Versant Database Fundamentals 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

Versant Object Database

This Chapter gives a brief overview of the Versant Object Database, its architecture and data management.

It also gives an clear insight to Versant features and its advantages. The techniques of using Versant to its best are also described.

This chapter describes:

- Introducing Versant Object Database (VOD)
- Versant Features
- Storage Architecture
- Software Structure
- Language Interfaces
- Why Versant?
- Versant Implementing Tips

INTRODUCING VERSANT OBJECT DATABASE

The Versant Object Database is an Object Database Management System (ODBMS). It has been designed to ease development and enhance performance in complex, distributed and heterogeneous environments. It is very useful where applications are written in Java and/or C++.

It is a complete, e-infrastructure software that simplifies the process of building and deploying transactional, distributed applications.

As a standalone database, the Versant ODBMS is designed to meet customers' requirements for high performance, scalability, reliability and compatibility with disparate computing platforms and corporate information systems.

Versant Object Database has established a reputation for exceeding the demands of mission critical enterprise business applications providing reliability, integrity and performance. The efficient multi-threaded architecture, internal parallelism, balanced client-server architect and efficient query optimization of Versant ODBMS delivers unsurpassed levels of performance and scalability.

The Versant Object Database includes the Versant ODBMS, the C++ and Java language interfaces, the XML toolkit and Asynchronous Replication framework.

The main advantage is that all Versant products included in the Versant Object Database, work cohesively together, conform to the same compiler (C++, JDK) versions and adopt the same release schedule.

Versant Object Database consists of the following components:

Versant Object Database Components

COMPONENT	DETAILS
Versant ODBMS	Versant Object Database Management System.
GUI Tools	Versant GUI Tools.
JDO Interface	JDO Interface for Versant ODBMS.
Java Versant Interface	Java Language Interface for Versant ODBMS.
C++/Versant	C++ Language Interface for Versant ODBMS.
VAR	Versant Asynchronous Replication for Versant ODBMS.
Versant XML Toolkit	XML Toolkit for Versant ODBMS.
Vedding	Fault Tolerant Versant ODBMS Server.
HABACKUP	Backup Solution for use with High Availability Server
Vorkout	Versant Online Database Reorganization Tool
Warm Standby	Used for an Incremental Roll Forward recovery.

The Versant Object Database 7.0 release, focuses on improved query performance capabilities, better X/Open XA interface and a more reliable Fault Tolerant Server.

There are improved Security and Authentication features along with an enhanced set of GUI tools.

VERSANT FEATURES

Versant is an object database management system that includes all features needed for scalable production databases in a distributed, heterogeneous workgroup environment.

Following is a brief overview of the Versant Features:

Data as Objects

Versant models data as objects. The Versant implementation of objects allows:

- Custom definition of complex data types.
- Encapsulation of data and code.
- Inheritance of data and code.
- Code reuse.
- Polymorphism.
- Unique identification of objects.

Database Features

Versant database features include:

- Persistent storage of data.
- Concurrent access by multiple users.
- Concurrent access to a session by multiple processes or threads.
- Multiple sessions, including sessions containing no processes.
- Transaction management.
- Recovery from system failures.
- Navigational and search condition queries.
- Remote database connections.
- Data versions.
- User defined security.
- Two-phase commits.
- You can create a distributed database system containing a combined total of 2^16 databases.

- Multiple standard language interfaces.
- Heterogeneous platforms.
- Use of multiple threads by the database server.
- Use of multiple latches by the database server.
- Unique indexes.

C++/Versant Interface

The C++/Versant interface allows:

- Association of data with links and arrays of links.
- Extensible data types.
- Parameterized types.
- Embedding of objects.
- Containers and collections of objects.
- Support for the Standard Template Library
- Support for Rogue Wave Tools.h++
- Support for ODMG-93.
- Support for multiple compilers.

Java Versant interface

The Java Versant interface allows:

- Support for elemental data types as well as references.
- Extensible data types.
- Seamless support for garbage collection.
- User to specify class of persistence.
- Persistence by reachability.
- Support for JDK 1.2 Collections.
- Support for multi-threaded applications.
- Transparent event notification.
- Support for ODMG.
- Layered database APIs providing transparent and fundamental access to objects.

Database System

Versant database system features support:

- Distribution of data, including the ability to migrate objects.
- A client/server model of hardware utilization.
- Query processing on servers.
- Dynamic management of database schema.
- Tuning options for applications and databases.
- Object level locking.
- Object caching on the client and page caching on the server.
- Clustering of instances of a class.
- Either raw devices or files for data storage.
- Indexing for query optimization.
- Ability to turn locking and logging `ON` and `OFF`.
- On supported platforms and interfaces, multiple process or multiple thread applications.
- Data replication on numerous databases.
- Independence from OS passwords for DBA
- Support for password authentication for database utilities
- Support for External User Authentication (Plugins)
- Support for Windows Terminal Server

Database Administration

Versant database administration utilities support:

- Creating, expanding and deleting databases.
- Backing up data.
- User authorization.
- Custom system configurations.
- Modification of data definitions.
- Creation of classes at run time.

Application Programming

Application programming features support:

- All features of access languages, including flow of control.
- Versant provides application programming interfaces for C, C++, and Java. You can also access databases with SQL statements and parse the results with C or C++ routines using the Versant Query Language.
- Custom and third party libraries of code and data types.
- Multiple kinds of atomic work units, including transactions, checkpoints and savepoints.
- Predefined data types and management routines.
- Proprietary and third party programming tools.
- Control of process and shared memory, including explicit pinning of data.
- Error handling mechanisms.
- Application debugging facilities.

Physical Database

A Versant database system physically consists of:

- System files.
- Executable utilities.
- Header files.
- Link libraries.
- Class libraries for each language interface.
- Application development tools.
- At least one database consisting of storage and log volumes, which are files or raw devices.

Scalability & 64-bit support

The Versant ODBMS supports both the 64 and 32-bit models. The 32-bit model is specifically for customers with dependencies on other 32-bit products, whilst the 64-bit release allows massive scaling.

Versant ODBMS supports system backup and archive files larger than 2GB. On Windows platform, the database server is scalable when the machine is booted with /3GB switch.

Query

Virtual Attributes are supported which allow to define derived attributes that are dynamically computed from real class attributes.

The supported "Virtual Attributes" are:

1. Multi- attribute
2. Case Insensitive
3. Internationalized Strings

Versant Query Language 7.0

Complex Expressions in Query

- Support for attributes on LHS & RHS
- Support for mathematical operations
- Support for IS EMPTY operator
- Support for `toupper()`/`tolower()`
- Support for `o_list`
- Support for IN operator
- Support for EXISTS & FOR ALL operators
- Server-side sorting

SET Operators

Support for SET operators on multi-valued attributes (except for strings type).

Improved indexing capabilities

Btree indexing for multi-valued attribute of any Versant elementary type.

Improved stability of cursor query

New Query processing support and cursor changes

Querying on Candidate Collection

Instead of querying on an entire class, it will now be possible to restrict the query to only a subset of objects from the class.

Internationalization (i18N) Support

Versant supports the storage and retrieval of strings that use international character sets and in conjunction with the query enhancements, query them.

Versant has certified that 8-bit clean (UTF-8) character encodings, are stored and manipulated correctly internally, and it is possible to use such encodings for:

- Database names
- User names & passwords
- String attribute values

The query enhancements make it possible to build and use indexes, and query based on the character encoding and language (locale).

Open Transactions

Versant Open Trasnactions support:

- Multiple database connections
- Transaction timeout by TM
- Both local and XA transactions using the same database connection
- Improved error logging

Embedding Versant in Applications

Versant can be embedded in other applications. Some APIs are provided to manage a Versant environment completely via a API from C, C++ or Java. This functionality is significant for those who want to embed Versant in their own applications.

Versant XML Toolkit

The Versant XML Toolkit (VXML) adds XML/object mapping support to the Versant Object Database.

Using the command line tools or Java APIs, users can generate XML from defined object graphs and likewise generate objects from XML.

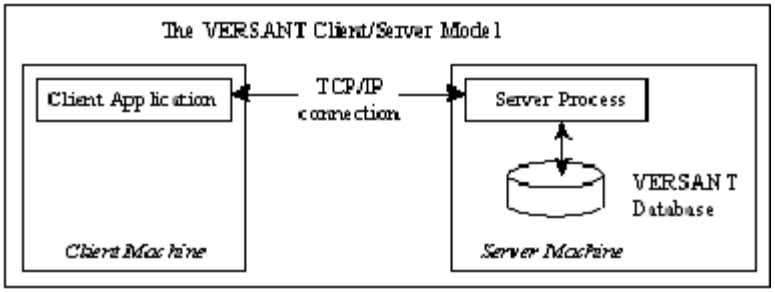
Scalable Operation

Versant is scalable, which means that it uses distributed resources in such a way that performance does not decrease as the system grows.

Versant is scalable because of the following features:

Client/server model

Versant uses a client/server model of computing. To Versant, the terms "client" and "server" refer to roles and not machines or processes. The client and server can be on the same machine or on different machines.



Client/Server Model

Your application is the database client and runs on a client machine. A client application can access multiple databases concurrently with other client applications.

The database that provides objects to an application is called the "server". One of the roles of the server is to process queries. A server can support concurrent access by numerous users.

The client and server communicate via a TCP/IP communication.

The queries are executed on the platform containing the data and the locks are set at the object level. Because of this, the network traffic is reduced to a short query message from the client to the server and the return of only the desired objects from the server to the client.

Processing queries on servers balances the database processing responsibilities between the client and the server. It can result in major performance gains in a large distributed database environment, by taking full advantage of available platform and network resources including parallel and scalable processors.

By contrast, a file/server query causes all objects of a class to be locked and passed over a network even if only one object is desired.

Locks

The Versant locking model provides for a high concurrency of multiple users.

Volumes

Objects are kept in one or more database *volumes*, which are storage places on disk. Database volumes can be files or raw devices which can be added incrementally either locally or at distributed locations.

Two-phase commits

To ensure data integrity when multiple, distributed databases are used, Versant performs updates with two-phase commits. Two-phase commits involve a procedure in which multiple databases communicate with each other to confirm that all changes in a unit of work are saved or rolled back together.

Lazy updates

Changes to class definitions do not paralyze operations. Instead, instances are updated the next time they are accessed, which is called a "lazy update." You can create or drop leaf classes, rename leaf or non-leaf classes and create or drop attributes and methods in leaf or non-leaf classes.

Schema management

To facilitate the use of distributed databases, you can ask an object the name of its class and then inspect its class definition. Routines are also provided for creating and modifying classes at runtime and for synchronizing class definitions among multiple databases.

Distributed Databases

Versant supports distributed databases with the following features:

Object migration

Objects can be migrated while applications still have transparent access to them. Object migration is possible, because objects have identifiers that stay with the object for its lifetime which means that the physical locations of objects are hidden from the application.

Recovery

To ensure data integrity when multiple, distributed databases are used, Versant performs updates with two-phase commits. Two-phase commits involve a procedure in which multiple databases communicate with each other to confirm that all changes in a unit of work are saved or rolled back together.

Heterogeneity

Objects can be moved among heterogeneous platforms and managed in databases on numerous hardware platforms to take advantage of available resources in a network.

Schema management

Class definitions can be managed at run time on both local and remote databases. Class definitions are stored with objects, which allow access to objects with applications that are running on different platforms and are using multiple interface languages.

Expansion

Databases can be created, deleted, and expanded on local and remote platforms. Database volumes can span devices and platforms.

Backup

Data on one machine can be backed up to remote sites, tapes, or files. Multiple distributed databases can be backed up to save their state at a given point in time. This gives transactional consistency across multiple databases.

Security

Access to databases and system utilities for security, is controlled through user authorization, which may be customized.

Session database

Versant implements the concept of a session database, which can be local or remote, which handles basic record keeping and logging for a session.

Connection database

Applications can connect to any number of local or remote databases and then manage objects in them as if they were local. You can work on objects in any number of databases at the same time in a distributed transaction.

Workgroup Support

Versant supports workgroups with the following features:

Locking

Locks allow multiple applications to access the same objects simultaneously in a co-operative, controlled, and predictable manner. To maximize concurrency, locks are applied at the object level.

Short locks reserve objects during a transaction typically lasting only seconds or minutes.

For more information see “Short Locks” on page 95.

Object level locking

Object level locking gives maximum access to data while providing “Read” and “Modification” guarantees.

Two types of databases

Versant allows you to create two kinds of databases: group databases, which are accessible to many users, and personal databases, which are only accessible to one user at a time.

Variety of transactions

Versant supports multiple kinds of atomic work units, including transactions, checkpoints, and savepoints.

Standard interfaces

Multiple standard language interfaces allow workgroup members to access data from applications written in multiple languages.The language specific interfaces map all capabilities and programming styles of a particular language to the object database model. Any action or data type that can be expressed in the interface language may become a part of a Versant database schema. Thus, programming languages are tightly bound to the database, but the database is not tightly bound to a particular language. There is no special Versant database language, thus Versant Database is not Language Specific.

Versant object databases are most commonly used with languages that implement the concept of either class or template and either delegation or inheritance. Versant can also be used with languages, such as C, which are not object oriented.

Versant provides language specific interfaces for C,C ++ and Java. Each language-specific interface consists of several libraries of precompiled routines and, for typed languages, predefined data types. An interface can include its own development tools, and can be used with other vendors' software development products.

Compilers and debugging facilities

Versant supports multiple compilers and debugging facilities.

Libraries

You can use Versant with custom and third party libraries of code and data types.

Tools

Versant supports a variety of proprietary and third party programming tools.

Shared memory cache

Database servers maintain a page cache in shared memory.

Custom lock models

For situations requiring unusual types of access, you can define your own locks.

Object migration

Because objects are persistently and uniquely identified, objects can be redistributed in a system of databases for performance improvements without affecting application code making object references.

Performance

The Performance Features of Versant include the following:

Object and page caches

During a database session, Versant maintains an object cache in virtual memory on the client machine and a page cache in shared memory on server machines with a database in use. There is one object cache per application and one page cache per operating database.

This approach combines the best of a page management scheme and an object management scheme, because the object cache provides fast access to objects needed in the current transaction while the page cache provides fast access to objects used in preceding transactions and to objects stored on the same page as a recently accessed object.

Memory management

Versant provides numerous mechanisms for managing application memory, including explicit pinning and releasing of data.

Process tuning

Versant allows you to set operating parameters for application and server processes.

Clustering

You can cluster instances of a class on disk storage, which will improve query performance.

Raw devices

You can use either raw devices or files for data storage.

Indexing

You can create indexes on attributes, which will improve query performance.

Locking `ON` or `OFF`

You can turn locking and logging `ON` or `OFF` to improve performance.

Turning locking off is safe in a personal database, because there can be only one user of a personal database at a time.

Multiple processes and threads

You can use multiple processes or threads in a session. You can also establish multiple sessions.

Navigational queries

You can use object references to navigate to related objects.

Integrated Installation

A common, integrated installation offers a single GUI interface to install data-management and connectivity components.

This integrated installation, eliminates version conflicts when installing and deploying components of the Versant Object Database.

How to Use Versant?

To use Versant:

- Create data definitions and applications using Versant routines along with normal interface commands and methods.
- In your application, before saving or retrieving persistent objects, start a database "session." A session is a period of time in an application during which a Versant memory workspace exists in process memory or shared memory and during which you have access to at least one Versant database.

For the C and C++ interfaces:

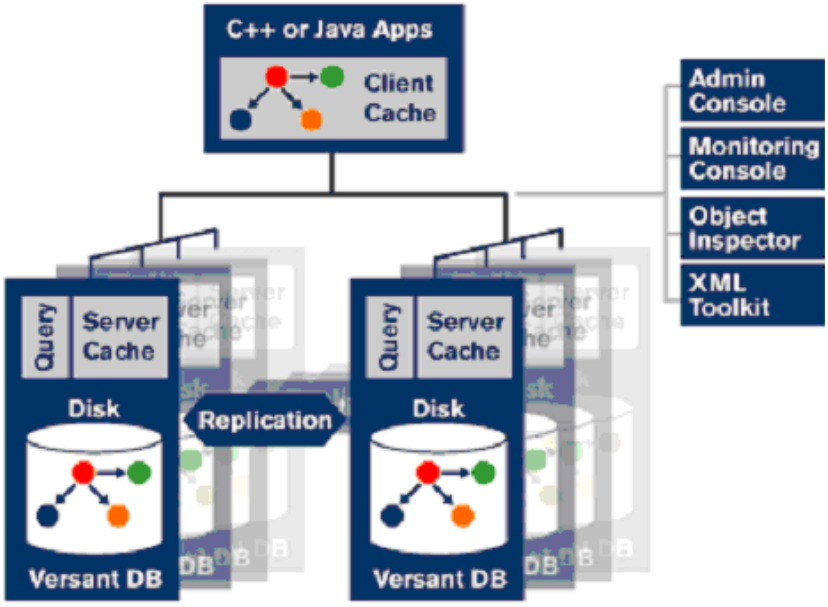
- Include relevant Versant header files and then compile and link with a Versant compiled code library to create an executable program.

For the Java interfaces:

- Use the JVI Enhancer that post-processes the compiled class byte-code to provide transparent persistence.

VERSANT ARCHITECTURE

Client - Server Architecture



Versant has a balanced client server architecture.

Both the client and server machines have limitations on resources and a balanced workload is necessary for optimal performance.

With Versant -

- Client and Server can run on the same or different machines
- The client is part of an C, C++ or Java application and runs within the scope of this application
- Client and Server normally run in separate processes, but can also run in one process.

Versant balances workload between the client and server:

- Versant Client manages
  - Database connections
  - Relevant persistent objects
- Versant Server manages
  - Disk files
  - Data storage/retrieval
  - Locking
  - Logging
  - Queries, Indexes

Versant Storage Architecture

Each database consists of a number of volumes, which are storage places on disk.

A volume can be either a file or a raw device.

Database volumes

The Database volumes are:

System Volume

The System Volume for each database is automatically created as a part of the initial process of creating a database. It is used for storing class descriptions and for storing object instances.

Data Volumes

Additional data volumes can be added to a database to increase capacity.

Logical Log Volume

The Logical Log volume contains transactions and redo information for logging recovery and rollback.

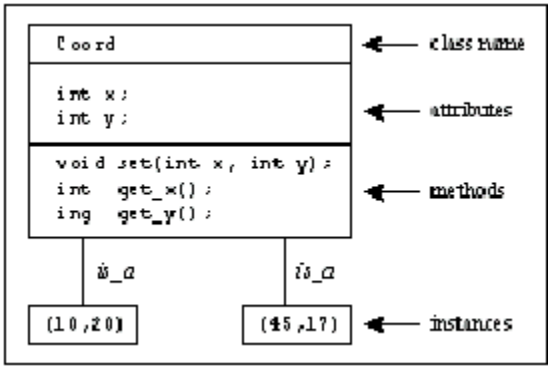
Physical Log Volume

The Physical Log Volume contains physical data information for logging and recovery.

The Logical Log Volume and Physical Log Volume are used to record transaction activities and provide information for roll back and recovery.

Logical log and physical log volumes are created when a database is created.

The basic storage architecture is as follows:



Basic Storage Architecture

There is no structural difference between personal and group databases.

The maximum number of databases that can be combined to form a distributed database system is 2^{16}.

For the C and C++ interfaces -

The applications and Versant libraries are stored as files. When you create classes or an application, you write a program that includes appropriate `Versant.h` specification files. You then compile your program and then link it with appropriate `Versant.a` library files.

With the C++ Interface, you must also insert your classes into a database with the Schema Change Utility `sch2db`.

VERSANT INTERNAL STRUCTURE

Internally Versant is composed of several software modules.

To understand how Versant uses processes, you need to know that Versant is structured into several modules.

Versant Manager

From a developer's viewpoint, programs using language interfaces drive all database and object management actions by communicating with a module called "Versant Manager". Versant Manager then communicates with a module called "Versant Server".

Versant Manager performs object caching, object validation and management of queries, schema, links and versions.

The portion of Versant that runs on the same machine as a client application is called Versant Manager.

The Versant Manager has the following features:

- It presents objects to an application,
- It manipulates classes,
- It caches objects in virtual memory,
- It provides transaction management via a two-phase commit protocol,
- It distributes requests for queries, updates, links and versions to server processes that manage databases,
- It manages database sessions,
- It establishes connections to databases, which may be on the same machine or another machine,
- It converts objects from Versant database format to client machine format.

Application programs communicate with Versant Manager through language specific interfaces. Versant Manager in turn communicates with one or more Versant Server modules, which manage databases.

Versant Manager functionally consists of several subsidiary software modules.

The Versant Schema Manager, creates and manipulates class definitions. The Versant Query Processor supports queries for particular objects and iteration over classes of objects. Other modules cache objects and provide session and distributed transaction support.

Versant Manager is structurally divided into two parts:

- one part is associated with an application and an object cache
- another part is associated with a Versant Server and a server page cache.

The structural organization, like the functional organization, is transparent to developers and users, but it provides the internal basis for flexibility in system configurations.

To create an application capable of accessing a Versant database, you link Versant Manager into your application. It then appears to your application that Versant Manager moves objects from the database into and out of your application. This happens automatically, and your application need not explicitly control object movement as objects will be supplied as your application accesses them.

Versant Server

Versant Server performs object retrieval, object update, page caching, query support and management of storage classes, indexes, transactions, logging and locking.

The portion of Versant that runs on the machine where data is stored or retrieved from is called Versant Server. Versant Server is a interface between Versant Managers and operating systems.

Versant Server has the following features:

- It evaluates objects on disk or in the server page cache per queries sent by client applications,
- It performs disk and storage management tasks such as object retrieval, object update, page caching, manages storage classes, indexes, transactions, logging, and locking,
- It defines transactions and locks objects,
- It maintains files that provide for logging and recovery,
- It manages indexes.

The term "Versant Server" refers to a software module and should not be confused with the term "server", which refers to a machine running Versant Server software.

Versant Server is the base level of a Versant object database management system. It interfaces with the operating system to retrieve and store data in database volumes and communicates with Versant Managers.

At the Versant Server level an object is a simple physical data structure consisting of a fixed number of fields. Each Versant Server accesses one database.

When a client application begins a database session, it automatically runs both Versant Manager and Versant Server.

Network and Virtual layers

Between a Versant Server and an operating system, is a Virtual System Layer specific to the hardware platform. The Virtual System Layer provides portability across hardware boundaries.

Internal communications occur through network layers. The network layers translate messages as appropriate to the network protocol. Due to this form of internal communication, a Versant Manager may communicate transparently with all parts of a distributed database network.

To provide for heterogeneous hardware, Versant implements its client/server architecture with additional software modules called the "Virtual System Layer" and the "Network Layer".

The Virtual System Layer isolates operating system code and provides portability across hardware boundaries. To allow portability, you must use Versant elemental and/or Class Library data types in your programs and recompile your programs as appropriate for each platform.

The Network Layer translates messages to the appropriate network protocol and moves objects as network packets.

VERSANT LANGUAGE INTERFACES

Language-specific interfaces map the capabilities and styles of a programming language to the object database model.

Any action or data type that can be expressed in the interface language can become part of a Versant database schema. This means that you use standard language statements to manipulate Versant.

There is no special specific Versant database language.

Versant object databases are most commonly used with languages that implement the concept of either class or template and either delegation or inheritance. Versant can also be used with languages such as C, which are not object oriented.

Versant provides language specific interfaces for C, C++ and Java.

Each language-specific interface consists of several libraries of precompiled routines and for typed languages and predefined data types. An interface can include its own development tool and can be used with other vendors' software development products.

C/Versant Interface

The C/Versant interface can take advantage of most object model capabilities, including the ability to define embedded, association, inheritance and version relationships. C/Versant also implements all Versant database functionality as both functions and utilities.

There is, however, no messaging concept in the pure C environment: when you define methods they are executed as functions. Also, there is no runtime binding to functions, inheritance is limited to virtual inheritance, and there is no concept of private versus public functions.

C++/Versant Interface

The C++/Versant interface implements the full object model. Methods are provided to define embedded, association, inheritance and version relationships, either at compile time or at runtime. C++/Versant also implements complete Versant database functionality as both functions and utilities.

C++/Versant functionality is available through standard C++. This includes dynamic access to the database schema for runtime type checking, locating subclass and superclass information, and identifying class attributes. Any persistent class can have transient and persistent instances simultaneously.

Sharing C++ Objects

Although Versant stores objects in such a way that they can be accessed by C++ applications, there are differences in the C++ languages that limit object sharing.

Following is an explanation of the Versant and C++ data models:

Database object model

A Versant database stores schema information in class objects and data in instance objects. The information in the class objects is slightly different depending on what language defined the class, but the objects are accessible to all supported language interfaces.

C++ data model

Classes defined by C++ may have multiple inheritance, the attribute names of superclasses may be the same as attribute names in derived classes, and classes may be embedded as attributes. Associations among objects may be defined with links.

In order to allow multiple inheritance and class embedding, Versant assigns a unique database name to each attribute. This name normally consists of the class name concatenated with the attribute name. If the attribute type is another class, the database attribute name is a concatenation of the class name, the embedded class name and the attribute name.

For C++/Versant applications, the storage layout of an object is defined at compile time. Even though the storage layout may be different among different C++ compilers, at run time the compiler assumes that the object has a fixed size and layout, and that each attribute is of a particular domain.

Java Versant Interface

Java Versant Interface (JVI) implements the full object model. Methods are provided to define embedded association and inheritance relationships, either at compile time or at runtime. JVI also implements complete Versant database functionality as both functions and utilities.

JVI functionality is available through standard Java. This includes dynamic access to the database schema for runtime type checking, locating subclass and superclass information and identifying class attributes. Any persistent class can have transient and persistent instances simultaneously.

For more information about the JVI language binding, please refer to the *Java Versant Interface Usage Manual*.

Versant Standards

Versant Object Technology Corporation is involved in efforts to define industry standards. Versant is committed to and/or working on the following:

- ISO 9001 Registration for Company-wide Quality Management Systems.
- Object Management Group (OMG).
- Object Database Management Group (ODMG).
- ANSI (American National Standards Institute) X3H2 Structured Query Language, X3H7 Object-Oriented Information Systems, X3J4 Object-Oriented COBOL Task Group, X3J16 C++.

WHY VERSANT?

Following are some of the advantages of the using the Versant Object Database.

Versant is Multi-featured

Versant is designed for use as a high-speed, production database in a complex, scalable, heterogeneous, client/server, workgroup environment. Accordingly, it has numerous features that allow you to perform almost any database task.

If you are an experienced database developer or object programmer, then you will immediately appreciate the power and flexibility that Versant offers.

