

The ORDER BY clause specifies the sorting of rows retrieved by the SELECT statement. SQL does not guarantee the sort order of rows unless the SELECT statement includes an ORDER BY clause.

Ascending order is the default ordering. The descending order will be used only if the keyword DESC is specified for that column.

Each expr is an expression of one or more columns of the tables specified in the FROM clause of the SELECT statement. Each posn is a number identifying the column position of the columns being selected by the SELECT statement.

The selected rows are ordered on the basis of the first expr or posn and if the values are the same then the second expr or posn is used in the ordering.

The ORDER BY clause if specified should follow all other clauses of the SELECT statement.

A query expression followed by an optional ORDER BY clause can be specified. In such a case, if the query expression contains set operators, then the ORDER BY clause can specify only the positions.

For example:

```
-- Get a merged list of customers and suppliers
-- sorted by their name.
     (SELECT name, street, state, zip
      FROM customer
      UNION
      SELECT name, street, state, zip
      FROM supplier)
      ORDER BY 1 ;
```

If expr or posn refers to a character column, the reference can include an optional COLLATE clause. The COLLATE clause specifies a collation sequence supported by the underlying storage system.

For notes on character sets and collations, please refer “Specifying the Character Set for Character Data Types” on page 140 in "Chapter 5 - Versant ReVind Common Language Elements".

(See the documentation for your underlying storage system for details on any supported collations.)

Example:

```
SELECT name, street, city, state, zip
     FROM customer
     ORDER BY name ;
FOR UPDATE Clause
```

```
FOR UPDATE [ OF [table].column_name, ... ] [ NOWAIT ]
```

The FOR UPDATE clause specifies update intention on the rows selected by the SELECT statement.

If FOR UPDATE clause is specified, WRITE locks are acquired on all the rows selected by the SELECT statement.

If NOWAIT is specified, an error is returned when a lock cannot be acquired on a row in the selection set because of the lock held by some other transaction. Otherwise, the transaction would wait until it gets the required lock or until it times out waiting for the lock.

SET CONNECTION

```
SET CONNECTION connection_name ;
```

where

```
connection_name::
      { character_literal | host_variable | DEFAULT }
```

SET CONNECTION sets the database associated with the connection name as the current database.

If DEFAULT is specified, there should exist a DEFAULT connection (this could have been previously achieved through a CONNECT TO DEFAULT statement). All SQL statements are executed for the current database.

Examples:

```
SET CONNECTION "salesdb";
SET CONNECTION DEFAULT;
```

Authorization	None
SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	DISCONNECT, SET CONNECTION

SET TRANSACTION ISOLATION

```
SET TRANSACTION ISOLATION LEVEL { 0 | 1 | 2 | 3 };
```

Choose the isolation level for future transactions. If a transaction is currently active, SET TRANSACTION generates an error.

The isolation level specifies the degree to which one transaction is isolated from the effects of concurrent access of the database by other transactions. The appropriate level of isolation depends on how a transaction needs to be isolated from effects of another transaction. Higher isolation levels provide greater data consistency to the user's transaction but reduce access to data by concurrent transactions.

The following table shows how the isolation level integers correspond to SQL-92 standard isolation level keywords:

Number	Isolation Level
1	UNCOMMITTED READ
0	COMMITTED READ
2	REPEATABLE READ
3	SERIALIZABLE

The isolation level `SERIALIZABLE` guarantees the highest consistency and is the default. The isolation level `READ UNCOMMITTED` guarantees the least consistency.

The ANSI/ISO SQL standard defines isolation levels in terms of the of the inconsistencies they allow:

Dirty read	Allows the transaction to read a row that has been inserted or modified by another transaction, but not committed. If the other transaction rolls back its changes, the transaction will have read a row that never existed, in the sense that it was never committed.
Non-repeatable read	Allows the transaction to read a row that another transaction modifies or deletes before the next read operation. If the other transaction commits the change, the transaction will receive modified values, or discover the row is deleted, on subsequent read operations.
Phantom	Allows the transaction to read a range of rows that satisfies a given search condition, but to which another transaction adds rows before a another read operation using the same search condition. The transaction receives a different collection of rows with the same search condition.

Arguments are:

`READ UNCOMMITTED`

Allows dirty reads, non-repeatable reads, and phantoms.

`READ COMMITTED`

Prohibits dirty reads; allows non-repeatable reads and phantoms.

`REPEATABLE READ`

Prohibits dirty reads and non-repeatable reads; allows phantoms.

`SERIALIZABLE`

The default. Prohibits dirty reads, non-repeatable reads, and phantoms. It guarantees that the concurrent transactions will not affect each other and they will behave as if they were executing serially, not concurrently.

Authorization	None
SQL Compliance	SQL-92. Extension: the numeric syntax is not standard, but the semantics it corresponds to are standard.
Environment	Embedded SQL and interactive
Related Statements	`LOCK TABLE, COMMIT, ROLLBACK`

UPDATE

```
UPDATE table_name
    SET assignment, assignment, ...
  [ WHERE search_condition ]

assignment ::
      column = { expr | NULL }
  | ( column, column, ... ) = ( expr, expr, ... )
  | ( column, column, ... ) = ( query_expression )
```

Updates the columns of the specified table with the given values that satisfy the `search_condition`.

If the optional `WHERE` clause is specified, then only rows that satisfy the search_condition are updated. If the `WHERE` clause is not specified then all rows of the table are updated.

The expressions in the SET clause are evaluated for each row of the table if they are dependent on the columns of the target table.

If a query expression is specified on the right hand side for an assignment, the number of expressions in the first SELECT clause of the query expression must be the same as the number of columns listed on the left hand side of the assignment.

If a query expression is specified on the right hand side for an assignment, the query expression must return one row.

If a table has primary/candidate keys and if the columns to be updated are part of the primary/candidate key, a check is made as to whether there exists any corresponding row in the referencing table. If so, the UPDATE operation fails.

Example:

```
UPDATE orders
          SET qty = 12000
          WHERE order_no = 1001 ;
UPDATE orders
          SET (product) =
                    (SELECT item_name
                     FROM items
                     WHERE item_no = 2401
                    )
          WHERE order_no = 1002 ;
UPDATE orders
          SET (amount) = (2000 * 30)
          WHERE order_no = 1004 ;
UPDATE orders
          SET (product, amount) =
                         (SELECT item_name, price * 30
                          FROM items
                          WHERE item_no = 2401
                         )
          WHERE order_no = 1002 ;
```

Authorization	The user executing this statement must have: • DBA privilege. • UPDATE privilege on all the specified columns of the target table and SELECT privilege on all the other tables referred to in the statement.
SQL Compliance	SQL-92, ODBC Extended SQL grammar. Extensions: assignments of the form `( column, column, ... ) = ( expr, expr, ... )`
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	SELECT, OPEN, FETCH, search conditions, query expressions

UPDATE STATISTICS

```
UPDATE STATISTICS [ FOR table_name ]
```

Queries system tables and updates table and column statistics:

- The number of rows in the table (the cardinality)
- The approximate number of occurrences of a value in each column

If the underlying storage system does not return its own information, the optimizer uses the information from UPDATE STATISTICS to calculate a query strategy for a particular SQL statement.

Until a user, application, or SQL script issues an UPDATE STATISTICS statement, the optimizer bases query strategies on values it generates from various defaults. These values will not lead to the best performance, so it is good practice for database administrators to periodically update statistics.

UPDATE STATISTICS only works on tables that have indexes defined on them.

Arguments are:

`table_name`

Specifies a single table on which to update statistics. The default is to update statistics on all tables in the database.

.

Authorization	To issue the `UPDATE STATISTICS` statement for all tables in the database, the user must have DBA privilege or `SELECT` privilege on all the tables in the database. To issue the `UPDATE STATISTICS` statement for a specific table, the user must be the owner or have `SELECT` privilege on the table.
SQL Compliance	Extension
Environment	Interactive SQL, ODBC applications
Related Statements	`SET DISPLAY COST ON` (Interactive SQL) `SET DISPLAY COST ON` (Interactive SQL)

CHAPTER 7

Versant ReVind Function Reference

This Chapter gives a detailed description of the Versant ReVind functions.

The Chapter explains the following in detail:

- Functions
- Aggregate Functions
- Scalar Functions

FUNCTIONS

Functions are a type of SQL expression that return a value based on the argument they are supplied. SQL supports two types of functions:

- Aggregate functions calculate a single value for a collection of rows in a result table (if the function is in a statement with a `GROUP BY` clause, it returns a value for each group in the result table). Aggregate functions are also called set or statistical functions. Aggregate functions cannot be nested.
- Scalar functions calculate a value based on another single value. Scalar functions are also called value functions. Scalar functions can be nested.

AGGREGATE FUNCTIONS

AVG

```
AVG ( { [ALL] expression } | { DISTINCT column_ref } )
```

The aggregate function AVG computes the average of a collection of values. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the average.

- Null values are eliminated before the average value is computed. If all the values are null, the result is null.
- The argument to the function must be of type SMALLINT, INTEGER, NUMERIC, REAL or FLOAT.
- The result is of type NUMERIC.

Example

```
SELECT AVG (salary)
    FROM employee
    WHERE deptno = 20 ;
```

COUNT

```
COUNT ( { [ALL] expression } | { DISTINCT column_ref } | * )
```

The aggregate function COUNT computes either the number of rows in a group of rows or the number of non-null values in a group of values.

- The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the count.
- If the argument to COUNT function is '*', then the function computes the count of the number of rows in group.
- If the argument to COUNT function is not '*', then null values are eliminated before the number of rows is computed.
- The argument column_ref or expression can be of any type.
- The result of the function is of INTEGER type. The result is never null.

Example

```
SELECT COUNT (*)
    FROM orders
    WHERE order_date = SYSDATE ;
```

MAX

```
MAX ( { [ALL] expression } | { DISTINCT column_ref } )
```

The aggregate function `MAX` returns the maximum value in a group of values.

- The specification of `DISTINCT` has no effect on the result.
- The argument `column_ref` or `expression` can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is `null` if the result set is empty or contains only null values.

Example

```
SELECT order_date, product, MAX (qty)
    FROM orders
    GROUP BY order_date, product ;
```

MIN

```
MIN ( { [ALL] expression } | { DISTINCT column_ref } )
```

The aggregate function `MIN` returns the minimum value in a group of values.

- The specification of `DISTINCT` has no effect on the result.
- The argument `column_ref` or `expression` can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is `null` if the result set is empty or contains only null values.

Example

```
SELECT MIN (salary)
    FROM employee
    WHERE deptno = 20 ;
```

SUM

```
SUM ( { [ALL] expression } | { DISTINCT column_ref } )
```

The aggregate function `SUM` returns the sum of the values in a group. The keyword `DISTINCT` specifies that the duplicate values are to be eliminated before computing the sum.

- The argument `column_ref` or `expression` can be of any type.
- The result of the function is of the same data type as that of the argument except that the result is of type `INTEGER` when the argument is of type `SMALLINT` or `TINYINT`.
- The result can have a null value.

Example

```
SELECT SUM (amount)
    FROM orders
    WHERE order_date = SYSDATE ;
```

SCALAR FUNCTIONS

ABS function (ODBC compatible)

`ABS ( expression )`

The scalar function `ABS` computes the absolute value of `expression`.

Example

```
SELECT ABS (MONTHS_BETWEEN (SYSDATE, order_date))
    FROM orders
    WHERE  ABS (MONTHS_BETWEEN (SYSDATE, order_date)) > 3 ;
```

NOTES:-

- The argument to the function must be of type `TINYINT, SMALLINT, INTEGER, NUMERIC, REAL` or `FLOAT`.
- The result is of type `NUMERIC`.
- If the argument `expression` evaluates to `null`, the result is `null`.

ACOS function (ODBC compatible)

`ACOS ( expression )`

The scalar function `ACOS` returns the arccosine of `expression`.

Example

```
select acos (.5) 'Arccosine in radians' from syscalctable;
ARCCOSINE IN RAD
----------------
1.047197551196598
1 record selected
select acos (.5) * (180/ pi()) 'Arccosine in degrees' from
syscalctable;
```

```
ARCCOSINE IN DEG
----------------
59.999999999999993
1 record selected
```

NOTES:-

- ACOS takes the ratio (expression) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

  The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.
- expression must be in the range -1 to 1.
- expression must evaluate to an approximate numeric data type.

ADD_MONTHS function (extension)

```
ADD_MONTHS ( date_expression, integer_expression )
```

The scalar function ADD_MONTHS adds to the date value specified by the date_expression, the given number of months specified by integer_expression and returns the resultant date value.

Example

```
SELECT *
FROM   customer
WHERE  ADD_MONTHS (start_date, 6) > SYSDATE ;
```

NOTES:-

- The first argument must be of DATE type.
- The second argument to the function must be of numeric type.
- The result is of type DATE.
- If any of the arguments evaluate to null, the result is null.

ASCII function (ODBC compatible)

```
ASCII ( char_expression )
```

The scalar function `ASCII` returns the ASCII value of the first character of the given character expression.

Example

```
SELECT ASCII ( zip )
     FROM  customer ;
```

NOTES:-

- The argument to the function must be of type character.
- The result is of type `INTEGER`.
- If the argument `char_expression` evaluates to null, the result is `null`.

ASIN function (ODBC compatible)

```
ASIN ( expression )
```

The scalar function `ASIN` returns the arcsine of `expression`.

Example

```
select asin (1) * (180/ pi()) 'Arcsine in degrees' from syscalctable;
ARCSINE IN DEGRE
----------------
90.000000000000000
1 record selected
 select asin (1) 'Arcsine in radians' from syscalctable;
ARCSINE IN RADIA
----------------
1.570796326794897
1 record selected
```

NOTES:-

- `ASIN` takes the ratio (`expression`) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

  The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.
- `expression` must be in the range `-1` to `1`.
- `expression` must evaluate to an approximate numeric data type.

ATAN function (ODBC compatible)

```
ATAN ( expression )
```

The scalar function `ATAN` returns the arctangent of `expression`.

Example

```
select atan (1) * (180/ pi()) 'Arctangent in degrees' from syscalctable;
ARCTANGENT IN DE
---------------
45.000000000000000
1 record selected
select atan (1) 'Arctangent in radians' from syscalctable;
ARCTANGENT IN RA
---------------
0.785398163397448
1 record selected
```

NOTES:-

- `ATAN` takes the ratio (`expression`) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

  The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.
- `expression` must be in the range `-1` to `1`.

- expression must evaluate to an approximate numeric data type.

ATAN2 function (ODBC compatible)

```
ATAN2 ( expression1 , expression2 )
```

The scalar function ATAN2 returns the arctangent of the x and y coordinates specified by expression1 and expression2.

Example

```
select atan2 (1,1) * (180/ pi()) 'Arctangent in degrees' from syscalctable;
ARCTANGENT IN DE
----------------
45.000000000000000
1 record selected
select atan2 (1,1) 'Arctangent in radians' from syscalctable;
ARCTANGENT IN RA
----------------
0.785398163397448
1 record selected
```

NOTES:-

- ATAN2 takes the ratio of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.
- expression1 and expression2 specify the x and y coordinates of the end of the hypotenuse opposite the angle.
- The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.
- Both expression1 and expression2 must evaluate to approximate numeric data types.

CASE (SQL-92 Compatible)

```
case-expr::
searched-case-expr | simple-case-expr
searched-case-expr::
CASE
     WHEN search_condition THEN { result-expr | NULL }
     [ ... ]
     [ ELSE expr | NULL ]
END
simple-case-expr::
CASE primary-expr
     WHEN expr THEN { result-expr | NULL }
     [ ... ]
     [ ELSE expr | NULL ]
END
```

The CASE scalar function is a type of conditional expression.

For a summary of all the conditional expressions, please refer “Conditional Expressions” on page 173 in "Chapter 5 - Versant ReVind Common Language Elements".

The general form of the CASE scalar function specifies a series of search conditions and associated result expressions. It is called a searched case expression. SQL returns the value specified by the first result expression whose associated search condition evaluates as true. If none of the search conditions evaluate as true, the CASE expression returns a null value (or the value of some other default expression if the CASE expression includes the ELSE clause).

CASE also supports syntax for a shorthand notation, called a simple case expression, for evaluating whether one expression is equal to a series of other expressions.

NOTES:-

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.

Arguments are:

CASE

The CASE keyword alone, not followed by primary-expr, specifies a searched case expression. It must be followed by one or more WHEN-THEN clauses each that specify a search condition and corresponding expression.

```
WHEN search_condition THEN { result-expr | NULL }
```

`WHEN` clause for searched case expressions. SQL evaluates `search condition`. If `search_condition` evaluates as true, `CASE` returns the value specified by `result-expr` (or `null`, if the clause specifies `THEN NULL`).

If `search_condition` evaluates as false, SQL evaluates the next `WHEN-THEN` clause, if any, or the `ELSE` clause, if it is specified.

```
CASE primary-expr
```

The `CASE` keyword followed by an expression specifies a simple case expression. In a simple case expression, one or more `WHEN-THEN` clauses specify two expressions.

A simple case expression can always be expressed as a searched case expression. Consider the following general simple case expression:

```
CASE primary-expr
     WHEN expr1 THEN result-expr1
     WHEN expr2 THEN result-expr2
     ELSE expr3
END
```

The preceding simple case expression is equivalent to the following searched case expression:

```
CASE
     WHEN primary-expr = expr1 THEN result-expr1
     WHEN primary-expr = expr2 THEN result-expr2
     ELSE expr3
END
```

```
WHEN expr THEN { result-expr | NULL }
```

`WHEN` clause for simple case expressions. SQL evaluates *expr* and compares it with `primary-expr` specified in the `CASE` clause. If they are equal, `CASE` returns the value specified by `result-expr` (or `null`, if the clause specifies `THEN NULL`).

If `expr` is not equal to `primary-expr`, SQL evaluates the next `WHEN-THEN` clause, if any, or the `ELSE` clause, if it is specified.

```
[ ELSE { expr | NULL } ]
```

In both searched case expressions and simple case expressions, the `ELSE` clause specifies an optional expression whose value SQL returns if none of the conditions specified in

WHEN-THEN clauses were satisfied. If the CASE expression omits the ELSE clause, it is the same as specifying ELSE NULL.

Examples

The following example shows a searched case expression that assigns a label denoting tables as system tables if they begin with the letters sys. Note that this example can not be reformulated as a simple case expression, since it specifies a relational operator other than =.

```
SELECT tbl,
       CASE
          WHEN tbl like 'sys%' THEN 'System Table'
          ELSE 'Not System table'
       END
FROM systables;
TBL                                 SEARCHED_CASE(TBLSY
---                                 -------------------
systblspaces                        System Table
systables                           System Table
syscolumns                          System Table
sysindexes                          System Table
sysdbauth                           System Table
systabauth                          System Table
syscolauth                          System Table
sysviews                            System Table
syssynonyms                         System Table
sysdblinks                          System Table
sys_keycol_usage                    System Table
sys_ref_constrs                     System Table
sys_chk_constrs                     System Table
sys_tbl_constrs                     System Table
sys_chkcol_usage                    System Table
sysdatatypes                        System Table
syscalctable                        System Table
systblstat                          System Table
```

The following example shows a searched case expression and an equivalent simple case expression:

```
– Searched case expression:
SELECT tbl,
   CASE
      WHEN tbltype = 'S' THEN 'System Table'
      ELSE 'Not System table'
   End
FROM systables;
– Equivalent simple case expression:
SELECT tbl,
   CASE tbltype
      WHEN  'S' THEN 'System Table'
      ELSE 'Not System table'
   END
FROM systables;
```

CAST function (SQL-92 compatible)

```
CAST ( { expression | NULL } AS data_type [(length)] )
```

The scalar function CAST converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

The length option for the data_type argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 30 bytes.

If the expression evaluates to null, the result of the function is null. Specifying NULL with the CAST function is useful for set operations, such as UNION, that require two table to have the same structure. CAST NULL allows you to specify a column of the correct data type so a table with a similar structure to another, but with fewer columns, can be in a union operation with the other table.

The CAST function provides a data-type-conversion mechanism compatible with the SQL-92 standard.

Use the CONVERT function, enclosed in the ODBC escape clause {fn }, to specify ODBC-compliant syntax for data type conversion.

See the “CONVERT function (ODBC compatible)” on page 249 for more information.

Example

The following SQL example uses CAST to convert an integer field from a catalog table to a character data type:

```
SELECT CAST(fld AS CHAR(25)), fld  FROM systpe.syscalctable;
CONVERT(CHARACTER(25),FLD)          FLD
--------------------------         ---
100                                 100
1 record selected
```

CEILING function (ODBC compatible)

```
CEILING ( expression )
```

The scalar function CEILING returns the smallest integer greater than or equal to expression.

Example

```
SELECT CEILING (32.5) 'Ceiling'
     FROM SYSTPE.SYSCALCTABLE;
```

NOTE:- Expression must evaluate to a numeric data type.

CHAR function (ODBC compatible)

```
CHAR ( integer_expression )
```

The scalar function CHAR returns a character string with the first character having an ASCII value equal to the argument expression. CHAR is identical to CHR but provides ODBC-compatible syntax.

Example

```
SELECT *
     FROM   customer
     WHERE  SUBSTR (zip, 1, 1) = CHAR (53) ;
```

NOTES:-

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type character.
- If the argument integer_expression evaluates to null, the result is null.

CHARTOROWID (extension)

```
CHARTOROWID ( char_expression )
```

The scalar function `CHARTOROWID` returns a `ROWID` contained in the input argument in character form. The representation of a row identifier depends on the storage manager. The format of the `char_expression` argument to this function varies between storage managers.

Data Integration Notes

This function works only on local tables.

Example

The following example shows the character-string format for a row identifier supplied as an argument to `CHARTOROWID` (note that the format shown is specific to the Versant ReVind storage manager):

```
SELECT *
     FROM   customer
     WHERE  ROWID = CHARTOROWID ('00000032.1001.0000');
```

The following example shows the character-string format for a row identifier supplied as an argument to `CHARTOROWID`. In this example, the format for a row identifier is an integer (delimited as a character string by single quotes).

```
SELECT ROWID, FLD FROM SYSCALCTABLE;
ROWID                          FLD
-----                          ---
0                              100
1 record selected
-- CHARTOROWID requires single quotes around its argument
SELECT * FROM SYSCALCTABLE WHERE ROWID = CHARTOROWID ('0');
         FLD
         ---
         100
1 record selected
```

General Notes

- The argument to the function must be of type `character`.
- The result is of internal `ROWID` type as defined by the storage manager.

- If the argument char_expression evaluates to null, the result is null.
- The SQL statement execution returns error if the result of the input character expression does not contain a character string in the proper format for a row identifier, as defined by the storage manager.

CHR function (extension)

```
CHR ( integer_expression )
```

The scalar function CHR returns a character string with the first character having an ASCII value equal to the argument expression.

Example

```
SELECT *
     FROM   customer
     WHERE  SUBSTR (zip, 1, 1) = CHR (53) ;
```

NOTES:-

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type character.
- If the argument integer_expression evaluates to null, the result is null.

COALESCE (SQL-92 compatible)

```
COALESCE ( expression1, expression2 [ , ... ] )
```

The COALESCE scalar function is a type of conditional expression.

For a summary of all the conditional expressions, please refer “Conditional Expressions” on page 173 in "Chapter 5 - Versant ReVind Common Language Elements".

COALESCE specifies a series of expressions, and returns the first expression whose value is not null. If all the expressions evaluate as null, COALESCE returns a null value.

The COALESCE syntax is shorthand notation for a common case that can also be represented in a CASE expression. The following two formulations are equivalent:

```
COALESCE ( expression1 , expression2 , expression3 )
CASE
   WHEN expression1 IS NOT NULL THEN expression1
   WHEN expression2 IS NOT NULL THEN expression2
   ELSE expression3
```

Example

```
SELECT COALESCE(end_date, start_date) from job_hist;
```

NOTES:-

- This function is not allowed in a `GROUP BY` clause.
- Arguments to this function cannot be query expressions.

CONCAT function (ODBC compatible)

```
CONCAT ( char_expression, char_expression )
```

The scalar function `CONCAT` returns a concatenated character string formed by concatenating argument one with argument two.

The `CONCAT` scalar function is similar to the concatenation operator. However, the concatenation operator allows easy concatenation of more than two character expressions, while the `CONCAT` function requires nesting.

Example

```
SELECT name, empno, salary
     FROM customer
     WHERE project = CONCAT('US',proj_nam);
```

NOTES:-

- Both the arguments must be of type `CHARACTER` or `VARCHAR`.
- The result is of type `VARCHAR`.
- If any of the argument expressions evaluates to null, the result is `null`.
- The trailing blanks for the first argument are removed.

CONVERT function (SQL-92 compatible)

```
CONVERT ( char_expression USING conversion_name )
```

The SQL-92 compatible CONVERT function converts the characters in char_expression to another storage format. The details of the conversion are dependent on the underlying storage system. See the documentation for the underlying storage system for which conversions it supports, if any.

Example

This example converts character strings (customers' street addresses) from a variable number of bytes per character (multi-octet) to two bytes per character. It presumes the underlying storage system supports a conversion called MULTI_TO_DOUBLE_OCTET.

```
SELECT CONVERT (street USING MULTI_TO_DOUBLE_OCTET)
     FROM customer ;
```

NOTES:- The storage representation for a character set is called the form of use of the character set. The form of use for the default ASCII character set is a single byte (or octet) containing a number designating a particular ASCII character. Other character sets, such as Unicode, use two or more bytes (or a varying number of bytes, depending on the character) for each character.

For additional notes on character sets, please refer “Specifying the Character Set for Character Data Types” on page 140 in "Chapter 5 - Versant ReVind Common Language Elements".

Support for specific conversion_names is completely dependent on the underlying storage system. When CONVERT refers to a conversion_name that is not supported by the underlying storage system, SQL generates an error.

CONVERT function (extension)

```
CONVERT ( 'data_type[(length)]', expression )
```

The scalar function CONVERT converts an expression to another data type. The first argument is the target data type. The second argument is the expression to be converted to that type.

The length option for the data_type argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 30 bytes.

If the expression evaluates to null, the result of the function is null.

The CONVERT function syntax is similar to but not compatible with the ODBC CONVERT function. Enclose the function in the ODBC escape clause {fn }, to specify ODBC-compliant syntax.

For more information, please refer “CONVERT function (ODBC compatible)” on page 249.

Example

The following SQL example converts an integer field from a catalog table to a character string:

```
SELECT CONVERT('CHAR', fld), fld  FROM systpe.syscalctable;
CONVERT(CHAR,FLD)                              FLD
---------------------                         ---
100                                            100
1 record selected
SELECT CONVERT('CHAR(35)', fld), fld
     FROM systpe.syscalctable;
CONVERT(CHAR(35),FLD)                          FLD
---------------------                         ---
100                                            100
1 record selected
```

CONVERT function (ODBC compatible)

```
{fn CONVERT (expression , data_type ) }
data_type::
   SQL_BIGINT
|   SQL_BINARY
|   SQL_BIT
|   SQL_CHAR
|   SQL_DATE
|   SQL_DECIMAL
|   SQL_DOUBLE
|   SQL_FLOAT
|   SQL_INTEGER
|   SQL_LONGVARBINARY
|   SQL_LONGVARCHAR
|   SQL_REAL
|   SQL_SMALLINT
|   SQL_TIME
|   SQL_TIMESTAMP
|   SQL_TINYINT
|   SQL_VARBINARY
|   SQL_VARCHAR
```

The ODBC scalar function `CONVERT` converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

If the expression evaluates to null, the result of the function is `null`.