If you are new to databases and/or object programming, the number of Versant features is so large that Versant may be overwhelming at first. Accordingly, no matter what your background, we recommend training, either by Versant or by third party vendors. Although our customers and industry studies report order of magnitude gains in productivity with the use of object programming and object databases, most also report that an initial period of training in the concepts involved has a high payoff.

Versant also offers on-site consulting, which can help during the start of the programming phase of a project.

For information about Versant training classes or on-site consultation, please call the Versant Training and Consulting Group.

Once you understand the underlying concepts, you will find Versant to be logical and predictable. Versant is an expression of the object model of data with familiar and natural models for database management and interface usage. However, the object model, normal database procedures and protocols, and Versant must all be understood in order to create efficient application programs.

Versant is Powerful and Flexible

Versant has most of the features and options found in modern database systems.

For example, Versant has four different ways to retrieve objects, seven types of locks and numerous transaction alternatives. Among other things, you can move objects among databases, change schemas, tune performance, create classes at run time, invoke database administration utilities at run time and move databases after they have been created.

If you have a database background, you can understand that Versant provides database features that go far beyond mere object storage. To benefit from these features, you must know how to use them. For example, To use the Versant to its best you must know when to use update locks or else, in a workgroup environment, your program will tend to get deadlock errors. You must use group operations to retrieve and update large numbers of objects, to reduce network traffic. You must use cursor queries instead of simple queries to reduce the size of the result set. Also, you must remember to turn locking and logging `ON` when using Versant in a production environment, and, as with any production system, you must remember to perform frequent database backups.

Versant uses Object Languages

Versant supports all computational and control features found in its interface languages. It also extends these languages to provide database, memory management and process control features.

For example, Versant has numerous functions that control memory, five options for session memory workspaces, low-level and high-level variable-length storage types, provision for controlled use of shared memory by multiple processes, and multiple language, hardware, and compiler interfaces.

If you have a database background, you must learn about the best way to implement various tasks. For example, you must know that although links are always safe, you must not use pointers to unpinned objects.

Versant Implements the Object Model of Data.

Versant implements the object model of data, which provides an extraordinary amount of power and flexibility.

For example, Versant allows you to define your own data types and establish inheritance, embedded, and association relationships among objects. The advantages of using objects include dramatic gains in programming productivity and the ability to create applications that were not previously practical. However, these gains depend upon an understanding of the object model.

It is important to have knowledge about the differences between `is-a` (inheritance), `has-a` (association), and `contains-a` (embedded) relationships, and that most objects are found by following links rather than by performing queries.

If you have a procedural programming background, you must learn about messages and classes and learn that switch statements are rarely needed.

Versant Extends the OS Functionality

Versant extends operating system functionality, which provides important benefits in a heterogeneous, workgroup environment.

For example, Versant provides heterogeneity among platforms, compilers, and languages and provides controlled use of shared memory. It uses a client/server model, which means that Versant is scalable and provides beneficial use of additional hardware resources.

Versant Concepts are Orthogonal but Mutually Supportive

Versant is a complete database system with orthogonal session, locking, interface, usage, versioning, and object management models. Because they are orthogonal, the usage, database, and interface models provide flexibility; however, they must be used together in a consistent manner in order to achieve results. For example, you must be in a session before you can interface with a database.

Some elements of Versant, such as the data model and database administration, are language independent. Other elements, such as the interface model, rules, conventions, and usage are language specific. Also, Versant users have, inevitably, different backgrounds and skill levels.

Since interface dependent and interface independent elements are orthogonal but interrelated, a systematic presentation of Versant is necessarily arbitrary. There is no sequential, lock-step way to explain Versant or, for that matter, any other database system.

IMPLEMENTING VERSANT

Versant should be used in the context of a good software engineering practice.

The SDLC Perspective

As with any programming project, the principal phases of a Versant application development effort are analysis/design, development/testing, and deployment/ maintenance.

Following are brief comments related to each of these project phases.

Analysis

You must have a firm understanding of your situation and project goals before beginning development of your application.

Design

Experience suggests that the design phase of software development is even more critical to the success of object oriented projects than it is to procedural programming projects. The reason is that the object programming development lifecycle becomes shorter with a higher percentage of the time spent understanding the problem to be solved and designing the needed classes. If you have a good design for the classes in your application, implementation proceeds far more smoothly than with non-object oriented languages. We recommend using a design methodology such as Booch and Rumbaugh along with a CASE tool to record your design.

Development/Testing

During the Development phase of software development, it is critical to understand that database transactions and the support of multiple users are significant considerations. Experience suggests that you cannot "retrofit" transactions and concurrency into an application that assumes all of its data is in memory and, thus, ready to be modified at will. Also, you should carefully identify whether there are any bottleneck objects that must be write-locked by many transactions and will thus be a disproportionate source of lock contention.

Important development issues are:

1. Review of the object model (classes and associations) in the light of implementation issues.
2. Navigation model (how do you move from object to object.)
3. Transaction model (how do you maximize concurrency and define units of work.)
4. Distribution model (how to maximize available resources.)
5. Usage model (how users interact with the application and what tasks are most performed.)
6. Performance tuning.

Testing is a key component of any development project. Test to be certain that each of your classes do what you expect. Test your database code not just with a single user, but also with the use load you expect in production. Often, multi-user testing uncovers subtle flaws in an application's locking and data-sharing strategy.

Class-level unit testing is essential early in development to catch minor design problems before they turn into major rework headaches. Integration testing and multi-user testing is essential to ensure the system works as you intend. Acceptance testing is critical to ensure that the application meets the user's expectations.

Because design and development work interact, you will probably want to iterate through the design and development stages numerous times. You will probably also want to iterate through development and deployment, where a "semi-final" deployed product results from each iteration. For example, performance tuning is an issue that is often considered during development/ deployment iterations.

Deployment / Maintenance

You should design and implement your application with deployment and maintenance in mind. How will users build, maintain, backup and restore your database? Where will your software reside on the system? How will you reliably move your software from your development environment to your user's machine? How will you handle upgrades to your software and database schema?

Versant provides tools and services to assist with all these tasks, and you can use third party tools with Versant, but the primary responsibility to ensure that each phase is successful rests with you, the software engineer and project manager.

Implementing Tips

Based upon our experience with customers, following are some implementing tips for using Versant. These are a few suggestions based upon common problems.

Spend time on initial design

Most developers do not spend enough time mapping logical designs to physical designs. The most important thing is to identify the various objects that will be used and the operations that will be performed on them.

Prototype with a realistic amount of data

Most projects run fast with fifty objects, but many design issues, bugs, and performance problems tend to surface only when databases similar in size to the target production environment are used.

Test with multiple users

When an application is deployed, concurrency issue is often the most important factor in performance.

Use the same hardware as the user

Developers tend to have high performance machines, so mostly the problems are not foreseen. It is important to determine the exact hardware of the end user while running an application, maybe with more modest machines.

For example, an application may run quickly on a machine with a large amount of physical memory but become sluggish on a smaller machine due to virtual memory paging.

Gather and log performance data on a regular basis

We suggest that you write monitoring code that can be turned `ON` and `OFF` and also write external scripts that log resource consumption in various sectors of your programs. This kind of Data is extremely useful both in debugging and in performance tuning.

Use your computers around the clock

When you are not using your machine, you can use some scripts to perform database administration, run batch jobs, prepare reports, and run long test programs.

One of the most important factors in successful deployment is, rigorous testing under a wide variety of conditions. Sometimes, problems don't show up until a program has been running for days or weeks or tested with extremely large numbers of objects or users.

If you run your machine constantly, every time you run into a problem, you can add a unit test to a growing suite of automated tests that can be re-run every time you make a change of any kind.

Keep transactions short, hold locks for a minimum amount of time.

Open transactions hold locks, so avoid "think times" in which a transaction is open while the application waits for user input. Holding key objects locked will block other users from doing work.

Alternatives are to use timeouts during screen input/output or to gather all input from the user before updating objects.

Use update locks when appropriate

In a multi-user environment, you cannot assume that you will always be the next person to convert a read lock to a write lock.

For example, programming the following sequence of events is deadlock prone:

```
begin transactionread objects
write lock selected objects
update selected objects
end transaction
```

Update locks allow you to read objects without blocking other readers. They have the additional advantage of guaranteeing you the next write lock.

Invest in trainings on a regular basis

Training in object languages and typicallyVersant, pays for itself ten times over. There is so much power and flexibility in using objects that taking time to just learn what can be done is time well spent.

For more information on Trainings, please contact Versant support.

CHAPTER 2

Objects

This chapter describes Versant objects and Versant object data model.

Following topics are covered:

- Object Types
- Object Elements
- Object Characteristics
- Object Status
- Object Relationships
- Object Migration
- Object Sharing

OBJECT TYPES

Versant databases model data as objects.

A database "object" is a software construction that can encapsulate data and can reference to other objects. Usually, the generic term "object" refers to an object containing data, but it can also refer to other kinds of objects.

Software objects are a response to programming issues, inherent in large and complex software applications. Objects are useful if you want to organize large amounts of code, handle complex data types, model graph structured data, perform navigational queries, and/or make frequent modifications to applications.

The term "object" has numerous meanings, which are clear in context.

Following is listing of object types:

Instance object

A database object that holds data is called an "instance object". In a Versant database, an object has a class that defines what kinds of data are associated with the object. Elemental (immediate) values are stored as typed values.

Class object

When you create a class, Versant creates a special kind of object, called a "class object" or “schema object," that stores your data type definition in a database.

You can define classes to contain any kind of data, including values, images, sounds, documents, or references to other objects. Only a very few data types are difficult to implement in object databases.

For example, bitfields can cause problems in heterogeneous systems, and pointers to functions are difficult to implement although easy to avoid.

C++/Versant: Objects are defined in a class using normal C++ mechanisms. Versant creates for each class an associated runtime type identifier object that contains additional information needed by C++. This runtime type identifier object is an instance of the C++/Versant class `PClass`. Each instance object has an `is_a` pointer to its type identifier object in order to identify its class.

Transient object

A transient object exists only while the program that created it, is running.

Persistent object

A "persistent object" is an object that can be stored in a Versant database. Persistent objects must derive from the Versant `PObject` class.

When a complex object is read from a database, only those portions of it, which are examined by the user will actually be loaded into memory, allowing "large" objects to be lazily instantiated on a per object basis. Performance options are provided so that you can explicitly control how referenced objects are read in.

C++/Versant: Both transient and persistent objects are dereferenced with the same syntax.

Stand-alone object

C++/Versant: A stand-alone object is one created with the C++ new operator, with a C++/Versant new operator or function, or an object defined as a local or global variable.

Embedded object

C++/Versant: An embedded object is an object that is an attribute of another object.

OBJECT ELEMENTS

Although objects are used as if they were a single structure, the implementation of objects involves several discrete elements.

Object attributes

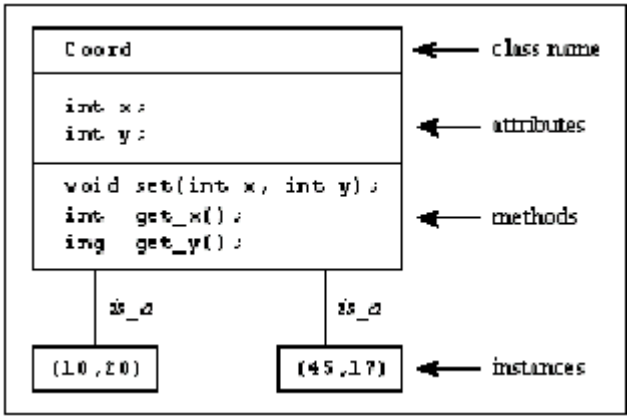
C++/Versant — The parts of an object that store data and associations with other objects are called "attributes". An attribute is persistent when the object containing it is persistent. An attribute can be a literal whose elemental value is self-contained, another object with its own attributes, an object that is an array of other objects, or a reference to another object.

Attributes are first defined in normal C++ class files. After class files have been compiled, attribute definitions are loaded into a database and stored in class objects. Actual data are stored in instance objects.

Object methods

C++/Versant: The parts of an object that store executable units of program code are called methods. Methods are defined and implemented in normal C++ class files. After the class files have been compiled, methods are linked with an application.

C++/Versant Example: For example, consider a C++ class named `Coord`.



Attributes of class `Coord` are the coordinates `x` and `y`. Methods of class `Coord` are `set()`, which sets the values of the coordinates, `get_x()`, which returns the value of the `x` coordinate, and `get_y()`, which returns the value of the `y` coordinate.

OBJECT CHARACTERISTICS

Following are characteristics of persistent objects:

Object identity

One of the strongest concepts in object technology is object identity, because it makes possible features such as persistent references to other objects and the ability to migrate objects among distributed databases without having to change code that accesses the objects.

Versant assigns each persistent object a unique identifier called its logical object identifier or `loid`. Logical object identifiers are composed of two parts: a database identifier and an object identifier.

The database identifier portion of a loid is an identifier for the creation database that is unique among all databases in a system of databases. This identifier is created by Versant when the database is created.

To ensure that database identifiers are unique, all databases in a system of databases have their identifiers stored in a single database system file named osc-dbid. When you create a database, your system must be able to modify the system file, but once a database is created you do not need to be connected to a network to perform tasks such as creating or managing objects in a local database.

The object identifier portion of a loid is an identifier that is unique within the creation database. This identifier is created by Versant when the object is created. Because the object portion of the identifier is based only upon other objects within the creation database, you only have to be connected to the creation database when you create an object.

The object identifier is built upon a database identifier which is unique within a system of databases. Thus the object identifier is unique within a database, thus assuring that it is unique among all objects of the system.

The overhead in memory of a Versant object identifier, on 32-bit CPU architectures, is approximately twenty-four bytes per persistent object, and four bytes for a non-persistent object. A link requires eight bytes. For a particular database system, you can address 2^64 objects in 2^16 databases.

Object migration

The ability to migrate objects makes distributed databases practical. After an object has been created, you may want to migrate it in order to place it physically closer to where it is most

often used, which will reduce network traffic, or migrate it because disk space is filling up on the machine containing the creation database.

Object and database identifiers do not change when an object or database is moved or changed. Identifiers are never reused, even after an object or database has been deleted.This means that code that references an object does not have to be changed each time an object is migrated.

Each persistent object must have a class object associated with it that contains its class definition. When you migrate an object to a new database, if it is not already defined in the target database, a copy of its class object is also migrated.

Migrating an object instance includes migration of its definition in a class object, causing a particular class object may be duplicated in numerous databases. However, once a class object has been migrated, it exists as an independent object in that database. This means that to use a migrated object, you do not have to be connected to its creation database.

If classes of the same name are defined differently in different databases, an attempt to migrate an object with a different class definition will be automatically blocked. You can then use any of a variety of mechanisms, such as a "synchronize class" routine, to resolve the differing class definitions.

Dynamic binding

When you define a new class that specializes its base class, you may want to redefine methods, such as a "print yourself" method, for the new class. Inherited methods can be redefined in sub-classes.

Polymorphism

To use a method, you call an object method with a message. For a method to respond, the message must have both the correct name and the correct signature. The number and data types of the arguments for a method are called its signature.

You can overload a method to work for several classes and a varying number of arguments; this is called polymorphism and can simplify control structures significantly. In effect, polymorphism binds a message to a particular method at run time in a manner similar to case statements.

Situation modeling

An important advantage of using the object model is its ability to model situations in a realistic manner that can be understood by others. Using classes to classify individual elements of a situation and then using embedded, inheritance and reference relationships to relate classes to one another results in a model that directly reflects the real situation. This is much more

understandable than a flat system of normalized tables whose logic must be reassembled each time they are used in each application that accesses them.

Schema modification

After you have completed a complex application, you will almost certainly want to extend or modify its underlying classes to accomplish new tasks.

The set of object definitions associated with a particular database is called the "schema" of that database. You can add to and change the schema of a database at any time. In many cases, you will be able to redefine classes by extending them to hold additional data, perform new tasks, and perform existing tasks differently without changing applications that use the classes.

Predefined types

Versant predefines many elemental and class types that you can use with embedded, inheritance, and association relationships.

The following types available in all interfaces:

Vstr — A vstr provides low overhead, variable length storage for one or many elemental values.

Link — A link stores a reference to another object.

Link vstr — A link vstr provides variable length storage for one or many references to other objects.

C/Versant

C/Versant also predefines the following type:

List — A list provides ordered variable length storage.

C++/Versant

C++/Versant also predefines the following types:

Bi-link — A bi-link stores a bi-directional reference.

Bi-link vstr — A bi-link vstr provides variable length storage for one or many bi-directional references.

Array — An array is an ordered collection of elements accessed by an index which is an integer.

Set — A set is an unordered collection of unique elements.

Dictionary — A dictionary is a collection that maps keys to values.

List — A list provides ordered variable length storage.

Date and Time — Date and time attribute types.

The C++/Versant interface also provides facilities to create the following types:

Parameterized — Parameterized types allow single definitions for classes that differ only by the data type of their contents. This is useful for classes, such as sets, that have identical functionality and differ only in the data type of their contents.

Run-time — Classes that are defined at run-time.

Versant also allows you to use third party class libraries to define your classes.

OBJECT STATUS

When you create or access a database object, Versant maintains information about its current status.

Following is a list of the status information maintained about each object.

Object transient, persistent status

When you create an object, Versant notes whether it is transient or persistent. Within the scope of a transaction, transient objects are treated the same as persistent objects, but when you perform a commit, only persistent objects are saved to a database.

Object lock status

Locks provide access guarantees to objects. For example, a write lock guarantees that you are the sole user of an object, which is important if you want to update the object.

Object dirty status

When you change an object, you must mark it as "dirty," which means that it will be updated at the next commit. This improves performance significantly, because at the time of a commit Versant does not have to compare the contents of each object you have used with its original contents in order to determine which objects need to be updated.

Object pin status

C/Versant and C++/Versant: To manage memory efficiently and to improve access to objects of current interest, Versant maintains an object cache in virtual memory that contains all the objects accessed during a transaction. An object in the cache can be "pinned," which means that it will not be swapped out of the object cache back to its source database. A pinned object is not guaranteed to be in process memory, just in virtual memory.

Normally, pinning occurs automatically when you access an object, but you can explicitly unpin objects if you are accessing a large number of objects and virtual memory is limited. Versant also allows you to set nested pin regions to make it easier to pin and unpin specific sets of objects.

OBJECT RELATIONSHIPS

Object embedded relationship

C++/Versant: An "embedded relationship" is created when you use an object as an attribute of another object. An embedded relationship can also be called a "contains-a" or "containment relationship." When you create instances, embedded objects are accessed via the containing object.

An object that contains an aggregation of embedded objects provides referential integrity and avoids orphaned data. Using embedded objects can improve performance, because access to an aggregation of objects is done in one step, and only one entity is locked. Embedding predefined types such as dates or custom designed sets can also save a lot of programming time.

Some C++/Versant classes can be used only as embedded objects, while others can be used as stand-alone objects.

Object inheritance relationship

When you define objects, you can specify a hierarchical structure of base and derived classes. In this hierarchy, descendant objects "inherit" data and methods from their ancestors. This means that you can store data in an instance as if it had all of the attributes that are in its ancestors, and you can manipulate an instance as if it had all of the methods that are available to its ancestors.

Inheritance is an efficient way to describe a situation: base classes describe once what is common to all of its derived classes, then each derived class describes only what is new or different about itself. Since derived classes inherit both attributes and methods, inheritance enables stable data and code definitions to be reused, extended, and customized.

An inheritance relationship is called an "`is-a`" relationship, because the derived class is a special form of its base class.

C++/Versant: In C++, there can be two kinds of inheritance: single inheritance and multiple inheritance. Single inheritance describes an object related to ancestors in a stack structure. Multiple inheritance describes an object related to ancestors in a tree of stacks.

Object association relationship

An important feature of the object model is the ability to associate objects that are logically related but that are of different types. The ability to associate objects allows you to create graph structured data relationships. For example, while inheritance is useful to define an employee as a kind of person, you must create an association to relate an employee to a department.

C++/Versant: The association of objects is similar to using a C++ pointer from one object to another, assuming that both objects have been assigned a virtual memory address.

However, pointers are not valid for objects in a database but not in memory and pointers are unreliable for objects that are not pinned in virtual memory

To create persistent database pointers, C++/Versant uses "links". Links are sometimes called "smart pointers", because they are valid regardless of the location of the object, whether in memory or in a database, and they remain valid even if you move the object from one database to another. The transparency of links is important, because it allows you to write code that does not depend upon the memory or database location of objects. Link relationships are called "`has-a`" relationships, because one object "has a" link to another object.

There are several performance advantages to using links. Just as the fastest way to find an object in memory is to use a pointer, it is much faster to find an object in a database using a link rather than a query. This means you can move quickly among related objects by traversing the graph of relationships rather than performing repeated queries and joins. Links also improve performance because linked objects are retrieved only when you decide that you actually want the object by dereferencing the link.

You can use either links or arrays of links as attributes. You can create one-to-one, one-to-many, many-to-one, and many-to-many associations using the predefined Versant link and link vstr data types. Versant also has data types for "bi-links," which provide referential integrity and cascaded deletes, and for containers, arrays, sets, lists, and dictionaries, which allow you to associate many objects into a single structure.

OBJECT MIGRATION

Versant allows you to move or migrate objects from one database to another.

Migrating an object moves it permanently from one database to another.

Migrating an object is useful when you want to distribute data to take advantage of available hardware, reduce network traffic by placing data near users, and/or move data from a development environment to a production environment.

Actions

Migrate object

```
c     o_migrateobj()
      o_migrateobjs()

c++   migrateobj()
      migrateobjs()
```

Object identity

When you migrate an object, the object identifier of the object is not changed. This means that you do not have to change code that references an object or update other objects that have links to the migrated object even though the object has been moved to a new location.

Locks

Migration routines acquire a write lock if the object does not already have one. This ensures that you are the sole user of the object when you migrate it.

Schema

When you migrate an object, copies of class and superclass objects are made in the target database. If a class or superclass exists with the same name in the target database but with a different definition, an error will be returned. To resolve differences in schema definitions, you can use the "synchronize" method or other interface specific mechanisms.

Commits

A migration is not persistent until committed. If multiple objects are migrated in a transaction, either all the objects will be migrated if the transaction commits successfully or all objects will be returned to their original databases.

Links to objects in personal databases

When you migrate objects created in a personal database to a group database, make sure that the migrated objects do not reference objects left in a personal database.

OBJECT SHARING

Sharing C and C++ Objects

This explains the mixing of C/C++ Objects. It informs about the addition of the header files and C/C++ functions, so that the database can be used in any C or C++ application.

The best way to handle objects created with C++/Versant is with C++ applications and vice versa. To handle both C and C++ objects in the same application, the easiest way is to write in C++ and use the C++ interface to handle the C++ objects and simultaneously use the C Interface to handle the C objects. The only times that it makes sense to mismatch interfaces is when you want a C object to reference a C++ object, or vice versa.

Use the following approach to access C objects with the C++ Interface:

- Include the C interface header file in the following manner:

  ```
  extern "C" { #include "omapi.h" }; // right way
  ```

  If you include the C interface header file as:

  ```
  #include "omapi.h" // wrong way
  ```

  Your C++ compiler will think that the C/Versant functions in `omapi.h` should have a C++ linkage, which will result in linker errors for each C/Versant function referenced.

  Also, you should include `cxxcls/pobject.h` before `omapi.h`.
- For a link, use LinkAny or `o_object` instead of Link<type>.
- For vstrs, use either Vstr<type> or VstrAny.
- For link vstrs, use LinkVstrAny instead of LinkVstr<type>.
- To retrieve objects from a database, use the C function `o_locateobj()` instead of the C++ dereference operators ->, `type*`, or `PObject*`.
- To acquire a write lock and mark an object for update, use `o_preptochange()` instead of the C++ `PObject::dirty()` method.
- To modify an attribute, acquire a write lock, and mark an object for update, use `o_setattr()` instead of a C++ method.
- To create a new C object, use `o_createobj()` rather than the C++ `O_NEW_PERSISTENT()` syntax.

To access C++ objects with the C Interface, use `o_locateobj(), o_setattr(),` and `o_getattr()` in a normal manner to retrieve and access C++ objects.

If your C++ object uses simple inheritance, you can use casting to a `struct` pointer to access the fields. Objects deriving from `PObject` have one attribute of type `PClass*` that will show at the top of your objects. You cannot run the methods on an object created with the C Interface since

the internal C++ pointers are not initialized properly.

All C++/Versant collection classes may be used.

You can use C/VERSANT to access and manipulate all C++/Versant collection classes if you implement hash functions in both C++ and C, that operate in the two environments in the same way. In other words, for each member, your hash function must return the same hash value in both C++ and C.

Sharing JVI and C++ Objects

JVI enables transparent sharing of C++ objects with Java programs.

The sharing is enabled through generating "Java Foreign View" source files that correspond to the schema defined from C++. These `.java` source files must be compiled and enhanced with other classes in the Transparent JVI program.

The Foreign View approach has the following features:

- The object sharing is one-directional.
- The Transparent JVI program can access database schema for a class created from a language other than Java. Sharing the schema of a class defined from a Transparent JVI progra with other languages transparently, is not addressed.
- Transparent sharing of the object attributes is supported, but not the methods.
- However, you can write methods in the generated source files of classes and then use these methods as you would in any other persistent Java class.

For more information, on object sharing and a step-by-step tutorial, please refer to the on-line tutorial provided with your JVI installation.

CHAPTER 3

Sessions

This Chapter explains basic Session concepts in Versant.

Following topics are covered:

- Session Boundaries
- Session Memory Areas
- Session Elements
- Session Types
- Units of Work
- Session Operations

SESSION BOUNDARIES

Applications perform Versant work in sessions.

You must start a session to use Versant databases, methods, data types, and persistent objects. The only exceptions are methods that set session and environment parameters.

A process or thread can be in only one session at a time.

C++/Versant: When you are using C++/Versant, transient objects of Versant data types and transient objects of classes derived from `PObject` that were created before a session should not be modified in a session. Transient objects of Versant data types and transient objects of classes derived from PObject that were created or modified during a session should be deleted before the end of the session.

SESSION MEMORY AREAS

When you start a session, Versant creates the following session memory elements:

The object cache, cache table, and session tables may be created in client machine memory. The server page cache is in shared memory on the machine containing the session database. All these memory areas are maintained by Versant.

Object cache

An object cache in virtual memory improves access to objects used in a transaction.

Object cache table

A cached object descriptor table tracks the location of all objects referenced during a session.

Session tables

Various session information tables track your processes, connected databases, and transactions.

Server page cache

Associated with the session database and each connection database is a page cache for recently accessed objects.

SESSION ELEMENTS

The following Elements are also associated with sessions:

Session database

A session database is the initial default database. It is used to manage transactions that occur during a session and to coordinate two-phase commits.

Either a personal database or a group database can serve as a session database. Only one personal database can be accessed in a session, which must be the session database. More than one group database can be accessed from a session database.

Specification of a session database does not change the database association of an object.

To change the database used as the session database, you must end the session and start a new session.

Session name

The session name is used as the name of the transactions that occur during the session.

Default database

A database designated as the default database is the location of new persistent objects and is used by many functions when a null database name is specified as a parameter. Initially the default database is the session database, but you can change the default after a session has started.

Connected databases

Once a session has started, you can connect to other databases. Each database connection starts a new server process on the database machine.

Transaction

When you start a "standard" session, Versant begins keeping track of your activities in a transaction.

SESSION TYPES

When you begin a session, depending upon your interface language, you may be able to specify the kind of session that you want.

Versant provides the following session options:

Standard session

In a standard session, you are always in a transaction.

Almost all tasks can be performed using transactions with commits, checkpoint commits, rollbacks, and savepoints. The standard Versant locking model satisfies most concurrency requirements.

If you do not make a specification for session type when you start a session, you will start a standard session.

Multiple threads and multiple sessions

You can start a session in which you can place one or more threads. If you start this kind of session, you can also start additional sessions, and each of the additional sessions can have zero or any number of threads in it

See also “Thread Management” on page 179 in "Chapter 9 - Thread and Session Management".

Optimistic locking session

An optimistic locking session suppresses object swapping, prevents automatic lock upgrades, and provides automatic collision notification.

See also “Optimistic Locking Actions” on page 111 in "Chapter 5 - Locks and Optimistic Locking".

UNITS OF WORK

Following is the hierarchy of possible Versant units of work in various kinds of sessions. (Some interfaces do not allow all types of sessions, such as thread sessions.)

```
Standard session
    Transaction
        Savepoint
Standard session with threads
    Transaction
       Savepoint
```

Sessions can be a sequence of none to many. Transactions are a sequence of one to many within a session. Savepoints are a sequence of none to many within a transaction.

See also “Transaction Overview” on page 84 in "Chapter 4 - Transactions".

SESSION OPERATIONS

Begin session

Start a session, start a transaction.

```
c     o_beginsession()
c++   beginsession()
```

End session

End a session, commit the current transaction, disconnect the application process from all databases, close all session memory workspaces, and commit, rollback, or continue the current transaction.

```
c     o_endsession()
c++   endsession()
```

End process and end session

End a session if it has not already ended, terminate the application process, and either commit or roll back the current transaction, depending upon the option supplied.

```
c     o_exit()
c++   exit()
```

Starting Session Firewall

You must start a session before using Versant functions, methods, and persistent objects.

The only exceptions are functions and methods that set session parameters.

Connecting to the Database

You must explicitly connect with a database before accessing it.

To connect with the session database, use a "begin session" function or method.

After beginning a session, to connect with other databases, use a "connect database" function or method. Connecting to a database does not change the default database. To change the default database, use a "set default database" function or method.

CHAPTER 4

Transactions

This Chapter explains Versant Transactions.

Following topics are covered:

- Transaction Overview
- Actions
- Hierarchy
- Usage Notes

TRANSACTION OVERVIEW

A transaction is a logical unit of work whose results are either saved or abandoned as a group.

Transactions are an essential database concept, because they ensure that data is always in a known state, even in a distributed database environment.

As soon as you start a session, Versant begins keeping track of your activities in a transaction.

Transaction States are:

Atomic: When a transaction ends, the results of all actions taken in the transaction are either saved or abandoned. This is an important database feature, because it ensures that data is always in a consistent and known state.

Durable: Results are either saved permanently or abandoned permanently - no further undo or redo operations are possible.

Independent: When you are operating on locked objects in a transaction, no other user can intrude upon your work. While in a transaction, you can operate on your objects as if you were the sole user of a database.

Coordinated: Objects in a transaction are locked, which means that your work is co-ordinated with other users in a workgroup environment.

Distributed: A two phase commit protocol ensures that data is always in a known state even when you are working with objects in numerous databases in a distributed database environment.

Ever-present: You are always in a transaction. When you end a transaction, another is automatically started for you.

In advanced cases the User may want to create special kinds of transactions. In such case the User can start sessions with multiple processes in a transaction.

A session is also a unit of work.

For more information refer to “Session Types” on page 79, in "Chapter 3 - Sessions".

TRANSACTION ACTIONS

Commit transaction

A commit saves your actions to the databases involved and releases short locks. A commit also releases objects from cache memory, erases all savepoints and starts a new transaction. A commit makes no changes to transient objects.

Changes made in a transaction are not visible to other users until you commit them. This means that others cannot be confused by seeing partial changes. However, objects that have been flushed to their database (by swapping, queries, or deletions) will be visible to users using null locks in dirty reads.

```
c      o_xact()
c++    commit()
       xact()
```

Checkpoint commit transaction

A checkpoint commit performs a save and holds object locks. A checkpoint commit also maintains objects in cache memory, erases all savepoints and starts a new transaction. A checkpoint commit makes no changes to transient objects.

Changes made in a transaction are not visible to other users until you commit them. This means that others cannot be confused by seeing partial changes if they are using read locks. However, objects that have been flushed to their database (by swapping, queries, or deletions) will be visible to users using null locks in dirty reads.

```
c      o_xact()
c++    checkpointcommit()
       xact()
```

Rollback transaction

A rollback abandons your actions, releases short locks, and restores databases to conditions at the last commit or checkpoint commit. A rollback also releases objects from cache memory, erases all savepoints, and starts a new transaction. A rollback makes no changes to transient objects.

If your application or machine crashes, Versant will automatically roll back the current incomplete transaction.

```
c     o_xact()
c++   rollback()
      xact()
```

Begin session

Beginning a session starts a transaction.

```
c     o_beginsession()
c++   beginsession()
```

End session

Ending a session performs a transaction commit.

```
c     o_endsession()
c++   endsession()
```

Set savepoint

Setting a savepoint creates a snapshot of database conditions to which you can selectively return without ending the transaction. Setting a savepoint makes no changes to databases, locks, transient objects, or the memory cache.

When a savepoint places a record of current conditions in the database log, changes to an object are visible to other users if they access the object with a null lock in a dirty read.You can set as many savepoints as you want in a normal transaction.

Using savepoints is a way of working down a set of instructions toward a solution and then backtracking if some sub-set of the solution fails.

Savepoints are not compatible with optimistic locking, because setting a savepoint flushes objects to their databases, which resets locks.

```
c     o_savepoint()
c++   savepoint()
```

Undo to savepoint

Undoing a savepoint returns database conditions to the immediately previous savepoint; if there is no immediately previous savepoint, conditions will be restored to those that existed at the last commit.

Undoing a savepoint makes no changes to databases, locks, or transient objects, but it does invalidate the object memory cache.

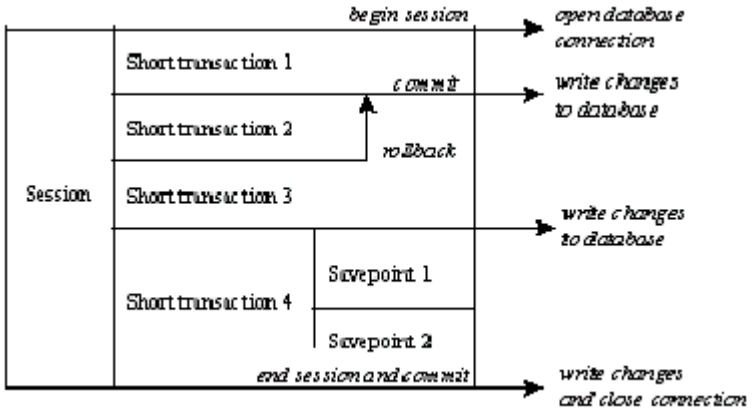
```
c      o_undosavepoint()
c++    undosavepoint()
```

TRANSACTION HIERARCHY

The transaction hierarchy is:

```
Session
   Transaction
      Savepoint
```

For example:



Changes in transactions 1, 3, and 4 are committed to the database, while changes made in transaction 2 are discarded.

Memory Effects of Transaction

After a transaction routine has executed, the following effects take place on memory:

Memory effects of a transaction	undo savepoint	rollback	checkpoint commit	commit	end session
Cache and cache table Invalidate object cache Invalidate cod table	 yes no	yes no	no no	yes no	 no no
Afterwards for C Can use `o_object` in a variable Can use a vstr Can use pointer to persistent obj	yes yes no	yes no no	yes yes yes	yes no no	no no no
Afterwards for C++ Can use link held in a variable Can use transient object created in a session modified in a session Can use pointer to persistent obj	yes yes yes no	yes yes yes no	yes yes yes yes	yes yes no	no no no no

USAGE NOTES

Following are usage notes related to transactions:

Turn logging ON

To use transactions in your application, you must enable logging in your database.

For more information, on turning logging ON or OFF, please refer to "logging" parameter in the chapter "Database Profiles" in the *Versant Database Administration Manual*.

Turn locking ON

To use short locking in your application, you must enable locking.

For more information, on turning locking ON or OFF, please refer to "locking" parameter in the chapter "Database Profiles" in the *Versant Database Administration Manual*.

Keep them short

Keep transactions short, because locks held by transactions block other users and the work which is not committed will be lost if the application or machine crashes.

Commit when states are consistent

Commit a transaction only when the states of the objects you are working with are internally consistent.

Build in transactions

Build transactions into your application from the beginning. Transactions are essential to data consistency and workgroup concurrency, yet they can be hard to retrofit into an application as they are a major structural element.

Do not commit with a link to a transient object

If you have an attribute of a persistent object that contains a link, link vstr, bilink, or bilink vstr, cannot contain a link to a transient object at the time of a commit.

CHAPTER 5

Locks and Optimistic Locking

This Chapter explains Versant Locks. Short Locks and Short Intention locks are explained in detail.

Following topics are covered:

- Locks and Transactions
- Short Locks
- Short Intention Locks
- Short Lock Precedence
- Optimistic Locking
- Multiple Read Inconsistencies
- Optimistic Locking Protocol
- General Usage Notes
- Optimistic Locking Examples

LOCKS AND TRANSACTIONS

Overview

Locking of objects is done in multi-user systems to preserve integrity of changes; so that one persons changes do not accidently overwrite anothers.

Locks on data allow multiple processes to access the same objects in the same database, at the same time in a co-operative, controlled, and predictable manner. Thus locks are very essential in a multi-user environment. If there is only one user of a database, there is no need of locks.

Locks and transactions are conceptually related.

Short locks and transactions

Transactions are by definition, units of work in which concurrency is provided by short locks. Typically for concurrency reasons, short locks are held for short amounts of time.

Short locks are released when a transaction ends with a commit or rollback. Accordingly, for convenience in managing short locks, transactions typically define units of work that take a short amount of time, such as seconds or minutes. However, since Versant provides mechanisms for upgrading and downgrading locks on specific objects and for writing specific objects to a database, transactions do not have to be short.

Ending a session also ends the current transaction (you can specify ending a session with either a transaction commit or rollback). A short lock does not survive a system disruption.

Features of Locks

Following are features of Versant locks:

A lock provides an access guarantee.

A lock on an object provides an access guarantee. The access guarantee is provided by blocking certain kinds of actions by other users.

Locks are applied to objects.

To maximize concurrency, locks are applied at the object level.

Any object can be locked.

Locks can be set on all kinds of objects: instance objects, class objects etc.

Locking a class object locks all objects of the class.

Placing a lock on a class object has the effect of placing the same lock on all instances of a class. This is far faster than retrieving all objects of a class and placing individual locks on each object.

Linked objects are not locked.

Locking an object does not lock objects which are the targets of links in the locked object.

Locks have precedence.

The precedence of locks is: write > update > read > null. An exception to this rule occurs if you are using multiple processes in the same session. In this case, different processes can place different locks if they are compatible. For example, different processes can request and receive a read lock and an update lock on the same object, but the net effect is the same to outside users: the highest lock prevails.

An object can have only one lock in a transaction.

In a particular transaction, an object can have only one lock on it at a time. If you separately request different locks on the same object, the lock placed depends upon the relative precedence of the locks. For example, if you request a read lock on an object for which you already hold a write lock, the read lock request is ignored; if you request a write lock on an object for which you already hold a read lock, the read lock is replaced by a write lock.

Locking an object also prevents the class object from being changed.

When you place a read, update, or write lock, Versant internally places a special kind of lock on the class object which prevents it from being changed while the instance object is locked.

You can turn locks off.

You can enable or disable short locks.

Strict two-phase locking is used.

Versant uses a strict, two-phase locking strategy. By "strict" it means that locks are set before work starts, rather than at commit time when locks might not be available. Strict locking prevents other applications from modifying an object while you are using it and is the appropriate strategy in a multiple user environment. "Two-phase locking" indicates that all locks are gathered in one phase and then released in a second phase, when the transaction ends.

Implicit locking is used.

By default, Versant uses an "implicit" locking strategy in which methods automatically obtain locks as needed. For example, marking an object as dirty automatically obtains a write lock. You can override the Versant implicit locking strategy in several ways. You can explicitly upgrade locks, change the default lock, and/or change the lock wait time.

SHORT LOCKS

Short locks are set at the object level for concurrency control. There are various types of short locks viz. write locks, update locks, read locks and null locks (used for dirty reads).

Short Lock Types

Following are the types of short lock modes and their access guarantees.

Short write lock

A write lock guarantees that you are the sole user of an object and that you are looking at the current state of an object. It is useful when you want to update an object. A write lock provides its guarantee by blocking all other requests for a write, read, or update lock on a particular object. A request for a write lock is blocked if the object already has a read or update lock.

```
c      WLOCK
c++    WLOCK
```

Short update lock

An update lock allows you to read an object and get the next available write lock on it. It is useful if you want to look at an object now while knowing that you will later want to update it. An update lock provides its guarantee by blocking all other requests for a write or update lock. It does not block other requests for a read lock, but if another user tries to change a read lock to a write lock, that request may cause a deadlock error that they must handle or have their application terminated. A request for an update lock is blocked if the object already has a write or update lock.

```
c        ULOCK
c++      ULOCK
```

For more information refer to “Short Lock Protocol” on page 102, in "Chapter 5 - Locks and Optimistic Locking".

Short read lock

A read lock guarantees that an object will not be changed while you are looking at it. It is useful when you want to look at an object but not change it. A read lock provides its guarantee by blocking all other requests for a write lock. It does not block other requests for a read or update lock. A request for a read lock is blocked if the object already has a write lock.

```
c         RLOCK
c++       RLOCK
```

Short null lock

A null lock, also called a "snapshot" lock, provides no access guarantees and, strictly speaking, is not a lock at all. Specifying a null lock is useful when you want to look at the current state of an object without placing a lock or waiting for other locks to be released. Looking at the current state of an object without placing a lock is sometimes called a "dirty read," because there are no guarantees that the object will not be changed while you are looking at it. That is, a null lock does not block other requests for a write, read, or update lock. A request for a null lock is never blocked.

```
c         NOLOCK
c++       NOLOCK
```

Short Lock Interactions

A lock provides a guarantee for a specific set of actions. To provide their guarantees, certain kinds of locks block other locks. For example, a write lock blocks a read lock, so that the process with the write lock is the sole user of the object.

There can be two kinds of lock interactions:

Interactions of normal locks

Interactions of normal locks occur when a process requests a lock on a object that already has a lock: to be granted, the guarantees of the new lock must be compatible with the guarantees of any existing locks. These interactions are the same for both instance and class objects.

Interactions of normal and intention locks

Interactions of normal and intention locks occur when a process requests a lock on an instance and the system then attempts to place a matching intention lock on its class object: to be granted, the guarantees of the new intention lock must be compatible with the guarantees of any existing locks.

The interactions of normal locks are relatively straightforward. For example, multiple read locks on the same object are compatible, but an object can have only one write lock.

The interactions of normal and intention locks are more logically indirect, because instance locks cause class object intention locks:

- A lock on an instance object can block a request for a lock on a class object.

For example, a read lock on an instance of a class blocks a request for a write lock on a class object, because a read lock on an instance also sets an intention read lock on the class object, and an intention read lock blocks a write lock.

- A lock on a class object can block a request for a lock on an instance object.

  For example, a write lock on a class object blocks a read lock on an instance of the class because to get a read lock on an instance, you must also get an intention read lock on the class object. However, a write lock blocks an intention read lock.

This behavior means that

- The definition of a class cannot change while an object is locked.
- Locking a class object with a normal, non-intention lock has the effect of locking all instances of a class.

Locks interact in the following ways:

Lock Type	Description
write lock	Blocks write, read/intention write, intention write, update, read and intention read
read/intention write lock	Blocks write, read/intention write, intention write, update and read locks.
intention write lock	Blocks write, read/intention write, update and read locks.
update lock	Blocks write, read/intention write, intention write and update locks.
read lock	Blocks write, read/intention write, and intention write locks.
intention read lock	Blocks write lock.
null lock	Blocks nothing.

The following table summarizes how locks interact.

Short Lock Interactions	write	read/ inten. write	inten. write	update	read	inten. read	null
write	blocks	blocks	blocks	blocks	blocks	blocks	

read/inten. write	blocks	blocks	blocks	blocks	blocks
intention write	blocks	blocks		blocks	blocks
update	blocks	blocks	blocks	blocks	
read	blocks	blocks	blocks		
intention read	blocks				
null					

Blocking of locks

The following summarizes how short locks interact.

This lock...	*...blocks this lock:* write	update	read	null
write	blocks	blocks	blocks	---
update	blocks	blocks	---	---
read	blocks	---	---	---
null	---	---	---	---

When you request a short lock, the system immediately tries to obtain it. A short lock is granted on an object if the following conditions are true:

- There are no incompatible short locks.

For example, a read lock is not granted on an object that already has a write lock.

If your request succeeds, the system places your short lock immediately.

If your request for a short lock, is blocked by an incompatible short lock, your application pauses either until the object becomes available or until your request times out. If your request times out, you will receive a "lock timeout" error. The length of time a request waits is determined by the value of the `lock_wait_timeout` parameter in the Server Process Profile.

Deadlocks

Potential deadlocks are detected immediately and a "would-cause-deadlock" error is sent to the user creating the potential deadlock. For example, if two users have read locks on an object and then both request a write lock, the first user to request a write lock is blocked, and the second user receives a "would-cause-deadlock" error message. (In this case, the first user would receive the lock if a lock wait time-out did not occur.)

Versant directly handles complex, single-database deadlocks with any number of clients and any number of locked objects. (An example of a "complex" deadlock is "A waiting for B waiting for C waiting for D waiting for A".) Versant uses timeout mechanisms to detect multiple database deadlocks.

Short Lock Actions

Set short lock explicitly

The following set a short lock explicitly.

Component	Function	Description
c	`o_acquireslock()`	set short lock
	`o_upgradelock()`	upgrade lock
c++	`acquireslock()`	set short lock
	`upgradelock()`	upgrade lock

Set short lock implicitly

The following set a short lock implicitly or allow you to specify a short lock in a parameter.

Component	Function	Description
c	`o_acquireilock()`	set intention lock
	`o_deleteobj()`	delete object
	`o_dropattr()`	drop attribute
	`o_dropclass()`	drop class
	`o_dropinst()`	drop instances
	`o_gdeleteobjs()`	delete objects
	`o_getclosure()`	find object and linked objects
	`o_greadobjs()`	get objects

	o_locateobj()	get object
	o_new_attr()	create attribute
	o_preptochange()	set object dirty
	o_refreshobj()	refresh object
	o_refreshobjs()	refresh objects
	o_renameattr()	rename attribute
	o_select()	find object
	o_setdirty()	dirty object
c++	(type*)()	cast link
	acquireilock()	set intention lock
	delete()	delete object
	deleteobj()	delete object
	dirty()	set object dirty
	gdeleteobjs()	delete objects
	getclosure()	find object and linked objects
	greadobjs()	get objects
	locateobj()	get object
	operator*()	dereference link or pointer
	operator ->()	dereference link or pointer
	operator delete	delete object
	preptochange()	set object dirty
	refreshobj()	refresh object
	refreshobjs()	refresh objects
	select()	find object
	setdirty()	dirty object

Release short lock explicitly

The following release short locks explicitly.

Component	Function	Description
c	o_downgradelock()	downgrade lock
c++	downgradelock()	downgrade lock

Release short lock implicitly

The following release short locks implicitly.

Component	Function	Description
c	o_endsession()	end session
	o_endtransaction()	commit or rollback
	o_exit()	exit process
	o_xact()	commit or rollback
	o_xactwithvstr()	commit or rollback
c++	abort()	rollback
	commit()	commit
	endsession()	end session
	exit()	exit process
	rollback()	rollback
	xact()	commit or rollback
	xactwithvstr()	commit or rollback

Get default short lock

Component	Function	Description
c++	get_default_lock()	get default lock

Set default short lock

When a session begins, the default short lock is a read lock. You can reset the default.

Component	Function	Description

c	o_beginsession()	begin session
	o_setdefaultlock()	set default lock
c++	beginsession()	begin session
	set_default_lock()	set default lock

No effect

None of the following affects short locks.

- checkpoint commit
- pin object
- release object
- set savepoint
- undo savepoint
- unpin object
- write objects
- zap object cache

Short Lock Protocol

The goal of a locking protocol is to maximize concurrent use of objects. Locking protocols are voluntary, but if not followed by all users, unexpected situations may occur.

A "deadlock" occurs when two transactions both hold read locks on the same object, both transactions then attempt to upgrade their locks to a write lock, and neither transaction can continue because both are waiting for the other to release their read lock.

To avoid deadlocks and maximize concurrency, the following voluntary locking protocol is recommended:

- If you want to snapshot read the current state of an object, then request a null lock. You will then get the object with no waiting and no blocking. Of course, the state of the object you get may become obsolete almost immediately.
- Use a read lock for objects that you know you do not want to modify. This allows others also to read the object.

- Use an update lock to read objects that you may later modify. This allows others also to read the object. If you decide to change the object, then request a write lock at that time. This keeps your write lock as short as possible.
- Never upgrade directly from a read to a write lock, because this can create a deadlock situation. Also, do not upgrade from a read lock to an update lock: get an update lock right from the beginning.

C++: Dirtying an object is the same as requesting a write lock.

- If you want to immediately change an object, then request a write lock from the beginning.
- If multiple applications are accessing the same group of objects, then you might also want to develop a protocol in which the different applications and/or transactions lock objects in the same order. The idea is to avoid a situation where one application asks for its locks starting with the beginning of a list of objects and another application asks for its locks starting with the end of the same list of objects (in which case, both applications would get some of the objects needed, but both would be blocked from getting all needed objects.)

Following are examples of typical usage of short locks.

Example using a write lock

Time 1 — Other users have a read lock on an instance.

Time 2 — You want to update the same instance, so you ask for a write lock on it. The request is blocked and your application waits until the request can be granted or the request times out.

Time 3 — Other users release their read locks by committing or rolling back their transactions.

Time 4 — If your request has not timed out, your application resumes, and you obtain a write lock on that instance.

Time 5 — You change the instance.

Time 6 — You commit the change and release the write lock.

Example using a read lock

Time 1 — Someone has a write lock on an instance of a class, and many others have read locks on other instances of the same class.

Time 2 — You want to create a report using the latest state of all objects at a particular moment in time. You request a read lock on the class object, but your request is blocked by the write lock on one instance of the class. Your application waits until the request can be granted or until the request times out.

If the request times out, if you are creating a report, you might want to consider asking for a null lock on the instances to gain immediate access, although you would not be guaranteed to be looking at the latest state of the objects. You may also get an inconsistent set of objects.

Time 3 — The user with a write lock on the instance executes a commit, which releases the write lock.

Time 4 — If your request has not timed out, your read lock on the class object is now granted and coexists with other read locks on instances of that class. This means that no other user can modify any instance of the class.

Time 5 — You can now print your report knowing that you are dealing with the latest state of all objects at that moment in time.

Example using an update lock

Time 1 — Numerous users are using a particular object, and they all have read locks on that object. But everyone has decided to follow recommended locking protocol for updates.

Time 2 — Someone decides to first read and then update an object and requests an update lock. The request is granted even though other users also have read locks.

Time 3 — You decide that you want to read and then update the same object, so you request an update lock. The request is blocked because an update lock already exists on that object. Your application waits.

Time 4 — The first user requests a write lock. This lock upgrade request is blocked by existing read locks by other users.

Time 5 — Other users with read locks finish and the user with the update lock gets a write lock.

Time 6 — The other user finishes, and the object becomes available. If your request has not timed out, your application resumes and acquires an update lock. You are now guaranteed to be the next person to get a write lock on that object.

Example that violates locking protocol

The following example illustrates lock interactions and lock precedences. In the example, "You" and "UserB" both try to use the same object. In the example, even though you follow the recommended protocol, your work is disrupted because "UserB" does not follow the recommended protocol.

Time 1

You have done nothing yet.

UserB requests a read lock.

UserB gets a read lock on the object.

Time 2

You request an update lock.

UserB still has a read lock.

You get the update lock.

Time 3

You have an update lock.

UserB requests a write lock in violation of protocol.

UserB waits because you have an update lock.

Time 4

You request an upgrade to a write lock per recommended protocol.

UserB continues to wait for a write lock.

You get a "would cause deadlock" error, because two users are now waiting for a write lock.

Time 5

Unless you handled the error and rolled back your transaction, your application continues to get a "would cause deadlock" error. Even though you followed the recommended protocol, UserB gets the write lock and continues.

The recommended protocol is for UserB to request an update lock at Time 1.

Short Locks and Queries

A query will return an array containing links to the returned objects. When you dereference an object in the array and bring it into memory, the default lock will be set on the object.

Depending on your interface language, there may be several forms of a select routine. Some will set the default lock on a class, and others will let you set an instance and/or class lock.

You can set an inheritance flag in a select statement.
If set to TRUE, the inheritance flag will cause the query to evaluate instances of both the specified class and also its subclasses. i.e.,If you do set the inheritance flag, your choice of short lock will be set on both the class object of the query class and on the class object of each subclass instance returned by the query.

If you want to lock all objects of the classes involved in the query, then specify a read or write lock in the query method. The effect will be the same as setting a read or write lock on all instances of the class and subclasses involved in the query. Although to improve performance, Versant actually only sets a read or write lock on the class objects for the instances to be returned.

If you do not want to lock objects returned by the query until you dereference them, then do not specify a short lock mode. In that case, Versant will set a special kind of read lock, called an "intention read lock", only on the class object. The effect is to prevent the class objects from being changed while you are looking at their instances. If you have specified either an intention lock or have specified a null lock in a query statement, then you must place instance locks separately, which will be done automatically when you dereference the links.

You do not need to know about intention locks unless you have special concurrency needs, because Versant automatically sets them on class objects whenever you set a lock on an instance object.

SHORT INTENTION LOCKS

This section explains intention locks. This is a topic you do not need to understand in order to use Versant.