The ODBC `CONVERT` function provides ODBC-compliant syntax for data type conversion. You must enclose the function with the ODBC escape clause `{fn }` to use ODBC-compliant syntax.

COS function (ODBC compatible)

```
COS ( expression )
```

The scalar function `COS` returns the cosine of `expression`.

Example

```
select cos(45 * pi()/180) 'Cosine of 45 degrees' from syscalctable;
COSINE OF 45 DEG
----------------
0.707106781186548
1 record selected
```

NOTES:- COS takes an angle (expression) and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

- expression specifies an angle in radians.
- expression must evaluate to an approximate numeric data type.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

CURDATE function (ODBC compatible)

CURDATE ()

CURDATE returns the current date as a DATE value. This function takes no arguments.

The value contained in this register is of INTEGER type. The host representation is of long integer type. SQL statements can refer to CURDATE anywhere they can refer to a DATE expression.

Example

```
INSERT INTO objects (object_owner, object_id, create_date)
           VALUES (USER, 1001, CURDATE()) ;
```

CURTIME function (ODBC compatible)

CURTIME ()

CURTIME returns the current time as a TIME value. This function takes no arguments.

The value contained in this register is of INTEGER type. The host representation is of long integer type. SQL statements can refer to CURTIME anywhere they can refer to a TIME expression.

Example

```
INSERT INTO objects (object_owner, object_id, create_time)
           VALUES (USER, 1001, CURTIME()) ;
```

DATABASE (ODBC compatible)

```
DATABASE [ ( ) ]
```

The scalar function `DATABASE` returns the name of the database corresponding to the current connection name. This function takes no arguments, and the trailing parentheses are optional.

Example

```
select database() from t2;
DATABASE
--------
steel
1 record selected
```

DAYNAME function (ODBC compatible)

```
DAYNAME ( date_expression )
```

Returns a character string containing the name of the day (for example, Sunday, through Saturday) for the day portion of `date_expression`. The argument `date_expression` can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Example

```
SELECT *
FROM orders
WHERE order_no = 342 and DAYNAME(order_date)='SATURDAY';


 ORDER_NO   ORDER_DATE   REFERENCE   CUST_NO
 342        08/10/1991   tdfg/101     10001

   1 record selected
```

DAYOFMONTH function (ODBC compatible)

```
DAYOFMONTH ( date_expression )
```

The scalar function DAYOFMONTH returns the day of the month in the argument as a short integer value in the range of 1 - 31.

Example

```
SELECT *
     FROM  orders
     WHERE DAYOFMONTH (order_date) = 14 ;
```

NOTES:-

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

DAYOFWEEK function (ODBC compatible)

```
DAYOFWEEK ( date_expression )
```

The scalar function DAYOFWEEK returns the day of the week in the argument as a short integer value in the range of 1 - 7.

Example

```
SELECT *
     FROM  orders
     WHERE DAYOFWEEK (order_date) = 2 ;
```

NOTES:-

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

DAYOFYEAR function (ODBC compatible)

```
DAYOFYEAR ( date_expression )
```

The scalar function DAYOFYEAR returns the day of the year in the argument as a short integer value in the range of 1 - 366.

Example

```
SELECT *
     FROM  orders
     WHERE DAYOFYEAR (order_date) = 300 ;
```

NOTES:-

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

DB_NAME (extension)

```
DB_NAME ( )
```

The scalar function DB_NAME returns the name of the database corresponding to the current connection name. It provides compatibility with the Sybase SQL Server function db_name.

Example

```
SELECT DB_NAME() FROM T2;
DB_NAME
-------
versantv4
1 record selected
```

DECODE function (extension)

```
DECODE ( expression, search_expression, match_expression
             [ , search_expression, match_expression ...]
             [ , default_expression ] )
```

The DECODE scalar function is a type of conditional expression.

For a summary of all the conditional expressions, please refer “Conditional Expressions” on page 173 in "Chapter 5 - Versant ReVind Common Language Elements".

The scalar function DECODE compares the value of the first argument expression with each search_expression and if a match is found, returns the corresponding match_expression. If no match is found, then the function returns default_expression. If default_expression is not specified and no match is found, the function returns a null value.

DECODE provides a subset of the functionality of CASE that is compatible with Oracle SQL syntax. Use a simple case expression for SQL-compatible syntax (see "CASE (SQL-92 Compatible)").

Example

```
SELECT ename, DECODE (deptno,
                        10, 'ACCOUNTS    ',
                        20, 'RESEARCH    ',
                        30, 'SALES       ',
                        40, 'SUPPORT     ',
                        'NOT ASSIGNED'
                    )
     FROM employee ;
```

NOTES:-

- The first argument expression can be of any type. The types of all search_expressions must be compatible with the type of the first argument.
- The match_expressions can be of any type. The types of all match_expressions must be compatible with the type of the first match_expression.
- The type of the default_expression must be compatible with the type of the first match_expression.
- The type of the result is the same as that of the first match_expression.

- If the first argument `expression` is `null` then the value of the `default_expression` is returned, if it is specified. Otherwise, `null` is returned.

DEGREES function (ODBC compatible)

```
DEGREES ( expression )
```

The scalar function `DEGREES` returns the number of degrees in an angle specified in radians by `expression`.

Example

```
SELECT DEGREES(3.14159265359) 'Degrees in pi Radians'
     FROM SYSTPE.SYSCALCTABLE;
```

NOTES:-

- `expression` specifies an angle in radians
- `expression` must evaluate to a numeric data type.

DIFFERENCE function (ODBC compatible)

```
DIFFERENCE ( string_exp1,string_exp2 )
```

The scalar function `DIFFERENCE` returns an integer value that indicates the difference between the values returned by the `SOUNDEX` function for `string_exp1` and `string_exp2`.

Example

```
SELECT DIFFERENCE(name,'Robets')
FROM customer
WHERE  name = 'Roberts';


DIFFEREN


       2
1 record selected
```

NOTES:-

- The arguments of the function can be of the type fixed length or variable length `CHARACTER`.

- The result is INTEGER.
- If the argument expression evaluates to null, the result is null.

EXP function (ODBC compatible)

```
EXP ( expression )
```

The scalar function EXP returns the exponential value of expression (e raised to the power of expression).

Example

```
SELECT EXP( 4 ) 'e to the 4th power'
     FROM SYSTPE.SYSCALCTABLE;
```

NOTE:- expression must evaluate to an approximate numeric data type.

FLOOR function (ODBC compatible)

```
FLOOR ( expression )
```

The scalar function FLOOR returns the largest integer less than or equal to expression.

Example

```
SELECT FLOOR (32.5) 'Floor'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTE:- expression must evaluate to a numeric data type.

GREATEST function (extension)

```
GREATEST ( expression, expression, ... )
```

The scalar function GREATEST returns the greatest value among the values of the given expressions.

Example

```
SELECT cust_no, name,
     GREATEST (ADD_MONTHS (start_date, 10), SYSDATE)
     FROM customer ;
```

NOTES:-

- The first argument to the function can be of any type. The types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluates to null, the result is `null`.

HOUR function (ODBC compatible)

```
HOUR ( time_expression )
```

The scalar function `HOUR` returns the hour in the argument as a short integer value in the range of `0 - 23`.

Example

```
SELECT *
     FROM  arrivals
     WHERE HOUR (in_time) < 12 ;
```

NOTES:-

- The argument to the function must be of type `TIME`.
- The argument must be specified in the format `hh:mi:ss`.
- The result is of type `SHORT`.
- If the argument expression evaluates to null, the result is `null`.

IFNULL function (ODBC compatible)

```
IFNULL( expr, value)
```

The scalar function `IFNULL` returns `value` if `expr` is `null`. If `expr` is not `null`; `IFNULL` returns `expr`.

Example

```
select c1, ifnull(c1, 9999) from temp order by c1;
c1      ifnull(c1,9999)
        9999
        9999
        9999
1       1
3       3
```

NOTE:- The data type of `value` must be compatible with the data type of `expr`.

INITCAP function (extension)

```
INITCAP ( char_expression )
```

The scalar function `INITCAP` returns the result of the argument character expression after converting the first character to upper case and the subsequent characters to lower case.

Example

```
SELECT INITCAP (name)
     FROM customer ;
```

NOTE:- The argument to the function must be of type `CHARACTER`.

- The result is of type `CHARACTER`.
- If the argument expression evaluates to null, the result is `null`.

INSERT function (ODBC compatible)

```
INSERT(string_exp1,start,length,string_exp2)
```

The scalar function `INSERT` returns a character string where length characters have been deleted from `string_exp1` beginning at `start` and `string_exp2` has been inserted into `string_exp1`, beginning at `start`.

Example

```
SELECT INSERT(name,2,4,'xx')
FROM customer
WHERE name = 'Goldman';
INSERT(NAME,2,4,XX)

Gxxan
1 record selected
```

NOTES:-

- The `string_exp` can be of the type fixed length or variable length `CHARACTER`.
- The start and length can be of the type `INTEGER, SMALLINT, TINYINT` or `BIGINT`.
- The result string is of the type `string_exp1`.
- If any of the argument expression evaluates to a null, the result would be a `null`.
- If `start` is negative or zero, the result string evaluates to a `null`.
- If `length` is negative, the result evaluates to a `null`.

INSTR function (extension)

```
INSTR ( char_expression, char_expression
          [, start_position [, occurrence]])
```

The scalar function `INSTR` searches for the character string corresponding to the second argument in the character string corresponding to the first argument starting at `start_position`. If `occurrence` is specified, then `INSTR` searches for the nth occurrence where `n` is the value of the fourth argument.

The position (with respect to the start of string corresponding to the first argument) is returned if a search is successful. Zero is returned if no match can be found.

Example

```
SELECT cust_no, name
     FROM  customer
     WHERE INSTR (LOWER (addr), 'heritage') > 0 ;
```

NOTES:-

- The first and second arguments must be of type `CHARACTER`.
- The third and fourth arguments, if specified, must be of type `INTEGER`.

- The values for specifying position in a character string starts from 1. That is, the very first character in a string is at position 1, the second character is at position 2 and so on.
- If the third argument is not specified, a default value of `1` is assumed.
- If the fourth argument is not specified, a default value of `1` is assumed.
- The result is of type `INTEGER`.
- If any of the argument expressions evaluates to null, the result is `null`.

LAST_DAY function (extension)

```
LAST_DAY ( date_expression )
```

The scalar function `LAST_DAY` returns the date corresponding to the last day of the month containing the argument date.

Example

```
SELECT *
     FROM  orders
     WHERE LAST_DAY (order_date) + 1 = '08/01/1991' ;
```

NOTES:-

- The argument to the function must be of type `DATE`.
- The result is of type `DATE`.
- If the argument expression evaluates to null, the result is `null`.

LCASE function (ODBC compatible)

```
LCASE ( char_expression )
```

The scalar function `LCASE` returns the result of the argument character expression after converting all the characters to lower case. `LCASE` is the same as `LOWER` but provides ODBC-compatible syntax.

Example

```
SELECT *
     FROM  customer
     WHERE LCASE (name) = 'smith' ;
```

NOTES:-

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

LEAST function (extension)

```
LEAST ( expression, expression, ... )
```

The scalar function LEAST returns the lowest value among the values of the given expressions.

Example

```
SELECT cust_no, name,
     LEAST (ADD_MONTHS (start_date, 10), SYSDATE)
     FROM  customer ;
```

NOTES:-

- The first argument to the function can be of any type. The types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluates to null, the result is null.

LEFT function (ODBC compatible)

```
LEFT ( string_exp, count )
```

The scalar function LEFT returns the leftmost count of characters of string_exp.

Example

```
SELECT LEFT(name,4)
FROM   customer
WHERE  name = 'Goldman';

LEFT(NAME,4)

Gold
```

```
1 record selected
```

NOTES:-

- The `string_exp` can be of the type fixed or variable length `CHARACTER`.
- The count can be of the type `INTEGER, SMALLINT, BIGINT, TINYINT`.
- If any of the arguments of the expression evaluates to a null, the result would be `null`.
- If the count is negative, the result evaluates to a `null`.

LENGTH function (ODBC compatible)

```
LENGTH ( char_expression )
```

The scalar function `LENGTH` returns the string length of the value of the given character expression.

Example

```
SELECT name 'LONG NAME'
     FROM  customer
     WHERE LENGTH (name) > 5 ;
```

NOTES:-

- The argument to the function must be of type `CHARACTER` or `VARCHAR`.
- The result is of type `INTEGER`.
- If the argument expression evaluates to null, the result is `null`.

LOCATE function (ODBC compatible)

```
LOCATE( char-expr1 , char-expr2, [start-position] )
```

The scalar function `LOCATE` returns the location of the first occurrence of `char-expr1` in `char-expr2`. If the function includes the optional integer argument start-position, `LOCATE` begins searching `char-expr2` at that position. If the function omits the start-position argument, `LOCATE` begins its search at the beginning of `char-expr2`.

`LOCATE` denotes the first character position of a character expression as `1`. If the search fails, `LOCATE` returns `0`. If either character expression is null, `LOCATE` returns a `null` value.

Example

The following example uses two string literals as character expressions. LOCATE returns a value of 6:

```
SELECT LOCATE('this', 'test this test', 1) FROM TEST;
LOCATE(THIS,
------------
           6
1 record selected
```

LOG10 function (ODBC compatible)

```
LOG10 ( expression )
```

The scalar function LOG10 returns the base 10 logarithm of expression.

Example

```
SELECT LOG10 (100) 'Log base 10 of 100'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTE:- expression must evaluate to an approximate numeric data type.

LOWER function (SQL-92 compatible)

```
LOWER ( char_expression )
```

The scalar function LOWER returns the result of the argument character expression after converting all the characters to lower case.

Example

```
SELECT *
     FROM  customer
     WHERE LOWER (name) = 'smith' ;
```

NOTES:-

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

LPAD function (extension)

```
LPAD ( char_expression, length [, pad_expression] )
```

The scalar function `LPAD` pads the character string corresponding to the first argument on the left with the character string corresponding to the third argument so that after the padding, the length of the result is `length`.

Example

```
SELECT LPAD (name, 30)
     FROM customer ;
SELECT LPAD (name, 30, '.')
       FROM customer ;
```

NOTES:-

- The first argument to the function must be of type `CHARACTER`.
- The second argument to the function must be of type `INTEGER`.
- The third argument, if specified, must be of type `CHARACTER`.
  - If the third argument is not specified, the default value is a string of length `1` containing one blank.
  - If `L1` is the length of the first argument and `L2` is the value of the second argument, then:
  - If `L1` is less than `L2`, the number of characters padded is equal to `L2 - L1`.
  - If `L1` is equal to `L2`, no characters are padded and the result string is the same as the first argument.
  - If `L1` is greater than `L2`, the result string is equal to the first argument truncated to first `L2` characters.
- The result is of type `CHARACTER`.
- If the argument expression evaluates to null, the result is `null`.

LTRIM function (ODBC compatible)

```
LTRIM ( char_expression [ , char_set ] )
```

The scalar function `LTRIM` removes all the leading characters in `char_expression`, that are present in `char_set` and returns the resultant string. Thus, the first character in the result is guaranteed to be not in `char_set`. If the `char_set` argument is omitted, the function removes the leading and trailing blanks from `char_expression`.

Example

```
SELECT name, LTRIM (addr, ' ')
     FROM  customer ;
```

NOTES:-

- The first argument to the function must be of type `CHARACTER`.
- The second argument to the function must be of type `CHARACTER`.
- The result is of type `CHARACTER`.
- If the argument expression evaluates to null, the result is `null`.

MINUTE function (ODBC compatible)

```
MINUTE ( time_expression )
```

The scalar function `MINUTE` returns the minute value in the argument as a short integer in the range of `0 - 59`.

Example

```
SELECT *
     FROM  arrivals
     WHERE MINUTE (in_time) > 10 ;
```

NOTES:-

- The argument to the function must be of type `TIME`.
- The argument must be specified in the format `HH:MI:SS`.
- The result is of type `SHORT`.
- If the argument expression evaluates to null, the result is `null`.

MOD function (ODBC compatible)

```
MOD ( expression1, expression2 )
```

The scalar function `MOD` returns the remainder of `expression1` divided by `expression2`.

Example

```
SELECT MOD (11, 4) 'Modulus'
     FROM SYSTPE.SYSCALCTABLE;
```

NOTES:-

- Both `expression1` and `expression2` must evaluate to exact numeric data types.
- If `expression2` evaluates to zero, `MOD` returns `zero`.

MONTHNAME function (ODBC compatible)

`MONTHNAME ( date_expression )`

Returns a character string containing the name of the month (for example, January, through December) for the month portion of `date_expression`. Argument `date_expression` can be name of a column, the result of another scalar function, or a date or timestamp literal.

Example

```
SELECT *
FROM orders
WHERE order_no =346 and MONTHNAME(order_date)='JUNE';
ORDER_NO   ORDER_DATE   REFERENCE   CUST_NO
346        06/01/1991   87/rd       10002
   1 record selected
```

MONTH function (ODBC compatible)

`MONTH ( time_expression )`

The scalar function `MONTH` returns the month in the year specified by the argument as a short integer value in the range of `1 - 12`.

Example

```
SELECT *
     FROM  orders
     WHERE MONTH (order_date) = 6 ;
```

NOTES:-

- The argument to the function must be of type `DATE`.

- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

MONTHS_BETWEEN function (extension)

```
MONTHS_BETWEEN ( date_expression, date_expression )
```

The scalar function MONTHS_BETWEEN computes the number of months between two date values corresponding to the first and second arguments.

Example

```
SELECT MONTHS_BETWEEN (SYSDATE, order_date)
     FROM  orders
     WHERE order_no = 1002 ;
```

NOTES:-

- The first and the second arguments to the function must be of type DATE.
- The result is of type INTEGER.
- The result is negative if the date corresponding to the second argument is greater than that corresponding to the first argument.
- If any of the arguments expression evaluates to null, the result is null.

NEXT_DAY function (extension)

```
NEXT_DAY ( date_expression, day_of_week )
```

The scalar function NEXT_DAY returns the minimum date that is greater than the date corresponding to the first argument for which the day of the week is same as that specified by the second argument.

Example

```
SELECT NEXT_DAY (order_date, 'MONDAY')
     FROM  orders ;
```

NOTES:-

- The first argument to the function must be of type DATE.

- The second argument to the function must be of type CHARACTER. The result of the second argument must be a valid day of week ('SUNDAY', 'MONDAY' etc.)
- The result is of type DATE.
- If any of the argument expressions evaluates to null, the result is null.

NOW function (ODBC compatible)

```
NOW ( )
```

NOW returns the current date and time as a TIMESTAMP value. This function takes no arguments.

NULLIF (SQL-92 compatible)

```
NULLIF ( expression1, expression2 )
```

The NULLIF scalar function is a type of conditional expression.

For a summary of all the conditional expressions,please refer “Conditional Expressions” on page 173 in "Chapter 5 - Versant ReVind Common Language Elements".

The NULLIF scalar function returns a null value for expression1 if it is equal to expression2. It's useful for converting values to null from applications that use some other representation for missing or unknown data.

NOTES:-

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.
- The NULLIF expression is shorthand notation for a common case that can also be represented in a CASE expression, as follows:

  ```
  CASE
     WHEN expression1 = expression2 THEN NULL
     ELSE expression1
  ```

Example

This example uses the NULLIF scalar function to insert a null value into an address column if the host-language variable contains a single space character.

```
INSERT INTO employee (add1) VALUES (NULLIF (:address1, ' '));
```

NVL function (extension)

```
NVL ( expression, expression )
```

The scalar function `NVL` returns the value of the first expression if the first expression value is not null. If the first expression value is null, the value of the second expression is returned.

The `NVL` function is not ODBC compatible. Use the `IFNULL` function for ODBC-compatible syntax.

Example

```
SELECT salary + NVL (comm, 0) 'TOTAL SALARY'
     FROM  employee ;
```

NOTES:-

- The first argument to the function can be of any type.
- The type of the second argument must be compatible with that of the first argument.
- The type of the result is the same as the first argument.

OBJECT_ID function (extension)

```
OBJECT_ID ('table_name')
```

The scalar function `OBJECT_ID` returns the value of the id column in the systpe.systables, plus one. This function provides compatibility with the Sybase SQL Server function `object_id`.

Arguments are:

`table_name`

The name of the table for which `OBJECT_ID` returns an identification value.

Example

```
select id, object_id(tbl), tbl from vsqldba.systables
   1  where owner = 'systpe';
         ID OBJECT_ID(TB TBL
         -- ------------ ---
          0            1 systblspaces
          1            2 systables
          2            3 syscolumns
          3            4 sysindexes
          4            5 systsfiles
          5            6 syslogfiles
          6            7 sysdbbackup
          7            8 syslogbackup
          8            9 sysdbsyncpt
          9           10 sysdbsuuid
         10           11 syssyssvr
         11           12 sysusrsvr
.
.
```

PI function (ODBC compatible)

`PI ( )`

The scalar function `PI` returns the constant value of pi as a floating point value.

Example

```
SELECT PI ()
     FROM VSQLDBA.SYSCALCTABLE;
```

POWER function (ODBC compatible)

`POWER ( expression1 , expression2 )`

The scalar function `POWER` returns `expression1` raised to the power of `expression2`.

Example

```
SELECT POWER ( 3 , 2) '3 raised to the 2nd power'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTES.

- `expression1` must evaluate to a numeric data type.
- `expression2` must evaluate to an exact numeric data type.

PREFIX function (extension)

`PREFIX(char_expression, start_position, char_expression)`

The scalar function `PREFIX` returns the substring of a character string starting from the position specified by start position, and ending before the specified character.

Arguments are:

`char_expression`

An expression that evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to null, `PREFIX` returns `null`.

`start_position`

An expression that evaluates to an integer value. `PREFIX` searches the string specified in the first argument starting at that position. A value of `1` indicates the first character of the string.

`char_expression`

An expression that evaluates to a single character. `PREFIX` returns the substring that ends before that character. If `PREFIX` does not find the character, it returns the substring beginning with `start_position,` to the end of the string. If the expression evaluates to more than one character, `PREFIX` ignores all but the first character.

Example

```
SELECT C1, C2, PREFIX(C1, 1, '.') FROM T1;
C1           C2  PREFIX(C1,1,.
--           --  -------------
test.pref    .   test
pref.test    s   pref
2 records selected
SELECT C1, C2, PREFIX(C1, 1, C2) FROM T1;
C1           C2  PREFIX(C1,1,C
--           --  -------------
test.pref    .   test
pref.test    s   pref.te
2 records selected
SELECT C1, C2, PREFIX(C1, 1, 'Q') FROM T1;
C1           C2  PREFIX(C1,1,Q
--           --  -------------
test.pref    .   test.pref
pref.test    s   pref.test
2 records selected
```

QUARTER function (ODBC compatible)

```
QUARTER ( time_expression )
```

The scalar function `QUARTER` returns the quarter in the year specified by the argument as a short integer value in the range of `1 - 4`.

Example

```
SELECT *
     FROM  orders
     WHERE QUARTER (order_date) = 3 ;
```

NOTES:-

- The argument to the function must be of type `DATE`.
- The argument must be specified in the format `MM/DD/YYYY`.

- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

RADIANS function (ODBC compatible)

```
RADIANS ( expression )
```

The scalar function RADIANS returns the number of radians in an angle specified in degrees by expression.

Example

```
SELECT RADIANS(180) 'Radians in 180 degrees'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTES:-

- expression specifies an angle in degrees.
- expression must evaluate to a numeric data type.

RAND function (ODBC compatible)

```
RAND ( [ expression ] )
```

The scalar function RAND returns a randomly-generated number, using expression as an optional seed value.

Example

```
SELECT RAND(3) 'Random number using 3 as seed value'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTE:- expression must evaluate to an exact numeric data type.

REPLACE function (ODBC compatible)

```
REPLACE ( string_exp1,string_exp2,string_exp3 )
```

The scalar function REPLACE replaces all occurrences of string_exp2 in string_exp1 with string_exp3.

Example

```
SELECT REPLACE ( name,'mi','moo' )
FROM customer
WHERE name = 'Smith';

REPLACE(NAME,MI,MOO)

Smooth
1 record selected
```

NOTES:-

- `string_exp` can be of the type fixed or variable length `CHARACTER`.
- If any of the arguments of the expression evaluates to null, the result is `null`.
- If the replacement string is not found in the search string, it returns the original string.

RIGHT function (ODBC compatible)

`RIGHT ( string_exp, count )`

The scalar function `RIGHT` returns the rightmost count of characters of `string_exp`.

Example

```
SELECT RIGHT(fld1,6)
FROM   test100
WHERE  fld1 = 'Afghanistan';
RIGHT(FLD1,6)

nistan
1 record selected
```

NOTES:-

- The `string_exp` can be of the type fixed or variable length `CHARACTER`.
- The count can be of the type `INTEGER, SMALLINT, BIGINT, TINYINT`.
- If any of the arguments of the expression evaluates to a null, the result would be `null`.
- If the count is negative, the result evaluates to a `null`.

REPEAT function (ODBC compatible)

```
REPEAT ( string_exp,count )
```

The scalar function **REPEAT** returns a character string composed of `string_exp` repeated `count` times.

Example

```
SELECT REPEAT(fld1,3)
FROM   test100
WHERE  fld1 = 'Afghanistan'

Results

REPEAT(FLD1,3)

AfghanistanAfghanistanAfghanistan

1 record selected
```

NOTES:-

- The string exp. can be of the type fixed length or variable length `CHARACTER`.
- The count can be of the type `INTEGER, SMALLINT, BIGINT, TINYINT`.
- If any of the arguments of the expression evaluates to a null, the result would be `null`.
- If the count is negative or zero, the result evaluates to a `null`.

ROWID (extension)

```
ROWID
```

`ROWID` returns the row identifier of the current row in a table. This function takes no arguments. The `ROWID` of a row is determined when the row is inserted into the table. Once assigned, the `ROWID` remains the same for the row until the row is deleted. At any given time, each row in a table is uniquely identified by its `ROWID`.

The format of the row identifier returned by this function varies between storage managers.

Selecting a row in a table using its `ROWID` is the most efficient way of selecting the row.

Example

```
SELECT *
FROM customers
WHERE ROWID = '10';
```

ROWIDTOCHAR (extension)

```
ROWIDTOCHAR ( expression )
```

The scalar function ROWIDTOCHAR returns the character form of a ROWID contained in the input argument. The representation of a row identifier depends on the storage manager. The format of the argument to this function is defined by the storage manager. See the documentation for your storage manager for details.

This function works only on local tables.

The following example uses ROWIDTOCHAR to convert a row identifier from its internal representation to a character string. This example is specific to the Versant storage manager's representation of a row identifier:

```
SELECT cust_no,
   SUBSTR (ROWIDTOCHAR (ROWID),  1, 8) 'PAGE NUMBER',
   SUBSTR (ROWIDTOCHAR (ROWID), 10, 4) 'LINE NUMBER',
   SUBSTR (ROWIDTOCHAR (ROWID), 15, 4) 'TABLE SPACE NUMBER'
   FROM customer ;
```

NOTES:-

- The argument to the function must be a ROWID, as defined by the storage manager.
- The result is of CHARACTER type.
- If the argument expression evaluates to null, the result is null.

RPAD function (extension)

```
RPAD ( char_expression, length [, pad_expression] )
```

The scalar function RPAD pads the character string corresponding to the first argument on the right with the character string corresponding to the third argument so that after the padding, the length of the result would be equal to the value of the second argument length.

Example

```
SELECT RPAD (name, 30)
     FROM customer ;

     SELECT RPAD (name, 30, '.')
     FROM customer ;
```

NOTES:-

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type INTEGER.
- The third argument, if specified, must be of type CHARACTER.
- If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If L1 is the length of the first argument and L2 is the value of the second argument, then:
  - If L1 is less than L2, the number of characters padded is equal to L2 - L1.
  - If L1 is equal to L2, no characters are padded and the result string is the same as the first argument.
  - If L1 is greater than L2, the result string is equal to the first argument truncated to first L2 characters.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

RTRIM function (ODBC compatible)

```
RTRIM ( char_expression [ , char_set ] )
```

The scalar function RTRIM removes all the trailing characters in char_expression, that are present in char_set and returns the resultant string. Thus, the last character in the result is guaranteed to be not in char_set. If the char_set argument is omitted, the function removes the leading and trailing blanks from char_expression.

Example

```
SELECT RPAD ( RTRIM (addr, ' '), 30, '.')
     FROM customer ;
```

NOTES:-

- The first argument to the function must be of type CHARACTER.

- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

SECOND function (ODBC compatible)

```
SECOND ( time_expression )
```

The scalar function SECOND returns the seconds in the argument as a short integer value in the range of 0 - 59.

Example

```
SELECT *
     FROM  arrivals
     WHERE SECOND (in_time) <= 40 ;
```

NOTES:-

- The argument to the function must be of type TIME.
- The argument must be specified in the format HH:MI:SS.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

SIGN function (ODBC compatible)

```
SIGN ( expression )
```

The scalar function SIGN returns 1 if expression is positive, -1 if expression is negative, or zero if it is zero.

Example

```
SELECT SIGN(-14) 'Sign'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTE:- The expression parameter must evaluate to a numeric data type.

SIN function (ODBC compatible)

```
SIN ( expression )
```

The scalar function `SIN` returns the sine of `expression`.

Example

```
select sin(45 * pi()/180) 'Sine of 45 degrees' from syscalctable;
SINE OF 45 DEGRE
----------------
0.707106781186547
1 record selected
```

NOTE:- `SIN` takes an angle (`expression`) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

- `expression` specifies an angle in radians
- `expression` must evaluate to an approximate numeric data type.

To convert degrees to radians, multiply degrees by `Pi/180`. To convert radians to degrees, multiply radians by `180/Pi`.

SOUNDEX function (ODBC compatible)

```
SOUNDEX ( string_exp )
```

The scalar function `SOUNDEX` returns a four-character soundex code for character strings that are composed of a contiguous sequence of valid single- or double-byte roman letters.

Example

```
SELECT SOUNDEX('Roberts')
FROM vsqldba.syscalctable;
```

SPACE function (ODBC compatible)

```
SPACE ( count )
```

The scalar function `SPACE` returns a character string consisting of `count` spaces.

Example

```
SELECT CONCAT(SPACE(3), name)
FROM customer
WHERE name = 'Roberts';

CONCAT (   ,NAME)

   Roberts
1 record selected
```

Example

- The `count` argument can be of type `INTEGER, SMALLINT, BIGINT, or TINYINT`.
- If `count` is null, the result is `null`.
- If `count` is negative, the result is `null`.

SQRT function (ODBC compatible)

```
SQRT ( expression )
```

The scalar function `SQRT` returns the square root of `expression`.

Example

```
SELECT SQRT(28) 'square root of 28'
     FROM VSQLDBA.SYSCALCTABLE;
```

NOTES:-

- The value of `expression` must be positive.
- `expression` must evaluate to an approximate numeric data type.

SUBSTR function (extension)

```
SUBSTR ( char_expression, start_position [, length ] )
```

The scalar function `SUBSTR` returns the substring of the character string corresponding to the first argument starting at `start_position` and `length` characters long. If the third argument `length` is not specified, substring starting at `start_position` up to the end of `char_expression` is returned.

Example

```
SELECT name, '(', SUBSTR (phone, 1, 3) , ')',
               SUBSTR (phone, 4, 3), '-',
               SUBSTR (phone, 7, 4)
     FROM customer ;
```

NOTES:-

- The first argument must be of type `CHARACTER`.
- The second argument must be of type `INTEGER`.
- The third argument, if specified, must be of type `INTEGER`.
- The values for specifying position in the character string start from 1: The very first character in a string is at position 1, the second character is at position 2 and so on.
- The result is of type `CHARACTER`.
- If any of the argument expressions evaluates to null, the result is `null`.

SUBSTRING function (ODBC compatible)

```
SUBSTRING ( char_expression, start_position [, length ] )
```

The scalar function `SUBSTRING` returns the substring of the character string corresponding to the first argument starting at `start_position` and `length` characters long. If the third argument `length` is not specified, substring starting at `start_position` up to the end of `char_expression` is returned. `SUBSTRING` is identical to **SUBSTR** but provides ODBC-compatible syntax.

Example

```
SELECT name, '(', SUBSTRING (phone, 1, 3) , ')',
               SUBSTRING (phone, 4, 3), '-',
               SUBSTRING (phone, 7, 4)
     FROM customer ;
```

NOTES:-

- The first argument must be of type `CHARACTER`.
- The second argument must be of type `INTEGER`.
- The third argument, if specified, must be of type `INTEGER`.

- The values for specifying position in the character string start from 1: The very first character in a string is at position 1, the second character is at position 2 and so on.
- The result is of type `CHARACTER`.
- If any of the argument expressions evaluates to null, the result is `null`.

SUFFIX function (extension)

```
SUFFIX(char_expression, start_position, char_expression)
```

The scalar function `SUFFIX` returns the substring of a character string starting after the position specified by `start_position` and the second `char_expression`, to the end of the string.

Arguments are:

`char_expression`

An expression that evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to null, `SUFFIX` returns `null`.

`start_position`

An expression that evaluates to an integer value. `SUFFIX` searches the string specified in the first argument starting at that position. A value of `1` indicates the first character of the string.

`char_expression`

An expression that evaluates to a single character. `SUFFIX` returns the substring that begins with that character. If `SUFFIX` does not find the character after start_position, it returns `null`. If the expression evaluates to more than one character, `SUFFIX` ignores all but the first character.

Example

```
SELECT C1, C2, SUFFIX(C1, 6, '.') FROM T1;
C1           C2  SUFFIX(C1,6,.
--           --  -------------
test.pref    .
pref.test    s
2 records selected
SELECT C1, C2, SUFFIX(C1, 1, C2) FROM T1;
C1           C2  SUFFIX(C1,1,C
--           --  -------------
test.pref    . pref
pref.test    s   t
2 records selected
SELECT C1, C2, SUFFIX(C1, 6, '.') FROM T1;
C1           C2  SUFFIX(C1,6,.
--           --  -------------
test.pref    .
pref.test    s
2 records selected
```

SUSER_NAME function (extension)

```
SUSER_NAME ( [user_id] )
```

The scalar function SUSER_NAME returns the user login name for the user_id specified in the input argument. If no user_id is specified, SUSER_NAME returns the name of the current user.

This function provides compatibility with the Sybase SQL Server function suser_name. It is identical to the USER_NAME function.

Data Integration Notes

When a data source for a Versant Integrator internal database is added through the ODBC Administrator utility, it requires a value for a User ID field. This function returns the string specified in that field for the currently open data source for Versant Integrator.

Example

```
select suser_name() from systpe.syscalctable;
SUSER_NAME
----------
searle
1 record selected
 select suser_name(104) from vsqldba.syscalctable;
SUSER_NAME(104)
---------------
dbp
1 record selected
  select id, tbl, owner from vsqldba.systables
   1  where owner = suser_name();
          ID TBL                                  OWNER
          -- ---                                  -----
          41 test                                 searle
          42 t2                                   searle
          43 t1                                   searle

3 records selected
```

SYSDATE function (extension)

`SYSDATE [ ( ) ]`

`SYSDATE` returns the current date as a `DATE` value. This function takes no arguments, and the trailing parentheses are optional.

SQL statements can refer to `SYSDATE` anywhere they can refer to a `DATE` expression.

Example

```
INSERT INTO objects (object_owner, object_id, create_date)
           VALUES (USER, 1001, SYSDATE) ;
```

SYSTIME function (extension)

```
SYSTIME [ ( ) ]
```

SYSTIME returns the current time as a TIME value. This function takes no arguments, and the trailing parentheses are optional.

SQL statements can refer to SYSTIME anywhere they can refer to a TIME expression.

Example:

```
INSERT INTO objects (object_owner, object_id, create_time)
           VALUES (USER, 1001, SYSTIME) ;
```

SYSTIMESTAMP function (extension)

```
SYSTIMESTAMP [ ( ) ]
```

SYSTIMESTAMP returns the current date and time as a TIMESTAMP value. This function takes no arguments, and the trailing parentheses are optional.

The following SQL example shows the different formats for SYSDATE, SYSTIME, and SYSTIMESTAMP:

```
SELECT SYSDATE FROM test;
SYSDATE
-------
09/13/1994
1 record selected
SELECT SYSTIME FROM test;
SYSTIME
-------
14:44:07:000
1 record selected
SELECT SYSTIMESTAMP FROM test;
SYSTIMESTAMP
------------
1994-09-13 14:44:15:000
1 record selected
```

TAN function (ODBC compatible)

```
TAN ( expression )
```

The scalar function `TAN` returns the tangent of `expression`.

Example

```
select tan(45 * pi()/180) 'Tangent of 45 degrees' from
   vsqldba.syscalctable;
TANGENT OF 45 DE
----------------
1.000000000000000
1 record selected
```

NOTES:- `TAN` takes an angle (`expression`) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

- `expression` specifies an angle in radians.
- `expression` must evaluate to an approximate numeric data type.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

TO_CHAR function (extension)

```
TO_CHAR ( expression [ , format_string ] )
```

The scalar function `TO_CHAR` converts the given expression to character form and returns the result. The primary use for `TO_CHAR` is to format the output of date-time expressions through the `format_string` argument.

Arguments are:

`expression`

An expression to be converted to character form.

To use the `format_string` argument, `expression` must evaluate to a date or time value.

`format_string`

A date-time format string that specifies the format of the output.

For details on format strings, See also “Date Format Strings” on page 181, and “Time Format Strings” on page 183 in "Chapter 5 - Versant ReVind Common Language Elements".

SQL ignores the format string if the `expression` argument does not evaluate to a date or time.

Example

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
---------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

NOTES:-

- The first argument to the function can be of any type.
- The second argument, if specified, must be of type `CHARACTER`.
- The result is of type `CHARACTER`.
- The `format` argument can be used only when the type of the first argument is `DATE`.
- If any of the argument expressions evaluates to null, the result is `null`.

TO_DATE function (extension)

```
TO_DATE ( date_lit )
```

The scalar function `TO_DATE` converts the given date literal to a date value.

Example

```
SELECT *
     FROM  orders
     WHERE order_date <= TO_DATE ('12/31/1991') ;
```

NOTES:-

- The result is of type DATE.
- Supply the date literal in any valid format.

For valid formats see “Date Literals” on page 176 in "Chapter 5 - Versant ReVind Common Language Elements".

TO_NUMBER function (extension)

```
TO_NUMBER ( char_expression )
```

The scalar function TO_NUMBER converts the given character expression to a number value.

Example

```
SELECT *
     FROM  customer
     WHERE TO_NUMBER (SUBSTR (phone, 1, 3)) = 603 ;
```

NOTES:-

- The argument to the function must be of type CHARACTER.
- The result is of type NUMERIC.
- If any of the argument expressions evaluates to null, the result is null.

TO_TIME function (extension)

```
TO_TIME ( time_lit )
```

The scalar function TO_TIME converts the given time literal to a time value.

Example

```
SELECT *
     FROM  orders
     WHERE order_date < TO_DATE ('05/15/1991')
           AND order_time < TO_TIME ('12:00:00') ;
```

NOTES:-

- The result is of type `TIME`.
- Supply the time literal in any valid format.

For valid formats, please refer “Time Literals” on page 178 in "Chapter 5 - Versant ReVind Common Language Elements".

TO_TIMESTAMP function (extension)

```
TO_TIMESTAMP ( timestamp_lit )
```

The scalar function `TO_TIMESTAMP` converts the given timestamp literal to a timestamp value.

Example

```
SELECT * FROM DTEST
WHERE C3 = TO_TIMESTAMP('4/18/95 10:41:19')
```

NOTES:-

- The result is of type `TIME`.
- Supply the timestamp literal in any valid format.

For valid formats, please refer “Timestamp Literals” on page 179 in "Chapter 5 - Versant ReVind Common Language Elements".

TRANSLATE function (SQL-92 compatible)

```
TRANSLATE ( char_expression USING translation_name )
```

The SQL-92 compatible `TRANSLATE` function translates the characters in `char_expression` to another character set. The details of the translation are dependent on the underlying storage system. See the documentation for the underlying storage system for which translations it supports, if any.

Example

This example translates customers' street addresses from ASCII to EBCDIC. It presumes the underlying storage system supports a translation called `ASCII_EBCDIC`.

```
SELECT name, TRANSLATE (street USING ASCII_EBCDIC)
     FROM customer ;
```

NOTE:- The character set associated with a character expression defines, among other characteristics, which set of characters (repertoire) it can contain. The TRANSLATE function provides a way to translate the characters from one repertoire another.

For notes on character sets, please refer “Specifying the Character Set for Character Data Types” on page 140 in "Chapter 5 - Versant ReVind Common Language Elements".

Support for specific translation_names is completely dependent on the underlying storage system. When TRANSLATE refers to a translation_name that is not supported by the underlying storage system, SQL generates an error.

TRANSLATE function (extension)

```
TRANSLATE ( char_expression, from_set, to_set )
```

The scalar function TRANSLATE translates each character in char_expression that is in from_set to the corresponding character in to_set. The translated character string is returned as the result. This function is similar to the Oracle TRANSLATE function.

Example

This example substitutes underscores for spaces in customer names.

```
SELECT TRANSLATE (customer_name, ' ', '_')
      "TRANSLATE Example" from customers;
TRANSLATE EXAMPLE
-----------------
Sports_Cars_Inc._____________________________
Mighty_Bulldozer_Inc._____________________________
Ship_Shapers_Inc._____________________________
Tower_Construction_Inc._____________________________
Chemical_Construction_Inc._____________________________
Aerospace_Enterprises_Inc._____________________________
Medical_Enterprises_Inc._____________________________
Rail_Builders_Inc._____________________________
Luxury_Cars_Inc._____________________________
Office_Furniture_Inc._____________________________
10 records selected
```

NOTES:-

- `char_expression, from_set`, and `to_set` can be any character expression.
- For each character in `char_expression`, `TRANSLATE` checks for the same character in `from_set`:
- If it is in `from_set, TRANSLATE` translates it to the corresponding character in `to_set` (if the character is the `nth` character in `from_set`, the `nth` character in `to_set`).
- If the character is not in `from_set TRANSLATE` does not change it.
- If `from_set` is longer than `to_set`, `TRANSLATE` does not change trailing characters in `from_set` that do not have a corresponding character in `to_set`.
- If either `from_set` or `to_set` is null, `TRANSLATE` does nothing.

UCASE function (ODBC compatible)

```
UCASE ( char_expression )
```

The scalar function `UCASE` returns the result of the argument character expression after converting all the characters to upper case. `UCASE` is identical to `UPPER`, but provides ODBC-compatible syntax.

Example

```
SELECT *
     FROM  customer
     WHERE UCASE (name) = 'SMITH' ;
```

NOTES:-

- The argument to the function must be of type `CHARACTER`.
- The result is of type `CHARACTER`.
- If the argument expression evaluates to null, the result is `null`.

UID function (extension)

`UID` returns an integer identifier for the user of the current transaction, as determined by the host operating system.

The value contained in this register is of `INTEGER` type. The host representation is of long integer type. SQL statements can refer to `UID` anywhere they can refer to an integer expression.

Example

```
INSERT INTO objects (owner_id, object_id)
           VALUES (UID, 1001) ;

SELECT *
    FROM objects
    WHERE owner_id = UID ;
```

UPPER function (SQL-92 compatible)

```
UPPER ( char_expression )
```

The scalar function UPPER returns the result of the argument character expression after converting all the characters to upper case.

Example

```
SELECT *
     FROM  customer
     WHERE UPPER (name) = 'SMITH' ;
```

NOTES:-

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

USER function (ODBC compatible)

```
USER [ ( ) ]
```

USER returns a character-string identifier for the user of the current transaction, as determined by the host operating system. This function takes no arguments, and the trailing parentheses are optional.

SQL statements can refer to USER anywhere they can refer to a character string expression.

Example

```
INSERT INTO objects (object_owner, object_id)
           VALUES (USER, 1001) ;

    SELECT *
    FROM objects
    WHERE object_owner = USER ;
```

USER_NAME function (extension)

```
USER_NAME ( [user_id] )
```

The scalar function USER_NAME returns the user login name for the user_id specified in the input argument. If no user_id is specified, USER_NAME returns the name of the current user.

The scalar function USER_NAME is identical to SUSER_NAME.

WEEK function (ODBC compatible)

```
WEEK ( time_expression )
```

The scalar function WEEK returns the week of the year as a short integer value in the range of 1 - 53.

Example

```
SELECT *
     FROM  orders
     WHERE WEEK (order_date) = 5 ;
```

NOTES:-

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

YEAR function (ODBC compatible)

```
YEAR ( date_expression )
```

The scalar function YEAR returns the year as a short integer value in the range of `0 - 9999`.

Example

```
SELECT *
     FROM  orders
     WHERE YEAR (order_date) = 1992 ;
```

NOTES:-

- The argument to the function must be of type `DATE`.
- The argument must be specified in the format `MM/DD/YYYY`.
- The result is of type `SHORT`.
- If the argument expression evaluates to null, the result is `null`.

CHAPTER 8

Java Stored Procedures

This Chapter describes the overview, advantages of stored procedures. It also explains the steps performed in creating stored procedures.

The Chapter covers the following in detail:

- Overview
- Advantages of Stored Procedures
- How Versant ReVind Interacts with Java

OVERVIEW

Stored procedures and triggers provides the ability to write Java routines that contain SQL statements and store those routines in any database over which Versant ReVind has been implemented. Tools and applications can then execute the procedures.

A **stored procedure** is a **snippet** of Java code embedded in an SQL CREATE PROCEDURE statement. The Java snippet can use all standard Java features as well as use Versant ReVind-supplied Java classes for processing any number of SQL statements.

A **trigger** is a special type of stored procedure that helps insure **referential integrity** for a database. Like stored procedures, triggers also contain Java code (embedded in a CREATE TRIGGER statement) and use Versant ReVind Java classes. However, triggers are automatically invoked ("fired") by certain SQL operations (an insert, update, or delete operation) on the trigger's target table.

ADVANTAGES OF STORED PROCEDURES

Stored procedures provide a very flexible, general mechanism to store in a database a collection of SQL statements and Java program constructs that enforce business rules and perform administrative tasks.
The ability to write stored procedures and triggers expands the flexibility and performance of applications that access a Versant ReVind environment:

- In a client/server environment, applications make only a single client/server request for the entire procedure, instead of one or more requests for each SQL statement in the stored procedure or trigger.
- Stored procedures and triggers are stored in compiled form (as well as source-code form), so execute much faster than a corresponding SQL script.
- Stored procedures can implement elaborate algorithms to enforce complex business rules. The details of the procedure implementation can change without requiring changes in an application that calls the procedure.

HOW VERSANT REVIND INTERACTS WITH JAVA

Versant ReVind's implementation of stored procedures allows use of standard Java programming constructs instead of requiring proprietary flow-control language. To do this, the **SQL engine** interacts with Java in the following ways:
When applications call a stored procedure, the SQL engine interacts with the Java **virtual machine** to execute the stored procedure and receive any results.

Following Illustrates the steps in creating a stored procedure:

Creating Stored Procedures

The Java source code that makes up the body of a stored procedure is not a complete program, but a snippet that Versant ReVind converts to a Java class when it processes a CREATE PROCEDURE statement.
Creating a stored procedure involves the following steps:

1. Some application or tool (interactive SQL, an SQL script, or an application) issues a CREATE PROCEDURE statement containing a Java snippet.
2. Versant ReVind adds code to the Java snippet to create a complete Java class and submits the combined code to the Java compiler.
3. Presuming there are no Java compilation errors, the Java compiler sends compiled **bytecode** back to Versant ReVind. If there are compilation errors, Versant ReVind passes the first error generated back to the application or tool that issued the CREATE PROCEDURE statement.
4. Versant ReVind stores both the Java source code and the bytecode form of the procedure in the underlying storage system.

Calling Stored Procedures

Once a stored procedure is created and stored in the database, any application (or other stored procedure) can execute it by calling it. You can call stored procedures from ODBC applications, JDBC applications, or directly from interactive SQL.
For instance, following shows an excerpt from an ODBC application that calls a stored procedure (order_parts) using the ODBC syntax { call procedure_name (param) }.

Example Executing a Stored Procedure Through ODBC

```
SQLUINTEGER Part_num;
SQLINTEGER  Part_numInd = 0;

// Bind the parameter.
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0
, 0,
                  &Part_num, 0, Part_numInd);

// Place the department number in Part_num.
Part_num = 318;

// Execute the statement.
SQLExecDirect(hstmt, "{call order_parts(?)}", SQL_NTS);
```

Executing a stored procedure involves the following steps:

1. The application calls the stored procedure through its native calling mechanism, for instance, used the ODBC call escape sequence.
2. Versant ReVind retrieves the compiled bytecode-form of the procedure and submits it to the Java virtual machine for execution.
3. For every SQL statement in the procedure, the Java virtual machine calls Versant ReVind.
4. Versant ReVind manages interaction with the underlying database system and execution of the SQL statements, and returns any results to the Java virtual machine.

The Java virtual machine returns results (output parameters and result sets) of the procedure to Versant ReVind, which in turn passes them to the calling application.

CHAPTER 9

Using Java Stored Procedures

This Chapter explains the usage of Java stored procedures in Versant ReVind.

The Chapter describes the following in detail:

- Stored Procedure Basics
- Stored Procedure Usage
- Using the Versant ReVind Java Classes

STORED PROCEDURE BASICS

Stored procedures extend the SQL capabilities of a database by adding control flow through Java program constructs that enforce business rules and perform administrative tasks.

Stored procedures can take advantage of the power of Java programming features. Stored procedures can:

- Receive and return input and output parameters
- Handle exceptions
- Include any number and kind of SQL statements to access the database
- Make calls to other procedures
- Used predefined and external Java classes

Functionality	Versant ReVind Java Class
Immediate (one-time) execution of SQL statements	`SQLIStatement`
Prepared (repeated) execution of SQL statements	`SQLPStatement`
Retrieval of SQL result sets	`SQLCursor`
Returning a procedure result set to the application	`DhSQLResultSet`
Exception handling for SQL statements	`DhSQLException`

What Is a Java Snippet?

The core of the stored procedure is the java_snippet. The snippet contains a sequence of Java statements. When it processes a CREATE PROCEDURE statement, VERSANT ReVind adds header and footer "wrapper" code to the Java snippet. This wrapper code:

- Declares a class with the name username_procname_SP (username is the user name of the database connection that issued the CREATE PROCEDURE statement and procname is the name supplied in the CREATE PROCEDURE statement).
- Declares a method within that class that includes the Java snippet.

When an application calls the stored procedure, the SQL engine calls the Java virtual machine to invoke the method of the username_procname_SP class.

STORED PROCEDURE USAGE

There are two parts to any stored procedure:

- The procedure specification must provide the name of the procedure and may include other optional clauses.
- The procedure body contains the Java code that executes when an application invokes the procedure.

A simple stored procedure requires only the procedure name in the specification and a statement that requires no parameters in the body, as shown in the procedure, it assumes a table called HelloWorld exists, and inserts a string into that table.

Example: Structure of a Simple Stored Procedure

```
CREATE PROCEDURE HelloWorld ()        Procedure Specification

BEGIN
     SQLIStatement Insert_HelloWorld = new SQLIStatement (
     "INSERT INTO HelloWorld(fld1) values ('Hello World!')");
     Insert_HelloWorld.execute();                          Procedure Body
END
```

From interactive SQL, you could execute the procedure as follows:

```
ISQL> CREATE TABLE helloworld (fld1 CHAR(100));
ISQL> CALL HelloWorld();
0 records returned
ISQL> SELECT * FROM helloworld;
FLD1
----
Hello World!
1 record selected
```

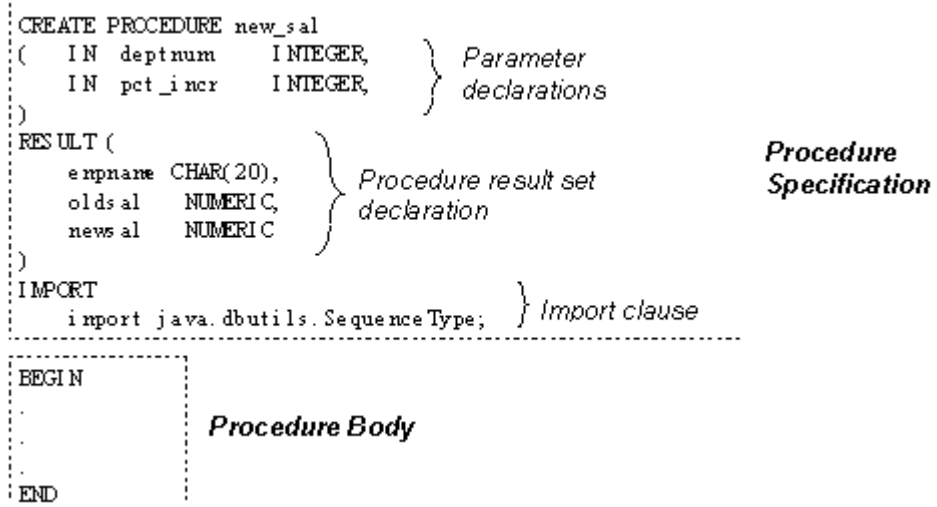
The procedure specification can also contain other clauses:

- Parameter declarations specify the name and type of parameters that the calling application will pass and receive from the procedure. Parameters can be input, output, or both.
- The procedure result set declaration details the names and types of fields in a result set the procedure generates. The result set is a set of rows that contain data generated by the procedure. If a procedure retrieves rows from a database table, for instance, it can store the rows in a result set for access by applications and other procedures.