Intention locks are relevant only to class objects.

The purpose of intention locks is to prevent changes to class objects while you are using an instance of the class. If your concurrency needs are unusual, then you may need to know about intention locks.

You can place a normal or intention lock on a class object. Intention locks have the same effect on a class object that a normal lock has on an instance object, except that intention locks do not block one another. For example, an intention read lock on a class object prevents it from changing, but an intention read lock does not block an intention write lock.

To maximize access to objects, an intention lock on a class object has no direct effect on instance objects. For example, an intention read lock on a class object does not block a write lock on an instance of that class. However, intention locks on class objects have an important indirect effect on instances, because to place a lock on an instance, a corresponding intention lock must also be set on the class object.

Intention locks are set implicitly on class objects when you request a lock on an instance. For example, if you set a read lock on an object, Versant will set an intention read lock on the class object.

If you use a query routine without specifying a lock mode argument, then the intention lock equivalent of the current default short lock is placed on the class objects for the instances returned.

You should not set intention locks on instances. Although setting an intention lock has the same effect as a normal lock, this is not good practice. However, an appropriate lock is automatically set while using the "get attribute" routine in C/Versant or dereference an object in C++/Versant.

Short Intention Lock Mode

Versant defines the following intention lock modes:

Intention read lock

An intention read lock on a class object prevents it from being changed.

An intention read lock on a class object is compatible with an intention write lock on the class, and a write lock on an instance of the class. An intention read lock is useful if you want to prevent others from changing a class object while you are using instances of the class.

An intention read lock is set implicitly by Versant on a class object when you request a read lock of an instance of the class.

A request for an intention read lock on a class object is blocked only if it has a write lock.

If you have an intention read lock on a class object, other users can still use instances of the class in a normal manner. They will also still be able to read the class object.

The terms "intention read lock" and "intention share lock" are synonymous.

```
c      IRLOCK
c++    IRLOCK
```

Intention write lock

An intention write lock on a class object prevents other users from reading or updating it but allows them to use instances of the class. An intention write lock is useful when you want to prevent the class object from being changed or read.

An intention write lock is set implicitly by Versant on a class object when you request a write lock on an instance of the class.

A request for an intention write lock on a class object is blocked if it has a read, update, or write lock. It will not be blocked by another intention read or intention write lock.

The terms "intention write lock" and "intention exclusive lock" are synonymous.

```
c      IWLOCK
c++    IWLOCK
```

Read with intention to write lock

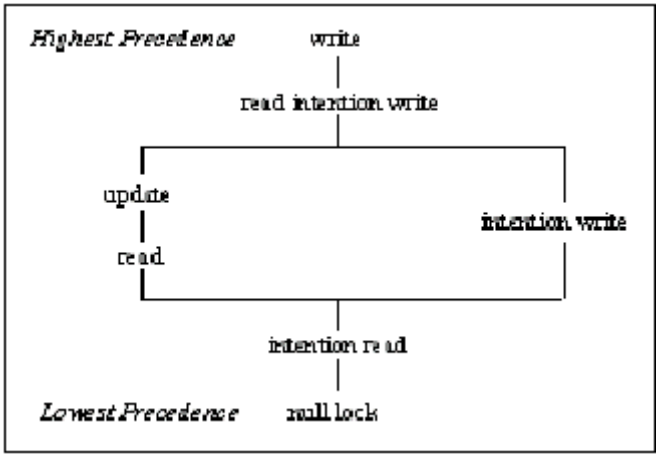
A read with intention to write lock prevents a class object from being changed and has the same effect as placing read locks on all instances of the class.

The terms "read with intention to write lock" and "share with intention exclusive lock" are synonymous.

```
c      RIWLOCK
c++    RIWLOCK
```

SHORT LOCK PRECEDENCE

The following shows lock precedence for both normal locks and intention locks. A write lock, at the top, has the highest precedence. A null lock has the lowest precedence.



Short Locks and the First Instance

Implicit Write lock on class object when inserting first instance

Normally, when you create or modify an instance of a class, Versant acquires an "Intention Write Lock" on the class object in order to prevent the class definition from being changed while you are creating or modifying an instance of that class. An Intention Write Lock allows multiple transactions to create or modify instances.

You may need to be aware of this behavior if you are installing a new production database or working in a development environment, because this behavior may mean that you may need to add application code to handle a deadlock or lock timeout. Alternately, whenever you define a class, you might want to insert a dummy instance and commit the change to initialize the class. The dummy instance can later be deleted, after other instances have been created.

This situation rarely occurs in production applications, because the case of an empty class is not common.

OPTIMISTIC LOCKING

The default locking mechanism in the Enterprise Objects Framework is optimistic locking.

Sometimes locking is too costly

Standard Versant read, update and write locks provide consistent and coherent access guaranteed in a multiple user environment. These guarantees are necessary and appropriate under most circumstances.

Sometimes however, you may want to read a large number of objects but update only a few of them. Or, you may be in a situation where there is only a small chance that the objects you want to work with will be updated by others. In this situation, holding a lock on all objects you access, can interfere with the work of others.

Accordingly, Versant provides optimistic locking features that allow you to work with objects without holding locks.

You do not have to use optimistic locking features to safely access and update unlocked objects, but the side effects of the alternative approaches might not be right for your situation.

Using Optimistic Locking Features

To use Versant optimistic locking features to work with unlocked objects, do the following.

- Add a time stamp attribute to class definitions.
- Start an optimistic locking session.
- Follow an optimistic locking protocol.
- Following an optimistic locking protocol will ensure the coherence and consistency normally provided by locks.
- Optionally use special methods.
- Optionally use performance related methods, which will allow fine-grain control over the objects you want to change.
- Optionally use event notification.
- Optionally use event notification methods, which will let you know when objects held with optimistic locks are modified.

By doing the above, you will get a shorter duration of locking, which will improve performance and improve concurrency of access among multiple users. At the same time, the time stamp validation at commit, delete, and group write time will still keep databases consistent.

Optimistic Locking Actions

Add time stamp attribute

To make a class "optimistic locking ready," add a time stamp attribute to the class definitions for the objects you will be accessing.

If an object has a time stamp attribute, the Versant commit, delete, group write, and check time stamp methods will automatically use the time stamp to detect obsolete copies in the object cache and prevent you from using the obsolete copies as the basis for changing the database object.

An obsolete copy will occur if, while you are working with an unlocked copy of an object, another user updates the original object in the database. If your object has a time stamp attribute, you will automatically be told if your commit, delete, and group write method will overwrite changes made by another user since you read your object. An obsolete copy will also occur if you try to modify an object that another user has already deleted.

Time stamps are a convenience feature. You could avoid having obsolete copies by first refreshing every unlocked object you wanted to change before proceeding to a commit. However, this would be costly, both because of the necessary fetching of objects and because of the time required to make detailed evaluations of each attribute in each object.

Once you add a time stamp attribute to an object, the time stamp attribute will be updated, no matter who does the updating and no matter the kind of session used. When an object with a time stamp attribute is created, its time stamp attribute is set to `0`. Thereafter, each update written to a database will increment the attribute by `1` during the normal commit and group write process.

You can rely on normal commit, group write, and delete methods to tell you if you have an obsolete copy, or you can detect obsolete copies in your object cache by using the "check time stamp" method before your commit.

When you detect an obsolete copy by checking time stamps, what you do next is up to you. For example, you might want to refresh your copy of the object, inspect it, and then reapply some or all of your modifications. Or you may simply want to abandon your update.

The procedure to add a time stamp attribute to a class is tightly bound to your interface language.

C/Versant

To define a time stamp attribute in a C class, add the following lines to your attribute descriptions:

```
static o_attrdesc clsAttrs[] =
{
   {TIMESTAMP_ATTR_NAME_STR,{"",""},
   TIMESTAMP_ATTR_TYPE_STR,
   O_SINGLE, NULL, 0},
   .
   .
};
```

Then, before calling `o_defineclass()`, add the following lines to your class definition `struct`:

```
typedef struct
{
   O_TS_TIMESTAMP,
   .
   .
} clsInstance;
```

C++/Versant

To define a time stamp attribute in a C++ class, add the following lines to your class definitions:

```
class SampleClass : public PObject
{
   public:
      O_TS_TIMESTAMP;
   .
   .
};
```

Begin optimistic locking session

In other kinds of sessions, as soon as you mark an object dirty, Versant will attempt to acquire a write lock on that object. In an optimistic locking session, all implicit lock upgrades are suppressed. Suppression of implicit lock upgrades allows you to continue working with your objects, as long as you like, without setting and holding a lock, which is the point of using optimistic locking. Of course, a write lock will still automatically be set on dirty objects when you commit.

An optimistic locking session will also suppress object swapping and provide automatic detection of obsolete copies.

```
c     o_beginsession() with O_OPT_LK
c++   beginsession() with O_OPT_LK
```

Check time stamps

Check time stamps to detect any obsolete copies in the object cache.

You must be careful to check both the objects marked dirty and the objects inter-related to them. When you consider inter-related objects, you should consider both hierarchical relationships (such as subclass objects) and logical relationships. An example of a logical relationship is when, say, you want to change B only if A is greater than 10. In this case, you should check both B (the changed object) and A (the logically related object) in order to prevent changing B if the value of A has changed since you read it.

```
c     o_checktimestamps()
c++   checktimestamps()
```

Downgrade short lock

Release locks on specified objects without flushing objects from the object cache.

```
c     o_downgradelocks()
c++   downgradelocks()
```

Delete objects

Delete objects in a `vstr` or collection. This method enhances performance by deleting a group of objects with the least possible number of network messages.

```
c     o_gdeleteobjs()
c++   gdeleteobjs()
```

Checkpoint commit transaction with group

Perform a checkpoint commit on objects in a vstr or collection and maintain objects in the object cache.

```
c     o_xactwithvstr()
c++   xactwithvstr()
```

Commit and retain cache

Rollback and retain cache

Most commit and rollback routines allow you optionally to prevent flushing of the object cache. (Exceptions are the C++/Versant `commit()` and `rollback()` methods.)

However, in an optimistic locking session, a better way to handle the object cache is to use commit and rollback routines that operate only on objects in a vstr or collection. However, if you use a group commit or rollback, you must be careful to place in the `vstr` or collection all objects marked dirty and all objects inter-related to them.

When you consider inter-related objects, you should consider both hierarchical relationships (such as subclass objects) and logical relationships. An example of a logical relationship is when, say, you want to change B only if A is greater than 10.

Commit and retain cache

```
c     o_xact()
c++   xact()
```

Commit a group of objects

```
c     o_xactwithvstr()
c++   xactwithvstr()
```

Rollback and retain cache

```
c     o_xact()
c++   xact()
```

Rollback a group of objects

```
c     o_xactwithvstr()
c++   xactwithvstr()
```

C/Versant and C++/Versant

An option in `o_xact()` and `xact()` allow you to prevent flushing the entire object cache after a commit. This might be useful, but normally you would use the commit with `vstr` routine in an optimistic locking session.

Drop read locks automatically

When you fetch an object, many routines will place a read lock on the object. If you are using optimistic locking, you will want to drop the read lock on the fetched object as soon as possible.

One way to drop a read lock is to track the objects fetched and then explicitly drop the lock with a routine such as the C/Versant `o_downgradelock()` function.

If you know that you always want to downgrade read locks on fetched objects, you can specify automatic lock downgrading. This approach does not require you to keep track of fetched objects and reduces network calls.

To automatically downgrade read locks, use a "set thread options" routine with the `O_DROP_RLOCK` option. Following is a description of the `O_DROP_RLOCK` option.

O_DROP_RLOCK

For this thread, drop read locks when objects are retrieved from a database.

This option will have no impact on intention read locks set on the class objects corresponding to the instance objects retrieved.

Read locks are held on instance objects during the duration of the read operation (which protects you from seeing uncommitted objects) and then dropped when the read operation is finished.

If you want to always drop read locks after fetching objects, this option reduces network calls, because it causes locks to be dropped by the database server process. By comparison, dropping locks with a down grade lock routine requires a separate network message from the application process to the server process.

This option may be used in a standard session or an optimistic locking session.

Thread option routines

C/Versant thread option functions:

`o_setthreadoptions()` — Set thread options.

`o_unsetthreadoptions()` — Restore thread options to their defaults.

`o_getthreadoptions()` — Get thread options.

C/Versant functions that are affected by `O_DROP_RLOCK`

The following routines, when invoked by a thread for which `O_DROP_RLOCK` has been set, will drop read locks on fetched objects.

`o_locateobj()` — Get object. A read lock will be dropped if the `lockmode` parameter is `RLOCK` or `IRLOCK`.

`o_pathselectcursor()`— Find object with cursor. Read locks will be dropped if the `instlock` parameter is specified as `RLOCK` or `IRLOCK` and `O_CURSOR_RELEASE_LOCK` is not specified in the `options` parameter. If you toggle `O_DROP_RLOCK` in the middle of a cursor select, the new option will take effect at the next retrieval of objects.

`o_fetchcursor()` — Get object with cursor created by `o_pathselectcursor()`. Read locks will be dropped if the instlock parameter in o_pathselectcursor() is specified as `RLOCK` or `IRLOCK` and `O_CURSOR_RELEASE_LOCK` is not specified in the options parameter. If you toggle `O_DROP_RLOCK` in the middle of a cursor fetch, the new option will take effect at the next retrieval of objects.

`o_greadobjs()` — Get objects. Read locks are dropped if the lockmode parameter if `RLOCK` or `IRLOCK`. This behavior is the same as setting the options parameter to be `O_DOWN_GRADE_LOCKS` parameter, except that only read and intention read locks are dropped.

`o_getclosure()` — Get object closure. Read locks are dropped if the lockmode parameter is `RLOCK` or `IRLOCK`.

`o_refreshobj()` — Refresh object. A read lock is dropped if the lockmode parameter is `RLOCK` or `IRLOCK`.

o_refreshobjs() — Refresh objects. Read locks are dropped if the lockmode parameter is RLOCK or IRLOCK.

o_getattr() — Get attribute value. Read locks are dropped if the default lock mode is RLOCK or IRLOCK.

o_getattroffset() — Get attribute offset. Read locks are dropped if the default lock mode is RLOCK or IRLOCK.

C++/Versant thread option methods

setthreadoptions() — Set thread options. The default is to not drop read locks when objects are fetched. Different threads in the same session may have different settings for thread options.

unsetthreadoptions() — Restore thread options to their defaults.

getthreadoptions() — Get thread options.

C++/Versant methods that are affected by O_DROP_RLOCK

The following routines, when invoked by a thread for which O_DROP_RLOCK has been set, will drop read locks on fetched objects.

locateobj() — Get object. A read lock will be dropped if the lockmode parameter is RLOCK or IRLOCK.

select() — Find object with cursor. Read locks will be dropped if the instlock parameter is specified as RLOCK or IRLOCK and O_CURSOR_RELEASE_LOCK is not specified in the options parameter. If you toggle O_DROP_RLOCK in the middle of a cursor select, the new option will take effect at the next retrieval of objects.

fetch() — Get object with cursor. Read locks will be dropped if the instlock parameter VCursor() is specified as RLOCK or IRLOCK and O_CURSOR_RELEASE_LOCK is not specified in the options parameter. If you toggle O_DROP_RLOCK in the middle of a cursor fetch, the new option will take effect at the next retrieval of objects.

greadobjs() — Get objects. Read locks are dropped if the lockmode parameter if RLOCK or IRLOCK. This behavior is the same as setting the options parameter to be O_DOWN_GRADE_LOCKS parameter, except that only read and intention read locks are dropped.

getclosure() — Get object closure. Read locks are dropped if the lockmode parameter is RLOCK or IRLOCK.

refreshobj() — Refresh object. A read lock is dropped if the lockmode parameter is RLOCK or IRLOCK.

`refreshobjs()` — Refresh objects. Read locks are dropped if the lockmode parameter is `RLOCK` or `IRLOCK`.

`get_attribute_value()` — Get attribute value. Read locks are dropped if the default lock mode is `RLOCK` or `IRLOCK`.

`getattroffset()` — Get attribute offset. Read locks are dropped if the default lock mode is `RLOCK` or `IRLOCK`.

Multiple read inconsistencies

If you perform multiple reads in separate operations and have set the `O_DROP_RLOCK` option `ON`, your objects may not be consistent, because operations may have been performed on your objects between your reads.

If you are going to use the `O_DROP_RLOCK` option, you must be able to tolerate multiple read inconsistencies.

Example

The danger of using implicit read lock downgrade mechanisms.

	Object cache (what you see)	Server cache (what database sees)
Time 1		Object A: state 1, no lock Object B: state 1, no lock
Time 2	You read Object A with a read lock.	Object A: state 1, read lock Object B: state 1, no lock
Time 3	You see Object A with state 1. Because automatic downgrading has been specified, your read lock is downgraded.	Object A: state 1, read lock Object B: state 1, no lock
Time 4	Another session modifies Object A and Object B and commits the changes.	Object A: state 2, write lock Object B: state 2, write lock
Time 5	You see Object A with state 1, but to the database it has state 2.	Object A: state 2, no lock Object B: state 2, no lock

Time 6	You read Object B with a read lock.	Object A: state 2, no lock Object B: state 2, read lock
Time 7	You see Object A with state 1 and Object B with state 2.	Object A: state 2, no lock Object B: state 2, no lock

What you see at Time 7 is inconsistent with both the initial and the current database states, because the objects were retrieved at different times and their read locks were dropped at different times.

OPTIMISTIC LOCKING PROTOCOL

In other kinds of sessions, locks ensure that your objects and databases are coherent and consistent, although you must set the right locks at the right time.

In an optimistic locking session, you must take responsibility for coherence and consistence by following an optimistic locking protocol.

Optimistic locking protocol assumes:

- The objects involved have time stamp attributes.
- You are working within an optimistic locking session.

When you use optimistic locking, you should observe the following protocol.

1. Start an optimistic locking database session.
2. Read objects with locks and pins.

   Fetch your objects from their databases with read locks and pins.

   The group of objects you bring into your object cache is called the "optimistic transaction set." Unless you change the default lock or override the default lock in a group read method, copying an object from a database will normally set a read lock and pin the object in your object cache.

   Read locks are necessary to ensure consistency among the objects in your optimistic transaction set. For example, if you did not fetch your objects with a read or stronger lock, a subclass object might be updated while you were fetching a superclass object, which would compromise consistency.

   Pins are necessary to avoid automatic flushing. Normally, if you run out of virtual memory for your cache, Versant will swap unpinned objects back to their databases at its own discretion. This could cause problems, because the next time you touched a swapped object, you would get a new copy which might not be consistent with the rest of your original optimistic transaction set.

3. Downgrade locks.

Once you have a consistent group of objects, use a "downgrade locks" routine to drop locks without changing the pin status. Alternately, for a thread, you can set automatic lock downgrading ON.

4. Browse and update.

   Using normal procedures, browse and update the optimistic transaction set.

   Since you are in an optimistic locking session, when you mark an object dirty, Versant will not set a write lock on the object.

   During this time, if you touch an object not fetched during the read phase, a read lock may be set by default. In C++, this can happen, for example, by dereferencing an object outside the optimistic transaction set. During this stage, you can, if you want, set the default lock to a null lock, but it is better to avoid fetching additional objects to prevent introducing an object not consistent with the original set of objects.

   Per normal behavior, schema evolution, and versioning methods will continue to implicitly set write locks on class objects. If you call these methods in an optimistic locking session, these implicit locks will be held until your enclosing transaction ends.

   For performance reasons, you will probably want to place all objects that you change into a vstr or collection. This will allow you to perform a commit, group write, or group delete only on the desired group of objects. If you place only the objects that you change into a vstr or collection (rather than all objects in the optimistic transaction set), be sure that you also place in the group all objects with inter-dependencies with the changed objects. Keeping inter-dependent objects together is important. For example, if you change an object, you will want to make sure that the time stamps for its associated objects are also checked, either at the time of a group commit or in an explicit time stamp checking step.

5. Check time stamps.

   Before proceeding to a commit, you should use the "check time stamp" routine to determine which of your changed objects have become obsolete.

   The check time stamp method will set the default lock on all objects checked, so if you do find objects that are obsolete, you may want to refresh your copy with the "refresh objects" routine.

6. Commit or rollback changes.

   At any time, you can save or roll back your updates with a transaction commit or rollback. You can either choose a method that will keep your objects in your cache, so that you can continue working with them, or choose a method that will flush objects to their databases.

At commit time, Versant does the following:

a. In the application process, Versant increments the time stamp (if it exists) of each dirty object by one.

b. Versant sends dirty objects from the object cache to their source databases in batches.

c. On each dirty object, Versant sets a write lock on its source object in its source database.

d. For each dirty object, Versant compares the time stamp in the dirty object with the time stamp in the source object.

e. If the time stamp of the dirty object is greater than the time stamp of the source object by one, Versant writes the dirty object to the database.

   If the time stamp of the dirty object is not greater than the time stamp of the source object by one, Versant does not write the dirty object to the database.

Steps a, d, and the check made in ‘e’ constitutes a "time stamp validation sequence." If all objects pass the time stamp validation, then the commit succeeds. If one or more objects fail the time stamp validation, then the commit fails.

At commit time, Versant will normally scan all objects in your object cache looking for those marked dirty and will flush your object cache at the end of the transaction. To improve performance, you can optionally use a method that will commit or rollback changes only to objects in a vstr or collection. This allows you to limit the scope of the objects to be locked during the commit and whose time stamps are to be compared.

If your commit fails

If your commit fails, perhaps because some objects have outdated time stamps or have been deleted by another user, you can either rollback or refresh the failed objects and try again.

Rolling back the transaction will restore the objects to their values before they were modified. Refreshing the failed objects will load the latest committed objects from the server.

If you refresh the objects, you should use a read lock. The refreshed objects will be different from the timestamp failed objects or they will have been deleted (in which case, you will see an "object not found" error.)

For example, suppose that you have two applications, `Application_1` and `Application_2`, who are both working with the same two objects, `ObjectA` and `ObjectB`. Now, suppose the following sequence of events occurs.

Action	Application_1	Application_2
Initial conditions	ObjectA	ObjectA
	Timestamp = 0	timestamp = 0
	Value = X	value = X
	ObjectB	ObjectB
	Timestamp = 0	timestamp = 0
	Value = Y	value = Y
Both modify some objects	ObjectA	ObjectA
	Timestamp = 0	timestamp = 0
	Value = X_1	value = X_2
	ObjectB	ObjectB
	Timestamp = 0	timestamp = 0
	Value = Y_1	value = Y
Application_2 commits and reads its results back	ObjectA	ObjectA
	Timestamp = 0	timestamp = 1
	Value = X_1	value = X_2
	ObjectB	ObjectB
	Timestamp = 0	timestamp = 0
	Value = Y_1	value = Y
Application_1 tries to commit but fails due to timestamp of ObjectA	ObjectA	ObjectA
	Timestamp = 0	timestamp = 1
	Value = X_1	value = X_2
	ObjectB	ObjectB
	Timestamp = 0	timestamp = 0
	Value = Y_1	value = Y

At this point, if Application_1 does a complete rollback, all objects would be flushed from the object cache.

Alternately, Application_1 could perform a "rollback and retain." For C and C++, this means using the option O_ROLLBACK_AND_RETAIN with a transaction routine such as o_xact() or xact(). In this case, Application_1 will see the following:

```
ObjectA
  timestamp = 0
  value     = X_2
ObjectB
  timestamp = 0
  value     = Y
```

If Application_1 refreshes each object, then Application_1 will see the following:

```
ObjectA
  timestamp = 1
  value     = X_2
ObjectB
  timestamp = 1
  value     = Y_1
```

In the above, note that the timestamps for dirty objects are incremented even though they fail timestamp validation. This is intended behavior and will only be experienced by the application whose commit fails because of timestamp failures.

USAGE NOTES

Compatibility with strict locking schemes

If you start any kind of session other than an optimistic locking or custom locking session, you will use "strict locking." Strict locking sessions use standard Versant locks to ensure database compatibility in a multi-user environment. All types of sessions use a two-phase commit protocol to ensure consistency among multiple databases.

Optimistic locking sessions are compatible with strict locking sessions run by other users. Once you have defined a time stamp attribute, it will be updated by commits and group writes performed in any kind of session.

Intra-session compatibility

Schema evolution and versioning methods implicitly set write locks on Versant system objects, regardless of the type of session. This means that if you use a schema evolution or versioning method in an optimistic locking session, you will really be running with a mixture of optimistic and strict locking.

Compatibility with savepoints

Savepoints are not compatible with optimistic locking, because setting a savepoint flushes objects to their databases, which resets locks.

Compatibility with C++/Versant links

The C++/Versant link and link vstr classes automatically obtain locks on database objects when you first touch an object within a transaction. This may cause confusion when using optimistic locking.

Delete behavior

The delete behavior can be controlled using the server parameter `commit_delete.` This parameter decides whether the object should be deleted immediately or at commit.

When this parameter is set to `ON`, the physical deletion of the object is delayed until commit time. This will ensure database consistency in case of rolling back objects with unique indices and maintain clustering if applicable.

When a delete operation is performed on any object, the operation is sent to the database immediately. If `commit_delete` is enabled, the objects are "`Marked for deletion`" in the

database and hold a Write lock. The objects are physically deleted at commit. In case of a rollback, they are unmarked and the original status is restored. If `commit_delete` is disabled the physical deletion of the object is done at the time when the delete operation is performed.

If an application has cached a set of objects, which are "`Marked for deletion`" by some other application, trying to access these objects will result in an error. The error will depend on the state of the object. If the object is physically deleted, that is the transaction has committed, the error "`object not found`" is returned. If the object is "`Marked for deletion`", it would have acquired a Write lock on the object. Trying to access such an object will return a "`lock wait timeout`" error. New queries will not include these objects in the result sets.

If `commit_delete` is set, the following difference in behavior will be seen.

Consider 2 processes working on the same set of objects.

Scenario 1: `commit_delete` set to `ON`

Action	Application_1	Application_2
Read objects	Object A Timestamp = 0 Value = X Object B Timestamp = 0 Value = Y	Object A Timestamp = 0 Value = X Object B Timestamp = 0 Value = Y
Application_2 modifies some objects	Object A Timestamp = 0 Value = X Object B Timestamp = 0 Value = Y	ObjectA Timestamp = 0 Value = X_2 ObjectB Timestamp = 0 Value = Y
Application_1 deletes the objects	Object A [Marked for deletion] Object B [Marked for deletion]	ObjectA Timestamp = 0 Value = X_2 ObjectB Timestamp = 0 Value = Y

Application_2 commits	Object A [Marked for deletion] Object B [Marked for deletion]	Receives error `SM_LOCK_TIMEDOUT`
Application_1 commits	Object A and Object B physically deleted	
Application_2 rolls back		

If `commit_delete` was set to `OFF`, then Application_2 would get the error `OB_NO_SUCH_OBJECT` since the objects would be immediately deleted.

Scenario 2: `commit_delete` set to `ON`

Action	**Application_1**	**Application_2**
Read objects	Object A Timestamp = 0 Value = X Object B Timestamp = 0 Value = Y	Object A Timestamp = 0 Value = X Object B Timestamp = 0 Value = Y
Application_2 deletes some objects	Object A Timestamp = 0 Value = X Object B Timestamp = 0 Value = Y	Object A [Marked for deletion] Object B [Marked for deletion]
Application_1 checks the timestamp for objects	Receives error `SM_LOCK_TIMEDOUT`	Object A [Marked for deletion] Object B [Marked for deletion]
Application_2 commits		Object A and Object B physically deleted

Application_1 checks the timestamp for objects	Receives error OB_NO_SUCH_OBJECT
Application_1 rolls back	

If commit_delete was set to OFF, Application_1 would have received error OB_NO_SUCH_OBJECT in both the cases.

Query behavior

If you use a query, remember that a query first flushes all dirty objects and marks them as clean before selecting objects. This means that if you update an object, perform a query, and then update the object again, you must mark the object dirty a second time to ensure that the second update is written to the database.

EXAMPLES

C++/Versant Example

The following C++/Versant example assumes that the Employee class has a time stamp attribute:

```
#include <cxxcls/vdate.h>
#include <stream.h>
#include "person.h"
#include "departme.h"
class Employee : public Person
{
protected:
   // attributes
    Link<Department>    department;
    o_u4b               number;
    o_float             salary;
    PString             job_description;
    VDate               last_raise;
public:
   O_TS_TIMESTAMP;
   // constructor and destructor
    Employee(
       const PString&   name,
       Department*      department,
       o_u4b            number,
       o_float          salary,
       const PString&   job_description);
   virtual ~Employee();
   // accessors
    virtual Department& get_department() const;
    virtual o_u4b       get_number() const;
    virtual o_float     get_salary();
    virtual PString     get_job_description() const;
   // modifiers
    virtual void
    set_department(Department* department);
    virtual void        set_number(o_u4b number);
    virtual void        set_salary(o_float salary);
    virtual void        set_job_description(
```

```
                          const PString& job_description);
   // inherits key methods from Person
};
```

Following is an optimistic locking example program named `opt_lock.cxx`.

```
 /*
 *  opt_lock.cxx
 */
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <cxxcls/try.h>
#include <cxxcls/vquery.h>
#include "employee.h"
#include "departme.h"

#include "dms.data"

void usage(char** argv)
{
   cout  << "usage is : " << argv[0] <<" <groupDB> <opt_lock[23]> "
   << endl;
   exit(1);
}
main
(
    int          argc,
    char         **argv
)
{
    o_bool       callDestructor;
    char         cmd[50];
    char         concurrentProgram[20];
    o_dbname     groupDB;
    int          i = 0;
    LinkVstrAny  timestampVstr;
    LinkVstrAny  deletedVstr;
    LinkVstrAny  lBackVstr;
    LinkVstrAny  lVstr;
    o_opts       optionArray;
```

```
     o_bool       rolledBack;
     o_bool       toPin;
    if (argc > 1)
    strcpy(groupDB, argv[1]);
    if (argc > 2)
    strcpy(concurrentProgram, argv[2]);
    cout << "\n\n\t\tWelcome to the Optimistic Lock Example Program!\n"
    << endl;
// Initialize DOM Interface and connect with group database
// Start an optimistic lock session so that the Versant
// front-end will pass to the Versant back-end
// the time stamp values of the  objects being written or
// deleted for a time-stamp verification.
    cout << "\n\t Starting an Optimistic Lock Session ...";
    optionArray[0] = O_OPT_LK;
    VSession *session = new VSession(NULL, groupDB, "first_session",
     optionArray);
    cout << " ...done\n" << endl;
    // Create instances
    Department* rd = O_NEW_PERSISTENT(Department)(DEPARTMENT1);
    // note the placement of the constructor arguments

    Employee* john = O_NEW_PERSISTENT(Employee)
                     (NAME1,rd,100022,35000.00,JOB1);
    Employee* lisa = O_NEW_PERSISTENT(Employee)
                     (NAME2,rd,100023,45000.00,JOB2);
    Employee* james = O_NEW_PERSISTENT(Employee)
                     (NAME3,rd,100024,61500.00,JOB3);

     cout << "\n\t Created instances for Department and Employee
     classes.\n"
     << endl;

    // Commit the Transaction
    session->commit();
    //  Find all employees in group database:
    LinkVstr<Employee> staff = PClassObj<Employee>.select
    (groupDB,TRUE,PPredicate());

    cout  << "\n\t Selected all Employees instances.\n" << endl;
    toPin = TRUE;
```

```
session->greadobjs( staff, groupDB, RLOCK, toPin );
cout  << "\n\t All employees objects are read.\n" << endl;
session->downgradelocks( staff, NOLOCK, &lBackVstr );
cout  << "\n\t Locks are downgraded.\n" << endl;
cout  << "\n\t Output from " << concurrentProgram << endl;
cout  << "\n\t ******************** " << endl;
sprintf(cmd, "%s %s", concurrentProgram, groupDB);
system(cmd);
cout << "\n\t ******************** " << endl;
//
// Take the first object selected. Conversion from
// Employee_link to Employee* happens.
//
Employee* worker = staff[0];
// Update an Employee object:
worker->set_salary(50000.00);
cout << "\n\t Updated an Employee instance. \n" << endl;
for( i = 1; i < staff.size(); i++ )
{
   if (staff[i] != NULL_LINK)
      lVstr.add(staff[i]);
}
callDestructor = FALSE;
try
{
session->gdeleteobjs( lVstr, groupDB, &timestampVstr,
&deletedVstr,callDestructor );
}
catch( PError& error )
{
cout  << "\n\t gdeleteobjs() comes across";
switch( error.errnum )
{
   case  OB_NO_SUCH_OBJECT:
   cout  << "\n\t\t<Error 5006>, OB_NO_SUCH_OBJECT." << endl;
   break;
   case  SM_E_KEY_NOT_FOUND:
   cout  << "\n\t\t<Error 1009>, SM_E_KEY_NOT_FOUND." << endl;
   break;
   case  OM_TR_INVALID_TS:
```

```
   cout  << "\n\t\t<Error 4036>, OM_TR_INVALID_TS:" << endl;
   cout  << "\n\t\tSome objects have been updated in the
            database." << endl;
   break;
   case  SM_LOCK_TIMEDOUT:
   cout  << "\n\t\t<Error 2903>, SM_LOCK_TIMEDOUT: ";
   cout  << "\n\t\tLock wait timed out." << endl;
   break;
   default:
   cout  << "\n\t\t<Error " << error.errnum << ">" << endl;
   break;
}
for( i = 0; i < (int)timestampVstr.size(); i++ )
{
   if (timestampVstr[i])
   {
   cout  << "\n\t\tObject " \
         << ((Link<Employee>)(timestampVstr[i]))->get_number();
   cout  << " was updated and committed before." << endl;
   cout  << "\n\t\tRefresh this object." << endl;
   }
}
for( i = 0; i < (int)deletedVstr.size(); i++ )
{
   if (deletedVstr[i])
   {
   cout  << "\n\t\tObject " \
         << ((Link<Employee>)(deletedVstr[i]))->get_number();
   cout  << " was deleted already." << endl;
   cout  << "\n\t\tRollback this object." << endl;
   }
}
}
end_try;  // for cfront only !!
// Release vstr memory
lBackVstr.release();
lVstr.release();
if (deletedVstr.size() > 0)
{
cout  << "\n\t Roll back the changes on objects which have ";
cout  << "already been deleted." << endl;
```

```
   cout  << "\n\t Roll back with the O_ROLLBACK_AND_RETAIN and ";
   cout  << "O_RELEASE_VSTR options " << endl;
0, session->xactwithvstr( (O_ROLLBACK_AND_RETAIN | O_RELEASE_VSTR),
   deletedVstr, &timestampVstr, &lBackVstr );
   cout  << "\n\t Rolled back " << deletedVstr.size();
   if (deletedVstr.size() > 1)
      cout << " objects." << endl;
   else
      cout << " object." << endl;
   }
   lBackVstr.release();
   if (timestampVstr.size() > 0)
   {
      VstrAny movedVstr;
   cout  << "\n\t Refresh the objects which have already
            been updated." << endl;
  session->refreshobjs( timestampVstr, groupDB, RLOCK, &movedVstr );
   cout  << "\n\t Refreshed " << timestampVstr.size();
   cout  << " Employee instances. Their current salary are:\n"<< endl;
   for (i = 0; i < (int)timestampVstr.size(); i++)
   {
      cout  << "\n\t "
            << ((Link<Employee>)(timestampVstr[i]))->get_number();
      cout  << "\t"
            << ((Link<Employee>)(timestampVstr[i]))->get_salary();
      cout  << endl;
   }
   // Delete the Employee object again:
   // Since refreshobjs() sets RLOCK on the objects, gdeleteobjs()
   // should succeed.
   session->gdeleteobjs( timestampVstr, groupDB, &lBackVstr,
   &deletedVstr, callDestructor );
   cout  << "\n\t Deleted the Employee instances again. \n" << endl;
   }
   // Release vstr memory
   lBackVstr.release();
   timestampVstr.release();
   deletedVstr.release();
   lVstr.add(staff[0]);
   // check the time stamp value
```

```
try
{
o_opts   options;
options[0] = O_INPUT_VSTR;
session->checktimestamps( options, lVstr, &timestampVstr,
&deletedVstr );
}
catch( PError& error )
{
cout  << "\n\t checktimestamps() comes across";
switch( error.errnum )
{
   case  OB_NO_SUCH_OBJECT:
   cout  << "\n\t\t<Error 5006>, OB_NO_SUCH_OBJECT:" << endl;
   break;
   case  SM_E_KEY_NOT_FOUND:
   cout  << "\n\t\t<Error 1009>, SM_E_KEY_NOT_FOUND:" << endl;
   break;
   case  OM_TR_INVALID_TS:
   cout  << "\n\t\t<Error 4036>, OM_TR_INVALID_TS:" << endl;
   cout  << "\n\t\tObject(s) has(ve) been updated in the database."
  << endl;
   cout  << endl;
   break;
   case SM_LOCK_TIMEDOUT:
   cout << "\n\t\t<Error 2903>, SM_LOCK_TIMEDOUT: ";
   cout << "\n\t\tLock wait timed out." << endl;
   break;
   default:
   rethrow;
}
for( i = 0; i < (int)timestampVstr.size(); i++ )
{
   if (timestampVstr[i])
   {
      cout  << "\n\t\tObject " \
            << ((Link<Employee>)(timestampVstr[i]))->get_number();
      cout  << " was updated and committed before." << endl;
      cout  << "\n\t\tRefresh this object." << endl;
   }
}
```

```
for( i = 0; i < (int)deletedVstr.size(); i++ )
{
   if (deletedVstr[i])
   {
      cout  << "\n\t\tObject " \
            << ((Link<Employee>)(deletedVstr[i]))->get_number();
      cout  << " was deleted already." << endl;
      cout  << "\n\t\tRollback this object." << endl;
   }
}
}
end_try;  // for cfront only !!
rolledBack = FALSE;
if (deletedVstr.size() > 0)
{
   cout  << "\n\t Roll back the changes on objects which have ";
   cout  << "already been deleted." << endl;
   cout  << "\n\t Roll back with the O_ROLLBACK_AND_RETAIN and ";
   cout  << "O_RELEASE_VSTR options." << endl;
   session->xactwithvstr( (O_ROLLBACK_AND_RETAIN |
   O_RELEASE_VSTR), 0,
         deletedVstr, &timestampVstr, &lBackVstr );
   cout  << "\n\t Rolled back " << deletedVstr.size();
   if (deletedVstr.size() > 1)
         cout  << " objects." << endl;
   else
         cout  << " object." << endl;
         rolledBack = TRUE;
}
lBackVstr.release();
if (timestampVstr.size() > 0)
{
   o_bool   moved;
   moved = FALSE;
   cout  << "\n\t Refresh the objects which have ";
   cout  << "already been updated." << endl;
   session->refreshobj( timestampVstr[0], RLOCK, &moved );
   cout  << "\n\t Refreshed an Employee instance." << endl;
   cout  << "\n\t Its current salary is ";
   cout  << ((Link<Employee>)(timestampVstr[0]))->get_salary()
```

```
         << endl;
   // Update the Employee object again:
   worker->set_salary(50000.00);
   cout << "\n\t Updated the Employee instance again \n" << endl;
}
// Release vstr memory
lBackVstr.release();
timestampVstr.release();
deletedVstr.release();
if (!rolledBack)
{
// If the above update has to be rolled
// back because the object has already been deleted,
// then do not commit the transaction.
// Otherwise, commit the transaction.
try
{
   cout  << "\n\t O_COMMIT_AND_RETAIN and O_INPUT_VSTR commit.\n"
         << endl;
   session->xactwithvstr( (O_COMMIT_AND_RETAIN | O_INPUT_VSTR),
   NULL, lVstr, &timestampVstr, &deletedVstr );
}
catch( PError& error )
{
   cout << "\n\t xactwithvstr() comes across ";
   switch( error.errnum )
   {
   case SM_LOCK_TIMEDOUT:
      cout << "<Error 2903>, SM_LOCK_TIMEDOUT: ";
      cout << "  Object is locked" << endl;
      break;
   case OB_NO_SUCH_OBJECT:
   {
   cout << "<Error 5006>, OB_NO_SUCH_OBJECT:" << endl;
      break;
   }
   case OM_TR_INVALID_TS:
   {
   cout  << "<Error 4036>, OM_TR_INVALID_TS:" << endl;
   cout  << "Some objects have been updated in the database."
            << endl;
```

```
            break;
         }
         default:
            rethrow;
         }
         for( i = 0; i < (int)timestampVstr.size(); i++ )
         {
         if (timestampVstr[i])
         {
            cout << "\n\t\tObject " \
                 << ((Link<Employee>)(timestampVstr[i]))->get_number();
         cout << " was updated and committed before." << endl;
         }
         }
         for( i = 0; i < (int)deletedVstr.size(); i++ )
         {
         if (deletedVstr[i])
         {
            cout << "\n\t\tObject " \
                 << ((Link<Employee>)(deletedVstr[i]))->get_number();
            cout << " was deleted and committed before." << endl;
         }
         }
      }
      end_try;  // for cfront only !!
      cout << "\n\t Committed the transaction.\n" << endl;
      }
      // Release vstr memory
      lBackVstr.release();
      lVstr.release();
      staff.release();
      timestampVstr.release();
      deletedVstr.release();

      cout << "\n\t Ended the session. \n" << endl;
      cout << "\n\t End of the Example! \n" << endl;
      delete session;
   } // end main
```

CHAPTER 6

Schema Evolution

This Chapter explains Versant Schema Evolution.

Following topics are covered:

- Schema Evolution
- Classes
- Attributes
- Propagating Schema Changes
- Verifying Schema

SCHEMA EVOLUTION

Overview

Schema Evolution is the process wherein existing schema in the database is changed to reflect the changes in user classes.

For each class of object that is stored in a Versant database, the class definition is also stored. This stored definition (schema) must be identical to the class definition in your program files to avoid unpredictable results. In addition, when you change a class definition, existing instances of the changed class must adapt to the new definition. This section tells how to update stored class and instance objects with evolving class definitions.

Most kinds of schema changes do not require explicit conversion of instances. Each time your applications access an object, Versant automatically adjusts the object's structure to the current class definition as needed. This "lazy updating" feature spreads the cost of instance conversions over time, and avoids the need to shut out applications while global conversions are performed. The exceptional schema changes that require global conversions are noted below.

Lazy updating does not alter object identifiers, so references to affected objects remain valid after the conversion. For this reason, lazy updating is preferable to manual conversion involving the creation of new objects.

Multiple schema versions in the database can impact your query performance, as the same query will be repeated for different schema versions. In this case you might want to evolve all instances of the schema to the latest schema version. VOD provides a tool that allows you to do a server-side explicit instance conversion without writing any special applications to do so. For more information about using this tool, please refer to the chapter “Database Utilities” in the *Versant Database Administration Manual.*

These are the two schema utilities which are used for schema evolution -

`sch2db`	To load the class definitions in the schema definition file in the database.
`schcomp`	To create a schema description file and a schema definition file, if given an implementation file.

All changes that are to be made to the database schema by `sch2db` can only be communicated to `sch2db` through the schema (.sch) files. Any time a user's class definition changes or a new persistent class is added to the user's header files, the schema files will have to be regenerated using the schema compiler.

Each programming language requires different mechanisms for communicating schema changes to a Versant Database. The following summarizes the basic mechanisms and provides an overview of the language-specific (C/C++) variants.

The following changes can be made to the schema using schema evolution:

- Add a new class to the database
- Drop an attribute from a class
- Add an attribute to a class.
- Completely redefine a class, by dropping the class definition in the database and adding the current one.
- Rearrange the attributes within a class.
- Rename attributes in a class.

CLASSES

Adding a Class

Adding a new class has no impact on existing objects in the database, so no special precautions are necessary.

The new class is defined in the usual language-specific way. The class name must be unique within the relevant databases.

If the new class is intended to become a superclass of any existing class, and must therefore be inserted into an inheritance chain, each intended subclass must be redefined and its instances must be converted.

```
c     o_defineclass()
c++   Use schcomp utility to generate .sch file from .imp file
```

If the `.sch` files include definitions for persistent classes that were not defined in the database before, those classes can be added during evolution.

New classes should be leaf classes. You can always add a class as a subclass of any existing class to the database, but ***you cannot add a class as a superclass of an existing class.***

For example:

Assume schema loaded in the database was generated using the following header file:

```
Class A : public PObject
{
   int x;
};
```

Classes that derive from `A` are leaf classes and can be added to the database using schema evolution. But the definition of class `A` cannot be changed to, say:

```
class A: public PObject, public B
{
   int x;
};
```

where class `B` is another persistent class

Deleting a Class

Deleting a class requires a language-specific drop-class method or function, shown below.

All instances of the target class are deleted from the database, as well as subclasses and their instances.

With the C++/Versant interface, deleting a class also deletes any class with an attribute containing a link to the class being dropped.

For example:

If class `B` has an attribute `Link<A>` that references an instance of `A`, then dropping `A` also drops `B`. This prevents dangling references.

```
c     o_dropclass()
c++   dropclass()
util  dropclass
```

Renaming a Class

With the C or C++ interfaces, renaming a class invalidates instances in a way that cannot be accommodated via lazy updating, as most other schema changes can. Therefore, a multi-step conversion is required as follows:

Add a new class that has the desired class name.

1. Create and run a program to create instances of the new class based on the instances of the old class.
2. Repair any references to the old instances.
3. Delete the old class, and with it the old instances.

For example:

Suppose you want to rename the `Book` class so its name is `Publication`:

1. Add a new `Publication` class, based on the `Book` class definition.
2. Create and run a program to create instances of `Publication` based on instances of `Book`.
3. For each reference to a `Book`, substitute a reference to the equivalent `Publication`.

4. Delete the Book class, and with it, all instances of Book.

Changing an Inheritance Tree

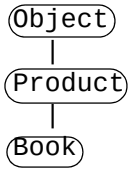
With the C or C++/Versant interfaces, changing a class's inheritance chain invalidates instances in a way that cannot be accommodated via lazy updating, as most other schema changes can. Therefore, a multi-step conversion is required as follows:

1. Rename the class whose inheritance chain is being altered (see "Renaming a Class" above).
2. Add a new class that has the original class name and the new inheritance chain.
3. Create and run a program to create instances of the new class based on the instances of the old class.
4. Repair any references to the old instances.
5. Delete the old class, and with it the old instances.

For example:

Suppose a hypothetical Book class currently inherits from Object.

You want it to inherit from a newly created Product class, resulting in the following inheritance chain:



The steps for accomplishing this conversion would be:

Rename Book to OldBook

1. Add a new Book class that has Product as its superclass. (Product attributes, such as price, might also need to be removed from Book's definition.)
2. Create and run a program to create instances of Book based on instances of OldBook.
3. For each reference to an OldBook, substitute a reference to an equivalent Book.
4. Delete the OldBook class, and with it all instances of OldBook.

ATTRIBUTES

You can add, delete or rename an attribute in a single step.

Versant updates instances automatically as they are accessed, initializing a new attribute's value to zero or NULL.

If you modify the attributes in a superclass, subclass instances are also updated automatically. Data in a renamed attribute is not affected. Data in other attributes also is not affected.

With the C++/Versant interface, adding or dropping an embedded class or struct requires manual conversion of the class and its instances.

```
c     o_newattr(), o_dropattr(), o_renameattr()
c++   Modify schema files, then use sch2db to update
      database.
```

Use `synclass()` to synchronize other databases.

For more information on `synclass()`, please refer to the Versant C++ Reference Manual.

Adding an attribute to a class (C++)

If a persistent class includes a new attribute that is currently not in the database schema, `sch2db` will add this new attribute. If a super class has a new attribute then the current subclass will also inherit the new attribute.

Adding an attribute has no effect on the data in the existing attributes.

When an attribute is added to a class, the attribute will automatically be added to all subclasses. The type of the new attribute can be any valid class or elementary type. The value of a new attribute will be initialized to zero or `NULL`.

Dropping an attribute from a class (C++)

If the database class definition includes an attribute that has been dropped from the class definition in the header files, the schema in the database can be updated using schema evolution. Dropping an attribute implies loss of data in that attribute both in instances of that class and instances of its subclasses.

For example:

If the original definition of `class A` when schema was loaded into the database was:

```
class A {
   int x;
   int y;
};
```

and new schema is generated using the definition

```
class A {
   int x;
};
```

Then, if `sch2db` is used to evolve the schema in the database with the new schema, `sch2db` recognizes that attribute `y` in the database should be dropped and issues the following message:

```
Schema Changes needed:
Class 'A' drop attribute 'A::y'
Ok to apply changes?
```

If you approve the change, attribute `y` will be dropped.

If an attribute type is a structure or class, the whole block of attributes which is an expansion of the structure/class will be dropped.

For example, suppose you have:

```
struct x{
   int a;
   int b;
};
class A : public PObject
{
   x attribute1;
   int attribute2;
};
```

Then, if the schema for class `A` is stored in the database, components of structure `x`, namely `a` and `b` are also part of the database schema. If `attribute1` is dropped from class `A`, then

during schema evolution for class A, the a and b components of attribute1 will also be dropped. When an attribute is dropped, data in remaining attributes is not lost.

Redefining attribute in a class (C++)

It is quite possible that a persistent class has undergone significant changes in that its inheritance hierarchy has changed.

In such a case, it may not be possible for the -e (evolution) option of the sch2db utility, to evolve the schema. Then the only way to change the schema is to drop the class and then redefine it.

For example, if the definition for class A when schema was defined in the database was:

```
class A:public PObject
{
   o_4b x;
};
```

and is now updated to:

```
class A: public PObject, public B
{
   o_4b y;
};
```

then when sch2db is used with the -e option, sch2db will respond with

```
Error  : <8555> SCAP_CLASS_DIFFERS: Class 'A' already
 exists in the database in a different form
```

In this case, to change the schema in the database you must use the -f (force) option. Then, sch2db will drop the class and then redefine it. This does mean that all the data stored in the instances of the class being dropped and redefined is lost.

If a non-leaf class is dropped all of its subclasses will also be dropped. In the above example, B is not a leaf class, since class A inherits from B. If B is dropped, then A, being a subclass of B, also gets dropped.

Rearranging attributes in a class (C++)

If you want to change the attribute ordering in a class, then you should regenerate the schema files and evolve the database schema by using `sch2db` with the `-e` option. The `sch2db` utility will not report any changes if the only change as a result of evolution was a rearrangement of attributes.

Renaming attributes in a class (C++)

If you want to change an attribute name, you can use the option `-r` (rename) of the `sch2db` utility, that changes the attribute name without losing data associated with this attribute.

For example, consider `class A`:

```
class A : public PObject {
   int x;
   int y;
};
```

If you want to change the name of attribute `x` to `p` without losing the data associated with `x` in all instances of `A`, then modify class `A` to reflect the new name of attribute, regenerate the schema files, and run sch2db with the `-r` option.

For example:

```
sch2db –d mydb -r myschema.sch
```

The `sch2db` utility will respond with:

```
Schema changes needed:
Class 'A' renames attribute 'A::x' as 'A::p'
Ok to apply changes?
```

If you say "Ok", then `sch2db` will change the name of attribute `x` to `p` in the database.

NOTE:- It is important to note that if the type of `x` is changed from `int` to some other type, `sch2db` cannot perform the rename and will complain that Class `A` already exists in a different form.

Care should be taken in using the `-r` op*tion* if the attribute positions are switched.

In the above example, if the definition of `class A` was changed so that it now is:

```
class A
{
   int y; // position switched
   int renamed_attrribute_x; // attribute x renamed
};
```

and `sch2db` is used with the `-r` option, then for `class A` what was previously attribute `x` will be renamed to `y` and what was `y` will be renamed to `renamed_attribute_x`. Inadvertently, data that should have been stored under `y` was switched to `rename_attribute_x`, and data that should have been stored for `renamed_attribute_x` is now stored as `y`.

Changing an Attribute's Data Type

Changing an attribute's data type requires multiple steps, but the steps differ depending on whether you are using a typed or untyped programming language.

In C/C++ the steps are:

Rename the attribute.

1. Add a new attribute with the new data type.
2. Create and run a program that casts each old-attribute value to the new data type, and then stores the resulting value in the new attribute.
3. Delete the old attribute.

For example, to change the data type of `Book's` price attribute from `int` to `float`:

Rename the price attribute to oldprice.

1. Add a new attribute named `price`, with float as its data type.
2. Create and run a program that casts each book's `oldprice` integer as a `float`, and stores the result in the price attribute.
3. Delete the `oldprice` attribute.

When the data type is inferred, no special action is required. Versant accommodates values of any recognized data type automatically. If the database contains old instances, however this results in a mix of old and new data types for the attribute, which may cause problems for your applications. In that case, you would need to create an informal program to fetch each old object, recast the target attribute's value to the new data type, and then update the object in the database.

PROPAGATING SCHEMA CHANGES

Communicating a schema change to one or more Versant databases depends on the programming language.

Class changes that are propagated in this way do not affect object identifiers, so references to the affected objects within an application remain valid even after the class is changed. The few exceptions to this rule are noted in the discussions of specific kinds of changes above.

Instances of a changed class, or of its subclasses, are aligned with the new class definition only as they are accessed. This lazy updating mechanism spreads the cost of schema evolution over time, and avoids the need to perform a global sweep that would shut out normal database usage.