  Note: The names specified in the result-set declaration are not used within the stored procedure body. Instead, methods of the VERSANT ReVind Java classes refer to fields in the result set by ordinal number, not by name.)
- The import clause specifies which packages the procedure needs from the Java core API. By default, the Java compiler imports the java.lang package. The IMPORT clause must list any other packages the procedure uses. (VERSANT ReVind automatically imports the packages it requires.)

Following example shows a more complex procedure specification that contains these elements:

Example: Structure of a More Complex Stored Procedure



Setting Up Your Environment to Write Stored Procedures

Before you create stored procedures, you need to have a Java development environment running on your system. Versant ReVind stored procedures support the following environments:

On UNIX and Windows: JDK™ Version 1.4

Also, make sure environment variables include the settings shown in the table below:

TABLE 1. Java-Related Environment Variables

Variable	Purpose and Setting
TPEROOT	The main Versant ReVind directory. `SET TPEROOT=d:\v6050`
CLASSPATH	Directories containing Java classes. Must at least include the Versant ReVind and main Java class directories. `SET CLASSPATH=%TPEROOT%\lib\vsql.jar;%SystemRoot%\java\lib`

JAVA_COMPILER	Must include the Java compiler `javac.exe`. It is used when compiling generated java files, required for JSP.
	`Windows - SET JAVA_COMPILER = C:\SDK-Java.20\BIN; . .`
	`Unix - UNSET JAVA_COMPILER`

Writing Stored Procedures

Use any text editor to write the `CREATE PROCEDURE` statement, and save the source code as a text file. That way, you can easily modify the source code and try again if it generates syntax or Java compilation errors.

Submit the file containing the `CREATE PROCEDURE` statement to interactive SQL as a script, as shown in example below.

Example: Submitting Scripts to Create Stored Procedures

```
$ type helloworldscript.sql
SET ECHO ON;
SET AUTOCOMMIT OFF;
CREATE PROCEDURE HelloWorld ()

BEGIN
     SQLIStatement Insert_HelloWorld = new SQLIStatement (
     "INSERT INTO HelloWorld(fld1) values ('Hello World!')");
     Insert_HelloWorld.execute();
END
commit work;
$ isql -s hello_world_script.sql example_db
SET AUTOCOMMIT OFF;
CREATE PROCEDURE HelloWorld ()

BEGIN
     SQLIStatement Insert_HelloWorld = new SQLIStatement (
     "INSERT INTO HelloWorld(fld1) values ('Hello World!')");
     Insert_HelloWorld.execute();
END
;
commit work;
$
```

The Java snippet within the `CREATE PROCEDURE` statement does not execute as a standalone program. Instead, it executes in the context of an application call to the method of the class created by the SQL engine. This characteristic has the following implications:

- It is meaningless for a snippet to declare a main method, since it will never be executed.
- If the snippet declares any classes, it must instantiate them within the snippet to invoke their methods.

If your Versant ReVind environment was built using the network model, the SQL engine redirects the standard output stream to a file, `sql_server.log`. This means method invocations such as `System.out.println` will not display messages on the screen, but instead writes them to that file.

Invoking Stored Procedures

How applications call stored procedures depends on their environment.

From ODBC

From ODBC, applications use the ODBC `call escape` sequence:

```
{ call proc_name [ ( parameter [ , … ] ) ] }
```

Use parameter markers (question marks used as placeholders) for input or output parameters to the procedure. You can also use literal values for input parameters only. VERSANT stored procedures do not support return values in the ODBC escape sequence.

Embed the escape sequence in an ODBC SQLExecDirect call to execute the procedure. Following example shows a call to a stored procedure named order_parts that passes a single input parameter using a parameter marker:

Example: Invoking a Stored Procedure from an ODBC Application

```
SQLUINTEGER Part_num;
SQLINTEGER Part_numInd = 0;
//Bind the parameter.
SQLBindParameter (hstmt, l, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0,
&Part_num = 318;
//Execute the statement.
SQLExecDirect(hstmt, "{call order_parts(?)}", SQL_NTS);
```

From JDBC

The JDBC call escape sequence is the same as in ODBC:

```
{ call proc_name [ ( parameter [ , … ] ) ] }
```

Embed the escape sequence in a JDBC CallableStatement.prepareCall method invocation. shows the JDBC code parallel to the ODBC code excerpt shown in:

Example: Invoking a Stored Procedure from a JDBC Application

```
try {
```

```
    CallableStatement statement;
    int Part_num = 318;

    // Associate the statement with the procedure call
    // (conn is a previously-instantiated connection object)
    statement = conn.prepareCall("{call order_parts(?)}");

    // Bind the parameter.
    statement.setInt(1, Part_num);

    // Execute the statement.
    statement.execute();
}
```

From Interactive SQL

Example: Invoking a Stored Procedure from Interactive SQL

```
CALL order_parts (318);
```

Debugging Stored Procedures

If there's a compiler error, the SQL engine returns the error at create time and does not create the procedure. For example, invokes an ISQL script that tries to create a procedure that fails and generates a Java compiler error.

Example: Java Compilation Error During Stored Procedure Creation

```
C:\example_scripts>isql -s type_mismatch.sql testdb
error(-20141): error in compiling the stored procedure
```

You can write messages to either of the log files with the common methods log and err, which write a character-string message to the sql_server.log and sqlerr.log files, respectively.

You can include the values of variables or return values of methods in the string using the standard Java concatenation operator (+) as shown in:

Example: Logging Messages With the log and err Methods

```
 SQLCursor select_syscalctable = new SQLCursor (
```

```
  "SELECT fld FROM systpe.syscalctable ");
     select_syscalctable.open();
     select_syscalctable.fetch();
     log ("Any records? Found returned " + select_syscalctable.found());
.
.
.
```

The log invocation in writes the following line to the sql_server.log file:

```
Any records? After fetch, found returned true
```

Transactions and Stored Procedures

Any updates done by a stored procedure become part of the transaction that called the procedure. The transaction behavior is the same as if the sequence of SQL statements in the procedure were directly executed by the calling application.

Stored procedures cannot contain COMMIT or ROLLBACK statements.

Stored Procedure Security

- For creating a stored procedure, a user needs to have RESOURCE or DBA privileges.
- The DBA privilege entitles a user to execute any stored procedure.
- The DBA privilege entitles a user to drop any stored procedure.
- The owner of a stored procedure is given EXECUTE and DROP privilege on that procedure at creation time, by default.
- The privileges on a procedure can be granted to another user or to public either by the owner of that procedure or by DBA.
- When a procedure is being executed on behalf of a user with EXECUTE privilege on that procedure, for the objects that are accessed by the procedure, the procedure owner's privileges are checked and not the user's. This enables a user to execute a procedure successfully even when the user does not have the privileges to directly access the objects that are accessed by the procedure, so long as the user has EXECUTE privilege on the procedure.

USING THE VERSANT REVIND JAVA CLASSES

This section describes how the VERSANT ReVind Java classes are used to issue and process SQL statements in Java stored procedures.

To process SQL statements in a stored procedure, you need to know whether the SQL statement generates results (in other words, if the statement is a query) or not. SELECT statements, for example, generate results: they retrieve data from one or more database tables and return the results as rows in a table.

Whether a statement generates such an SQL result set dictates which VERSANT ReVind Java classes you use to issue it:

- To issue SQL statements that do not generate results (such as INSERT, GRANT, or CREATE), use either the SQLIStatement class (for one-time execution) or the SQLPStatement class (for repeated execution).
- To issue SQL statements that generate results (SELECT and, in some cases, CALL), use the SQLCursor class to retrieve rows from a database or another procedure's result set.

In either case, if you want to return a result set to the application, use the DhSQLResultSet class to store rows of data in a procedure result set. You must use DhSQLResultSet methods to transfer data from an SQL result set to the procedure result set for the calling application to process it. You can also use DhSQLResultSet methods to store rows of data generated internally by the procedure.

In addition, VERSANT ReVind provides the DhSQLException class so procedures can process and generate Java exceptions through the try, catch, and throw constructs.

Passing Values To SQL Statements

Stored procedures need to be able to pass and receive values from SQL statements they execute. They do this through the setParam and getValue methods.

The setParam Method: Pass Input Values to SQL Statements

The setParam method sets the value of an SQL statement's parameter marker to the specified value (a literal, a procedure variable, or a procedure input parameter).

The setParam method takes two arguments:

```
setParam ( marker_num , value ) ;
```

- `marker_num` is an integer that specifies which parameter marker in the SQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).
- `value` is a literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Example: Setting SQL Statement Input Parameters: The setParam Method

```
        SQLIStatement insert_cust = new SQLIStatement (
            "INSERT INTO customer VALUES (?,?) ");

        insert_cust.setParam (1, cust_number);
        insert_cust.setParam (2, cust_name);
.
.
.
```

The getValue Method: Pass Values from SQL Result Sets to Variables

The getValue method of the SQLCursor class assigns a single value from an SQL result set (returned by an SQL query or another stored procedure) to a procedure variable or output parameter.

The format and arguments for getValue are similar to setParam:

```
getValue ( col_num , variable ) ;
```

- `col_num` is an integer that specifies which column of the result set is of interest. `getValue` retrieves the value in the currently-fetched record of the column denoted by `col_num`. (1 denotes the first column of the result set, 2 denotes the second, and so on).
- `variable` specifies the procedure variable or output parameter name that will receive the value from the result set.

Example: Using the getValue Method to Pass Values from Result Sets

```
    StringBuffer ename = new StringBuffer (20) ;
```

```
    BigDecimal esal = new BigDecimal () ;

    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
        empcursor.Fetch ();
        if (empcursor.Found ())
        {
                empcursor.getValue (1, ename);
                empcursor.getValue (2, esal);
.
.
```

In the SELECT statement in the example it was clear that the result set had two columns, `name` and `sal`. If the SELECT statement had used a wildcard in its select list (`SELECT * FROM EMP`) you have to know the structure of the EMP table in order to correctly specify the column numbers in the getValue method.

Passing Values To and From Stored Procedures: Input and Output Parameters

Applications need to pass and receive values from the stored procedures they call. They do this through input and output parameters declared in the procedure specification.

Applications can pass and receive values from stored procedures using input and output parameters declared in the stored procedure specification. When it processes the CREATE PROCEDURE statement, the SQL engine declares Java variables of the same name. This means the body of the stored procedure can refer to input and output parameters as if they were Java variables declared in the body of the stored procedure.

Procedure result sets are another way for applications to receive output values from a stored procedure. Procedure result sets provide output in a row-oriented tabular format. See section Returning a Procedure Result Set to Applications: the RESULT Clause and DhSQLResultSet

Parameter declarations include the parameter type (IN, OUT, or INOUT), the parameter name, and SQL data type.

Declare input and output parameters in the specification section of a stored procedure, as shown in:

```
CREATE PROCEDURE order_entry (
    IN  cust_name    CHAR(20),
    IN  item_num     INTEGER,
    IN  quantity     INTEGER,
    OUT status_code  INTEGER,
    INOUT order_num  INTEGER
)
.
.
.
```

When the `order_entry` stored procedures executes, the calling application passes values for the `cust_name`, `item_num`, `quantity`, and `order_num` input parameters. The body of the procedure refers to them as Java variables. Similarly, Java code in the body of `order_entry` processes and returns values in the `status_code` and `order_num` output parameters.

Implicit Data Type Conversion Between SQL and Java Types

When the SQL engine creates a stored procedure, it converts the type of any input and output parameters from the SQL data types in the procedure specification to Java wrapper types.

The Java.lang package defines classes for all the primitive Java types that "wrap" values of the corresponding primitive type in an object. The SQL engine converts the SQL data types declared for input and output parameters to one of these wrapper types as shown in the table.

You must be sure to use wrapper types when declaring procedure variables to use as arguments to the getValue, setParam, and set methods. These methods take objects as arguments and will generate compilation errors if you pass a primitive type to them.

The following example shows two stored procedures. The first tries to use a variable declared as the Java `int` primitive type as an argument to the SQLResultSet.set method, and will generate a compilation error. The second correctly declares the variable using the integer wrapper class.

Example: Using Wrapper Types as Arguments Versant ReVind Classes

```
CREATE PROCEDURE type_mismatch( )
RESULT (res_int INTEGER )
BEGIN
     // This won't work: Create a variable as (primitive) type int:
     int pvar_int = 4;

     // Transfer the value from the procedure variable to the result set.
     SQLResultSet.set(1,pvar_int); //Error!

     // Insert the row into the procedure result set.
     SQLResultSet.insert();

     // Close the SQL result set.
     select_syscalctable.close();
END
CREATE PROCEDURE type_okmatch( )
RESULT (res_int INTEGER )
BEGIN
     // Create a variable as (wrapper) type Integer:
     Integer pvar_int = new Integer(4);

     // Transfer the value from the procedure variable to the result set.
     SQLResultSet.set(1,pvar_int); //Success!

     // Insert the row into the procedure result set.
     SQLResultSet.insert();

     // Close the SQL result set.
     select_syscalctable.close();
END
```

When the SQL engine submits the Java class it creates from the stored procedure to the Java compiler, the compiler checks for data-type consistency between the converted parameters and variables you declare in the body of the stored procedure.

To avoid type mismatch errors, use the data-type mappings shown in following table, for declaring parameters and result-set fields in the procedure specification and the Java variables in the procedure body.

Mapping between SQL and Java Data Types:

SQL Type	Java Wrapper Type	SQL Type	Java Wrapper Type
CHAR	StringBuffer	REAL	Float
VARCHAR	StringBuffer	FLOAT	Double
LONGVARCHAR	StringBuffer	DOUBLE PRECISION	Double
NUMERIC	java.math.BigDecimal	BINARY	byte[]
DECIMAL	java.math.BigDecimal	VARBINARY	byte[]
MONEY	java.math.BigDecimal	LONGVARBINARY	byte[]
BIT	Boolean	DATE	java.sql.Date
TINYINT	byte[1]	TIME	java.sql.Time
SMALLINT	Integer	TIMESTAMP	java.sql.Timestamp
INTEGER	Integer		
BIGINT	Long		

Executing an SQL Statement

If an SQL statement does not generate a result set, stored procedures can execute it in either one of the two ways:

- Immediate execution, using methods of the SQLIStatement class, executes a statement once.
- Prepared execution, using methods of the SQLPStatement class, prepares a statement so you can execute it multiple times in a procedure loop.

Executable SQL Statements:

ALTER TABLE	CREATE INDEX
CREATE SYNONYM	CREATE PROCEDURE
CREATE TABLE	CREATE TRIGGER
CREATE VIEW	DELETE
DROP INDEX	DROP SYNONYM

DROP PROCEDURE	DROP TABLE
DROP TRIGGER	DROP VIEW
GRANT	INSERT
RENAME	REVOKE
UPDATE	UPDATE STATISTICS

Immediate Execution

Use immediate execution when a procedure needs to execute an SQL statement only once. Following is an example of immediate execution:

```
CREATE PROCEDURE insert_customer (
   IN  cust_number INTEGER,
   IN  cust_name   CHAR(20)
)
BEGIN
    SQLIStatement insert_cust = new SQLIStatement (
       "INSERT INTO customer VALUES (?,?) ");
    insert_cust.setParam (1, cust_number);
    insert_cust.setParam (2, cust_name);
    insert_cust.execute ();
END
```

This example inserts a row in a table. The constructor for SQLIStatement takes the SQL INSERT statement as its only argument. In this example, the statement includes two parameter markers.

Prepared Execution

Use prepared execution when you need to execute the same SQL statement repeatedly. Prepared execution avoids the overhead of creating multiple SQLIStatement objects for a single statement.

The advantage of prepared execution comes when you have the same SQL statement executed from within a loop. Instead of creating an object during each pass through the loop, prepared execution creates an object once and only passes input parameters for each execution of the statement.

Once a stored procedure creates an SQLPStatement object, it can execute it multiple times, supplying different values for each execution.

Following is an example of prepared execution:

```
     CREATE PROCEDURE prepared_insert_customer (
        IN  cust_number INTEGER,
        IN  cust_name   CHAR(20)
     )
     BEGIN
         SQLPStatement p_insert_cust = new SQLPStatement (
            "INSERT INTO customer VALUES (?,?) ");
.
.
.
         int i;
         for (i=0; i<new_custs.length; i++)
            {
            p_insert_cust.setParam (1, new_custs[i].cust_number);
            p_insert_cust.setParam (2, new_custs[i].cust_name);
            p_insert_cust.execute ();
            }
     END
```

Retrieving Data: The SQLCursor Class

Methods of the SQLCursor class let stored procedures retrieve rows of data.

When stored procedures XE “stored procedure:creating an object from the SQLCuror class" create an object from the SQLCursor class, they pass as an argument an SQL statement that generates a result set. The SQL statement is either a SELECT or CALL statement:

A SELECT statement queries the database and returns data that meets the criteria specified by the query expression in the SELECT statement.

A CALL statement invokes another stored procedure that returns a result set specified by the RESULT clause of the CREATE PROCEDURE statement.

Either way, once the procedure creates an object from the SQLCursor class, the processing of results sets follows the same steps:

1. Open the cursor with the SQLCursor.Open method
2. Check whether there are any records in the result set with the SQLCursor.Found method.

Try to fetch a record with the SQLCursor.Fetch method.

Check whether the fetch returned a record with the SQLCursor.Found method.

If the fetch operation returned a record, assign values from the result-set record's fields to procedure variables or procedure output parameters with the SQLCursor.getValue method.

Process the data in some manner.

If the fetch operation did not return a record, exit the loop.

3. Close the cursor with the SQLCursor.close method

Following is an example that uses SQLCursor to process the result set returned by an SQL SELECT statement:

```
CREATE PROCEDURE get_sal ()
BEGIN
    StringBuffer ename = new StringBuffer (20) ;
    BigDecimal esal = new BigDecimal () ;

    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
    empcursor.fetch ();
        while (empcursor.Found ())
        {
            empcursor.getValue (1, ename);
            empcursor.getValue (2, esal);
            // [... do something with the values here ...]
        }
    empcursor.close () ;
END
```

Stored procedures also use SQLCursor objects to process a result set returned by another stored procedure. Instead of a SELECT statement, the SQLCursor constructor includes a CALL statement that invokes the desired procedure.

Following shows an excerpt from a stored procedure that processes the result set returned by another procedure, get_customers:

```
    SQLCursor cust_cursor = new SQLCursor (
```

```
    "CALL get_customers (?) ") ;

cust_cursor.setParam (1, "NE");
cust_cursor.Open () ;
for (;;)
{
    cust_cursor.Fetch ();
    if (cust_cursor.Found ())
    {
            cust_cursor.getValue (1, cust_number);
            cust_cursor.getValue (2, cust_name) ;
    }
    else
            break;
}

cust_cursor.close () ;
```

Returning a Procedure Result Set to Applications: The RESULT Clause and DhSQLResultSet

The `get_sal` procedure is used in the SQLCursor.getValue method to store the values of a database record in individual variables. But the procedure did not do anything with those values, and they would be overwritten in the next iteration of the loop that fetches records.

The DhSQLResultSet class provides a way for a procedure to store rows of data in a procedure result set so the rows can be returned to the application that calls it. There can only be one procedure result set in a stored procedure.

A stored procedure must explicitly process a result set to return it to the calling application:

- Declare the procedure result set through the RESULT clause of the procedure specification
- Populate the procedure result set in the body of the procedure using the methods of the DhSQLResultSet class

When the SQL engine creates a Java class from a CREATE PROCEDURE statement that contains the RESULT clause, it implicitly instantiates an object of type DhSQLResultSet, and calls it SQLResultSet. Invoke methods of the SQLResultSet instance to populate fields and rows of the procedure result set.

Extends to return a procedure result set. For each row of the SQL result set assigned to procedure variables, the procedure:

- Assigns the current values in the procedure variables to corresponding fields in the procedure result set with the DhSQLResultSet.Set method
- Inserts a row into the procedure result set with the DhSQLResultSet.Insert method

Example: Returning a Procedure Result Set From a Stored Procedure

```
CREATE PROCEDURE get_sal2 ()
RESULT (
    empname CHAR(20),
    empsal   NUMERIC,
)
BEGIN
    StringBuffer ename = new StringBuffer (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
    do
    {
        empcursor.Fetch ();
        if (empcursor.Found ())
        {
                empcursor.getValue (1, ename);
                empcursor.getValue (2, esal);
            SQLResultSet.Set (1, ename);
            SQLResultSet.Set (2, esal);
            SQLResultSet.Insert ();
        }
    } while (empcursor.Found ()) ;
    empcursor.close () ;
END
```

Handling Null Values

Stored procedures need to routinely set and detect null values:

- Stored procedures XE “stored procedure:handling null values" may need to set the values of SQL statement input parameters or procedure result fields to null.
- Stored procedures must check if the value of a field in an SQL result set is null before assigning it through the SQLCursor.getValue method. (The SQL engine generates a runtime error if result-set field specified in getValue is null.)

Setting SQL Statement Input Parameters and Procedure Result Set Fields to Null

Both the setParam method and set method (for procedure result set fields; take objects as their value arguments. You can pass a null reference directly to the method or pass a variable which has been assigned the null value. shows using both techniques to set an SQL input parameter to null.

Example: Passing Null Values to setParam and set

```
CREATE PROCEDURE test_nulls( )
BEGIN
     Integer pvar_int1      = new Integer(0);
     Integer pvar_int2      = new Integer(0);
     Integer pvar_int3;
     pvar_int3 = null;
     SQLIStatement insert_t1 = new SQLIStatement
     ( "INSERT INTO systpe.t1 (c1,c2, c3) values (?,?,?) ");
     insert_t1.setParam(1, new Integer(1)); // Set to non-null value
     insert_t1.setParam(2, null);           // Set directly to null
     insert_t1.setParam(3, pvar_int3);      // Set indirectly to null
     insert_t1.execute();
END
```

Assigning Null Values from SQL Result Sets: The SQLCursor.wasNULL Method

If the value of the field argument to the SQLCursor.getValue method is null, the SQL engine returns a runtime error:

```
(error(-20144): Null value fetched.)
```

This means you must always check whether a value is null before attempting to assign a value in an SQL result set to a procedure variable or output parameter. The SQLCursor class provides the wasNULL method for this purpose.

The SQLCuror.wasNULL method returns TRUE if a field in the result set is null. It takes a single integer argument that specifies which field of the current row of the result set to check.

Illustrates using wasNULL.

Example: Checking SQL Result Sets for Null Values with wasNULL

```
CREATE PROCEDURE test_nulls2( )
RESULT ( res_int1 INTEGER ,
         res_int2 INTEGER ,
         res_int3 INTEGER )
BEGIN
     Integer pvar_int1      = new Integer(0);
     Integer pvar_int2      = new Integer(0);
     Integer pvar_int3      = new Integer(0);
     SQLCursor select_t1 = new SQLCursor
     ( "SELECT c1, c2, c3 from t1" );

     select_t1.open();
     select_t1.fetch();
     while ( select_t1.found() )
     {
         // Assign values from the current row of the SQL result set
         // to the pvar_intx procedure variables. Must first check
         // whether the values fetched are null: if they are, must set
         // pvars explicitly to null.
 if ((select_t1.wasNULL(1)) == true)
            pvar_int1 = null;
 else
            select_t1.getValue(1,pvar_int1);
 if ((select_t1.wasNULL(2)) == true)
            pvar_int2 = null;
 else
            select_t1.getValue(2,pvar_int2);
 if ((select_t1.wasNULL(3)) == true)
            pvar_int3 = null;
```

```
 else
            select_t1.getValue(3,pvar_int3);
         // Transfer the value from the procedure variables to the
         // columns of the current row of the procedure result set.
         SQLResultSet.set(1,pvar_int1);
         SQLResultSet.set(2,pvar_int2);
         SQLResultSet.set(3,pvar_int3);
         // Insert the row into the procedure result set.
         SQLResultSet.insert();

         select_t1.fetch();
     }
     // Close the SQL result set.
     select_t1.close();
END
```

Handling Errors

VERSANT ReVind stored procedures XE “stored procedure:handling errors" use standard Java try/catch constructs to process exceptions.

Any errors in SQL statement execution result in creation of an DhSQLException class object. When VERSANT ReVind detects an error in an SQL statement, it throws an exception. The stored procedure should use try/catch constructs to process such exceptions. The getDiagnostics method of the DhSQLException class object provides a mechanism to retrieve different details of the error.

The getDiagnostics method takes a single argument whose value specifies which error message detail it returns.

Stored procedures can also throw their own exceptions by instantiating a DhSQLException object and throwing the object when the procedure detects an error in execution. The conditions under which the procedure throws the exception object are completely dependent on the procedure.

Please refer Chapter 13 “Error Messages” on page 415, for error messages and the associated SQLSTATE and VERSANT error code values.

CHAPTER 10

Java Class Reference

This Chapter provides reference material on the Versant ReVind Java classes and methods.

The Chapter covers the following in detail:

- Overview
- DhSQLException
- DhSQLResultSet
- SQLCursor
- SQLIStatement
- SQLPStatement

OVERVIEW

This chapter lists all the methods in the Versant ReVind Java classes and shows which classes declare them. Following sections are arranged alphabetically and describe each class and its methods in more detail. Some methods are common to more than one class.

TABLE 1. Methods in the Versant ReVind Java Classes

Method	SQLIStatement	SQLPStatement	SQLCursor	DhSQLResultSet	DhSQLException	Purpose
`setParam`	Y	Y	Y			Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter)
`makeNULL`	Y	Y	Y			Sets the value of an SQL statement's input parameter to null
`execute`	Y	Y				Executes the SQL statement
`rowCount`	Y	Y	Y			Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement
`open`			Y			Opens the result set specified by the SELECT or CALL statement
`close`			Y			Closes the result set specified by the SELECT or CALL statement
`fetch`			Y			Fetches the next record in a result set
`found`			Y			Checks whether a fetch operation returned a record
`wasNULL`			Y			Checks if the value in a fetched field is null

getValue			Y			Stores the value of a fetched field in the specified procedure variable or procedure output parameter
set			Y	Y		Sets the field in the currently-active row of a procedure's result set to the specified value (a literal, procedure variable, or procedure input parameter)
makeNULL			Y			Sets a field of the currently-active row in a procedure's result set to null
insert			Y			Inserts the currently-active row into a procedure's result set
getDiagnostics					Y	Returns the specified detail of an error message
log	Y	Y	Y	Y	Y	Writes a message to the file *sql_server.log*. Inherited by all the Versant ReVind classes. See section.
err	Y	Y	Y	Y	Y	Writes a message to the file *sqlerr.log*. Inherited by all the Versant ReVind classes. See section.

DHSQLEXCEPTION

Description

The DhSQLException class extends the general java.lang.Exception class to provide detail about errors in SQL statement execution. Any such errors raise an exception with an argument that is an SQLException class object. The getDiagnostics() method retrieves details of the error.

Constructors

```
public DhSQLException(int ecode, String errMsg)
```

Parameters

ecode

The error number associated with the exception condition.

errMsg

The error message associated with the exception condition.

Example

This procedure illustrates using the DhSQLException constructor to create an exception object called excep. It then throws the excep object under all conditions.

```
CREATE PROCEDURE sp1_02()
BEGIN
     // raising exception
     DhSQLException
excep = new DhSQLException(666,new String("Entered the t
st02 procedure"));
     if (true)
     throw excep;
END
```

DhSQLException.getDiagnostics

Returns the requested detail about an exception.

Format

```
public String getDiagnostics(int diagType)
```

Returns

A string containing the information specified by the diagType parameter, as shown in table.

Parameters

diagType

One of the values shown in table below.

TABLE 2. Argument Values for DhSQLException.getDiagnostics

Argument Value	Returns
RETURNED_SQLSTATE	The SQLSTATE returned by execution of the previous SQL statement.
MESSAGE_TEXT	The condition indicated by RETURNED_SQLSTATE.
CLASS_ORIGIN	Not currently used. Always returns null.
SUBCLASS_ORIGIN	Not currently used. Always returns null.

Throws

DhSQLException

Example

```
try
{
    SQLIStatement insert_cust = new SQLIStatement (
     "INSERT INTO customer VALUES (1,2) ");
}
catch (DhSQLException e)
{
    errstate = e.getDiagnostics (RETURNED_SQLSTATE) ;
    errmesg  = e.getDiagnostics (MESSAGE_TEXT) ;
    ...
}
```

DHSQLRESULTSET

Description

Methods of the DhSQLResultSet class populate a result set that the stored procedure returns to the application that called the procedure.

The Java code in a stored procedure does not explicitly create DhSQLResultSet objects. Instead, when the SQL engine creates a Java class from a CREATE PROCEDURE statement that contains a RESULT clause, it implicitly instantiates an object of type DhSQLResultSet, and calls it SQLResultSet.

Procedures invoke methods of the SQLResultSet instance to populate fields and rows of the result set.

Constructors

No explicit constructor

Parameters

None

Throws

DhSQLException

DhSQLResultSet.insert

Inserts the currently-active row into a procedure's result set.

Format

```
public void insert()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

DhSQLResultSet.makeNULL

Sets a field of the currently-active row in a procedure's result set to null. This method is redundant when using with the DhSQLResultSet.set method to set a procedure result-set field to null.

Format

```
public void makeNULL(int field)
```

Returns

None

Parameters

field

An integer that specifies which field of the result-set row to set to null. (1 denotes the first field in the row, 2 denotes the second, and so on.)

Throws

DhSQLException

Example

```
CREATE PROCEDURE test_makeNULL2(
    IN char_in CHAR(20)
RESULT ( res_char CHAR(20) , res_vchar VARCHAR(30))
BEGIN

    SQLResultSet.set(1,char_in);
    SQLResultSet.makeNULL(2);

END
```

DhSQLResultSet.set

Sets the field in the currently-active row of a procedure's result set to the specified value (a literal, procedure variable, or procedure input parameter).

Format

```
public void set(int field, Object val)
```

Returns

None

Parameters

field

An integer that specifies which field of the result-set row to set to the value specified by *val*. (1 denotes the first field in the row, 2 denotes the second, and so on.)

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the field.

Throws

DhSQLException

Example

SQLCURSOR

Description

Methods of the SQLCursor class retrieve rows of data from a database or another stored procedure's result set.

Constructors

```
SQLCursor (String statement)
```

Parameters

statement

Throws

DhSQLException

Example

The following excerpt from a stored procedure instantiates an SQLCursor object called cust_cursor that retrieves data from a database table:

```
SQLCursor empcursor = new SQLCursor (
    "SELECT name, sal FROM emp " ) ;
```

The following excerpt from a stored procedure instantiates an SQLCursor object called cust_cursor that calls another stored procedure:

```
SQLCursor cust_cursor = new SQLCursor (

  "CALL get_customers (?) ") ;
```

SQLCursor.close

Closes the result set specified by the SELECT or CALL statement.

Format

```
public void close()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

```
.
.
.
        if (cust_cursor.Found ())
        {
                cust_cursor.getValue (1, cust_number);
                cust_cursor.getValue (2, cust_name) ;
        }
        else
                break;
    }

    cust_cursor.close () ;
```

SQLCursor.fetch

Fetches the next record in a result set, if there is one.

Format

```
public void fetch()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

```
for (;;)
{
    cust_cursor.Fetch ();
    if (cust_cursor.Found ())
    {
            cust_cursor.getValue (1, cust_number);
            cust_cursor.getValue (2, cust_name) ;
    }
    else
            break;
}
```

SQLCursor.found

Checks whether a fetch operation returned a record.

Format

```
public boolean found ()
```

Returns

True if the previous call to fetch() returned a record, false otherwise.

Parameters

None

Throws

DhSQLException

Example

```
for (;;)
{
```

```
        cust_cursor.Fetch ();
        if (cust_cursor.Found ())
        {
                cust_cursor.getValue (1, cust_number);
                cust_cursor.getValue (2, cust_name) ;
        }
        else
                break;
    }
```

SQLCursor.getValue

Stores the value of a field from a fetched row in the specified procedure variable or procedure output parameter.

Format

```
public void getValue(int field, Object var)
```

Returns

None

Parameters

field

An integer that specifies which field of the fetched record is of interest. getValue retrieves the value in the currently-fetched record of the column denoted by field. (1 denotes the first column of the result set, 2 denotes the second, and so on).

The value in the column denoted by field cannot be null, or the SQL engine returns an error:

```
(error(-20144): Null value fetched.)
```

This means you must always check whether a value is null before attempting to assign a value in an SQL result set to a procedure variable or output parameter. The SQLCursor class provides the wasNULL method for this purpose.

var

The variable or output parameter name that will receive the value from the result set.

Throws

DhSQLException

Example

```
.
.
.
     Integer pvar_int1      = new Integer(0);
     Integer pvar_int2      = new Integer(0);
     Integer pvar_int3      = new Integer(0);
     SQLCursor select_t1 = new SQLCursor
     ( "SELECT c1, c2, c3 from t1" );

     select_t1.open();
     select_t1.fetch();
     while ( select_t1.found() )
     {
         // Assign values from the current row of the SQL result set
         // to the pvar_intx procedure variables. Must first check
        // whether the values fetched are null: if they are, must set
         // pvars explicitly to null.
 if ((select_t1.wasNULL(1)) == true)
            pvar_int1 = null;
 else
            select_t1.getValue(1,pvar_int1);
 if ((select_t1.wasNULL(2)) == true)
            pvar_int2 = null;
 else
            select_t1.getValue(2,pvar_int2);
 if ((select_t1.wasNULL(3)) == true)
            pvar_int3 = null;
 else
            select_t1.getValue(3,pvar_int3);
.
.
.
```

SQLCursor.makeNULL

Sets the value of an SQL statement's input parameter to null. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes. This method is redundant when using with the setParam method to set an SQL statement's input parameter to null.

Format

```
public void makeNULL(int f)
```

Returns

None

Parameters

f

An integer that specifies which input parameter of the SQL statement string to set to null. (1 denotes the first input parameter in the statement, 2 denotes the second, and so on.)

Throws

DhSQLException

Example

```
CREATE PROCEDURE sc_makeNULL()
BEGIN
     SQLCursor select_btypes = new SQLCursor (
         "SELECT small_fld from sfns where small_fld = ? ");

     select_btypes.makeNULL(1);
     select_btypes.open();
     select_btypes.fetch();
.
.
.
     select_btypes.close();

END
```

SQLCursor.open

Opens the result set specified by the SELECT or CALL statement.

Format

```
public void open()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

```
.
.
.
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.Open () ;
.
.
.
```

SQLCursor.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public int rowCount()
```

Returns

An integer indicating the number of rows.

Parameters

None

Throws

DhSQLException

Example

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs BIGINT )
BEGIN
     SQLIStatement insert_test103 = new SQLIStatement (
     "INSERT INTO test103 (fld1) values (17)");

     insert_test103.execute();
     SQLResultSet.set(1,new Long(insert_test103.rowCount()));
     SQLResultSet.insert();
END
```

SQLCursor.setParam

Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the SQLCursor, SQLIStatement, and SQLPStatement classes.

Format

```
public void setParam(int f, Object val)
```

Returns

None

Parameters

f

An integer that specifies which parameter marker in the SQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).

val

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

Throws

DhSQLException

Example

```
CREATE PROCEDURE sps_setParam()

BEGIN

     // Assign local variables to be used as SQL input parameter
     references
     Integer ins_fld_ref   = new Integer(1);
     Integer ins_small_fld = new Integer(3200);
     Integer ins_int_fld   = new Integer(21474);
     Double  ins_doub_fld  = new Double(1.797E+30);
     StringBuffer  ins_char_fld  = new StringBuffer("Athula");
     StringBuffer  ins_vchar_fld = new StringBuffer("Scientist");
     Float   ins_real_fld  = new Float(17);

\    SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
       (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld) \
       values (?,?,?,?,?,?)" );

     insert_sfns1.setParam(1,ins_fld_ref);
     insert_sfns1.setParam(2,ins_small_fld);
     insert_sfns1.setParam(3,ins_int_fld);
     insert_sfns1.setParam(4,ins_doub_fld);
     insert_sfns1.setParam(5,ins_char_fld);
     insert_sfns1.setParam(6,ins_vchar_fld);
     insert_sfns1.execute();
END
```

SQLCursor.wasNULL

Checks if the value in a fetched field is null.

Format

```
public boolean wasNULL(int field)
```

Returns

True if the field is null, false otherwise.

Parameters

field

Throws

DhSQLException

Example

```
CREATE PROCEDURE test_wasNULL()
BEGIN

     int small_sp      = 0;

 SQLCursor select_btypes = new SQLCursor ("SELECT small_fld from
sfns");

     select_btypes.open();
     select_btypes.fetch();
     if ((select_btypes.wasNULL(1)) == true)
            small_sp = null;
     else
            select_btypes.getValue(1,small_sp);

     select_btypes.close();

END
```

SQLISTATEMENT

Description

Methods of the SQLIStatement class provide for immediate (one-time) execution of SQL statements that do not generate a result set.

Constructors

```
SQLIStatement (String statement)
```

Parameters

statement

An SQL statement that does not generate a result set. Enclose the SQL statement in double quotes.

Throws

DhSQLException

Example

```
     CREATE PROCEDURE insert_customer (
        IN  cust_number INTEGER,
        IN  cust_name   CHAR(20)
     )
     BEGIN
         SQLIStatement insert_cust = new SQLIStatement (
            "INSERT INTO customer VALUES (?,?) ");
.
.
.
     END
```

SQLIStatement.execute

Executes the SQL statement. This method is common to the SQLIStatement and SQLPStatement classes.

Format

```
public void execute()
```

Returns

None

Parameters

None

Throws

DhSQLException

Example

```
CREATE PROCEDURE insert_customer (
   IN  cust_number INTEGER,
   IN  cust_name   CHAR(20)
)
BEGIN
    SQLIStatement insert_cust = new SQLIStatement (
       "INSERT INTO customer VALUES (?,?) ");
    insert_cust.setParam (1, cust_number);
    insert_cust.setParam (2, cust_name);
    insert_cust.execute ();
END
```

SQLIStatement.makeNULL

```
CREATE PROCEDURE sis_makeNULL()
BEGIN

     SQLIStatement insert_sfns1 = new SQLIStatement ("INSERT INTO sfns \
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld) \
      values (?,?,?,?,?,?)" );
     insert_sfns1.setParam(1,new Integer(66));
     insert_sfns1.makeNULL(2);
     insert_sfns1.makeNULL(3);
     insert_sfns1.makeNULL(4);
     insert_sfns1.makeNULL(5);
     insert_sfns1.makeNULL(6);
     insert_sfns1.execute();
```

```
END
```

SQLIStatement.rowCount

SQLIStatement.setParam

SQLPSTATEMENT

Description

Methods of the SQLPStatement class provide for prepared (repeated) execution of SQL statements that do not generate a result set.

Constructors

```
SQLPStatement (String statement)
```

Parameters

statement

Throws

DhSQLException

Example

```
     SQLPStatement pstmt = new SQLPStatement (
         "INSERT INTO T1 VALUES (?, ?) ") ;
.
.
.
```

SQLPStatement.execute

```
     SQLPStatement pstmt = new SQLPStatement (
         "INSERT INTO T1 VALUES (?, ?) ") ;

     pstmt.setParam (1, 10);
     pstmt.setParam (2, 10);
     pstmt.execute ();

     pstmt.setParam (1, 20);
     pstmt.setParam (2, 20);
     pstmt.execute ();
```

SQLPStatement.makeNULL

```
CREATE PROCEDURE sps_makeNULL()
BEGIN
\   SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns
      (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld) \
      values (?,?,?,?,?,?)" );

     insert_sfns1.setParam(1,new Integer(666));
     insert_sfns1.makeNULL(2);
     insert_sfns1.makeNULL(3);
     insert_sfns1.makeNULL(4);
     insert_sfns1.makeNULL(5);
     insert_sfns1.makeNULL(6);
     insert_sfns1.execute();
END
```

SQLPStatement.rowCount

SQLPStatement.setParam

CHAPTER 11

JDBC

This Chapter provides a brief overview of JDBC and describes it's architecture, setup, connections.

The Chapter explains the following in detail:

- JDBC Introduction
- Setup
- Connecting to a Database from within a JDBC Client
- Managing Transactions Explicitly to Improve Performance
- JDBC Conformance Notes
- Sample Program Source Code

JDBC INTRODUCTION

Overview

JDBC allows applications to connect to any database using the same set of Java interfaces. Those interfaces allow programs to embed standard Structured Query Language (SQL) statements that update and retrieve data in the database.

Because the Java interfaces and SQL syntax are independent of any particular database implementation, JDBC makes feasible applications that can connect to different database environments without any modification.

JDBC Architecture

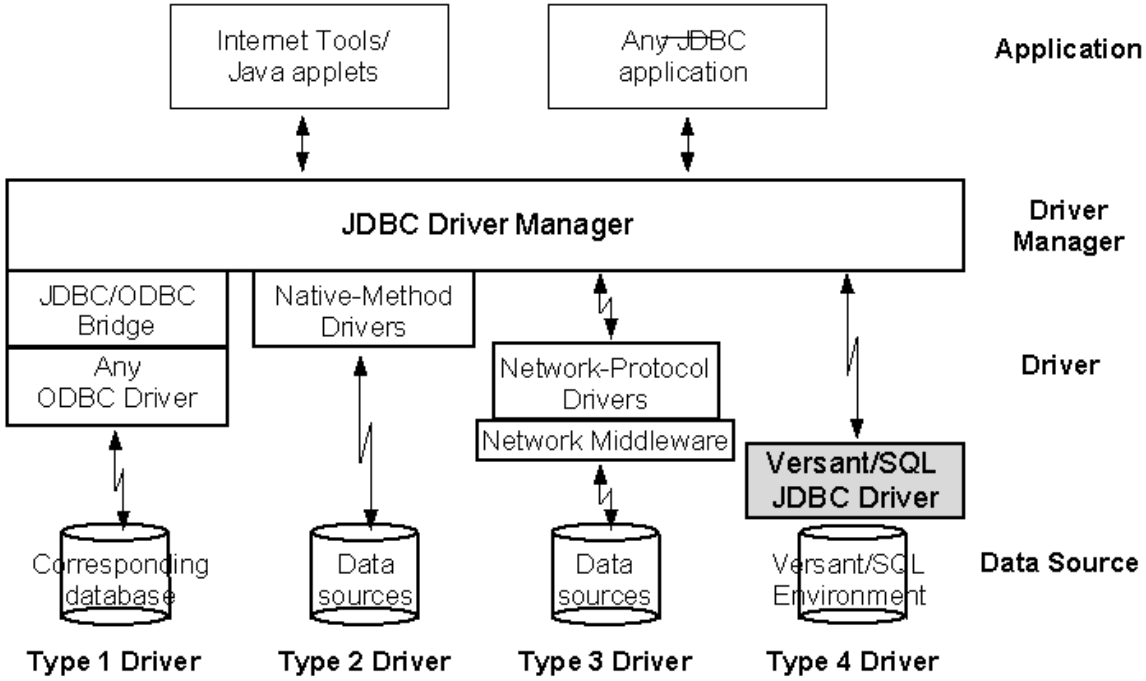
JDBC insulates Java applications from variations in database-access implementations through the JDBC API, a set of class libraries distributed as a standard part of core Java. Instead of using calls to vendor-specific interfaces, JDBC applications use the JDBC API.

The JDBC API is distributed as the package `java.sql` and is included with the JavaSoft JDK and Microsoft Java SDK (Version 2.0, or later), so any environment that supports a recent Java compiler can be used to develop JDBC applications.

Calls to the JDBC API are managed by the JDBC driver manager. The JDBC driver manager can support multiple drivers connecting to different databases. When an application tries to connect to a particular database, the driver manager loads the appropriate JDBC driver and routes subsequent calls through the driver.

A JDBC driver is a database-specific piece of software that receives calls from the JDBC driver manager, translates them into a form that the database can process, and returns data to the application.

The following shows the different components of the JDBC architecture:



Types of JDBC Drivers

JDBC drivers can either be entirely written in Java so that they can be downloaded as part of an applet, or they can be implemented using native methods to bridge to existing database access libraries.

JavaSoft defines four different types of JDBC drivers, as shown in the above illustration and outlined in the following sections:

JDBC-ODBC Bridge Drivers

Type 1 drivers translate calls to JDBC methods into calls to Microsoft Open Database Connectivity (ODBC) functions. Bridge drivers allow JDBC applications immediate access to database connectivity provided by the existing array of ODBC drivers.

Both the JavaSoft JDK and Microsoft Java SDK include JDBC-ODBC bridge drivers.

ODBC architecture requires that the ODBC driver manager and (typically) the ODBC drivers themselves be loaded on each client system. The requirement for software resident on client

systems means that JDBC-ODBC bridge drivers will not work with Java applets run from an Internet browser. Browsers do not allow applets to run another program on the client to which they are downloaded. (In general, JDBC-ODBC bridge drivers will not work in environments that restrict Java applications from reading and writing files or running other programs.)

JDBC-ODBC bridge drivers are still useful in corporate networks, or for use by application server code written in Java in a 3-tier architecture. In such an environment, the application server has intermediary software, such as Blue Lobster's Aptivity, that receives requests from browsers and other Internet applications. The intermediary software in turn calls the JDBC driver manager when it receives a database request.

Native-Method Drivers

Type 2 drivers contain Java code that calls "native" C or C++ methods already implemented by database vendors.

Like an ODBC driver, a native-method driver must be installed on each client or server that uses it, and thus has the same limitations as the JDBC-ODBC bridge drivers. A typical use of native-method drivers is on application servers.

Network-Protocol All-Java Drivers

Type 3 drivers are completely written in Java. They translate JDBC calls into a database-independent network protocol which is in turn translated to a DBMS protocol by middleware on a network server.

This type of driver can thus connect many Java clients to many different databases. The specific protocol used depends on the vendor.

Type 3 drivers are the most flexible since they do not require any driver software resident on client systems and can allow a single driver to provide access to multiple databases.

Native-Protocol All-Java Drivers (VSQL JDBC Driver)

Type 4 drivers are also written completely in Java, but do not rely on middleware. They convert JDBC calls directly into the network protocol used by a particular database. This approach allows a direct call from the client system to the database server. Also, since there is no client-resident software, it also is practical for Internet applications.

Type 4 drivers provide the best performance.

The Versant JDBC Driver is a Type 4 driver.

JDBC Compared to ODBC

Generally speaking, JDBC is to Java what Microsoft's Open Database Connectivity (ODBC) interface is to the C language.

Both JDBC and ODBC:

- Provide a vendor-independent API that allows the same application to connect to different vendors' databases and retrieve and update data using standard SQL statements.
- Adopt the architecture of imposing a driver manager between applications and vendor-supplied drivers that translate between the standard API and a vendor's proprietary implementation.
- Are based on the X/Open SQL call-level interface specification.

JDBC proponents cite these advantages of JDBC over ODBC:

- JDBC applications enjoy the platform-independence of Java, which lends itself to Internet applications. ODBC applications must at a minimum be recompiled to run on a different operating-system/hardware combination.
- JDBC does not require software on each client system, which also recommends it for Internet applications.
- JDBC is much simpler and easier to learn than ODBC.
- JDBC is not primarily targeted for PC application development, which makes for faster implementation outside the Windows environment.

SETUP

Set up Java environment

You must have a supported Java development or runtime environment on each system that uses the JDBC Driver.

Please see your Release Notes for the version of JavaSoft JDK that is supported.

Set up the JDBC Driver on the Web Server

In a Web server environment, the JDBC driver and Java applets that use it reside on a host system. No additional driver software is required on the client machine. Client applications must support a Java virtual machine compatible with JavaSoft's JDK Version 1.4 or later. (Internet browsers such as Netscape or Internet Explorer meet this requirement.)

Client applications invoke a JDBC applet through a Web page on the server. The browser downloads both the applet and the JDBC Driver from the server (usually in compressed format) and runs the applet. The Java applet opens a database connection (see following) and accesses the database using the JDBC API.

A general JDBC security restriction is that applets can only open a database connection from the server from which they are downloaded. That means the host system must be running both the HTTP Web server and the Versant server process.

See your Release Notes for details of managing the server process.

To set up the JDBC Driver for an applet on a Web server, complete these steps:

1. Copy compiled class files for the JDBC Driver and the applet to a directory accessible to the Web page that will invoke the applet.
2. Compress all the class files into a single Java Archive (JAR) file.
3. Create the Web page that will invoke the applet.

The following describes these steps:

1. Copy JDBC Driver and Applet class files.

On both Windows and UNIX, the `vsql.jar` file under the `lib` directory comprises of all the class files required for the JDBC Driver. Copy this jar file from that directory to a directory accessible to the Web page. Do the same for the applet's class file.

For example:

```
cd $webroot

mkdir test

cd test

cp -i $TPEROOT/lib/vsql.jar .

cp -i /applet_test/DhJDBCApplet.class .
```

2. Compress class files Into Java Archive (JAR) files.

This step is optional but recommended. JAR files greatly reduce the number of connections a browser must make to the Web server to download required classes.

For example, from the directory you created in the previous step:

```
jar -cvf DhJDBCTest.jar *.class

adding: ClRqTypes.class (in=1287) (out=736) (deflated 42%)

adding: CntlIface.class (in=696) (out=375) (deflated 46%)

adding: CntlIfaceCS.class (in=2143) (out=1116) (deflated 47%)

adding: CntlIfaceSS.class (in=2133) (out=1056) (deflated 50%)

.

.

.

ls -al *.jar

-rw-r--r-- 1 systpe staff 132534 Sep 2 16:58 DhJDBCTest.jar
```

3. Create a Web Page that invokes the Applet

At a minimum, the page must include the `APPLET` tag that invokes the applet.

For example, the following page includes little else but the `APPLET` tag. In this case, the `APPLET` tag specifies the `VJDBCApplet.class` and `VJDBCTest.jar` files from the preceding sections, as well as class-name and connection parameters to pass to the applet.

```
<html>
<head>
  <title>Test</title>
</head>
<body>
  <p>
  Here is the VJDBCApplet test applet!
  <center>
  <applet code="VJDBCApplet.class"
          archive="VJDBCTest.jar"
          width=500
          height=400>
     <param name=Driver value="com.versant.jdbc.VersantDriver">
     <param name=URL value="jdbc:versant:T:isis:jdbcdb:5020">
     <param name=User value="systpe">
     <param name=Password value="dummy">
  </applet>
  </center>
</body>
</html>
```

Set up the JDBC Driver on the Application Server

In an application server environment, the system on which the JDBC application runs also has the JDBC driver installed. This configuration provides good performance when users are on the same system or can execute the JDBC application across a network.

To set up the JDBC Driver, you must have access to a system (UNIX or Windows) where the Versant ReVind libraries and executable files have been built, as described in your *Release Notes*. On both Windows and UNIX, the `vsql.jar` file under the `lib` directory comprises of all these class files.

If the application server is a different system than the system used to build Versant ReVind libraries and executables, you may need to copy the `vsql.jar` file to the application server. Then, you need to set the `CLASSPATH` environment variable to point to the `vsql.jar` file.

The following describes these steps:

1. Copy JDBC Driver Files to the Application Server (if necessary).

This step is not necessary if the application server has access to the disk containing the class files (through a Windows network or NFS-mounted disks on UNIX).

Otherwise, you must first copy files to the application server. On UNIX systems and Windows systems, use any available utility to copy the class files to the application server.

On both Windows and UNIX, the class files are comprised in the `..lib\vsql.jar` file.

On Windows, for example:

```
mkdir \v6050\lib

cd \v6050\lib

ftp -i ktwo

Connected to ktwo.

220 ktwo Microsoft FTP Service (Version 2.0).

User (ktwo:(none)): systpe

331 Password required for systpe.

Password:

230 User systpe logged in.

ftp> cd v6050\lib

250 CWD command successful.
```

```
ftp> binary

200 Type set to I.

ftp> mget vsql.jar

200 PORT command successful.

150 Opening BINARY mode data connection for VersantCA.class(1069
bytes).

226 Transfer complete.

.

.
```

2. Set Environment Variables

Whether the JDBC Driver class files reside locally or on network-served disks, you must set the `CLASSPATH` environment variable to point to the class files.

On both Windows and UNIX, the CLASSPATH environment variable must point to the `vsql.jar` file.

Windows

On Windows, you must set the `CLASSPATH` environment variable in two places:

1. In the initialization file `VSQL.INI`, in the windows directory (the directory specified by the `windir` system variable on Windows). Edit this file and add a line similar to the following:

   ```
   CLASSPATH=c:\jdk1.3\jre\lib\rt.jar;c:\v6050\lib\vsql.jar
   ```

   Note that the `CLASSPATH` setting in `VSQL.INI` cannot include references to other environment variables, such as to `%TPEROOT%`.

2. At the Windows command prompt, as in the following example:

   ```
   set classpath

   CLASSPATH=c:\jdk1.3\jre\lib\rt.jar

   set CLASSPATH=%CLASSPATH%;c:\v6050\lib\vsql.jar
   ```

For the environment variables to persist across different processes, set them using the Window Control Panel's `System` utility as system variables.

Also, on Windows, make sure the PATH environment variable includes the directory %TPEROOT%\bin:

```
set PATH=%TPEROOT%/bin;%PATH%
```

UNIX

Make sure the CLASSPATH variable points to the vsql.jar file.

For example:

```
% setenv CLASSPATH ".:$TPEROOT/lib/vsql.jar:${JDKHOME}/jre/lib/rt.jar"
```

Run the Sample Application

The Versant ReVind $TPEROOT/demo/jdbc directory includes a test Java application, JDBCTest.class, that exercises the JDBC interfaces implemented by the JDBC Driver. The application connects to the specified database (or creates a default connection hardcoded in the application) and presents a limited interactive SQL interface.