No special action is required when migrating objects. If an object is moved to a database in which its class has not been loaded, Versant automatically migrates the class object as well as the instance. If the class is already defined in both the source and target databases, an error is generated unless the two class definitions are identical.

For C/Versant interface:

Since C language does not support class definitions, the C/Versant interface requires that you create a C program to implement each set of schema changes.

The C/Versant interface provides a specific method for implementing each type of schema change, such as `o_dropclass()` or `o_renameattr()`.

For C++/Versant interface:

In the C++/Versant interface, the Schema Change Utility (`sch2db`) is used to propagate changes in schema (.sch) files to a database.

To reload a class definition in multiple databases, you can run the Schema Change utility separately on each database, or run it on one database and then reconcile differences in other databases with the `synclass()`method.

For more information about the Schema Change utility please refer to the *Versant Database Administration manual* and for the `synclass()` method, please refer to the *C++/Versant Reference Manual*.

VERIFYING SCHEMA

Why Verify?

At any given moment, the schema as known to the database may or may not be identical to the schema as known to your application code, depending on how recently and how rigorously the developers in the team have synchronized these two views of the schema.

To avoid such mismatch, especially on large, multi-developer projects, it's a good idea to confirm the schema before testing or deploying your applications.

Each of the Versant interfaces for class-based languages, such as C++ provides a facility for confirming the schema. In each case, the nature of the facility is tailored to the development environment.

Using sch2db

The Schema Change Utility (`sch2db`), which is normally used to change class definitions in a Versant database, can also be used to report mismatches without making any changes.

This reporting feature is enabled with the `-n` flag on the command line that is used to invoke `sch2db`.

Incase of an Application Failure

If an application terminates abnormally and you are running with logging turned OFF, you will probably have to reset the database. This is because with recovery features turned OFF, Versant will not be given an opportunity to restore the database to a consistent state.

To reset the database

1. Run the `stopdb` utility.
2. Restart the database with `startdb`.

   If logging and recovery are turned ON, then the next time you try to use the database, Versant will automatically rollback all incomplete transactions that may be corrupting the database and restore conditions to a consistent state.

Versant uses shared memory and semaphores for its inter-process communications. If an application does not finish normally, shared memory resources may not be released.

To release shared memory resources after a failed application

1. Run the `stopdb` utility.

   If stopdb fails for some reason, manually remove shared memory if any, by using the UNIX `ipcrm` command.
2. Restart the database with `startdb`.
3. If step 1 does not succeed, you will have to forcibly stop the database.
   For information on how to forcibly stop a database, call Versant Customer Support.

   You can use the UNIX `ipcs` command to verify that all shared memory and semaphores have been released.

CHAPTER 7

Memory Architecture

This Chapter explains Versants Memory Management.

The information given here is provided for situations when you want to control memory usage explicitly.

Following topics are covered:

- Memory Management
- Object Cache
- Server Page Cache

MEMORY MANAGEMENT

When you start a session, Versant uses three kinds of memory-

Object cache

The application memory area includes an object cache for objects accessed in the current transaction and related object cache management tables.

Server page cache

The database memory area includes a page cache for recently accessed objects and related database cache management tables.

Process memory

Either one or two processes (depending upon circumstances) are used to manage database and application memory.

For more information on the types of Versant memory usage please refer to the following Chapters:

- Chapter on “Sessions" for an explanation of non-memory elements of a session, including the session database, connection databases and transactions.
- Chapter on "Statistics Collection" for information on collecting performance information about the object cache, page cache, and application process.
- Chapter on "Performance Tuning and Optimization" for information on improving memory usage.
- Chapter on "Database Profiles" in the *Versant Database Administration Manual* for information on parameters you can use to fine tune database and application performance.

C++/VERSANT NOTE: C++/Versant default memory management is suitable for most situations. For example, Versant automatically caches and pins objects in application virtual memory when you dereference them. Also, Versant automatically maintains a page cache in server shared memory.

OBJECT CACHE

To improve access to objects in a transaction, when you start a database session, Versant creates an object cache.

Also created are various information tables to track your processes, connected databases, and transactions.

One of these information tables is the cached object descriptor table, sometimes abbreviated as "cod table". It is a hash table keyed on logical object identifiers.

Objective

Object caches are for the sole use of a particular application.

The object cache is in process virtual memory on the machine running the application.

Normally, the object cache contains objects accessed in the current transaction and is cleared whenever a transaction commits or rolls back.

When an application requests an object, Versant first searches the object cache, then the server cache for the database to which the object belongs, and then the database itself.

Since Versant uses an object cache, both transient and persistent Versant objects and transient non-Versant objects created within a session are no longer valid when the session ends. Similarly, instances created outside a session are no longer valid when a session begins. For this reason, you should avoid using in a session any objects that were created outside a session.

Also, since most Versant classes make use of the object cache, you should avoid using instances of Versant classes outside of a session. Exceptions are the `Vstr<type>` and `PString` classes which have an alternative storage mechanism when the object cache does not exist. However, even for these exceptions, you cannot use in a session any instances created outside of a session.

C++/Versant example: For example, if you create an instance of PString before a session, it will be invalid when the session begins. If you create an instance during a session, it will be invalid after the session ends.

```
main()
{  dom = new PDOM;
   // example of creating a PString before a session
   PString s1 = "before session";
   {  // example of creating a PString in a session
```

```
      dom -> beginsession( "mydb", NULL );
      // string s1 is now invalid
      PString s2 = "inside session";
      dom -> endsession();
      // string s2 is now invalid
   }
}
```

Pinning Objects

When you access a persistent object by dereferencing a pointer or using the `locateobj()` method, it is retrieved from its database, placed in the object cache, and "pinned" in the cache. "Pinning" means that the object will physically remain in the cache until the pin is released.

Pins are released when a transaction commits or rolls back. Pins are held when a transaction checkpoint commits.

You can explicitly unpin a persistent object with the `unpinobj()` method. If you unpin an object, it becomes available for movement back to its database. Whether or not it will actually be removed from memory and returned to its database depends upon available virtual memory and internal Versant logic. When you unpin an object, a pointer to the object may become invalid, as its location in memory is no longer guaranteed.

Unpinning an object does not abandon any changes made to the object, and regardless of its location, all transaction safeguards of object integrity still apply after an object has been unpinned.

Releasing Object Cache

When you commit or rollback a transaction, the object cache is cleared, but the object cache table is maintained.

When the object cache is cleared, persistent objects marked as dirty are written to their database, and all persistent objects are dropped from object cache memory. To save changes but hold persistent objects in the object cache, you can perform a checkpoint commit.

As clearing the cache removes all persistent objects from memory, all pointers to them are invalid. However, after a commit or rollback, you can still use links and pointers to transient objects created or modified in the current session.

Releasing Objects

You can explicitly release an object from the object cache with the `releaseobj()` method, even if the object was previously pinned or marked as dirty.

Releasing an object can be useful if you have no further use for the object in the current transaction.

Releasing an object does not return it to its database, it just releases it from the object cache. If you have previously modified the object in the current transaction, its state will be indeterminate if it was previously unpinned, since it may or may not have been moved back to its database before being released. If you have previously explicitly written changes to the database with a group write, the changes will be saved to its database at the next commit or checkpoint commit.

Because releasing an object can cause its state to become indeterminate, it is usually used only with objects accessed with a null lock.

If you need to use a released object later in a transaction, you should explicitly retrieve it from its database again, with `locateobj()`, or by selecting and dereferencing it.

Cached Object Descriptor Table

Associated with object caches are tables that track their contents and information about the objects that they contain.

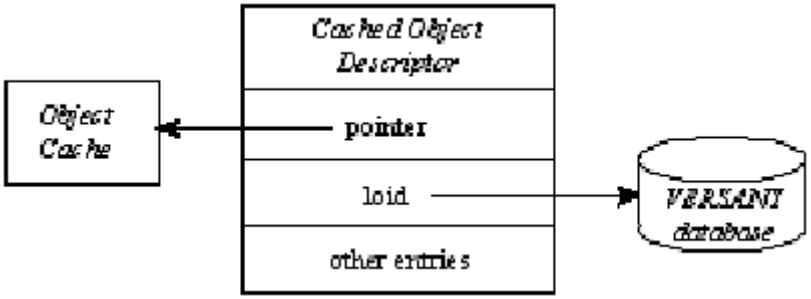
The memory table associated with an object cache is called the "cached object descriptor table", which is sometimes abbreviated to "cod table". It is a hash table keyed on logical object identifiers.

The cached object table contains an entry for each object accessed in the current database session. These entries are maintained regardless of the location of the object, either in memory or on disk, and survive the ending of individual transactions.

The cached object table entry for an object is called the object's "cached object descriptor" and, among other things, contains space for a pointer to the object, if it is in memory, and the object's unique identifier, its logical object identifier. The term "cached object descriptor" is often abbreviated as "cod", and the term "logical object identifier" is often abbreviated as "`loid`".

When you use a link, you are using an indirect reference to a persistent object through the cached object descriptor for the object. If the object is in the object cache, dereferencing a link uses the pointer entry in the cod. If the object is on disk, dereferencing a link uses the logical object identifier entry in the cod.

Cached Object Descriptor



The pointer and `loid` entries in a cached object descriptor can be in one of four states:

pointer	loid state	Meaning
null	null	Link is null.
null	non-null	Link is to an object on disk but not in memory.
non-null	null	Link is to a transient instance in memory.
non-null	non-null	Link is to an object on disk and in memory.

When a process requests an object using a link, Versant first checks the cached object descriptor table to determine whether the object has been previously accessed in the session. If the object is not in the cache, Versant will look for the object in all connected databases, bring it into memory, and return a cached object descriptor for the object.

For each object, besides a loid and a pointer, the cached object descriptor table also maintains the following information about the status of an object in the object cache:

Status of lock level

When you access an object, you implicitly or explicitly set a null, read, update or write lock. The C++/Versant default, which can be changed, is to set a read lock.

Status of pinning

When you access an object in C++/Versant, you implicitly pin the object in the object cache. You can override this behavior with explicit pin and unpin methods. You can access objects with pointers only if they have been pinned. To access an unpinned object, you must use an object link.

Status of update

When you change an object, you must mark it as dirty so that its database will be updated at the next commit.

Status of new

When you create a new persistent object, it is automatically marked for creation in the default database at the next commit.

Status of deleted

When you delete an object, it is automatically marked for deletion from its database at the next commit.

Since the cached object descriptor table maintains all the said information, when you access an object, all relevant information about the object is immediately known to the application. The intermediate level of indirection of cached object descriptors also allows Versant to manage memory as needed without affecting the application.

The size of the cached object table is limited only by available memory. There is one cached object table for each active database session.

Since a session has only one transaction active at any time, only that active transaction can access its cached object table. Since each transaction has its own object cache, it is possible to have duplicate copies of the same object residing in different caches of different transactions conducted by different applications.

When a transaction ends with a commit or rollback, cached objects and their short locks are released, but the cached object descriptors remain in the cached object table. Exceptions to this behavior are parent transactions that inherit locks when a descendant terminates, or when you perform a checkpoint commit that saves changes but maintains locks.

When a database session ends, both objects and cached object descriptors are freed, and all short locks are released.

Cached object descriptors and their states are transparent to an application. You do not need to be concerned with the cached object descriptor table except in the following situations:

Large transaction

By default, when you access an object, C++/Versant pins it in the object cache until the transaction ends.

If you are accessing an extremely large number of objects in a transaction, then it is conceivable that you might run out of virtual memory. In this case, you might want to control pinning explicitly.

If you do unpin an object, you need to remember that you then cannot access it with a pointer.

Long session

The cod table maintains an entry for each object accessed in a session. The cod table is not cleared until a session ends.

In a continuous session over an extremely long period of time, it is conceivable that you might run out of memory for the cod table. In this case, you might want to clear the cached object descriptor table.

Clearing the cached object descriptor table is not necessary until after you have accessed several million objects in a session.

If you do clear the cached object descriptor table, you need to remember that you lose all information about the object, including its update status.

Object Cache Management

To clear the cached object descriptor table, you can use the following explicit routines.

```
c     o_zapcods()
c++   zapcods()
```

To clear the cached object descriptor table with a "clean cods" option, you can use the following.

```
c     o_endpinregion()
      o_gwriteobjs()
      o_xact()
      o_xactwithvstr()
c++   endpinregion()
      gwriteobjs()
      xact()
      xactwithvstr()
```

SERVER PAGE CACHE

To improve access to objects used frequently by multiple applications, when you start a database, either explicitly with the startdb utility or implicitly by requesting a database connection with `connectdb()` or `beginsession()`, Versant creates a page cache for that database.

The database page cache contains disk pages for objects recently requested by an application. The database cache is in shared virtual memory on the machine containing the database so that server processes associated with multiple applications can access the same pages.

Also associated with the database page cache are tables which tracks its contents, track which objects have been locked by which application, and log file buffers.

Pages in the database cache, like objects in the object cache, can have either a "dirty" or "clean" status, which means that a page either does or does not contain objects that have been updated.

Pages are always written to their database when a session ends or a database is stopped. You can also control whether dirty pages are immediately saved to disk when a commit or direct group write occurs by enabling the server process profile parameter commit_flush.

In any case, if logging is turned `ON`, log files automatically track all changes so that the timing of page writing does not affect database integrity.

CHAPTER 8

Process Architecture

This chapter explains Versants Process architecture in details.

It also contains caution statements and recommendations in case of running an application with one process.

Following is covered in this chapter:

- Multi-Threaded Database Server
- Process Usage Notes
- Process Alternatives

MULTI-THREADED DATABASE SERVER

Overview

Beginning with Release 5, database connections start threads rather than processes. This database server architecture is invisible to applications, but the multi-threaded design saves systems resources (because a thread context is smaller than a process context) and improves performance (because thread context switching is faster than process context switching.)

The maximum number of concurrent connections to a database is determined by the server process transaction parameter. There is no limit on the number of concurrent connections, except those imposed by memory and system resources.

Two Process Model

When you link your application with the Versant two process library and/or you make a connection to a database that is not your session database or is not on your local machine, the following database server configuration is used. This situation is the "normal case." All of the following is invisible to an application.

Startup process

When you start a database, either with the startdb utility or by making a connection request, a startup process is created. This process creates an initial block of shared memory, sets up database control tables, mounts the database system volume, performs database physical and logical recovery, and, in general, prepares the database for use. The startup process then forks a cleanup process and then dies.

Cleanup process

The cleanup process stays in existence as long as the database is running. At regular intervals, it wakes up and removes unused system resources, such as dead threads. If a dead thread is found, the cleanup process will clean up the uncommitted transaction and any database resources left by the thread.

The cleanup process also responds to the first request for a database connection.

Server process

When the cleanup process receives the first request for a database connection, it starts the server process. The server process then stays in existence as long as the database is running.

When it receives a connection request, the server process creates a thread that becomes associated with the application session that requested the connection.

Server thread

All database requests by a particular session are handled by the server process thread created for it. The server thread remains in existence as long as the session that created it remains in existence, even if that session itself has no application threads in it.

Detailed sequence of events

Following is a detailed description of what happens when an application requests a database connection:

- The application sends a connection request to the TCP/IP `inetd` daemon on the machine containing the database.
- The inetd daemon starts the `ss` daemon associated with Versant in its configuration tables.
- The ss daemon checks to see whether the database has already been started.
- If the database has not been started, the ss daemon invokes the startup process, which forks the cleanup process and then terminates. The cleanup process then starts the server process and forwards the connection request to it.
- If the database is already running, the ss daemon forwards the connection request to the server process and then terminates.
- The server process creates a server thread for the connection.
- The server thread gets its port number and gives it to the ss daemon. The ss daemon then sends the port number to the application. The application process then sets up a connection between the server thread and the application.
- When the session ends, the server thread terminates.
- When the database is stopped with the stopdb utility, the server process and cleanup process terminate.

Backwards compatibility

For backwards compatibility, separate server processes are started for applications linked with libraries previous to Release 5.

One Process Model

When an application and database are running in a single process, there are no network calls.

An application and a database will run in a single process if all of the following are true:

- The application has been linked with the Versant single process library.
- The database is on the same machine as the application.
- The owner of the application process is the DBA of the database.
- The database is the "one-process database." The "one-process database" is the database associated with the first connection made by the process that satisfies rules 1-3 above.

In a process, there can be only a single one- process database at a time. However, if all connections to the one-process database are dropped by the application process (in all of its sessions,) then a different database can be used as an one-process database.

Per the above rules, if there are multiple threads in a session, they will all share the one-process connection associated with the session.

For a particular process, all one-process connections must be to the same database. The database by definition is the "one-process database" for that application process (including all of its sessions.)

Each connection to a database other than the one-process database will, of course, start a separate server thread on the connection database.

The first time a one process linked application connects to a database that satisfies the above conditions (linking + same machine + DBA + single 1P database), the connecting session will get a one process connection to that database. The application may create more sessions, which will also get one process connections if they connect to the same database. Any connection to any other database, even if it meets the three conditions, will be made in two process mode. This remains true until all of the application's one process connections have been terminated. In other words, a database can have more than one application process with a one process connection, but an application can have only one one-process database.

Valid situations

The following are examples of valid situations.

- Application Process 1 has a single one process connection to DatabaseA at the same time that Application Process 2 has a two process connection to DatabaseA.
- Application Process 1 has five sessions, each connected in one process mode to DatabaseA and connected in two process mode to DatabaseB. Application Process 2 has a session connected in one process mode to Database A. Application Process 3 has a one process connection to DatabaseB, and a two process connection to Database A.
- Application Process 1 has a session connected in one process mode to DatabaseA. It closes that connection, and then connects in one process mode to DatabaseB.

Invalid situations

The following is NOT OK.

- Application Process 1 has a session connected in one process mode to DatabaseA and another session connected in one process mode to DatabaseB.

  This does NOT work, because it violates the rule that any given application process may only have a one process connection to one database at a time.

Running with the one process model will improve performance, but it will slow linking and create a potential for database corruption. The potential for database corruption exists because an abnormal application exit may (or may not) leave the database in an indeterminate state. An example of an abnormal exit is one caused by use of Ctrl-C. This is an operating system issue for all databases running with a single image mode.

There is no automatic cleanup mechanism for threads using the one process model, because if you get an exception, the whole process dies. Leftover application resources will be freed if you do not catch the exception and the application process terminates. If you catch the exception, you can either explicitly end the session in which the thread was running or you can attempt to continue to use the session. If the exception was raised because data structures in the session were corrupted, continuing to use the session is likely to result in another exception.

PROCESS USAGE NOTES

When you start your database session with `beginsession()`, if you specify a session database on a different machine, you will run in two processes regardless of which Versant library you choose. Also, each connection to a group database after the session connection will also start a process on the machine containing the database.

If you decide to run with one process, we recommend the following:

- Enable signal blocking.

  This will block signals, such as Ctrl-C, which can terminate an application abnormally.

  Signal blocking is enabled by turning on the `signal_block` parameter in your application process profile file.

For more information, please refer to "signal_block" parameter in the chapter "Database Profiles" in the *Versant Database Administration Manual*.

- Optionally enable commit flushing.

  You may want to enable commit flushing, which will cause the database cache to be flushed to disk after each commit. However, turning on commit flushing will negate some of the performance gain achieved by running with one process if you are accessing the same objects in successive transactions (because the objects will have to be retrieved from disk with each access).

  Commit flushing is enabled by turning on the `commit_flush` parameter in your database server process profile file.

For more information, please refer to "commit_flush" parameter in the chapter "Database Profiles" in the *Versant Database Administration Manual*.

If your application does terminate abnormally, we recommend the following:

- Immediately stop the session database with the stopdb utility, which will release its shared memory.

  The primary reason for the above recommendations is that the separate memory areas for application and database caches are managed with different mechanisms.

- An application owns and controls its object cache, which is in process virtual memory.

  If any application exits abnormally, the application cache is released, and transaction mechanisms ensure that no problems related to the object cache occur when the application restarts.

- A database owns and controls its page cache, which is always in shared virtual memory. Page caches are for the use of any application using the database, and Versant swaps objects between page caches and disks at its own discretion.

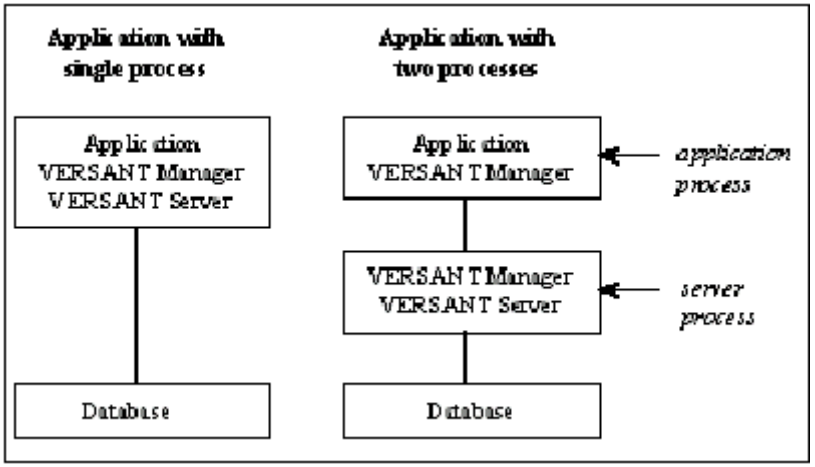
  There is no way for you to control database page caches from within an application. Database shared memory is released only when a database is explicitly stopped with the stopdb utility or when it times out per the `db_timeout` parameter in the server process `profile file`.

For more information, on stopdb please refer to the chapter "Database Utilities" and for db_timeout please refer to the chapter "Database Profiles" in the *Versant Database Administration Manual*.

If you run with two processes and an application exits, normally or abnormally, the database page cache is not affected. A separate process associated with the database will remain to run cleanup after the terminated process.

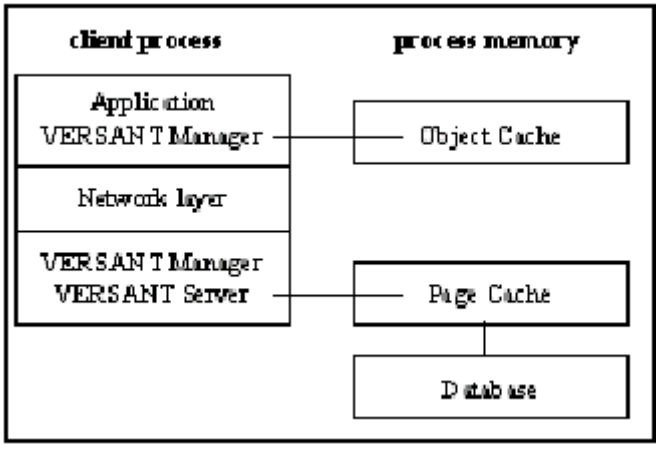
PROCESS ALTERNATIVES

Following are diagrams showing application process alternatives.



One Process Detail

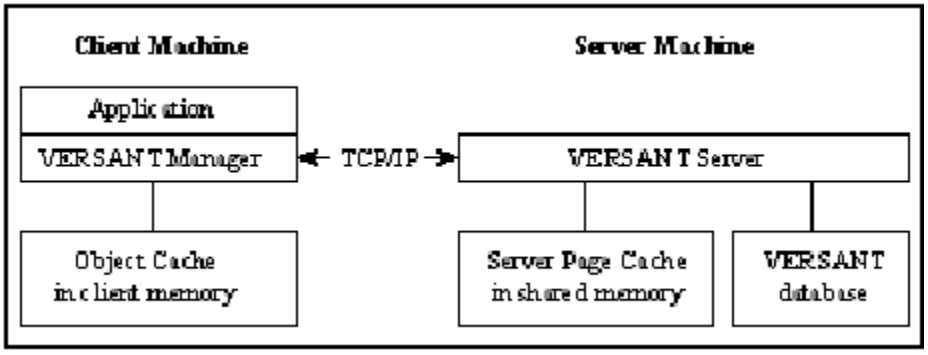
The following shows the One Process.



Two Process Detail

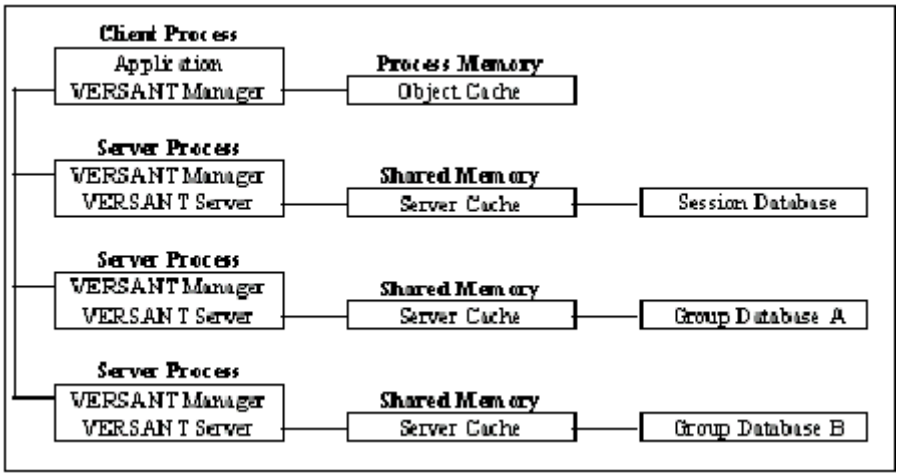
In this case, the application is running with two processes. The "client machine" and the "server machine" could be the same machine with the only difference being that the connection would be internal, rather than through a network connection.

The following shows the Two Processes.



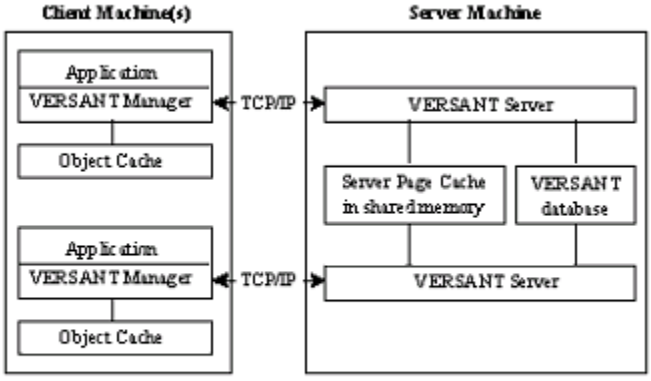
One Client Process Connected to Numerous Group Databases

We can One Client Process performing number of applications by connecting to Numerous Databases.



Multiple Users Sharing a Group Database

Multiple users can connect to the same group database. The following example shows two applications using the same group database as their session database.



CHAPTER 9

Thread and Session Management

This Chapter explains Thread and Sessions management. It also explains how to use multiple threads in one or more sessions.

Following topics are covered:

- Concepts and Terms
- Thread Management
- Thread and Session Alternatives
- Session Management
- Multiple Threads in the Same Session
- Thread Example Programs in C++

CONCEPTS

Following are Thread and Session concepts and some terms associated with it.