You can run the sample application to check the setup of the JDBC Driver in an application server environment.

For example, on Windows(using the JavaSoft JDK), the following command line supplies a connection URL to run the sample program:

```
java JDBCTest jdbc:Versant:T:ktwo:kirk2:5020
```

If you have a Java compiler, you can copy the sample program source code and modify it.

The source file is $TPEROOT\demo\jdbc\JDBCTest.java. A listing of JDBCTest.java is included in this chapter.

CONNECTING TO A DATABASE FROM WITHIN A JDBC CLIENT

The following describes operations that are common to all JDBC applications in terms of the methods they use. However, the arguments supplied to certain methods will expose some idioms that are Versant-specific. For example, in the URL that identifies a data source, the URL elements that will be shown will be those applicable to a Versant ReVind data source.

JDBC applications must perform two steps to connect to a database:

1. Load the JDBC driver
2. Connect to the driver

Load the JDBC Driver using Class.forName

The `Class.forName` method is used to load the JDBC driver, which is specified as `com.versant.jdbc.VersantDriver`, a fully-qualified class name. Because you configured the `CLASSPATH` environmental variable of the host operating system to include the pathname to the starting directory of this path-qualified file, the JVM will load and link the Versant/JDBC driver:

To load the JDBC Driver, use it as the argument to the `Class.forName` method:

```
// Load the driver

Class.forName("com.versant.jdbc.VersantDriver");
```

Connect to the JDBC Driver using DriverManager.GetConnection

To connect to a Versant ReVind database through the JDBC Driver, an application specifies:

- A database connection string in the form of a JDBC URL
- User authentication detail (user name and password)

Applications specify this information as arguments to the `DriverManager.GetConnection` method.

JDBC URL Connection String

`DriverManager.GetConnection` requires at least one argument, a character string specifying a database connection URL. To specify the Versant JDBC driver, a TCP/IP communications protocol, a sever hostname, and a database name, the URL takes the following form:

```
jdbc:versant:T:host_name:db_name:port
```

The URL string has the following components:

`jdbc:versant:T`	A conventional subprotocol string that identifies a Versant JDBC Driver and TCP/IP communications protocol.
`host_name`	Name of the server system where the database resides.
`db_name`	Name of the database.
`port`	The port number associated with the JDBC server on the host system. In most cases this component is optional. Java applets that are hosted on servers that do not use the default port number of 2223 must use this component to specify the correct port number.

For example, the default URL in the sample application is `jdbc:versant:T:isis:testdb:5020`. When passed to `DriverManager.GetConnection`, this URL specifies that the Versant JDBC Driver be used to connect to the database `testdb` on the server named `isis`.

User Authentication Detail

`DriverManager.GetConnection` accepts three variants of user authentication detail:

- User name and password passed as two character string arguments:

  ```
  Connection con = DriverManager.getConnection(url, "fred" "fredspasswd"
  );
  ```

- User name and password passed as a single properties object:

  ```
  Connection con = DriverManager.getConnection(url, prop );
  ```

  Note that the JDBC Driver expects the keys of the `prop` object to be named `user` and `password` when it processes the object. Application code must use those names when it populates the `prop` object:

  ```
  prop.put("user", userid);
  prop.put("password", passwd);
  ```

- User name and password omitted.

The JDBC Driver connects to the database with a blank username and null password:

```
Connection con = DriverManager.getConnection(url);
```

An Example of Connecting

The following example shows a code excerpt that illustrates loading the driver and connecting to the default server and database. This example uses the form of `DriverManager.GetConnection` that takes authentication information as a single Properties object.

Loading the JDBC Driver and Connecting to a Database:

```
String url    = "jdbc:versant:T:isis:testdb:5020";

String userid = "fred";

String passwd = "fredspasswd";



// Load the driver

Class.forName ("com.versant.jdbc.VersantDriver");



// Attempt to connect to a driver.  Each one

// of the registered drivers will be loaded until

// one is found that can process this URL.

java.util.Properties  prop = new java.util.Properties();

prop.put("user", userid);

prop.put("password", passwd);
```

```
Connection con = DriverManager.getConnection(url, prop);
```

MANAGING TRANSACTIONS EXPLICITLY TO IMPROVE PERFORMANCE

By default, new connections in JDBC applications are in "autocommit" mode. In autocommit mode every SQL statement executes in its own transaction:

- After successful completion, the JDBC Driver automatically commits the transaction.
- If the statement execution fails, the JDBC Driver automatically rolls back the transaction.

In autocommit mode, the JDBC Driver does not issue a commit after `SELECT` and `CALL` statements. The driver assumes these statements generate result sets and relies on the application to explicitly commit or roll back the transaction after it processes any result set and closes the statement.

You can change the transaction mode to "manual commit" by calling the `Connection.setAutoCommit` method. In manual commit mode, applications must commit a transaction by using the `Connection.commit` method and roll back a transaction by invoking the `Connection.rollback` method.

You will improve the performance of your programs by setting autocommit to false after creating a Connection object with the Connection.setAutoCommit method:

```
Connection con = DriverManager.getConnection ( url, prop);

con.setAutoCommit(false);
```

JDBC CONFORMANCE NOTES

Supported Data Types

The Versant ReVind ODBC Driversupports standard JDBC mapping of JDBC types corresponding Java types.

In the JDBC methods, `CallableStatement.getXXX` and `PreparedStatement.setXXX`methods, *XXX*, is a Java type:

For `setXXX` methods, the driver converts the Java type to the JDBC type, shown in the table below, before sending it to the database.

For `getXXX` methods, the driver converts the JDBC type returned by the database to the Java type before returning it to the `getXXX` method. Following are Java data types and corresponding JDBC data types:

Mapping Between Java and JDBC Data Types

Java Type	JDBC type
`boolean`	`BIT`
`byte`	`TINYINT`
`byte[]`	`VARBINARY or LONGVARBINARY`
`double`	`DOUBLE`
`float`	`REAL`
`int`	`INTEGER`
`java.math.BigDecimal`	`NUMERIC`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
`long`	`BIGINT`
`short`	`SMALLINT`
`String`	`VARCHAR or LONGVARCHAR`

Mapping Between JDBC and Java Data Types

JDBC type	Java type
BIGINT	long
BINARY	byte[]
BIT	boolean
CHAR	String
DATE	java.sql.Date
DECIMAL	java.math.BigDecimal
DOUBLE	double
FLOAT	double
INTEGER	int
LONGVARBINARY	byte[]
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
REAL	float
SMALLINT	short
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
TINYINT	byte
VARBINARY	byte[]
VARCHAR	String

DatabaseMetaData methods

Applications call methods of the `DatabaseMetaData` class to retrieve details about the JDBC support provided by a specific driver.

Getting Driver Information Through DatabaseMetadata

The following example shows an excerpt from the sample program that illustrates calling methods of `DatabaseMetadata`:

```
Connection con = DriverManager.getConnection ( url, prop);

.

// Get the DatabaseMetaData object and display

// some information about the connection


DatabaseMetaData dma = con.getMetaData ();


o.println("\nConnected to " + dma.getURL());

o.println("Driver       " +

dma.getDriverName());

o.println("Version      " +

dma.getDriverVersion());
```

Return Values for DatabaseMetaData Methods

Many of the `DatabaseMetaData` methods return lists of information as an object of type `ResultSet`. Use the normal `ResultSet` methods such as `getString` and `getInt` to retrieve the data from the result sets.

Following are `DatabaseMetaData` methods and their return values:

`allProceduresAreCallable()`

Can all the procedures returned by `getProcedures` be called by the current user?

Returns `False`.

`allTablesAreSelectable()`

Can all the tables returned by `getTable` be `SELECT`ed by the current user?

Returns `False`.

`dataDefinitionCausesTransactionCommit()`

Does a data definition statement within a transaction force the transaction to commit?
Returns `False`.

`dataDefinitionIgnoredInTransactions ()`

Is a data definition statement within a transaction ignored?
Returns `False`.

`doesMaxRowSizeIncludeBlobs()`

Did `getMaxRowSize()` include `LONGVARCHAR` and `LONGVARBINARY` blobs?
Returns `False`.

`getBestRowIdentifier(String, String, String, int, boolean)`

Get a description of a table's optimal set of columns that uniquely identifies a row.
Returns `(result .set)`

`getCatalogs()`

Get the catalog names available in this database.
Returns `(result .set)`

`getCatalogSeparator()`

What's the separator between catalog and table name?
Returns `null`.

`getCatalogTerm()`

What's the database vendor's preferred term for "catalog"?
Returns `null`.

`getColumnPrivileges(String, String, String, String)`

Get a description of the access rights for a table's columns.
Returns `(result .set)`

`getColumns(String, String, String, String)`

Get a description of table columns available in a catalog.
Returns `(result .set)`

`getCrossReference(String, String, String, String, String, String)`

Get a description of the foreign key columns in the foreign key table that reference the primary key columns of the primary key table (describe how one table imports another's key.) This should normally return a single foreign key/primary key pair (most tables only import a foreign key from a table once.) They are ordered by `FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ.`

Returns `(result .set)`

`getDatabaseProductName()`

What's the name of this database product?

Returns `"Versant/SQL."`

`getDatabaseProductVersion()`

What's the version of this database product?

Returns `"7.0.1.4"`

`getDefaultTransactionIsolation()`

What's the database's default transaction isolation level? The values are defined in java.sql.Connection.

Returns `TRANSACTION_READ_COMMITTED`.

`getDriverMajorVersion()`

What's this JDBC driver's major version number?

Returns `6`.

`getDriverMinorVersion()`

What's this JDBC driver's minor version number?

Returns `0`.

`getDriverName()`

What's the name of this JDBC driver?

Returns `"com.versant.jdbc.VersantDriver"`

`getDriverVersion()`

What's the version of this JDBC driver?

Returns `"7.0.1.4"`

`getExportedKeys(String, String, String)`

Get a description of the foreign key columns that reference a table's primary key columns (the foreign keys exported by a table).

Returns `(result .set)`

`getExtraNameCharacters()`

Get all the "extra" characters that can be used in unquoted identifier names (those beyond a-z, A-Z, 0-9 and _).

Returns `null`.

`getIdentifierQuoteString ()`

What's the string used to quote SQL identifiers? This returns a space " " if identifier quoting isn't supported.

Returns " "

`getImportedKeys(String, String, String)`

Get a description of the primary key columns that are referenced by a table's foreign key columns (the primary keys imported by a table).

Returns `(result .set)`

`getIndexInfo(String, String, String, boolean, boolean)`

Get a description of a table's indices and statistics.

Returns `(result .set)`

`getMaxBinaryLiteralLength()`

How many hex characters can you have in an inline binary literal?

Returns `2000`.

`getMaxCatalogNameLength()`

What's the maximum length of a catalog name?

Returns `32`.

`getMaxCharLiteralLength()`

What's the max length for a character literal?

Returns `2000`.

`getMaxColumnNameLength()`

What's the limit on column name length?

Returns `32`.

`getMaxColumnsInGroupBy()`

What's the maximum number of columns in a `"GROUP BY"` clause?
Returns `500`.

`getMaxColumnsInIndex()`

What's the maximum number of columns allowed in an index?
Returns `500`.

`getMaxColumnsInOrderBy()`

What's the maximum number of columns in an `"ORDER BY"` clause?
Returns `500`.

`getMaxColumnsInSelect()`

What's the maximum number of columns in a `"SELECT"` list?
Returns `500`.

`getMaxColumnsInTable()`

What's the maximum number of columns in a table?
Returns `500`.

`getMaxConnections()`

How many active connections can we have at a time to this database?
Returns `10`.

`getMaxCursorNameLength()`

What's the maximum cursor name length?
Returns `32`.

`getMaxIndexLength()`

What's the maximum length of an index (in bytes)?
Returns `500*2000`.

`getMaxProcedureNameLength()`

What's the maximum length of a procedure name?

Returns `32`.

`getMaxRowSize()`

What's the maximum length of a single row?

Returns `2000` .* 500

`getMaxSchemaNameLength()`

What's the maximum length allowed for a schema name?

Returns `32`.

`getMaxStatementLength()`

What's the maximum length of a SQL statement?

Returns `10000`.

`getMaxStatements()`

How many active statements can we have open at one time to this database?

Returns `100`.

`getMaxTableNameLength()`

What's the maximum length of a table name?

Returns `32`.

`getMaxTablesInSelect()`

What's the maximum number of tables in a `SELECT`?

Returns `100`.

`getMaxUserNameLength()`

What's the maximum length of a user name?

Returns `32`.

`getNumericFunctions()`

Get a comma separated list of math functions.

Returns `SQL_FN_NUM_ABS, SQL_FN_NUM_ACOS, SQL_FN_NUM_ASIN, SQL_FN_NUM_ATAN, SQL_FN_NUM_ATAN2, SQL_FN_NUM_CEILING, SQL_FN_NUM_COS, SQL_FN_NUM_DEGREES, SQL_FN_NUM_EXP, SQL_FN_NUM_COS, SQL_FN_NUM_DEGREES, SQL_FN_NUM_EXP, SQL_FN_NUM_FLOOR, SQL_FN_NUM_LOG10, SQL_FN_NUM_MOD, SQL_FN_NUM_PI, SQL_FN_NUM_POWER, SQL_FN_NUM_RADIANS, SQL_FN_NUM_RAND, SQL_FN_NUM_SIGN, SQL_FN_NUM_SIN, SQL_FN_NUM_SQRT, SQL_FN_NUM_TAN`

getPrimaryKeys(String, String, String)

Get a description of a table's primary key columns.

Returns `(result .set)`

getProcedureColumns(String, String, String, String)

Get a description of a catalog's stored procedure parameters and result columns.

Returns `(result .set)`

getProcedures(String, String, String)

Get a description of stored procedures available in a catalog.

Returns `(result .set)`

getProcedureTerm()

What's the database vendor's preferred term for "procedure"?

Returns `"procedure."`

getSchemas()

Get the schema names available in this database.

Returns `(result .set)`

getSchemaTerm()

What's the database vendor's preferred term for "schema"?

Returns `"Owner."`

getSearchStringEscape()

This is the string that can be used to escape '_' or '%' in the string pattern style catalog search parameters.

Returns `"\."`

getSQLKeywords()

Get a comma separated list of all a database's SQL keywords that are NOT also SQL92 keywords.

Returns `null`.

getStringFunctions()

Get a comma separated list of string functions.

Returns `SQL_FN_STR_ASCII., SQL_FN_STR_LTRIM, SQL_FN_STR_RTRIM, SQL_FN_STR_CONCAT, SQL_FN_STR_LENGTH, SQL_FN_STR_LOCATE`

getSystemFunctions()

Get a comma separated list of system functions.

Returns `SQL_FN_SYS_USERNAME., SQL_FN_SYS_IFNULL, SQL_FN_SYS_DBNAME`

getTablePrivileges(String, String, String)

Get a description of the access rights for each table available in a catalog.

Returns `(result .set)`

getTables(String, String, String, String [])

Get a description of tables available in a catalog.

Returns `(result .set)`

getTableTypes()

Get the table types available in this database.

Returns `(result .set)`

getTimeDateFunctions()

Get a comma separated list of time and date functions.

Returns `SQL_FN_TD_DAYOFMONTH., SQL_FN_TD_DAYOFWEEK, SQL_FN_TD_DAYOFYEAR, SQL_FN_TD_MONTH, SQL_FN_TD_QUARTER, SQL_FN_TD_WEEK, SQL_FN_TD_YEAR, SQL_FN_TD_HOUR, SQL_FN_TD_MINUTE, SQL_FN_TD_SECOND`

getTypeInfo()

Get a description of all the standard SQL types supported by this database.

Returns `(result .set)`

getURL()

What's the url for this database?

Returns `(the .URL)`

getUserName()

What's our user name as known to the database?

Returns `(the .userid)`

getVersionColumns(String, String, String)

Get a description of a table's columns that are automatically updated when any value in a row is updated.

Returns `(result .set)`

`isCatalogAtStart()`

Does a catalog appear at the start of a qualified table name? (Otherwise it appears at the end)

Returns `True`.

`isReadOnly()`

Is the database in read-only mode?

Returns `False`.

`nullPlusNonNullIsNull()`

Are concatenations between NULL and non-NULL values NULL? A JDBC-Compliant driver always returns true.

Returns `True`.

`nullsAreSortedAtEnd()`

Are NULL values sorted at the end regardless of sort order?

Returns `False`.

`nullsAreSortedAtStart()`

Are NULL values sorted at the start regardless of sort order?

Returns `False`.

`nullsAreSortedHigh()`

Are NULL values sorted high?

Returns `False`.

`nullsAreSortedLow()`

Are NULL values sorted low?

Returns `True`.

`storesLowerCaseIdentifiers()`

Does the database treat mixed case unquoted SQL identifiers as case insensitive and store them in lower case?

Returns `True`.

`storesLowerCaseQuotedIdentifiers()`

Does the database treat mixed case quoted SQL identifiers as case insensitive and store them in lower case?

Returns `False`.

`storesMixedCaseIdentifiers()`

Does the database treat mixed case unquoted SQL identifiers as case insensitive and store them in mixed case?

Returns `False`.

`storesMixedCaseQuotedIdentifiers()`

Does the database treat mixed case quoted SQL identifiers as case insensitive and store them in mixed case?

Returns `False`.

`storesUpperCaseIdentifiers()`

Does the database treat mixed case unquoted SQL identifiers as case insensitive and store them in upper case?

Returns `False`.

`storesUpperCaseQuotedIdentifiers()`

Does the database treat mixed case quoted SQL identifiers as case insensitive and store them in upper case?

Returns `False`.

`supportsAlterTableWithAddColumn()`

Is `"ALTER TABLE"` with add column supported?

Returns `True`.

`supportsAlterTableWithDropColumn()`

Is `"ALTER TABLE"` with drop column supported?

Returns `False`.

`supportsANSI92EntryLevelSQL()`

Is the ANSI92 entry level SQL grammar supported? All JDBC-Compliant drivers must return true.

Returns `True`.

supportsANSI92FullSQL()

Is the ANSI92 full SQL grammar supported?

Returns `False`.

supportsANSI92IntermediateSQL()

Is the ANSI92 intermediate SQL grammar supported?

Returns `False`.

supportsCatalogsInDataManipulation()

Can a catalog name be used in a data manipulation statement?

Returns `False`.

supportsCatalogsInIndexDefinitions()

Can a catalog name be used in an index definition statement?

Returns `False`.

supportsCatalogsInPrivilegeDefinitions()

Can a catalog name be used in a privilege definition statement?

Returns `False`.

supportsCatalogsInProcedureCalls()

Can a catalog name be used in a procedure call statement?

Returns `False`.

supportsCatalogsInTableDefinitions()

Can a catalog name be used in a table definition statement?

Returns `False`.

supportsColumnAliasing()

Is column aliasing supported? If so, the SQL AS clause can be used to provide names for computed columns or to provide alias names for columns as required.

Returns `True`.

supportsConvert()

Is the `CONVERT` function between SQL types supported?

Returns `True`.

supportsConvert(int, int)

Is CONVERT between the given SQL types supported?

Returns True.

supportsCoreSQLGrammar()

Is the ODBC Core SQL grammar supported?

Returns True.

supportsCorrelatedSubqueries()

Are correlated subqueries supported? A JDBC-Compliant driver always returns true.

Returns True.

supportsDataDefinitionAndDataManipulationTransactions ()

Are both data definition and data manipulation statements within a transaction supported?

Returns True.

supportsDataManipulationTransactionsOnly()

Are only data manipulation statements within a transaction supported?

Returns False.

supportsDifferentTableCorrelationNames()

If table correlation names are supported, are they restricted to be different from the names of the tables?

Returns False.

supportsExpressionsInOrderBy()

Are expressions in "ORDER BY" lists supported?

Returns False.

supportsExtendedSQLGrammar()

Is the ODBC Extended SQL grammar supported?

Returns False.

supportsFullOuterJoins()

Are full nested outer joins supported?

Returns False.

supportsGroupBy()

Is some form of "GROUP BY" clause supported?

Returns `True`.

supportsGroupByBeyondSelect()

Can a "`GROUP BY`" clause add columns not in the `SELECT` provided it specifies all the columns in the `SELECT`?

Returns `False`.

supportsGroupByUnrelated()

Can a "`GROUP BY`" clause use columns not in the `SELECT`?

Returns `False`.

supportsIntegrityEnhancementFacility()

Is the SQL Integrity Enhancement Facility supported?

Returns `False`.

supportsLikeEscapeClause()

Is the escape character in "`LIKE`" clauses supported? A JDBC-Compliant driver always returns true.

Returns `True`.

supportsLimitedOuterJoins()

Is there limited support for outer joins? (This will be true if `supportFullOuterJoins` is true.)

Returns `True`.

supportsMinimumSQLGrammar()

Is the ODBC Minimum SQL grammar supported? All JDBC-Compliant drivers must return true.

Returns `True`.

supportsMixedCaseIdentifiers()

Does the database treat mixed case unquoted SQL identifiers as case sensitive and as a result store them in mixed case? A JDBC-Compliant driver will always return false.

Returns `False`.

supportsMixedCaseQuotedIdentifiers()

Does the database treat mixed case quoted SQL identifiers as case sensitive and as a result store them in mixed case? A JDBC-Compliant driver will always return true.

Returns `True`.

`supportsMultipleResultSets()`

Are multiple ResultSets from a single execute supported?

Returns `False`.

`supportsMultipleTransactions ()`

Can we have multiple transactions open at once (on different connections)?

Returns `True`.

`supportsNonNullableColumns()`

Can columns be defined as non-nullable? A JDBC-Compliant driver always returns true.

Returns `True`.

`supportsOpenCursorsAcrossCommit()`

Can cursors remain open across commits?

Returns `False`.

`supportsOpenCursorsAcrossRollback()`

Can cursors remain open across rollbacks?

Returns `False`.

`supportsOpenStatementsAcrossCommit()`

Can statements remain open across commits?

Returns `False`.

`supportsOpenStatementsAcrossRollback()`

Can statements remain open across rollbacks?

Returns `False`.

`supportsOrderByUnrelated()`

Can an `"ORDER BY"` clause use columns not in the `SELECT`?

Returns `True`.

`supportsOuterJoins()`

Is some form of outer join supported?
Returns `True`.

`supportsPositionedDelete()`

Is positioned `DELETE` supported?
Returns `True`.

`supportsPositionedUpdate()`

Is positioned `UPDATE` supported?
Returns `True`.

`supportsSchemasInDataManipulation()`

Can a schema name be used in a data manipulation statement?
Returns `True`.

`supportsSchemasInIndexDefinitions()`

Can a schema name be used in an index definition statement?
Returns `True`.

`supportsSchemasInPrivilegeDefinitions()`

Can a schema name be used in a privilege definition statement?
Returns `True`.

`supportsSchemasInProcedureCalls()`

Can a schema name be used in a procedure call statement?
Returns `True`.

`supportsSchemasInTableDefinitions()`

Can a schema name be used in a table definition statement?
Returns `True`.

`supportsSelectForUpdate()`

Is `SELECT` for `UPDATE` supported?
Returns `True`.

`supportsStoredProcedures()`

Are stored procedure calls using the stored procedure escape syntax supported?

Returns `True`.

`supportsSubqueriesInComparisons()`

Are subqueries in comparison expressions supported? A JDBC-Compliant driver always returns true.

Returns `True`.

`supportsSubqueriesInExists()`

Are subqueries in 'exists' expressions supported? A JDBC-Compliant driver always returns true.

Returns `True`.

`supportsSubqueriesInIns()`

Are subqueries in 'in' statements supported? A JDBC-Compliant driver always returns true.

Returns `True`.

`supportsSubqueriesInQuantifieds()`

Are subqueries in quantified expressions supported? A JDBC-Compliant driver always returns true.

Returns `True`.

`supportsTableCorrelationNames()`

Are table correlation names supported? A JDBC-Compliant driver always returns true.

Returns `True`.

`supportsTransactionIsolationLevel(int)`

Does the database support the given transaction isolation level?

Returns `True`.

`supportsTransactions ()`

Are transactions supported? If not, commit is a no-op and the isolation level is `TRANSACTION_NONE`.

Returns `True`.

`supportsUnion()`

Is `SQL UNION` supported?

Returns `True`.

`supportsUnionAll()`

Is `SQL UNION ALL` supported?

Returns `True`.

`usesLocalFilePerTable()`

Does the database use a file for each table?

Returns `False`.

`usesLocalFiles()`

Does the database store tables in a local file?

Returns `False`.

Error Messages

The error messages are generated by the driver, along with associated `SQLSTATE` and Versant error code values.

For more information, please refer Chapter 13 “Error Messages” on page 415.

SAMPLE PROGRAM SOURCE CODE

JDBCTest.java

```
//
// Description:Test program for JDBC API interface. This Java
// application will connect to a Versant JDBC driver, issue a
// select statement and display all result columns and rows
//
package com.versant.jdbc;
import java.net.URL;
import java.sql.*;
import java.io.*;
class  VersantTest
{
  protected static final boolean debugFlag = true;
  protected static final PrintStream o = System.out;
  protected static final String ColString = "COLUMNS";
  protected static final String TblString = "TABLES";
  protected static final String ViewString = "VIEWS";
  protected static final String CallString = "CALL";
  protected static final int MAX_SQLSTMTLEN = 1000;
```

```
public static void main (String args[])
{
  try
  {
    String url  = null;
    String userid = null;
    String passwd = null;
    String query  = null;

    switch (args.length)
    {
    case 0:
      url   = "jdbc:Versant:T:isis:testdb:5020";
      userid= "systpe";
      passwd= "dummy";
      break;
    case 1:
      url   = args[0];
      userid= "systpe";
      passwd= "dummy";
      break;
    case 2:
```

```
    url   = args[0];
    userid= args[1];
    passwd= "dummy";
    break;
  case 3:
    url   = args[0];
    userid= args[1];
    passwd= args[2];
    break;
  default:
    System.out.println("\nUsage: Java/jview VersantTest " +
                " [[[<url>]  <userid>] <passwd>] ");

    System.exit(0);
  }

  // Load the driver
  Class.forName ("com.versant.jdbc.VersantDriver");
  java.io.PrintStream    pStream = null;

  if (debugFlag)
```

```
{
  //  Create  a PrintStream using System.out
  pStream = new java.io.PrintStream(System.out, true);
}
else
{
  //  Create  PrintStream using a file.
  java.io.FileOutputStream outFile   =
    new java.io.FileOutputStream("dhjdbc.log");
  pStream   = new java.io.PrintStream(outFile, true);
}
// Enable JDBC tracing
DriverManager.setLogStream(pStream);

// Attempt to connect to a driver. Each one
// of the registered drivers will be loaded until
// one is found that can process this URL
java.util.Properties  prop = new java.util.Properties();
prop.put("user", userid);
prop.put("password", passwd);

// We have to add any other options as additional
```

```
// properties in the prop argument.
// e.g., prop.put\("Caller", "VersantTest"\);
Connection con = DriverManager.getConnection (
      url, prop);

// If we were unable to connect, an exception
// would have been thrown. So, if we get here,
// we are successfully connected to the URL

// Check for, and display any warnings generated
// by the connect.
checkForWarning (con.getWarnings ());
// o.println("\nConnected to " + url);

// Get the DatabaseMetaData object and display
// some information about the connection
DatabaseMetaData dma = con.getMetaData ();
o.println("\nConnected to " + dma.getURL());
o.println("Driver     " + dma.getDriverName());
o.println("Version    " + dma.getDriverVersion());
o.println("");
```

```
byte [] bArray = new byte [MAX_SQLSTMTLEN + 1];
boolean       rs_exists = false;
ResultSet     rs   = null;
PreparedStatement pstmt = null;
CallableStatement callstmt = null;
while (true)
{
  o.print ("DhTest> ");
  int l = System.in.read (bArray, 0, MAX_SQLSTMTLEN);
  query = new String (bArray, 0, 0, l);
  query = query.trim();

  if (query.endsWith(";"))
  {
    query = query.substring(0, query.length() - 1);
    query = query.trim();
  }

  if (query.length() == 0)
    continue;
  if (query.equalsIgnoreCase("quit"))
    break;
```

```
// Execute the statement.
try
{
  if (tablesCmd(query))
  {
    String [] types = {"TABLE","SYSTEM TABLE"};
    String   t_patrn= query.substring(TblString.length());
    t_patrn = t_patrn.trim();
    if (t_patrn.length() == 0)
      rs = dma.getTables("","","%",types);
    else
      rs = dma.getTables("","",t_patrn,types);
    rs_exists = true;
  } else if (viewsCmd(query) == true)
  {
    String [] types = {"VIEW"};
    String   v_patrn= query.substring(ViewString.length());
    v_patrn = v_patrn.trim();
    if (v_patrn.length() == 0)
      rs = dma.getTables("","","%",types);
    else
```

```
      rs = dma.getTables("","",v_patrn,types);
    rs_exists = true;
  } else if (columnsCmd(query) == true)
  {
    String   t_patrn= query.substring(ColString.length());
    t_patrn = t_patrn.trim();
    if (t_patrn.length() == 0)
    {
      //rs = dma.getColumns("","","%","%");
      o.println("Specify a table pattern");
      continue;
    }
    else
      rs = dma.getColumns("","",t_patrn,"%");
    rs_exists = true;
  } else if (callStmt(query) == true)
  {
    callstmt = con.prepareCall(query);
    setCallParams(callstmt);
    rs_exists = callstmt.execute();
    if (rs_exists == true)
    {
```

```
      rs = callstmt.getResultSet();
    } else
    {
      int updCount = callstmt.getUpdateCount();
      o.println( "RowCount is " + updCount);
    }
  } else if (query.equalsIgnoreCase("types"))
  {
    rs = dma.getTypeInfo();
    rs_exists = true;
  }else
  {
    pstmt = con.prepareStatement(query);
    setParams(pstmt);
    rs_exists = pstmt.execute();
    if (rs_exists == true)
    {
      rs = pstmt.getResultSet();
    } else
    {
      int updCount = pstmt.getUpdateCount();
```

```
        o.println( "RowCount is " + updCount);
      }
    }
    if (rs_exists == true)
    {
      // Display all columns and rows from the result set
      dispResultSet (rs);
      rs.close();
      rs_exists = false;
    }
  }
  catch (SQLException ex)
  {
    o.println(ex.getMessage());
    continue;
  }
  // Close the statement
  if (pstmt != null)
    pstmt.close();
  pstmt = null;
  if (callstmt != null)
    callstmt.close();
```

```
      callstmt = null;
    }
    // Close the connection
    con.close();
  }
  catch (SQLException ex)
  {
    // A SQLException was generated. Catch it and
    // display the error information. Note that there
    // could be multiple error objects chained
    // together
    while (ex != null)
    {
      o.println ("SQLState: " +
                 ex.getSQLState ());
      o.println ("Message:  " +
                 ex.getMessage ());
      o.println ("VendorCode:   " +
                 ex.getErrorCode ());
      ex = ex.getNextException ();
      o.println ("");
```

```
    }
  }
  catch (java.lang.Exception ex)
  {
    // Got some other type of exception. Dump it.
    ex.printStackTrace ();
  }
  finally
  {
    o.println ("@VersantTest:finally");
  }
}
//------------------------------------------------------------------------
// checkForWarning
// Checks for and displays warnings. Returns true if a warning
// existed
//------------------------------------------------------------------------
private static boolean checkForWarning (SQLWarning warn)
  throws SQLException
  {
    boolean rc = false;
    // If a SQLWarning object was given, display the
```

```
        // warning messages. Note that there could be
        // multiple warnings chained together
        if (warn != null)
        {
          o.println ("\n *** Warning ***\n");
          rc = true;
          while (warn != null)
          {
            o.println ("SQLState: " +
            warn.getSQLState ());
            o.println ("Message:  " +
            warn.getMessage ());
            o.println ("Vendor:   " +
            warn.getErrorCode ());
            o.println ("");
            warn = warn.getNextWarning ();
          }
        }
        return rc;
      }
                                                                    //
  ------------------------------------------------------------------
```

```
// dispResultSet
// Displays all columns and rows in the given result set
//---------------------------------------------------------------------
private static void dispResultSet (ResultSet rs)
  throws SQLException
{
    int i,j;
    // Get the ResultSetMetaData. This will be used for
    // the column headings
    ResultSetMetaData rsmd = rs.getMetaData ();
    // Get the number of columns in the result set
    int numCols = rsmd.getColumnCount ();
    // Display column headings
    for (i=1; i<=numCols; i++)
    {
      if (i > 1)
        o.print(",");
      String label  = rsmd.getColumnLabel(i);
      o.print(label);
    }
    o.println();
    for (i=1; i<=numCols; i++)
```

```
{
  if (i > 1)
    o.print("-");
  String label  = rsmd.getColumnLabel(i);
  for (j = 0; j < label.length(); j++)
    o.print("-");
}
o.println();
// Display data, fetching until end of the result set
while (rs.next ())
{
  // Loop through each column, getting the
  // column data and displaying
  for (i=1; i<=numCols; i++)
  {
    if (i > 1) System.out.print(",");
    o.print(rs.getString (i));
  }
  o.println("");
  // Fetch the next result set row
}
```

```
}
//---------------------------------------------------------------------
// tablesCmd returns true if its a Tables Command
// else false
//---------------------------------------------------------------------
private static boolean tablesCmd (String query)
{
  String u_query = query.toUpperCase();
  if ((u_query.startsWith(TblString)) ||
     (u_query.startsWith(TblString + " ")))
    return true;
  return  false;
}
//---------------------------------------------------------------------
// ViewsCmd returns true if its a Views Command
// else false
//---------------------------------------------------------------------
private static boolean viewsCmd (String query)
{

  String u_query = query.toUpperCase();
  if ((u_query.startsWith(ViewString)) ||
```

```
      (u_query.startsWith(ViewString + " ")))
      return true;
    return  false;
  }
                                                                    //
-------------------------------------------------------------------
  // columnsCmd returns true if its a Columns Command
  // else false
                                                                    //
-------------------------------------------------------------------
  private static boolean columnsCmd (String query)
  {

    String u_query = query.toUpperCase();
    if ((u_query.startsWith(ColString)) ||
      (u_query.startsWith(ColString + " ")))
      return true;
    return  false;
  }
                                                                    //
-------------------------------------------------------------------
  // CallStmt returns true if its a Procedure Call Statement
  // else false
```

```
//---------------------------------------------------------------------
private static boolean callStmt (String query)
{
  String u_query = query.toUpperCase();
  if ((u_query.startsWith(CallString)) ||
    (u_query.startsWith(CallString + " ")))
    return true;
  return  false;
}
//---------------------------------------------------------------------
// setParams prompts for parameters and sets them.
//---------------------------------------------------------------------
private static void  setParams (PreparedStatement stmt)
  throws SQLException
  {
    byte [] bArray  = new byte [MAX_SQLSTMTLEN + 1];
    String param = null;
    int paramCount = ((VersantPreparedStatement)stmt).getNparams();
    if (paramCount > 0)
      o.println("Parameters Required = " + paramCount );
    try
    {
```

```
            for (int i = 0; i < paramCount; i++)
            {
              o.print ("type parameter no " + i +" : ");
              int l = System.in.read (bArray, 0, MAX_SQLSTMTLEN);

              param = new String (bArray, 0, 0, l);
              param = param.trim();
              if (param.equalsIgnoreCase("null"))
                stmt.setNull(i+1, java.sql.Types.CHAR);
              else
                stmt.setString (i+1, param);
            }
          }

          catch (java.lang.Exception ex)
          {
            // Got some other type of exception. Dump it.
            ex.printStackTrace ();
          }
        }
                                                                        //
    ------------------------------------------------------------------
```

```
// setCallParams prompts for parameters and sets them.
//---------------------------------------------------------------------
private static void  setCallParams (CallableStatement stmt)
  throws SQLException
  {
    byte [] bArray  = new byte [MAX_SQLSTMTLEN + 1];
    String param = null;
    int paramCount = ((VersantCallableStatement)stmt).getNparams();
    if (paramCount > 0)
      o.println("Parameters Required = " + paramCount );
    try
    {
      for (int i = 0; i < paramCount; i++)
      {
        o.println ("Parameter " + i );
        while (true)
        {
          o.print ("  Type IN , OUT or INOUT : ");
          int l = System.in.read (bArray, 0, MAX_SQLSTMTLEN);

          String paramtype = new String (bArray, 0, 0, l);
          paramtype = paramtype.trim();
```

```
paramtype = paramtype.toUpperCase();
if (paramtype.equalsIgnoreCase("IN"))
{
  o.print ("  Type ParamValue : ");
  l = System.in.read (bArray, 0, MAX_SQLSTMTLEN);

  String paramvalue = new String (bArray, 0, 0, l);
  paramvalue = paramvalue.trim();
  if (paramvalue.equalsIgnoreCase("null"))
    stmt.setNull(i+1, java.sql.Types.CHAR);
  else
    stmt.setString (i+1, paramvalue);
  break;
} else if (paramtype.equalsIgnoreCase("INOUT"))
{
  o.print ("  Type ParamValue : ");
  l = System.in.read (bArray, 0, MAX_SQLSTMTLEN);

  String paramvalue = new String (bArray, 0, 0, l);
  paramvalue = paramvalue.trim();
  if (paramvalue.equalsIgnoreCase("null"))
```

```
            stmt.setNull(i+1, java.sql.Types.CHAR);
          else
            stmt.setString (i+1, paramvalue);
          stmt.registerOutParameter (i+1,
                        java.sql.Types.CHAR);
          break;
        } else if (paramtype.equalsIgnoreCase("OUT"))
        {
          stmt.registerOutParameter (i+1,
                        java.sql.Types.CHAR);
          break;
        }
      }
    }
  }

  catch (java.lang.Exception ex)
  {
    // Got some other type of exception. Dump it.
    ex.printStackTrace ();
  }
}
```

```
    }
```

CHAPTER 12

Performance Improvement Tips

This Chapter provides tips to improve the query performance.

The Chapter explains following in detail:

- Usage Notes
- Optimizing certain non-intuitive Queries
- De-normalization

USAGE NOTES

Query performance can be achieved at three levels:

- Database query engine implementer
- Database administrator
- Database application developer

Following addresses various items that can be performed with respect to Database administrator and Database application developer inorder to improve query performance:

- Versant ODBMS does not allow to create composite indexes with more than one field (attributes). Create as many meaningful indices as possible. You can create either `btree` or `hash` indices. Having many indices can improve the performance of the queries with multiple joins.
- Use Versant ODBMS clusters to store objects as it helps in improving performance.

  For more details refer to the Versant ODBMS documentation.

- Maintain the created indices.

  The Versant indices are maintained automatically. Maintaining indices also mean:

  - The `dba` should have a look at the indices periodically.
  - Drop the ones which are less used
  - Create new indices after analyzing the patterns of query processing requests and indices usages.

- Use `UPDATE STATISTICS` command to update the maintained statistical information. This helps query optimizer to use the most recent information to choose a plan based on this information. You can issue this on tables which have indexes defined on them. This particular command is an extension to SQL standard. The command will have maximum affect when you run this statement periodically in the background, when there is no or less activity on the Versant ReVind server. On large database (that has a large number of tables), this statement would take more time to complete. You can choose to update the statistics for selective few tables or classes which are frequently used. Alternatively, you can choose to update statistics for all tables and classes in the database.
- Tuning main-memory system parameters.

  The SQL engine uses main memory system for volatile data such as temporary tables and dynamic indices. This improves the performance of many queries such as joins. Depending on the amount of memory available on a system, certain queries may create very large

temporary tables to be stored in memory. In such cases, the main memory storage system swaps blocks of data to a disk file as required. On operating systems such as MS-Windows, the user can set the virtual memory size. You can tune this to a higher value so that here is lower disk i/o.

Following environment variables controls the characteristics of the usage of main memory to create temporary tables:

```
TPE_MM_SWAPSIZE
```

Specifies the size in kilobytes of swap file that the SQL engine uses, when it writes to the disk from the main memory cache. The default is 40,000KB. If you have more disk space, increasing the value will be beneficial.

```
TPE_MM_CACHESIZE
```

Specifies the size, in kilobytes, of the main memory cache used for temporary tables. The default value is 1MB. The SQL engine uses the cache for storing temporary tables for sorting and creating dynamic indices during processing. Increase the values in queries where the SQL engine needs to create temporary tables for statements like `UPDATE STATISTICS`.

- Use the single-process `'dhserver'` server wherein there is no separate Versant ODBMS server process. It only works when Versant ReVind is co-located with the database.
- For JDBC related performance improvement, use the java options with fine-tuned values:

`-ss<number>`	set the maximum native stack size for any thread
`-oss<number>`	set the maximum Java stack size for any thread
`-ms<number>`	set the initial Java heap size
`-mx<number>`	set the maximum Java heap size

Use the following command to start the JDBC driver daemon:

```
java versant.jdbc.JDBCDaemon
```

- Make use of `VQNeededClasses` class to speed-up the startup of the `DHSERVER` or `MISQL` sessions.

For more details, please refer to the section “Utility Tables” on page 23 in "Chapter 1 - SQL-to-Object Mapping".

- Use the query with `count(*)` aggregate functions cautiously. Please note that, SQL engine has to fetch the records and count them. Simple queries involving `count(*)`, such as:

```
select count(*) from employee
```

are optimized to return the number of objects without actually fetching them. In such cases, the SQL engine makes use of the cardinality information available. If you have a query such as

`select count(*)` from employee where salary > 100000 will not be able to make use of cardinality information directly. Here it will have to fetch the records satisfying the condition in the `WHERE` clause of the query and return the count. This could be time consuming.

- If the join order specified in a query is guaranteed to be optimized and you do not want the optimizer to work on join order optimization. You can use SQL syntax extension to achieve this:

  ```
  {versant ORDERED}
  ```

  This directs the SQL engine to join the tables in the order specified. Usage of this clause will override SQL engines join order optimization. This is useful in special cases when you know that a particular join order will give good results. Since this clause bypasses join order optimization, carefully test the queries that use it, to ensure the specified join order is faster than relying on the optimizer.

NOTE:- The braces are part of the required syntax and not a syntax convention.

For Example:

```
select sc.tbl, sc.coltype from syscolumns sc, systables st
{ versant ORDERD }
where sc.tbl = st.tbl and st.tyltype = ‘S’;
```

- Sometimes a query is corrected by changing the syntax. Watch for deferred mode updates, scalar and vector aggregates, nested iterations, sub-queries, sorts, and table scans.
- Use the SET DISPLAY COST to look at the costs of a query as evaluated by Versant ReVind, analyze the query displayed, and try to rearrange to produce better running query.

  For more details refer to the section “SET” on page 98 in "Chapter 3 - Versant Interactive SQL Tool".
- If you notice performance problems while using the ODBC- compliant tools such as MS-Access, refer to the vendor's documentation or help file for tips on usage with ODBC databases.
- Use the ODBC logging for analysis of the performance problems. One can use ODBC Driver Manager application Control Panel to set the ODBC tracing ON or OFF. For example, while using ODBC driver with MS-Access, and trying to connect to database as non-DBA, the queries can get slower. This is because MS-Access needs some special information in MsysConf table in the database. It queries this table numerous times when a non-dba user tries to access the database using MS-Access.

- Try to create indexes using `sqlcrtidx` with `btree` option. Avoid using hashed indexes. This can boost the `sort` optimization.

OPTIMIZING CERTAIN NON-INTUITIVE QUERIES

This section gives few examples of performance improvement that can be achieved by converting or rewriting “intuitive” queries to a different form which may appear “non-intuitive”.

'greater than' queries

This query with an index on `int_col`,

select * from table where int_col >3;

uses the index to find the first value that equals to 3. Next it scans forward to find the first value greater than 3. If there are many rows where `int_col` equals 3, the server has to scan numerous pages to find the first row where `int_col` is greater than 3.

It would probably be much more efficient to write the query like:

```
select * from table where int_col >= 4
```

This optimization is easier with integers.

NOT EXISTS tests

In subqueries, queries with clauses `EXISTS` and `IN` perform faster than `NOT EXISTS` and `NOT IN` when the values in `WHERE` clause are not indexed. For `EXISTS` and `IN`, clauses server can return true as soon as a single row matches. For NOT expressions, it has to match all the values.

COUNT vs. EXISTS

Do not use count aggregate in a subquery to do an existence check.

```
select * from table where 0 < (select count(*) from
table2)
```

Instead use `exists` or `in`:

```
select * from table where exists(select * from table2)
```

In the first query, it counts all matching rows. In the second query, it returns as soon as a single row matches, thus making the query faster.

DE-NORMALIZATION

Here are some de-normalization techniques that can help queries run faster. Normalization does not always mean faster queries. When de-normalization techniques are used, data integrity management techniques need to be incorporated at various levels.

- Adding derived columns can help eliminate joins and reduce the time needed to produce the aggregate values.
- Duplicating tables (classes). If a group of users regularly need only a subset of data, you can duplicate the critical table subset for that group.
- Queries can be structured to make use of the `SELFOID` column. Queries specifying joins that result in navigation are more efficient than those that do not.

CHAPTER 13

Error Messages

This Chapter lists the error messages generated by the various components of Versant ReVind.

The Chapter describes the following in detail:

- Error Messages Overview
- Versant ReVind Error Codes

ERROR MESSAGES OVERVIEW

You can receive error messages not only from the Versant ReVind engine, but also from underlying storage systems, including those Versant supplies for possible use with Versant ReVind implementations.
Error conditions associated with Versant components (and storage systems supplied by Versant) use the following ranges of error codes:

Error Number	Used By
15000 - 15999	Errors specific to the Versant ReVind flat-file storage system
16000 - 16999	Errors specific to the Versant ReVind main-memory storage system
20000 - 20999	Versant ReVind engine error message
25000 - 25999	Versant Integrator error messages
27181 – 27186	Versant ReVind Licensing Error
30000 - 30999	Versant ReVind network error messages

In addition to the Versant ReVind-specific error codes, error conditions have an associated `SQLSTATE` value. `SQLSTATE` is a 5-character status parameter whose value indicates the condition status returned by the most recent SQL statement. The first two characters of the `SQLSTATE` value specify the class code and the last three characters specify the subclass code.

NOTE:- Class codes of `a-h` and `0-4` are reserved by the SQL standard. For those class codes only, subclass codes of a-h and `0-4` are also reserved by the standard.

Class codes of `i-z` and `5-9` are specific to database implementations such as Versant ReVind. All subclass codes in those classes are implementation defined.

The file that contains the error message text and codes is `$TPEROOT/lib/dherrors` and `$TPEROOT/lib/dherrors_cust`.
Other storage systems on which Versant ReVind is implemented will likely generate their own error messages. See the documentation for your storage system for details.

VERSANT REVIND ERROR CODES

Error Codes, SQLSTATE Values, and Messages

The following table lists the Versant ReVind error messages, ordered by error code number, and shows the corresponding SQLSTATE values for each message:

Error Code	SQLSTATE Value	Class Condition	Subclass Message
00000	00000	Successful completion	***status okay
100L	02000	no data	**sql not found.
15001	60601	Versant ReVind ff errors	FF- File IO error
15002	60602	Versant ReVind ff errors	FF- No more records
15003	42603	Access violation error	FF- Table already exists
15004	22604	Data exception	FF- Invalid record number
15005	60605	Versant ReVind ff errors	FF- Record deleted
15006	60606	Versant ReVind ff errors	FF- Invalid type
15007	60607	Versant ReVind ff errors	FF- Duplicate value
15008	08608	Connection exception	FF- Database exists
15009	08609	Connection exception	FF- No database found
15010	60610	Versant ReVind ff errors	FF- Version mis-match
15011	60611	Versant ReVind ff errors	FF- Virtual file cache exceeded
15012	60612	Versant ReVind ff errors	FF- Physical file open error
15013	60613	Versant ReVind ff errors	FF- Corrupt virtual file handle
15014	22614	Data exception	FF- Overflow error
16001	22701	Data exception	MM- No data block
16002	70702	Versant ReVind MM errors	MM- Bad swap block
16003	70703	Versant ReVind MM errors	MM- No cache block
16004	22704	Data exception	MM- Invalid row number
16005	70705	Versant ReVind MM errors	MM- Invalid cache block

16006	70706	Versant ReVind MM errors	MM- Bad swap file
16007	70707	Versant ReVind MM errors	MM- Row too big
16008	70708	Versant ReVind MM errors	MM- Array initialized
16009	70709	Versant ReVind MM errors	MM- Invalid chunk number
16010	70710	Versant ReVind MM errors	MM- Can't create table
16011	70711	Versant ReVind MM errors	MM- Can't alter table
16012	70712	Versant ReVind MM errors	MM- Can't drop table
16020	70713	Versant ReVind MM errors	MM- TPL ctor error
16021	70714	Versant ReVind MM errors	MM- Insertion error
16022	70715	Versant ReVind MM errors	MM- Deletion error
16023	70716	Versant ReVind MM errors	MM- Updation error
16024	70717	Versant ReVind MM errors	MM- Fetching error
16025	70718	Versant ReVind MM errors	MM- Sorting error
16026	70719	Versant ReVind MM errors	MM- Printing error
16027	70720	Versant ReVind MM errors	MM- TPLSCAN ctor error
16028	70721	Versant ReVind MM errors	MM- Scan fetching error
16030	70722	Versant ReVind MM errors	MM- Can't create index
16031	70723	Versant ReVind MM errors	MM- Can't drop index
16032	70724	Versant ReVind MM errors	MM- IXSCAN ctor error
16033	70725	Versant ReVind MM errors	MM- IX ctor error
16034	70726	Versant ReVind MM errors	MM- IX deletion error
16035	70727	Versant ReVind MM errors	MM- IX appending error
16036	70728	Versant ReVind MM errors	MM- IX insertion error
16037	70729	Versant ReVind MM errors	MM- IX scan fetching error
16040	70730	Versant ReVind MM errors	MM- Begin transaction
16041	70731	Versant ReVind MM errors	MM- Commit transaction
16042	40000	Transaction rollback	***MM- Rollback transaction
16043	70732	Versant ReVind MM errors	MM- Mark point
16044	70733	Versant ReVind MM errors	MM- Rollback savepoint

16045	70734	Versant ReVind MM errors	MM- Set & Get isolation
16050	70735	Versant ReVind MM errors	MM- TID to char
16051	70736	Versant ReVind MM errors	MM- char to TID
20000	50501	Versant ReVind rds error	SQL internal error
20001	50502	Versant ReVind rds error	Memory allocation failure
20002	50503	Versant ReVind rds error	Open database failed
20003	2a504	Syntax error	Syntax error
20004	28505	Invalid auth specs	User not found
20005	22506	Data exception	Table/View/Synonym not found
20006	22507	Data exception	Column not found/specified
20007	22508	Data exception	No columns in table
20008	22509	Data exception	Inconsistent types
20009	22510	Data exception	Column ambiguously specified
20010	22511	Data exception	Duplicate column specification
20011	22512	Data exception	Invalid length
20012	22513	Data exception	Invalid precision
20013	22514	Data exception	Invalid scale
20014	22515	Data exception	Missing input parameters
20015	22516	Data exception	Subquery returns multiple rows
20016	22517	Data exception	Null value supplied for a mandatory (not null) column
20017	22518	Data exception	Too many values specified
20018	22519	Data exception	Too few values specified
20019	50520	Versant ReVind rds error	Can not modify table referred to in subquery
20020	42521	Access rule violation	Bad column specification for group by clause

20021	42522	Access rule violation	Non-group-by expression in having clause
20022	42523	Access rule violation	Non-group-by expression in select clause
20023	42524	Access rule violation	Aggregate function not allowed here
20024	0a000	feature not supported	Sorry, operation not yet implemented
20025	42526	Access rule violation	Aggregate functions nested
20026	50527	Versant ReVind rds error	Too many table references
20027	42528	Access rule violation	Bad field specification in order by clause
20028	50529	Versant ReVind rds error	An index with the same name already exists
20029	50530	Versant ReVind rds error	Index referenced not found
20030	22531	Data exception	Table space with same name already exists
20031	50532	Versant ReVind rds error	Cluster with same name already exists
20032	50533	Versant ReVind rds error	No cluster with this name
20033	22534	Data exception	Tablespace not found
20034	50535	Versant ReVind rds error	Bad free percentage specification
20035	50536	Versant ReVind rds error	At least column spec or null clause should be specified
20036	07537	Dynamic sql-error	Statement not prepared
20037	24538	Invalid cursor state	Executing select statement
20038	24539	Invalid cursor state	Cursor not closed
20039	24540	Invalid cursor state	Open for non select statement

20040	24541	Invalid cursor state	Cursor not opened
20041	22542	Data exception	Table/View/Synonym already exists
20042	2a543	Syntax error	Distinct specified more than once in query
20043	50544	Versant ReVind rds error	Tuple size too high
20044	50545	Versant ReVind rds error	Array size too high
20045	08546	Connection exception	File does not exist or not accessible
20046	50547	Versant ReVind rds error	Field value not null for some tuples
20047	42548	Access rule violation	Granting to self not allowed
20048	42549	Access rule violation	Revoking for self not allowed
20049	22550	Data exception	Keyword used for a name
20050	21551	Cardinality violation	Too many fields specified
20051	21552	Cardinality violation	Too many indexes on this table
20052	22553	Data exception	Overflow error
20053	08554	Connection exception	Database not opened
20054	08555	Connection exception	Database not specified or improperly specified
20055	08556	Connection exception	Database not specified or Database not started
20056	28557	Invalid auth specs	No DBA access rights
20057	28558	Invalid auth specs	No RESOURCE privileges
20058	40559	Transaction rollback	Executing SQL statement for an aborted transac-tion
20059	22560	Data exception	No files in the table space
20060	22561	Data exception	Table not empty