Process

A process consists of memory areas, data structures and resources.

A process always has atleast one resource called a "Thread."

Thread

A Thread is the part of a process that performs work.

By definition, when you start a process, you also create a Thread, so a process always has at least one Thread.

A process can start additional Threads with mechanisms specific to its operating system.

When a process starts additional Threads, each additional Thread has some memory areas and data structures of its own, but starting an additional Thread does not require as many resources as starting another process.

When the process that created a Thread ends, that Thread also ends.

Session

A session is a Versants unit of work.

Associated with a Session is an object cache, a set of database connections, a set of locks, a transaction, and various session and cache support tables.

Thread and Session Usage

A Thread can use Versant routines that access and manage persistent objects and databases only when it is in a Session.

If a process has many Threads, any Thread may start a Session.

A Session can have no or many Threads in it.

Threads can open and close Sessions at will, although a Thread can be in only one Session at a time.

A thread can open a Session from within a Session.

A Thread may associate with a database Session and may exit, leaving the Session open for a subsequent Thread.

Transaction

As a unit of work, a session can contain a sequence of one or many transactions.

Each Transaction in each Session maintains its own set of locks. Accordingly, within the scope of the standard Versant lock model, a commit or rollback made in a Transaction in one Session has no effect on a Transaction in another Session.

When multiple threads access the same Session, all Threads in the Session are in the same database transaction. This means that if one thread commits or rolls back a Transaction, then the cache is cleared, and the Transaction is ended for all threads in that transaction.

Object cache

When you create multiple sessions, each session maintains its own object cache in application memory.

If multiple threads are in the same session, the object cache is accessible by all threads in that session. This means that objects acquired by one thread can be accessed and changed by other threads.

In a simple application that uses multiple threads in the same session, you should have each thread be responsible for a mutually exclusive set of objects. In this case, you only have to worry about the timing of your commits and rollbacks (which clear the cache). One approach to maintaining the cache is to perform checkpoint commits when all objects are ready to be saved and then explicitly release from the cache those objects you no longer need.

In a more complex application that uses multiple threads in the same session, the multiple threads may need to read and update the same objects. Since all threads operate in the same transaction, they share the same locks, which means that it is possible for one thread to be reading an object while another is updating it. More than one thread can update the same object at the same time. Since locks will not protect you, you must avoid conflicts either by your own application programming logic or by using thread exclusion mechanisms set on individual objects or groups of objects or on attributes within an object. In either case, you must protect the objects so that one thread changes an attribute only when all other threads are excluded.

Short lock

If you are using multiple sessions, it is as if each session were being run by a different user. This means that an object write locked by one thread in one session cannot be accessed by another thread in another session, even though the threads belong to the same process. In other words, lock requests by one thread in one session can be blocked by locks held by another thread in another session.

However, if you are using multiple threads in a single session, then locks acquired by one thread are also held by other threads, which means that locked objects can be updated by any thread and are not useful for thread exclusion.

THREAD MANAGEMENT

SPECIAL NOTE FOR C++:-

The Versant thread management strategy for the C++ interface has changed to comply with ODMG standards.

The class previously used to manage threads (`VThread`) is no longer required. Instead, all of the thread management operations have been added to the `VSession` class. Although the `VThread` class and methods are still supported by Versant for compatibility, we strongly recommend that new applications be created using VSession for thread management.

Thread Restriction

In a multi-threaded application, in a thread other than main thread of execution, user cannot create `PError` objects before calling a method of `VSession` or `PDOM`.

For more information, please refer to the section on "PError" in the chapter "Support Classes, Types and Macros" in the *C++/Versant Reference Manual*.

Thread Mechanisms

Start operating system thread

To start an operating system thread, use normal interface and operating system mechanisms.

For example:

```
thr_create(NULL, 0, t1_start, NULL, 0, &thr1);
```

Attach thread to a session

To attach a thread to a session, supply a session name to the following methods:

```
c     o_setsession()
c++   VSession::setsession()
```

Detach thread from a session

To detach a thread from a session without having it join a new session, use a "`set session`" routine with a `NULL` session name.

```
c     o_setsession()
c++   VSession::setsession()
```

Get thread parameters

The options for getting the Thread Parameters are:

Default lock

To get the default lock for a thread:

```
c++   VSession::get_default_lock()
```

Session pointer

To get a pointer to the current session object for a thread:

```
c++   VSession::current()
```

Session name

To get the name of the current session for a thread:

```
c++   VSession::get_session_name()
```

User data

To get user data for a thread:

```
c     o_thdataget()
c++   VSession::get_thread_user_data()
```

Set thread parameters

The options for setting the Thread.

Parameters are:

Default lock

To set the default lock for a thread:

```
c++   VSession::set_default_lock()
```

User data

To set user data for a thread:

```
c     o_thdataset()
c++   VSession::set_thread_user_data(const *)
```

Redefine thread exception handling in C++

Error handling

The following methods will replace the default error handling methods in C++/Versant.

```
c++   vpp_thread_terminate()
c++   vpp_thread_unexpected()
```

THREAD AND SESSION ALTERNATIVES

Following are threads and sessions alternatives.
The Versant C, C++, and Java interfaces support multiple threads and multiple sessions.

Multiple Threads in a single Session

You can place multiple threads in a single session, if your operating system supports multiple threads.

When you use multiple threads in a single session, you must take care to coordinate the threads so that they do not interfere with one another, because all threads access the same object cache.

When multiple threads access the same object cache, all threads are in the same transaction and share the same locks. Accordingly, when using "thread unsafe" methods, such as a commit, you must take care to coordinate your work so that one thread does not interfere with the work of other threads, usually by developing a "single writer, multiple reader" approach.

One or more Threads in multiple Sessions

You start multiple sessions and move threads among them at will; you can attach and detach a thread from a session as many times as you like.

Also, you can start multiple sessions even if your operating system doesn't support multiple threads.

When you place each thread into a separate session, Versant coordinates the work of the threads and uses locks and separate object caches to ensure that the threads do not interfere with one another.

A session is a unit of work. Opening multiple sessions allows you to define multiple units of work that are distinct. This can be useful when you want to create a complex application that responds to numerous events of a similar kind. For example, this approach is useful when you want to monitor and respond to hits on switches or a web site.

SESSION MANAGEMENT

Session Management also includes the Session Restrictions, which gives the guidelines on the restrictions during a Session.

Session Restrictions

There are some Session restrictions as follows:

- Custom locking cannot be used with the multiple threads option, `O_THREAD`.
- You cannot mix the use of Versant C and C++ thread related routines in the same session.
- A threaded process should not fork. Forking a threaded process will duplicate all threads in the process. This will lead to unpredictable results and must be avoided.
- A thread can be in only one session at a time.
- User cannot collect statistics on a per thread basis.

Session Mechanisms

Following are the Session Mechanisms:

Start session

The mechanisms for starting a session are:

```
c     o_beginsession()
c++   beginsession()
      VSession()
```

When you start a session, you must specify parameters to the routine that starts the session.

Following is an explanation of these parameters in the context of multiple threads and/or multiple sessions.

Session database name

Each session requires a database to be used as a persistent workspace. You can use the same database as a session database for each of many sessions, or use a different session database

for each session. As with all database connections, you must be a registered user of a database in order to use a database as a session database.

Session name

By default, the name of your first session will be your username, but you will probably want to supply a session name that will enable you to distinguish among sessions created later. In any case, when you start additional sessions, the session names must be unique within the scope of a particular application process.

Session options

Sessions also allow you to specify the kind of session that you want in one or more session options.

Following is an explanation of session options in the context of multiple threads and/or multiple sessions.

When you specify multiple options, use the `|` symbol to concatenate them. Option declarations last for the duration of the session.

Null option

If you specify `NULL`, you will start a "standard session" using standard Versant transactions and locks.

If you start a standard session, you will be able to start additional sessions (indeed, the `NULL` option is the usual case for multiple sessions.)

However, if you start a standard session, you will not be able to place multiple threads or processes in the session.

Optimistic lock option

If you specify `O_OPT_LK`, you will start an "optimistic locking session" that suppresses object swapping, prevents automatic lock upgrades, and provides automatic collision notification.

You can use the optimistic locking option with any other session option.

For more information, refer to “Optimistic Locking” on page 110 in "Chapter 5 - Locks and Optimistic Locking".

Multiple threads option

If you specify `O_THREAD`, you will start a "multiple threads" session that allows use of one or many threads in the same session.

In C/Versant, every function will be protected by a latch in a multiple threads session. In C++/Versant, you will need to coordinate the work of the threads.

See also ""Thread Safe" Methods in C++/Versant" on page 191.

Multi-session option

By default, a thread will join the session it starts. If you want to start a session but not join it, specify the "do not attach" option, `O_MULTI_SESSION`.

You must specify the `O_MULTI_SESSION` option if you want to use the same thread to start multiple sessions that will be available for immediate use at a later time. It is an error to start more than one session in a particular thread without specifying this option.

Read access option

If you specify `O_READ_ACCESS`, you will be able to view but not change data.

The read access option can be used with all other session options.

Write access option

If you specify `O_WRITE_ACCESS`, you will be able to edit data.

This access mode is not currently supported, but may be supported in future releases. At present, this mode is the same as `O_READ_ACCESS|O_WRITE_ACCESS.`

The write access option can be used with all other session options.

Read and write access option

If you specify `O_READ_ACCESS|O_WRITE_ACCESS`, you will be able to view and edit data.

This access mode is the default and does not have to be explicitly specified. It is compatible with all other session options.

C++ Usage Notes

If you want to use more than one thread or if you want to start more than one session, you must start your sessions by creating transient instances of class `VSession` with `VSession`().

Because the case of a single process in a single session is a subset of the general case of multiple threads / multiple sessions, you can use `VSession()` to start a single session that will have a single thread (the primary thread associated with the application process.)

The `beginsession()` method, which starts a session and places the starting thread into the session, continues to be available for backwards compatibility with previous Versant releases. However, if you use `beginsession()`, you will be able to start only a single session, and your thread can leave the session only by ending the session.

A thread can use Versant routines that access and manage persistent objects and databases only when it is in a session. If you create a session with `VSession()`, we recommend that you invoke Versant methods on your session instance.

However, for backwards compatibility, even if you start a session with `VSession()`, you can continue to use the global variable `::dom` of type `PDOM*` and invoke database methods on the `::dom` instance, as Versant is able to associate a thread with its current session.

C++/Versant example

The following starts a session by creating a transient instance of the `VSession` class.

```
o_opt options;
options[0] = O_THREAD;
VSession* session1 = new VSession(
   NULL,
   "db1",
   "session1",
   options);
```

After the session has started, you can send `PDOM` methods to an instance of `VSession`.

For example:

```
session1 ->commit();
```

Although we recommend that you send database methods to an instance of `VSession`, you can send `PDOM` methods to the `dom` global variable, as usual.

For example:

```
::dom->commit();
```

If you use the `dom` variable, it will delegate the method to the correct `VSession` object for you.

For example, if you want, say, `thread15` in `session4` to perform a commit, the thread sends the message with normal syntax:

```
::dom->commit();
```

In this case, the `dom` variable will automatically associate `thread15`, which called `commit()`, with `session4` and cause a commit only on the current transaction in `session4`.

End session

The mechanisms for ending a multiple session are:

```
c     o_endsession()
c++   endsession()
      ~VSession()
```

A thread does not have to be in a session to end it, but it can be there.

If one thread tries to end a session while another thread is still in that session, an error will be raised.

As with all sessions, ending a multiple session or a session with multiple processes or threads will commit and terminate the current transaction, close all database connections made in that session, and commit, rollback, or continue the current transaction.

C++/Versant: If you created the session with `VSession()`, you must close it with `~VSession()`.

Get session parameters

Default database

Get the default database for a session:

```
c++   get_default_db()
```

User data

Get user data for a session:

```
c     o_sedataget()
c++   get_user_data()
```

Number of threads

Get the number of threads currently attached to a session:

```
c++   get_thread_count()
```

Set session parameters

Default database

Set the default database for a session:

```
c++   set_default_db()
```

User data

Set user data for a session:

```
c     o_sedataset()
c++   set_user_data()
```

MULTIPLE THREADS IN SINGLE/SAME SESSION

There can be Multiple Threads in the same Session. The Threads have Safe and Unsafe Methods. This Section gives us detailed description on these.

If you use a single thread per session, the work of each thread will be protected by the session boundaries. If you place multiple threads in the same session, however, you will need to coordinate their activities.

Following are usage notes related to the use of multiple threads in the same session. The behavior varies depending upon the interface you are using.

C/Versant

For the C/Versant interface, all entry functions into the C/Versant kernel library are protected by a mutex latch. This means that one thread cannot update an object while any other C/Versant library function is reading or updating it.

C++/Versant

The C++/Versant interface is built on top of the C/Versant library. Most C++/Versant methods call the corresponding C library routine to perform database operations. However some of the operations, like dereferencing a link, may not call C/Versant if the object is already in cache.

Attach a thread to the current session by using `VSession::set_session()`. This will initialize the thread local variables used by C++/Versant.

"Thread Safe with Exceptions" Methods in C++/Versant

Some C++/Versant methods affect only individual objects or group of objects, but they still have an impact on other threads that happen to be using the objects in question. These methods are called "thread safe with exceptions."

For example, if you delete an object using one thread, then that object will no longer be available even if another thread is currently accessing the object. Because the delete changes only a single object rather than all objects, the delete method is considered "safe with exceptions" rather than "unsafe".

If you use a "thread safe with exceptions" method, you must protect the objects affected by the method by using a "one writer / multiple readers" exclusion mechanism (MR/SW) for each object or groups of objects. In a MR/SW mechanism, threads that will execute only thread-safe methods

enter the critical region as readers and the thread that will execute the thread-safe-with-exceptions methods will enter the region as a writer. Based on this scheme, the writer can execute thread-restricted methods only when no reader is in the critical region.

Similar exclusion mechanisms can be implemented on individual attributes of objects. For example, a `vstr` or link vstr used as an attribute of an object could be protected rather than the entire enclosing object to increase the concurrency. An iterator over a link vstr would enter as a reader; a method that adds to or removes an element from the link vstr would enter as a writer.

The following C++/Versant routines are "safe with exceptions". The list is not complete since all modifier methods on `vstr, link vstr`, and `collection` classes change the object and hence are thread-restricted.

Thread-safe-with-exceptions global operator

```
operator delete
```

Thread-safe-with-exceptions PDOM methods

```
checkloid()
freebeprofile()
freefeprofile()
gdeleteobjs()
gwriteobjs()
refreshobj()
refreshobjs()
releaseobjs()
xactwithvstr(), commit and rollback options
```

Thread-safe-with-exceptions PClass methods

```
dropclass()
dropinsts()
```

Thread-safe-with-exceptions PObject methods

```
deleteobj()
releaseobj()
```

```
unpinobj()
```

Thread-safe-with-exceptions Link<type> and LinkAny methods

```
deleteobj()
releaseobj()
```

Thread-safe-with-exceptions Vstr<type> and VstrAny methods

```
add()
append()
remove()
unordered_remove()
```

Thread-safe-with-exceptions LinkVstr<type> and LinkVstrAny methods

```
add()
append()
remove()unordered_remove()
```

“Thread Safe” Methods in C++/Versant

Some methods affect objects, but they are well behaved in terms of the impact on other threads. These methods are called "thread-safe."

For example, if you perform a checkpoint commit with one thread, objects will still be available in cache to other threads and all threads will see the same object states. If two threads try to dereference the same link at the same time, the end-impact on the cache is the same regardless of the order of execution.

All methods not noted above as thread-unsafe or thread-restricted are thread-safe. For example, the following methods are thread safe.

Thread safe global macro

```
O_NEW_PERSISTENT(....)
```

Thread safe PDOM methods

```
downgradelock()
downgradelocks()
xact(), if the action option is O_CHECKPOINT_COMMIT.
xactwithvstr(), if the action option is O_CHECKPOINT_COMMIT.
```

Thread safe PObject method

```
dirty()
```

Thread safe Link<type> and LinkAny methods

```
dirty()
locateobj()
operator->()
```

Thread safe PClass method

```
select(), if the option O_FLUSH_NONE is used.
```

“Thread Unsafe” Methods in C++/Versant

Some C++/Versant methods affect the cache globally and therefore require that no other threads be accessing any objects in the cache at the same time. These methods are called "thread unsafe".

For example, a database commit method is thread unsafe, because a commit performed by one thread will clear the object cache even if other threads are currently accessing objects in the cache.

If you use a thread-unsafe method, you must coordinate the threads with the thread synchronization mechanism provided by your operating system thread package. Typically a mutex and a thread counter is are used to synchronize the threads. An example is provided at the end of this note.

For operating systems that support conditional variables, a preferred synchronization mechanism is the conditional variable. Your the synchronization mechanism must ensure that

only one thread (the thread executing the thread-unsafe method) is accessing objects in the cache at the time the unsafe method is invoked.

The following C++/Versant methods are thread unsafe.

Thread unsafe PDOM methods

```
abort()
autorelease()
beginsession()
begintransaction()
commit()
connectdb()
detachsession()
disconnectdb()
endpinregion()
endsession()
endtransaction()
exit()
forkprocess()
joinsession()
resetoct()
rollback()
savepoint()
set_default_db()
set_default_lock()
undosavepoint()
xact(), when the action option is not O_CHECKPOINT_COMMIT
xactwithvstr(), when the action option is not O_CHECKPOINT_COMMIT
zapcods()
```

Thread unsafe PClass method

```
select(), when the option O_FLUSH_NONE is not used
```

EXAMPLE PROGRAMS IN C++

Following are the Thread example programs in C++:

Example 1 - Multiple Threads in Multiple Sessions

The following is an example of using multiple threads in multiple sessions.

testthrd.h

The following declares the class `TestThread` which will be used in the example program.

```
#include <cxxcls/pobject.h>
#include <cxxcls/vsession.h>
class TestThread : public PObject
{
   o_u4b   thread_id;
   o_u4b   session_id;
   o_4b    myid;
public:
   TestThread( o_4b _myid );
   void set_myid( o_4b );
   void print();
};
```

testthrd.cxx

The following implements the class TestThread.

```
#include <iostream.h>
#include "testthrd.h"
TestThread::TestThread(o_4b _myid) : myid(_myid)
{
   thread_id = thr_self();
   session_id = VSession::current()
       ->get_session()->get_session_index();
}
void
TestThread::set_myid(o_4b id)
```

```
{
   myid = id;
   dirty();
}
void
TestThread::print()
{
   cout << "thread_id = " << thread_id << "\t"
        << "session_id = " << session_id << "\t"
        << "myid = " << myid << endl;
}
```

schema.imp

The schema implementation file.

```
#include <cxxcls/pobject.h>
#include "testthrd.h"
O_CAPTURE_SCHEMA(TestThread);
```

main.cxx

The following program uses multiple threads in multiple sessions.

```
#include <stdlib.h>
#include <unistd.h>
#include <iostream.h>
#include <cxxcls/pobject.h>
#include "testthrd.h"

int t1_start();
int t2_start(char*);

typedef void* (*THREAD_FUNC)(void *);

int main(int argc, char** argv)
{
if(argc != 3)
{
     cerr << "Usage: " << argv[0]
     << " pdb gdb" << endl;
```

```
  exit(1);
}

char* pdb = argv[1];
char* gdb = argv[2];

// create 3 sessions which don't have any thread
// attached

o_opts options;
options[0] = O_MULTI_SESSION | O_THREAD;

VSession* s1 = new VSession("long1", pdb, "Session1", options);
VSession* s2 = new VSession("long2", pdb, "Session2", options);
VSession* s3 = new VSession("long3", gdb, "Session3", options);

// create 2 threads

thread_t t1, t2;
thr_create(NULL, 0, (THREAD_FUNC)t1_start, NULL, 0, &t1);
thr_create(NULL, 0, (THREAD_FUNC)t2_start, (void*)gdb, 0,&t2);

// wait until all threads return

int status;
thread_t departed;
while( thr_join(0,&departed,(void**)&status)==0 )
{
     cout << "thread " << departed
          << " returned with return value "
          << status << endl;
}

// end session by delete the VSession object

delete s1;
delete s2;
delete s3;

return 0;
```

```
}

int t1_start()
{
int thr_id = thr_self();

cout << "tid " << thr_id << ": join Session1..." << endl;

// this thread joins to Session1

//VThread mythrd("Session1");
VSession::set_session("Session1");

// create some persistent objects
// those objects should have a session_id=0 and
// thread_id=thr_id, they will be in database pdb
cout << "tid " << thr_id
     << ": create 10 TestThread instances" << endl;
for(int i=0; i<10; i++)
     O_NEW_PERSISTENT(TestThread)(i);

// A commit freshs the object cache
// The following commit is safe, because only one thread
// is in Session1. If there were other threads in
// Session1, a commit might disrupt their work unless
// you took care to coordinate threads

cout << "tid " << thr_id << ": doing commit..." << endl;
::dom->commit();

// now detach from Session1 and join to Session3

cout << "tid " << thr_id
<< ": detach from Session1 and join to Session3..."
 << endl;

//mythrd.set_session("Session3");
VSession::set_session("Session3");
// create some persistent objects
// those objects should have a session_id=2 and
// thread_id=thr_id, they will be in database gdb
```

```
cout << "tid " << thr_id
     << ": create another 10 TestThread instances" << endl;
for(i=0; i<10; i++)
     O_NEW_PERSISTENT(TestThread)(i);

// this commit is safe because only one thread attached
// to Session3

cout << "tid " << thr_id << ": doing commit..." << endl;
::dom->commit();

// switch back to Session1
cout << "tid " << thr_id
     << ": switch back to Session1..." << endl;

//mythrd.set_session("Session1");
VSession::set_session("Session1");

// find some objects
// select() flushs object from the cache to the backend
// you need to coordinate between the threads. It is
// safe here because only one thread in the Session1

LinkVstr<TestThread> l = PClassObject<TestThread>::Object().select(
NULL, 0, NULL_PREDICATE);

int size = l.size();
cout << "tid " << thr_id << ": selected " << size
     << " objects back" << endl;
for(i=0; i<size; i++)
     l[i]->print();

l.release();
::dom->commit();
cout << "******** thread " << thr_id << " done ********"
 << endl;

return 0;
}
```

```
int t2_start(char* dbname)
{
int thr_id = thr_self();
cout << "tid " << thr_id << ": join Session2..." << endl;

// this thread joins Session2

//VThread mythrd;
//mythrd.set_session("Session2");
VSession::set_session("Session2");

// get session handle
//VSession* h_sess = mythrd.get_session();

VSession* h_sess = VSession::current();

// create some persistent objects
// those objects should have session_id=1 and
// thread_id=thr_id, they will be in database pdb

cout << "tid " << thr_id << ": create 10 instances" << endl;
for(int i=0; i<10; i++)
     O_NEW_PERSISTENT(TestThread)(i);

// this commit is safe because only one thread in
// Session2

cout << "tid " << thr_id << ": doing commit..." << endl;
::dom->commit();

// find some objects

LinkVstr<TestThread> l = PClassObject<TestThread>::Object().select
(NULL, 0, NULL_PREDICATE);

int size = l.size();
cout << "tid " << thr_id << ": selected " << size
     << " objects back" << endl;
for(i=0; i<size; i++)
     l[i]->print();
```

```
// connect Session2 to gdb
// alternative, you could say ::dom->connectdb(dbname)

cout << "tid " << thr_id << ": connect to " << dbname << endl;
h_sess->connectdb(dbname);

// create 10 objects in the gdb. Note the use of new macro
// O_NEW_PERSISTENT1(db, class)
// those objects should have session_id=1 and
// thread_id=thr_id, they will be in database gdb

for(i=0; i<10; i++)
     O_NEW_PERSISTENT1(dbname,TestThread)(i);

cout << "tid " << thr_id << ": doing commit..." << endl;
::dom->commit();

// find some objects in gdb

l = PClassObject<TestThread>::Object().select(dbname,
      0, NULL_PREDICATE);
size = l.size();
cout << "tid " << thr_id << ": selected " << size
     << " objects back" << endl;
for(i=0; i<size; i++)
    l[i]->print();

l.release();

cout << "tid " << thr_id << ": disconnect from gdb..."
 << endl;
h_sess->disconnectdb(dbname);

// call cleanup before you call thr_exit()
// mythrd.cleanup();

cout << "******** thread " << thr_id << " done ********" << endl;

thr_exit(0);
```

```
return 0; // to make the compiler happy
}
```

makefile

Following is a sample `makefile`.

```
include ../makecxx.com
EXTRA_FLAGS=-mt
OBJS= main.o testthrd.o schema.o
EXEC= a.exe
all: $(EXEC)
a.exe: $(OBJS)
   $(CXXLINK) $(OBJS) $(CXX_LIBRARY)
$(OBJS): testthrd.h
schema.o: schema.cxx
schema.cxx: schema.imp
clean:
-rm $(TEMP_FILES) $(INSTANCE_FILE) schema.cxx $(EXEC)$(CXX_TO_NULL)
   -rm -rf Templates.DB
run_demo: all
   -stopdb -f $(PDB1)
   -removedb $(PDB1)
   -createdb $(PDB1)
   sch2db -y -D $(PDB1) schema.sch
   -stopdb -f $(GDB1)
   -removedb $(GDB1)
   -createdb $(GDB1)
   sch2db -y -D $(GDB1) schema.sch
   a.exe $(PDB1) $(GDB1)
```

Example 2 - Multiple Threads in Single Session

Following is a simple thread application:

In this application, the main thread starts a number of threads which perform some database operations. The main thread performs the thread unsafe methods `commit()` and `exit()` only when all the other threads are terminated. The thread synchronization is achieved by using a mutex and an integer counter. A better approach using conditional variable could be used on platforms where it is supported.