20061	22562	Data exception	Input parameter size too high
20062	42563	Syntax error	Full pathname not specified
20063	50564	Versant ReVind rds error	Duplicate file specification
20064	08565	Connection exception	Invalid attach type
20065	26000	Invalid SQL statement name	Invalid statement type
20066	33567	Invalid SQL descriptor name	Invalid sqlda
20067	08568	Connection exception	More than one database can't be attached locally
20068	42569	Syntax error	Bad arguments
20069	33570	Invalid SQL descriptor name	SQLDA size not enough
20070	33571	Invalid SQL descriptor name	SQLDA buffer length too high
20071	42572	Access rule violation	Specified operation not allowed on the view
20072	50573	Versant ReVind rds error	Server is not allocated
20073	2a574	Access rule violation	View query specification for view too long
20074	2a575	Access rule violation	View column list must be specified as expressions are given
20075	21576	Cardinality violation	Number of columns in column list is less than in select list
20076	21577	Cardinality violation	Number of columns in column list is more than in select list
20077	42578	Access rule violation	Check option specified for non-insertable view

20078	42579	Access rule violation	Given SQL statement is not allowed on the view
20079	50580	Versant ReVind rds error	More Tables cannot be created.
20080	44581	Check option violation	View check option violation
20081	22582	Data exception	No of expressions projected on either side of set-op don't match
20082	42583	Access rule violation	Column names not allowed in order by clause for this statement
20083	42584	Access rule violation	Outerjoin specified on a complex predicate
20084	42585	Access rule violation	Outerjoin specified on a sub_query
20085	42586	Access rule violation	Invalid Outerjoin specification
20086	42587	Access rule violation	Duplicate table constraint specification
20087	21588	Cardinality violation	Column count mismatch
20088	28589	Invalid auth specs	Invalid user name
20089	22590	Data exception	System date retrieval failed
20090	42591	Access rule violation	Table columnlist must be specified as expressions are given
20091	2a592	Access rule violation	Query statement too long.
20092	2d593	Invalid transaction termination	No tuples selected by the subquery for update
20093	22594	Data exception	Synonym already exists
20094	hz595	Remote database access	Database link with same name already exists
20095	hz596	Remote database access	Database link not found

20096	08597	Connection exception	Connect String not specified/incorrect
20097	hz598	Remote database access	Specified operation not allowed on a remote table
20098	22599	Data exception	More than one row selected by the query
20099	24000	Invalid cursor state	Cursor not positioned on a valid row
20100	4250a	Access rule violation	Subquery not allowed here
20101	2350b	Integrity constraint	No references for the table
20102	2350c	Integrity constraint	Primary/Candidate key column defined null
20103	2350d	Integrity constraint	No matching key defined for the referenced table
20104	2350e	Integrity constraint	Keys in reference constraint incompatible
20105	5050f	Versant ReVind rds error	Statement not allowed in readonly isolation level
20106	2150g	Cardinality violation	Invalid ROWID
20107	hz50h	Remote database access	Remote Database not started
20108	0850i	Connection exception	Remote Network Server not started.
20109	hz50j	Remote database access	Remote Database Name not valid
20110	0850k	Connection exception	TCP/IP Remote HostName is unknown.
20114	33002	Invalid SQL descriptor name	Fetched Value NULL & indicator var not defined
20115	5050l	Versant ReVind rds error	References to the table/ record present
20116	2350m	Integrity constraint	Constraint violation

20117	2350n	Integrity constraint	Table definition not complete
20118	4250o	Access rule violation	Duplicate constraint name
20119	2350p	Integrity constraint	Constraint name not found
20120	22000	Data exception	**use of reserved word
20121	5050q	Versant ReVind rds error	permission denied
20122	5050r	Versant ReVind rds error	procedure not found
20123	5050s	Versant ReVind rds error	invalid arguments to procedure
20124	5050t	Versant ReVind rds error	Query conditionally terminated
20125	0750u	Dynamic sql-error	Number of open cursors exceeds limit
20126	34000	Invalid cursor name	***Invalid cursor name
20127	07001	Dynamic sql-error	Bad parameter specification for the statement
20128	2250x	Data Exception	Numeric value out of range.
20129	2250y	Data Exception	Data truncated.
20132	5050u	Versant ReVind rds error	Revoke failed because of restrict
20134	5050v	Versant ReVind rds error	Invalid long datatype column references
20135	5050x	Versant ReVind rds error	Contains operator is not supported in this context
20135	m0m01	Versant ReVind diagnostics error	Diagnostics statement failed.
20136	5050z	Versant ReVind rds error	Contains operator is not supported for this datatype
20137	50514	Versant ReVind rds error	Index is not defined or does not support CONTAINS

20138	50513	Versant ReVind rds error	Index on long fields requires that it can push down only CONTAINS
20140	50512	Versant ReVind rds error	Procedure already exists
20141	85001	Versant ReVind Stored procedure Compilation	Error in Stored Procedure Compilation.
20142	86001	Versant ReVind Stored procedure Execution	Error in Stored Procedure Execution.
20143	86002	Versant ReVind Stored procedure Execution	Too many recursions in call procedure.
20144	86003	Versant ReVind Stored procedure Execution	Null value fetched.
20145	86004	Versant ReVind Stored procedure Execution	Invalid field reference.
20146	86005	Versant ReVind Triggers	Trigger with this name already exists.
20147	86006	Versant ReVind Triggers	Trigger with this name does not exist.
20148	86007	Versant ReVind Triggers	Trigger Execution Failed.
20211	22800	Data exception	Remote procedure call error
20212	08801	Connection exception	SQL client bind to daemon failed
20213	08802	Connection exception	SQL client bind to SQL server failed
20214	08803	Connection exception	SQL NETWORK service entry is not available
20215	08804	Connection exception	Invalid TCP/IP hostname
20216	hz805	Remote database access	Invalid remote database name
20217	08806	Connection exception	network error on server
20218	08807	Connection exception	Invalid protocol
20219	2e000	Invalid connection name	***Invalid connection name

20220	08809	Connection exception	Duplicate connection name
20221	08810	Connection exception	No active connection
20222	08811	Connection exception	No environment defined database
20223	08812	Connection exception	Multiple local connections
20224	08813	Connection exception	Invalid protocol in connect_string
20225	08814	Connection exception	Exceeding permissible number of connections
20226	80815	Versant ReVind snw errors	Bad database handle
20227	08816	Connection exception	Invalid host name in connect string
20228	28817	Invalid auth specs	Access denied (Authorization failed)
20229	22818	Data exception	Invalid date value
20230	22819	Data exception	Invalid date string
20231	22820	Data exception	Invalid number strings
20232	22821	Data exception	Invalid number string
20233	22822	Data exception	Invalid time value
20234	22523	Data exception	Invalid time string
20235	22007	Data exception	Invalid time stamp string
20236	22012	Data exception	Division by zero attempted
20238	22615	Data exception	Error in format type
20239	2c000	Invalid character set name	Invalid character set name specified.
20240	5050y	Versant ReVind rds errors	Invalid collation name specified.
20241	08815	Connection Exception	Service in use.
20300	90901	Versant/DBS errors	Column group column doesn't exist

20301	90902	Versant/DBS errors	Column group column already specified
20302	90903	Versant/DBS errors	Column group name already specified
20303	90904	Versant/DBS errors	Column groups haven't covered all columns
20304	90905	Versant/DBS errors	Column groups are not implemented in Versant storage
23000	22563	Versant ReVind Data exception	Table create returned invalid table id
23001	22564	Versant ReVind Data exception	Index create returned invalid index id
25001	i0i01	Versant ReVind odbc integrator	Operation is valid only on a linked table
25002	i0i02	Versant ReVind odbc integrator	Operation not allowed on a linked table
25003	i0i03	Versant ReVind odbc integrator	Copy object exists
25004	i0i04	Versant ReVind odbc integrator	Unknown copy object
25005	i0i05	Versant ReVind odbc integrator	Dropping table failed
25006	i0i06	Versant ReVind odbc integrator	Bad copy sql statement
25007	i0i07	Versant ReVind odbc integrator	Unknown data type
25008	i0i08	Versant ReVind odbc integrator	Bad insert statement
25009	i0i09	Versant ReVind odbc integrator	Fetch operation failed
25010	i0i10	Versant ReVind odbc integrator	Insert operation failed
25011	i0i11	Versant ReVind odbc integrator	Operation not started

25012	i0i12	Versant ReVind odbc integrator	Operation marked for abort
25013	i0i13	Versant ReVind odbc integrator	Commit operation failed
25014	i0i14	Versant ReVind odbc integrator	Create table failed
25015	i0i15	Versant ReVind odbc integrator	Bad sync sql statement
25016	i0i16	Versant ReVind odbc integrator	Sync object exists
25017	i0i17	Versant ReVind odbc integrator	Create sync object failed
25018	i0i18	Versant ReVind odbc integrator	Create copy object failed
25019	i0i19	Versant ReVind odbc integrator	Unknown sync object
25020	i0i20	Versant ReVind odbc integrator	Illegal column name
25021	i0i21	Versant ReVind odbc integrator	Duplicate column name
25022	i0i22	Versant ReVind odbc integrator	Install failure
25023	i0i23	Versant ReVind odbc integrator	Invalid sync mode
25024	i0i24	Versant ReVind odbc integrator	Download or snapshot table missing
25025	i0i25	Versant ReVind odbc integrator	Upload table missing
25026	i0i26	Versant ReVind odbc integrator	Update operation failed
25027	i0i27	Versant ReVind odbc integrator	Delete operation failed
25028	i0i28	Versant ReVind odbc integrator	Close cursor failed

25029	i0i29	Versant ReVind odbc integrator	No primary key
25030	i0i30	Versant ReVind odbc integrator	Missing row
25031	i0i31	Versant ReVind odbc integrator	Bad primary key
25032	i0i32	Versant ReVind odbc integrator	Update contention
25033	i0i33	Versant ReVind odbc integrator	Link table failed
25034	i0i34	Versant ReVind odbc integrator	Unlink table failed
25035	i0i35	Versant ReVind odbc integrator	Link data source failed
25036	i0i36	Versant ReVind odbc integrator	Unlink data source failed
25037	i0i37	Versant ReVind odbc integrator	Integrator internal error
25038	i0i38	Versant ReVind odbc integrator	Operation already started
25039	i0i39	Versant ReVind odbc integrator	Opening of copy sql stmt failed
25040	i0i40	Versant ReVind odbc integrator	sync object failed
25041	i0i41	Versant ReVind odbc integrator	Dropping copy object failed
25042	i0i42	Versant ReVind odbc integrator	Closing copy sql stmt failed
25043	i0i43	Versant ReVind odbc integrator	Failure to update metadata timestamp
25101	j0j01	Versant ReVind odbc trans layer	SQLAllocEnv failed
25102	j0j02	Versant ReVind odbc trans layer	SQLAllocConnect failed

25103	j0j03	Versant ReVind odbc trans layer	SQLConnect failed
25104	j0j04	Versant ReVind odbc trans layer	SQLGetConnectOption failed
25105	j0j05	Versant ReVind odbc trans layer	SQLSetConnectOption failed
25106	j0j06	Versant ReVind odbc trans layer	Failed to map stmt handle to UUID
25107	j0j07	Versant ReVind odbc trans layer	SQLSetParam failed
25108	j0j08	Versant ReVind odbc trans layer	SQLDisconnect failed
25109	j0j09	Versant ReVind odbc trans layer	SQLExecute failed
25110	j0j10	Versant ReVind odbc trans layer	SQLRowCount failed
25111	j0j11	Versant ReVind odbc trans layer	SQLSetParam failed
25112	j0j12	Versant ReVind odbc trans layer	SQLBindCol failed
25113	j0j13	Versant ReVind odbc trans layer	SQLPrepare failed
25114	j0j14	Versant ReVind odbc trans layer	SQLResultCols failed
25115	j0j15	Versant ReVind odbc trans layer	SQLDescribeCol failed
25116	j0j16	Versant ReVind odbc trans layer	SQLFreeStmt failed
25117	j0j17	Versant ReVind odbc trans layer	SQLFetch failed
25118	j0j18	Versant ReVind odbc trans layer	SQLTransact failed
25119	j0j19	Versant ReVind odbc trans layer	SQLAllocStmt failed

25120	j0j20	Versant ReVind odbc trans layer	SQLTables failed
25121	j0j21	Versant ReVind odbc trans layer	SQLColumns failed
25122	j0j22	Versant ReVind odbc trans layer	SQLStatistics failed
25123	j0j23	Versant ReVind odbc trans layer	ODBC Driver interface mismatch
25124	j0j24	Versant ReVind odbc trans layer	ODBC Driver metadata exceeds storage limits
25125	j0j25	Versant ReVind odbc trans layer	SQLGetInfo failed
25126	j0j26	Versant ReVind odbc trans layer	Operation not allowed on the read-only database
25127	j0j27	Versant ReVind odbc trans layer	Cannot update views-with-check-option on remote tables
25128	j0j28	Versant ReVind odbc trans layer	Query terminated as max row limit exceeded for a remote table
25131	j0j29	Versant ReVind odbc trans layer	Unable to read column info from remote table
27181	50525	Versant ReVind Licensing Error	Versant/SQL is not licensed
27182	50526	Versant ReVind Licensing Error	License expired for Versant/SQL
27183	50527	Versant ReVind Licensing Error	Environment variable not set
27184	50528	Versant ReVind Licensing Error	License file not found
27185	50529	Versant ReVind Licensing Error	Error in license file
27186	50530	Versant ReVind Licensing Error	Ignored text in license file

30001	5050w	Versant ReVind rds errors	Query aborted on user request
30002	k0k02	Versant ReVind network interface	Invalid network handle
30003	k0k03	Versant ReVind network interface	Invalid sqlnetwork INTERFACE
30004	k0k04	Versant ReVind network interface	Invalid sqlnetwork INTERFACE procedure
30005	k0k05	Versant ReVind network interface	INTERFACE is already attached
30006	k0k06	Versant ReVind network interface	INTERFACE entry not found
30007	k0k07	Versant ReVind network interface	INTERFACE is already registered
30008	k0k08	Versant ReVind network interface	Mismatch in pkt header size and total argument size
30009	k0k09	Versant ReVind network interface	Invalid server id
30010	k0k10	Versant ReVind network interface	Reply does not match the request
30011	k0k02	Versant ReVind network interface	Memory allocation failure
30031	k0k11	Versant ReVind network interface	Error in transmission of packet
30032	k0k12	Versant ReVind network interface	Error in reception of packet
30033	k0k13	Versant ReVind network interface	No packet received
30034	k0k14	Versant ReVind network interface	Connection reset
30051	k0k15	Versant ReVind network interface	Network handle is inproc-ess handle
30061	k0k16	Versant ReVind network interface	Could not connect to sql network daemon

30062	k0k17	Versant ReVind network interface	Error in number of arguments
30063	k0k18	Versant ReVind network interface	Requested INTERFACE not registered
30064	k0k19	Versant ReVind network interface	Invalid INTERFACE procedure id
30065	k0k20	Versant ReVind network interface	Requested server executable not found
30066	k0k21	Versant ReVind network interface	Invalid configuration information
30067	k0k22	Versant ReVind network interface	INTERFACE not supported
30091	k0k23	Versant ReVind network interface	Invalid service name
30092	k0k24	Versant ReVind network interface	Invalid host
30093	k0k25	Versant ReVind network interface	Error in tcp/ip accept call
30094	k0k26	Versant ReVind network interface	Error in tcp/ip connect call
30095	k0k27	Versant ReVind network interface	Error in tcp/ip bind call
30096	k0k28	Versant ReVind network interface	Error in creating socket
30097	k0k29	Versant ReVind network interface	Error in setting socket option
30101	k0k30	Versant ReVind network interface	Interrupt occured
40001	L0L01	Versant ReVind env error	Error in reading configuration

The following table lists the Versant driver error messages, ordered by error code number, and shows the corresponding description for each code:

Error Codes	Description
330001	Already exists
330002	Does not exist
330003	Duplicate
330004	Invalid table
330005	Schema evolution not supported through SQL
330006	SelfOID column creation failed
330007	Cannot create foreign key from SelfOID
330008	RowNumber column creation failed
330009	Table initialization failed
330010	Column creation failed
330011	Foreign key construction failed
330012	Class to table mapping failed
330013	Cursor does not exist
330014	Lock time out
330015	End of scan
330016	Lock would block
330017	No more elements
330018	No associated attributes
330019	Invalid type
330020	Could not get attribute
330021	Invalid OID
330022	Temporary tables not supported
330023	Invalid column
330024	Insertion failed
330025	Update failed
330026	Delete failed
330027	Index creation is not supported

330028	Object creation failed
330029	Object is not found
330030	Insertion to embed table is not supported
330031	Remove from embed table is not supported
330032	Cooperator is not supported
330033	Only update is allowed in a vstr table mapped from an array
330034	No more qualified elements in vstr table
330035	Invalid database number
330036	Remove from vstr table is not supported
330037	Versant type overflow error
330038	Versant type underflow error
330039	No privilege to login as vsqldba or Versant DBA
330040	Insertion is not allowed in this table
330041	Update is not allowed in this table
330042	Deletion is not allowed in this table
330043	Create table failed
330044	Delete table failed
330045	Cannot delete system table
330046	Table defined in multiple databases.
330047	General error
330048	Violation of unique constraint.
330049	Table level constraints not supported.
330050	Can reference only the SelfOID column.
330051	Run `schload` tool on the database.

Glossary

alias

A temporary name for a table or column specified in the FROM clause of an SQL query expression. Also called *correction name*. Derived tables and search conditions that join a table with itself must specify an alias. Once a query specifies an alias, references to the table or column must use the alias and not the underlying table or column name.

ASCII

(American Standard Code for Information Interchange) A 7-bit character set that provides 128 character combinations.

cardinality

Number of rows in a result table.

Cartesian product

Also called cross-product. In a query expression, the result table generated when a FROM clause lists more than one table but specifies no join conditions. In such a case, the result table is formed by concatenating every row of every table with all other rows in all tables.

client

Generally, in client/server systems, the part of the system that sends requests to servers and processes the results of those requests.

collation

The rules used to control how character strings in a character set compare with each other. Each character set specifies a collating sequence that defines relative values of each character for comparing, merging, and sorting character strings. In addition, storage systems may define additional collations that SQL statements specify with the COLLATE clause in column definitions, column references, and character-string references.

column alias

An alias specified for a column. See ***alias***.

constraint

Part of an SQL table definition that restricts the values that can be stored in a table. When you insert, delete, or update column values, the constraint checks the new values against the conditions specified by the constraint. If the value violates the constraint, it generates an error. Along with ***triggers***, constraints enforce ***referential integrity*** by insuring that a value stored in

the foreign key of a table must either be `null` or be equal to some value in the matching unique or primary key of another table.

correlation name

Another term for ***alias***.

cross product

Another term for ***Cartesian product***.

data dictionary

Another term for ***system catalog***.

delimited identifiers

Names in SQL statements enclosed in double quotation marks ("). Enclosing a name in double quotation marks preserves the case of the name and allows it to include reserved words and special characters. Subsequent references to a delimited identifier must also use enclosing double quotation marks.

derived table

A ***virtual table*** specified as a query expression in the `FROM` clause of another query expression.

form of use

The storage format for characters in a character set. Some character sets, such as ***ASCII***, require one byte (*octet*) for each character. Others, such as ***Unicode***, use two bytes, and are called multi-octet character sets.

join

A relational operation that combines data from two tables.

input parameter

In a stored procedure specification, an argument that an application must pass when it calls the stored procedure. In an SQL statement, a ***parameter marker*** in the statement string that acts as a placeholder for a value that will be substituted when the statement executes.

metadata

Data that details the structure of tables and indexes in the proprietary storage system. The SQL engine stores metadata in the system catalog.

octet

A group of 8 bits. Synonymous with byte and often used in descriptions of character-set encoding format.

output parameter

In a stored procedure specification, an argument in which the stored procedure returns a value after it executes.

parameter marker

A question mark (?) in a procedure call or SQL statement string that acts as a placeholder for an input or output parameter supplied at runtime when the procedure executes. The `CALL` statement (or corresponding ODBC or JDBC escape clause) use parameter markers to pass parameters to stored procedures, and the `SQLIStatement, SQLPStatement,` and `SQLCursor` objects use them within procedures.

primary key

A subset of the fields in a table, characterized by the constraint that no two records in a table may have the same primary key value, and that no fields of the primary key may have a `null` value. Primary keys are specified in a `CREATE TABLE` statement.

procedure result set

In a stored procedure, a set of data rows returned to the calling application. The number and data types of columns in the procedure result set are specified in the `RESULT` clause of the `CREATE PROCEDURE` statement. The procedure can transfer data from an *SQL result set* to the procedure result set or it can store data generated internally. A stored procedure can have only one procedure result set.

query expression

The fundamental element in SQL syntax. Query expressions specify a result table derived from some combination of rows from the tables or views identified in the `FROM` clause of the expression. Query expressions are the basis of `SELECT, DECLARE CURSOR, CREATE VIEW,` and `INSERT` statements.

referential integrity

The condition where the value stored in a database table's foreign key must either be `null` or be equal to some value in another table's the matching unique or primary key. SQL provides two mechanisms to enforce referential integrity: constraints specified as part of `CREATE TABLE` statements prevent updates that violate referential integrity, and ***triggers*** specified in `CREATE TRIGGER` statements execute a stored procedure to enforce referential integrity.

repertoire

The set of characters allowed in a character set.

result set

Another term for ***result table***.

result table

A temporary table of values derived from columns and rows of one or more tables that meet conditions specified by a query expression.

row identifier

Another term for ***tuple*** identifier.

search condition

The SQL syntax element that specifies a condition that is true or false about a given row or group of rows. Query expressions and `UPDATE` statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or `UPDATE` statement. Search conditions contain one or more predicates. Search conditions follow the `WHERE` or `HAVING` keywords in SQL statements.

selectivity

The fraction of a table's rows returned by a query.

server

Generally, in client/server systems, the part of the system that receives requests from clients and responds with results to those requests.

SQL diagnostics area

A data structure that contains information about the execution status (success, error or warning conditions) of the most recent SQL statement. The SQL-92 standard specified the diagnostics area as a standardized alternative to widely varying implementations of the SQLCA. Versant ReVind supports both the SQLCA and the SQL diagnostics area. The SQL `GET DIAGNOSTICS` statement returns information about the diagnostics area to an application, including the value of the `SQLSTATE` status parameter.

SQL engine

The core internal component of Versant ReVind. The SQL engine receives requests from the Versant ReVind ODBC driver, processes them, and returns results.

SQLCA

SQL Communications area: A data structure that contains information about the execution status (success, error or warning conditions) of the most recent SQL statement. The `SQLCA` includes an `SQLCODE` field, The `SQLCA` provides the same information as the SQL diagnostics area, but is not compliant with the SQL-92 standard. Versant ReVind supports both the `SQLCA` and the SQL diagnostics area.

SQLCODE

An integer status parameter whose value indicates the condition status returned by the most recent SQL statement. An `SQLCODE` value of zero means success, a positive value means warning, and a negative value means an error status. `SQLCODE` is superseded by `SQLSTATE` in the SQL-92 standard. Applications declare either `SQLSTATE` or `SQLCODE`, or both. SQL returns the status to `SQLSTATE` or `SQLCODE` after execution of each SQL statement.

SQL result set

In a stored procedure, the set of data rows generated by an SQL statement (`SELECT` and, in some cases, `CALL`).

SQLSTATE

A 5-character status parameter whose value indicates the condition status returned by the most recent SQL statement. `SQLSTATE` is specified by the SQL-92 standard as a replacement for the `SQLCODE` status parameter (which was part of SQL-89). `SQLSTATE` defines many more specific error conditions than `SQLCODE`, which allows applications to implement more portable error handling. Applications declare either `SQLSTATE` or `SQLCODE`, or both. SQL returns the status to `SQLSTATE` or `SQLCODE` after execution of each SQL statement.

storage interfaces

C++ routines called by the SQL engine that access and manipulate data in a proprietary storage system. A proprietary storage system must implement supplied storage stub templates to map the storage interfaces to the underlying storage system.

storage interfaces

Another term for ***stub*** interfaces.

storage manager

A completed implementation of Versant ReVind storage interfaces. A storage manager receives calls from the SQL engine and accesses the underlying proprietary storage system to retrieve and store data.

storage system

The proprietary database system that underlies a storage manager. Versant ReVind provides a SQL interface to a storage system through the SQL engine and its stub interfaces.

stub interfaces

Template routines provided with Versant ReVind for implementing access to proprietary storage systems. Once filled in for a particular storage system, the completed stubs are called ***storage managers***.

stubs

Another term for ***stub*** interfaces.

system catalog

Tables created by the SQL engine that store information about tables, columns, and indexes that make up the database. The SQL engine creates and manages the system catalog independent of the proprietary storage system.

system tables

Another term for ***system catalog***.

systpe

The default owner name for all system tables in a Versant ReVind database. Users must qualify references to system tables as `systpe.`*tablename*.

tid

Another term for ***tuple identifier***.

transaction

A group of operations whose changes can be made permanent or undone only as a unit. Once implementations add the ability to change data in the proprietary storage system, they must also implement transaction management to protect against data corruption.

trigger action time

The `BEFORE` or `AFTER` keywords in a `CREATE TRIGGER` statement. The trigger action time specifies whether the actions implemented by the trigger execute before or after the triggering `INSERT, UPDATE,` or `DELETE` statement.

trigger event

The statement that causes a trigger to execute. Trigger events can be SQL `INSERT, UPDATE,` or `DELETE` statements that affect the table for which a trigger is defined.

tuple identifier

A unique identifier for a tuple (row) in a table. Storage managers return a tuple identifier for the tuple that was inserted after an insert operation. The SQL engine passes a tuple identifier to the delete, update, and fetch stubs to indicate which tuple is affected. The SQL scalar function `ROWID` and related functions return tuple identifiers to applications.

unicode

A superset of the ASCII character set that uses two bytes for each character rather than ASCII's 7-bit representation. Unicode is able to handle 65,536 character combinations instead of ASCII's 128. Unicode includes alphabets for many of the world's languages. The first 128 codes of Unicode are identical to ASCII, with a second-byte value of zero.

URL

In general, a Universal Resource Locator specifies protocols and locations of items on the Internet. In JDBC, a database connection string in the form jdbc:*subprotocol:subname*. The Versant JDBC Driver format for database URLs is JDBC:versant:T:*host_name:db_name*.

Versant ReVind

The collection of products that opens databases by layering a standard SQL interface on any proprietary data storage system.

Versant ReVind environment

An open database created by the combination of Versant ReVind layered on top of any proprietary storage system.

view

A virtual table that recreates the result table specified by a `SELECT` statement. No data is stored in a view, but other queries can refer to it as if it were a table containing data corresponding to the result table it specifies.

virtual table

A table of values that is not physically stored in a database, but instead derived from columns and rows of other tables. SQL generates virtual tables in its processing of query expressions: the `FROM, WHERE, GROUP BY` and `HAVING` clauses each generate a virtual table based on their input.

vsqldba

The default owner name for all system tables in a Versant ReVind database. Users must qualify references to system tables as `vsqldba.`*`table-name`*.

Index