The application does the following:

- begins a session on a database
- spawns a number of threads to create new persistent instances
- waits for all threads to finish
- performs a commit
- ends the session
- begins a session on a database
- spawns a number of threads to dereference all instances, load them into databases, and call a virtual function on them
- waits for all threads to finish
- performs a commit
- ends the session
- begins a session on a database
- spawns a number of threads to dereference random instances based on a query using random numbers and call a virtual function on them
- waits for all threads to finish
- performs a commit
- ends the session
- begins a session on a database
- spawns a number of threads to load objects into memory. Some of the threads will cause exceptions. If an exception thrown, the thread is terminated.
- waits for all threads to finish
- performs a commit
- ends the session

Following is a listing of the sample program. This program is available in the Versant directory `..demo/cxx/threads` or `..demo\cxx\threads`.

class.h

```
#ifndef __CLASS_H
#define __CLASS_H
#ifdef __SVR4
#include <thread.h>
#define vpp_procSleep(x)     sleep(x)
#endif
#if defined(WIN32)
#include <windows.h>
#define vpp_procSleep(x)     Sleep(1000 * x)
#define vpp_mutexInit(x)     InitializeCriticalSection(x)
#define vpp_mutexLock(x)     EnterCriticalSection(x)
#define vpp_mutexUnlock(x)   LeaveCriticalSection(x)
#define vpp_thrExit(x)       ExitThread(x)
#define vpp_getTid()         GetCurrentThreadId()
typedef CRITICAL_SECTION     vpp_mut;
typedef DWORD                vpp_thr;
#endif /* WIN32 */

#include <cxxcls/pobject.h>

class Test : public PObject {
        o_4b value;
public:
        static o_4b callCount;
        Test(o_4b a = 0) :value(a) {}
        virtual void foo();
};
class TestTest : public virtual Test {
        o_4b    value2;
public:
        TestTest(o_4b a = 0) : Test(a),value2(2*a) {}
        void foo();
};

#endif
```

class.cxx

```
#include "class.h"

vpp_mut foo_mutex;
o_4b Test::callCount = 0;

void Test::foo()
{
        vpp_mutexLock(&foo_mutex);
        Test::callCount++;
        vpp_mutexUnlock(&foo_mutex);
}
void TestTest::foo()
{
        vpp_mutexLock(&foo_mutex);
        Test::callCount++;
        vpp_mutexUnlock(&foo_mutex);
}
```

main1.cxx

```
#include <stdlib.h>
#if !defined(_WIN32)
#include <unistd.h>
#endif /* _WIN32 */
#include "class.h"

#ifdef __SVR4
#include <thread.h>
typedef thread_tvpp_tid;
#define vpp_thrYield()thr_yield()
#endif

#if defined(_WIN32)
#include <windows.h>
typedef DWORDvpp_tid;
#define vpp_thrYield()Sleep(0)
```

```
#endif /* _WIN32 */

#include <cxxcls/vsession.h>

vpp_mut  sync_mutex; // create a barrier so that all
                     // child threads start together

vpp_mut  count_mutex;
extern  vpp_mut foo_mutex;

int count = 0;// To keep track of completed threads for
              // synchronization

char *gdbname; // database name

void *(*f)(char *);// function to be called in the
                   // session

void session(o_u1b operation);

int number_of_threads;
const int objects_per_thread = 50;

void my_terminate()
{
// increate the counter so that the main thread knows
// this thread is finished

vpp_mutexLock(&count_mutex);
        count++;
        vpp_mutexUnlock(&count_mutex);
::vpp_thread_terminate();
}

void my_unexpected()
{
// increate the counter so that the main thread knows
// this thread is finished
        vpp_mutexLock(&count_mutex);
        count++;
        vpp_mutexUnlock(&count_mutex);
```

```
::vpp_thread_unexpected();
}

void *database_create(char *arg)
{
// this thread creates some objects in the database
//VThread mythread("session");
VSession::set_session("session");
set_terminate(my_terminate);
set_unexpected(my_unexpected);

vpp_mutexLock(&sync_mutex);
vpp_mutexUnlock(&sync_mutex);

int me = atoi(arg);
int objval;

for (int j = 0; j < objects_per_thread; j++)
{
     objval = me * objects_per_thread + j;
     O_NEW_PERSISTENT(Test)(objval);
}

for (j = 0; j < objects_per_thread; j++)
{
     objval = me * objects_per_thread + j;
     O_NEW_PERSISTENT(TestTest)(objval);
}

vpp_mutexLock(&count_mutex);
count++;
vpp_mutexUnlock(&count_mutex);

//mythread.cleanup();
vpp_thrExit(0);
return 0; // make the compiler happy
}

void *database_deref(char *arg)
{
```

```
// this thread dereferences all the Test instances in
// the database
// VThread mythread("session");
VSession::set_session("session");
set_terminate(my_terminate);
set_unexpected(my_unexpected);
vpp_mutexLock(&sync_mutex);
vpp_mutexUnlock(&sync_mutex);

// select is OK because we have no dirty objects in the
// cache and flushing the cache has no impact on all
// threads or we could specify the O_FLUSH_NONE option

LinkVstr<Test> lv = PClassObject<Test>::Object().select(
                    gdbname,
                    1, // Yes, we want sub class instances
                    NULL_PREDICATE);

int size = lv.size();
fprintf(stderr, "Thread %d: Select got %d objects \n",
                 vpp_getTid(), size);

for (int i = 0; i < size; i++)
    lv[i]->foo();

lv.release();
vpp_mutexLock(&count_mutex);
count++;
vpp_mutexUnlock(&count_mutex);

//mythread.cleanup();
vpp_thrExit(0);
return 0; // make the compiler happy
}

void *database_deref_random(char *arg)
{
// this thread tests the deference of instances based on
// the query with random numbers
VSession::set_session("session");
set_terminate(my_terminate);
```

```
set_unexpected(my_unexpected);
vpp_mutexLock(&sync_mutex);
vpp_mutexUnlock(&sync_mutex);

int k = rand();
PPredicate pred = PAttribute("Test::value") == (o_4b) k;
LinkVstr<Test> lv = PClassObject<Test>::Object().select(
                    gdbname,
                    1, // Yes, we want sub class instances
                    pred);

int size = lv.size();

for (int i = 0; i < size; i++)
    lv[i]->foo();
lv.release();
vpp_mutexLock(&count_mutex);
count++;
vpp_mutexUnlock(&count_mutex);

//mythread.cleanup();
vpp_thrExit(0);
return 0; // make the compiler happy
}

void *database_error(char *arg)
{
// this thread tests the exception handling mechanism
VSession::set_session("session");
set_terminate(my_terminate);
set_unexpected(my_unexpected);
vpp_mutexLock(&sync_mutex);
vpp_mutexUnlock(&sync_mutex);
LinkVstr<Test> lv;

try
{
lv  = PClassObject<Test>::Object().select(
      gdbname,
      1, // Yes, we want sub class instances
```

```
      NULL_PREDICATE);

int size = lv.size();
if ((vpp_getTid() % 3) == 0)
     lv[0]->foo();
else if ((vpp_getTid() % 3) == 1)
lv[-1]->foo(); // this will throw an exception
else if ((vpp_getTid() % 3) == 2)
lv[size]->foo(); // this throws an exception too
}
catch (PError e)
{
e.print();
}

lv.release();
vpp_mutexLock(&count_mutex);
count++;
vpp_mutexUnlock(&count_mutex);

//mythread.cleanup();
vpp_thrExit(0);
return 0; // make the compiler happy
}

main(int argc,char **argv)
{
if (argc != 2)
{
fprintf(stderr,"Usage:   main1 <number-of-threads>\n");
exit(-1);
}

char *dbname = getenv("PDB1");
if (!dbname)
{
fprintf(stderr,"You should specify the database name with PDB1\n");
exit(-1);
}
gdbname = dbname;
number_of_threads = atoi(argv[1]);
```

```
vpp_mutexInit(&sync_mutex);
vpp_mutexInit(&count_mutex);
vpp_mutexInit(&foo_mutex);

session('c');// create objects
session('d');// dereference all objects
fprintf(stderr, "Dereferenced %d objects, should be
 %d\n",
Test::callCount,(number_of_threads * number_of_threads
 * objects_per_thread * 2));
Test::callCount = 0;
session('r');// dereference randomly
fprintf(stderr, "Dereferenced %d objects\n", Test::callCount);
fprintf(stderr, "Now testing exception handling in threads...\n");
session('e');// test error case
} // main

void session(o_u1b operation)
{
vpp_tid tid;

switch (operation)
{
case 'c':
f = database_create;
break;
case 'r':
f = database_deref_random;
break;
case 'd':
f = database_deref;
break;
case 'e':
f = database_error;
break;
default:
fprintf(stderr,"usage error\n");
exit(-1);
}
```

```
vpp_mutexLock(&sync_mutex);

o_opts option;
option[0] = O_THREAD ;

VSession *session = new VSession(
                                  "longtrans",
                                   gdbname,
                                  "session",
                                   option );
for (register int i = 0; i < number_of_threads; i++)
{
char *which = new char[8];
sprintf(which, "%d", i);

#if defined(__SVR4) && defined(VPP_CXXSUN40)
     o_u4b status;
typedef void* (*THREAD_FUNC)(void *);
status = thr_create(
                    NULL,// stack base, default
                    0,// stack size, default
                    (THREAD_FUNC)f,
                    which,// arguments
                    0,// creation flag, default
                    &tid// thread id returned
                    );
if (status != 0)
{
fprintf(stderr,"thr_create() fails with error %d\n",errno);
exit(-1);
}

#endif /* defined(__SVR4) && defined(VPP_CXXSUN40) */
#if defined(VPP_MSVC20)
    HANDLEtHandle;

    tHandle = CreateThread(
              NULL,/* security attributes */
              0,/* stack size, default */
              (LPTHREAD_START_ROUTINE)f,
```

```
                which,/* arguments */
                0,/* creation flags, default */
                &tid/* thread id returned */
                );

        if (tHandle == NULL)
        {
           printf("CreateThread fails with error %d\n",
                  GetLastError());
           exit(-1);
        }

#endif
fprintf(stderr,"started a thread with id %d\n",tid);
}

vpp_mutexUnlock(&sync_mutex);

// wait for all other threads to finish
int prevCount = 0;
fprintf(stderr,"In main thread, waiting for %d child
 threads...\n",
number_of_threads);
for (;;)
{
vpp_mutexLock(&count_mutex);
if (count != prevCount)
{
fprintf(stderr,"In main thread, %d child thread(s)
                completed.\n", count);
       prevCount = count;
}
if (count < number_of_threads)
{
vpp_mutexUnlock(&count_mutex);
vpp_thrYield();
continue;
}
else
{
```

```
vpp_mutexUnlock(&count_mutex);
break;
}
}
// now all threads have finished when we come out of
// the for loop
session->commit();
mainThread->set_session(0);
delete session;
count = 0;
}
```

schema.imp

```
#include "class.h"
O_CAPTURE_SCHEMA(Test);
O_CAPTURE_SCHEMA(TestTest);
```

makefile.sol

```
CCFLAGS=  -mt -I`oscp -p`/h -g
LIBS= -L`oscp -p`/lib/sun.4.0 -L`oscp -p`/lib  -lcxxcls
 -loscfe -lsocket -lnsl
a.out : class.o schema.o main1.o
        CC $(CCFLAGS) class.o schema.o main1.o $(LIBS)
class.o: class.h class.cxx
        CC $(CCFLAGS) -c class.cxx
main1.o: class.h main1.cxx
        CC $(CCFLAGS) -c main1.cxx
schema.cxx : class.h schema.imp
        schcomp -I`oscp -p`/h schema.imp
schema.o : schema.cxx
        CC $(CCFLAGS) -c schema.cxx
clean:
   rm -rf a.out *.o Templates.DB schema.cxx schema.sch *.i0 *.i1 *.i2
```

CHAPTER 10

Statistics Collection

This Chapter explains the performance monitoring statistics in details.

Following topics are covered:

- Statistical Information
- Statistics Quick Start
- Statistics Collection and Storage
- Statistics Names
- Statistics Operations
- Usage Notes

STATISTICAL INFORMATION

To help you measure and analyze the database performance, you can obtain the following kinds of statistical information.

Collecting Information

About an application process

You can monitor application activity, such as the number of objects read into the object cache.

About sessions

You can monitor session information, such as the number of dirty objects in the object cache. Session information and application process information are generally the same.

About connections

You can monitor connections made to a database, such as the number of seconds that applications spend waiting for locks. You can also monitor the time spent on Versant functions by using predefined function names in statistics tools and routines. You can monitor the time spent in your own functions by setting enter and exit points that trigger statistics collection.

About databases

You can monitor database information, such as the number of active transactions.

About latches

You can monitor latches per database or per connection.

About time

You can print timestamp along with the statistics.

About Derived statistics

You can derive your own statistics using combinations of other statistics along with numerical operators and statistical functions.

Collecting Statistics and Viewing

You can collect statistics in a file and view them with the command line utility `vstats`.

You can also collect statistics in the memory. When you collect statistics in the memory, you can view it in real time with the `vstats` utility or bring them into your application with a "`get statistics`" routine.

For information related to vstats, please refer to the *Versant Database Administration Manual*.

For information on how to use these statistics, refer to the Chapter “Performance Tuning and Optimization” on page 253.

STATISTICS QUICK START FOR PERFORMANCE MONITORING

There are some quick starting methods for collecting the statistics for performance monitoring.

You can access the performance monitoring functionality:

- with a direct connection,
- with automatic profiling,
- with interface routines.

Get Statistics with a Direct Connection

You can use the `vstats` utility to connect directly to a database and display database or connection statistics.

The advantages of using a direct connection are convenience and low overhead.

The disadvantages are lack of access to application statistics and lack of information on what operations a database is performing.

1. Select statistics, Turn statistics `ON`

The first step in using a direct connection is identifying the statistics of interest. Since there are so many statistics available, it is often useful to view a listing of all statistics.

To receive the complete list of statistics names, each with a one-line explanation, invoke the following from a command line:

```
vstats -summary
```

Now suppose we are interested in the performance of a database page cache. From the list of all statistics, we might choose the `db_data_reads` and `db_data_located` statistics as the ones, which we would like to examine.

For viewing the statistic, first make sure that collection of that statistic is turned `ON`. The most direct and convenient way to do this is by invoking `vstats` as follows:

```
vstats -database db1 -on db_data_reads \ db_data_located
```

2. View statistics for a database

Once a statistic is turned `ON`, we can view its value with the `-stats` option of `vstats`:

```
vstats -database db1 -stats “db_data_reads db1” \
“db_data_located db1”
```

This command will result in two columns of values being printed:

```
VERSANT Utility VSTATS Version 7.0.1.3
Copyright (c) 1989-2006 VERSANT Corporation
    0 = db_data_reads db1
    1 = db_data_located db1
    0         1
  =====     =====
   31       190
   35       200
   37       217
```

By default, a new line of values is printed every `5` seconds until you type `Ctrl-C` to break.

3. View statistics for a database connection

In the previous example, we viewed the per-database statistics `db_data_reads` and `db_data_located`.

For every connection, versions of these statistics, called `be_data_reads` and `be_data_located`, are also available.

To view per connection statistics, it is necessary to specify a connection identifier in addition to a database name. Connection identifiers are specified with the `-id` option of `vstats`.

To get information about all the current connections to a database, use the `-connections` option of `vstats`:

```
vstats -database db1 -connections
```

This command will cause a list of all connections to the specified database, to be printed. For example:

```
VERSANT Utility VSTATS Version 7.0.1.3
Copyright (c) 1989-2006 VERSANT Corporation
Connection ID to database 'db1':
Connection ID = 20334
User Name = 'george'
Session Name = 'example session'
Long Transaction = '0.0.0'
Server Process = 'gamehendge':20334
Client Process = 'gamehendge':20332
Protocol = TCP/IP
Server Port = 192.70.173.25:5019
Client Port = 192.70.173.25:1041
Connection ID = 20341
User Name = 'george'
Session Name = 'vstats -connections'
Long Transaction = '0.0.0'
Server Process = 'gamehendge':20341
Client Process = 'gamehendge':20339
Protocol = TCP/IP
Server Port = 192.70.173.25:5019
Client Port = 192.70.173.25:1043
```

From the resulting output, we can see that there are two connections. Connection `20334` is for an example program and connection `20341` is for the `vstats` command which we used to list the connections.

To view the `be_data_reads` and `be_data_located` statistics for the example program, we must do two things. First, we have to use the `-on` command for `vstats` to turn `ON` these statistics for the example program's connection:

```
vstats -database db1 -id 20334 \
-on be_data_reads be_data_located
```

Once the statistics are `ON`, we can use the `-stats` command to view them:

```
vstats -database db1 -id 20334 \
-stats “be_data_reads db1” “be_data_located db1”
```

As before, the `-stats` command will periodically print rows of statistic values.

For example:

```
VERSANT Utility VSTATS Version 7.0.1.3
Copyright (c) 1989-2006 VERSANT Corporation
    0 = be_data_reads db1
    1 = be_data_located db1
    0         1
  =====     =====
   23       123
   28       155
   32       188
```

4. Create statistics expressions

It is not necessary to limit yourself to the atomic statistics listed by the `vstats -summary` command. The `-stats` command understands standard arithmetic operations and a few built in functions.

For example, to print the page cache hit ratio for the example program:

```
vstats -database db1 -id 20334 \
-stats “delta be_data_located db1 \
/ (delta be_data_located db1 \
+ delta be_data_reads db1)”
```

This will result in output similar to the following:

```
VERSANT Utility VSTATS Version 7.0.1.3
Copyright (c) 1989-2006 VERSANT Corporation
    0 = delta be_data_located group
    / (delta be_data_located group
    + delta be_data_reads group)
    0
  =====
    -
   0.670
   NaN
   0.528
```

Note the use of the standard arithmetic `+`, `/`, and `()` operators.

Note also the use of the built in delta function which returns the difference between an expression's current value and it's value on the previous line. Since the delta function requires two lines of output before it can return a value, the first line of output for this expression is blank.

Finally, note the NaN ("not a number") value on the third line. This will happen when a divide by zero error occurs due to there being no change in either of the two atomic statistics. This is completely logical, since a read hit ratio is undefined over a time period where no reads occur.

Get Statistics with Automatic Profiling

The second method of accessing statistics also involves the vstats tool.

However, instead of taking input directly from the database, input is taken from a profile file generated by the application.

The advantages of automatic profiling are that it gives access to the application as well as the server statistics and that it provides information on individual database operations.

The disadvantage is that it can involve substantial performance overhead.

1. Create statistics profile file

The easiest way to generate a statistics profile file is by setting the VERSANT_STAT_FILE environment variable before running your application.

For example:

```
setenv VERSANT_STAT_FILE /tmp/app.vs
```

This will cause the application to automatically create the specified file and write statistics to it whenever certain database operations take place.

Although it is beyond the scope of this document, you should be aware that there are other environment variables, which can be set to limit the statistics, which are written to this file, thus increasing performance. By default, all application and server statistics are written to the file for all operations.

For more information on Environment Variables, please refer to the “Database Profiles” chapter, in the *Database Administration manual*.

Before running your application, be sure to turn ON any per database server statistics, which you are interested in viewing. Application statistics and per connection server statistics will be turned ON automatically.

As before, use the `vstats -on` command.

For example:

```
vstats -database db1 -on
```

In the above example, all statistics will be turned `ON` for all current connections, since no statistics or connection identifiers were specified.

Once the environment variables are set and the statistics are turned `ON`, simply run your application, and the profile file will be automatically generated.

2. View statistics from a profile file

You can examine the contents of the profile file with the `vstats` utility.

The syntax for examining statistics in a profile file is almost identical to the syntax used in the direct connection model. The only difference is that a file name is specified instead of a database name.

So for example, to show the change in `fe_real_time`, the `db_data_located` for `db1` and the `be_data_located` for `db2`, you could use the following command:

```
vstats -filename /tmp/app.vs -stats \
“delta fe_real_time” “db_data_located db1” \
“be_data_located db2”
```

Note that the database server statistics require database names, while the application statistic does not.

The above command will produce output similar to the following:

```
VERSANT Utility VSTATS Version 7.0.1.3
Copyright (c) 1989-2006 VERSANT Corporation
    0 = delta fe_real_time
    1 = db_data_located db1
    2 = be_data_located db2
    0         1         2
  =====     =====     =====
    -        59         3
    0.055    59         3      o_xact(db1,db2)
    0.216    60         4
```

```
    0.269    60          4       o_xact(db1,db2)
```

In the above output, there is an extra column, which is not present when using the direct connection model.

It provides information about what database operation that line of values is associated with. For each operation, there are two lines of output, one which shows values before the operation and one, which shows values after the operation.

To reduce screen clutter, only the second of these two lines is labelled with the operation. Note the list of database names provided along with each operation name. This shows for which databases server side statistics were collected and written to the profile file.

3. Create user defined collection points

It is possible that the statistics collection points built into the beginning and end of various low level database operations will prove to be inadequate. It is easy to specify additional points in your application where statistics should be written to the profile file.

Simply insert the "write statistics automatically" routine function into your application code wherever you would like statistics to be sampled and written to file.

If you would like to write statistics samples at the beginning and end of one of your application's operations, use the "write statistics on entering function" and "write statistics on exiting function" routines.

Get Statistics with Interface Routines

The third and final method for accessing statistics is through low level statistics routines for your interface.

Primitive flexibility are both the advantage and the disadvantage of this method.

To turn statistics ON, use the "turn statistics collection on" routine.

To turn statistics OFF, use the "turn statistics collection off" routine.

To get the current values of statistics, which are ON, use the "get statistics" routine.

To get a list of connection identifiers, which can be passed to these functions, use the "get activity information" routine.

STATISTICS COLLECTION AND STORAGE

Following are the ways you can collect performance and monitoring statistics about a Versant application and send them to a file.

Collect Statistics and Store them in a File

Set statistics collection ON and send to file

To collect any statistic for any database to which your application has made a connection, you can call "begin collection" routines from within the application. Statistics will be collected only for the activities of your own application and only for the duration of your session.

To collect any statistic for any database to which your application makes a connection, you can set the environment variable VERSANT_STAT_FILE to the name of a file and then run your application within its scope. Statistics will be collected only for the activities of your own application and only for the duration of your session.

You may want to filter the statistics collected to reduce file size and improve application performance. To filter the statistics collected and set other collection parameters, you can use the VERSANT_STAT_STATS, VERSANT_STAT_FUNCS, VERSANT_STAT_TIME, VERSANT_STAT_DBS and VERSANT_STAT_FLUSH environment variables.

For more information, on these variables, please refer to the chapter "Configuration Parameters" in the *Versant Database Administration Manual*.

Collecting statistics with environment variables is the same as invoking a "begin automatic collection" routine from within an application each time you make a database connection. Using a routine to begin automatic collection allows greater control, but it requires you to modify your application code.

To collect any statistic for any database and any application, you can invoke the vstats utility from the command line. Statistics will be collected only for connections in existence at the time vstats is invoked.

By default, the vstats utility sends statistics to stdout. To send statistics to a file, you can pipe the output of stdout.

For more information on vstats, please refer to the chapter “Database Utilities” in the *Versant Database Administration Manual*.

```
c     o_autostatsbegin()
env   VERSANT_STAT_FILE
util  vstats
```

Set statistics collection to off when using file

To stop collection started with a "begin automatic collection" routine, you can either end your session or use an explicit "end automatic collection routine."

```
c     o_autostatsend()
      o_endsession()
c++   endsession()
```

Collect statistics on function entry

If you use a "collect statistics in a file" routine or the VERSANT_STAT_FILE environment variable to collect statistics, you can use a "collect statistics on function entry" routine to collect and write statistics when entering a non-Versant function. If you use one of these routines, you must also use a "collect statistics on exiting" routine.

```
c     o_autostatsenter()
```

Collect statistics on function exit

If you use a "collect statistics in a file" routine or the VERSANT_STAT_FILE environment variable to collect statistics, you can use a "collect statistics on function exit" routine to both collect and write statistics when exiting a non-Versant function. If you use a "collect on entering" routine, you must also use a "collect on exiting" routine.

```
c     o_autostatsexit()
```

Collect statistics and write to file

A "collect and write" routine will collect the current set of statistics and write them to file along with a specified comment.

Use "collect on entering" and "collect on exiting" routines to add collection points at the beginning and end of your functions. Use "collect and write" any other time that you want to collect statistics.

```
c     o_autostatswrite()
```

View statistics in file

You can view statistics written to a file by using the `vstats` utility. (Else the text cannot be read when viewed with a text editor.)

With vstats utility, the text will be formatted and filtered per command line parameters. You can view the statistics file in real time by invoking `vstats` with the `-stdin` parameter, or you can view statistics at a later time by invoking `vstats` with the `-file` parameter.

```
util  vstats
```

Collect Statistics in Memory and View in Real Time

Following are the ways in which you can collect statistics and send them to the memory.

Set statistics collection `ON` and send to memory

To collect any database and/or connection statistic associated with a particular database, you can set the `stat` server process configuration parameter for that database.

After setting `stat`, the specified database statistics will be collected whenever the database starts, and specified connection statistics will be collected whenever any application connects to the database.

To collect any statistic for a particular database connection already in existence, you can invoke a "collect statistics" routine from within an application. Until the database stops and restarts, calling these methods will override any setting made in the `stat` server process configuration parameter that applies to the specified connection. Their effect on the specified connection will persist even after the end of the session in which they are called.

Invoking these methods has no effect on statistics collection separately being performed by `vstats`, `VERSANT_STAT_FILE`, or `beginAutoCollectionOfStatistics:`.

```
c     o_collectstats()
util  vstats
cfg   stat
```

Set statistics collection OFF when using memory

You can override `stat` by calling a general or specific "turn `OFF`" routine.

```
c     o_collectstats()
```

View statistics in the memory

To view any statistic being collected for any database and any application, in real time, you can specify a "direct connection" to one or more databases, when you invoke the `vstats` utility.

When you invoke `vstats`, you can ask for statistics by connection or database. Statistics are printed to `stdout`.

```
util  vstats
```

Get statistics from the memory

To view any statistic for a particular database and particular database connection, you can invoke a "get statistics" routine from within an application. Statistics are sent to an array from which you can fetch them.

Values are sent for all collected statistics at the moment when the routine is invoked.

```
c     o_getstats()
```

Get Statistics

The `vstats` utility and the "turn `ON` collection," "turn `OFF` collection", and "get statistics" routines allow you to Set or Get statistics for a specified database connection.

Get connection information, lock information, transaction information

Following are ways that you can get information about current connections, locks and transactions so that you can specify the required parameters.

```
c     o_getactivityinfo()
      o_gettransid()
c     gettransid()
util  vstats
dbtool
```

Get object information, Get volume information

The `dbtool` utility can also get information about objects and storage volumes associated with a particular database.

For more information on the dbtool utility, please refer to the Database Utilities Reference in the *Database Administration Manual*.

Derived Statistics

You can derive useful statistics using combinations of other statistics along with numerical operators and statistical functions.

To predefine derived statistics, you can create definitions in the statistics configuration file. When it starts, `vstats` will read and use the configuration file and compute the derived statistics.

To turn on the collection of derived statistics you have to use the individual statistic names instead.

For example:

The statistic `db_inst_cache_hit_ratio db` is defined as delta

`db_data_located db / (delta db_data_located db + delta db_data_reads db).`

To monitor this derived statistic you have to turn on the collection for `db_data_located` and `db_data_reads`.

The name of the configuration file depends upon your operating system.

```
For unix - .vstatsrc
For win  - vstats.ini
```

STATISTICS NAMES

Each statistic has a name and stores a numeric value. The symbolic names are used as parameters in functions and methods.

Following are statistics that you can collect and the names of functions for which you can collect statistics.

Function Statistics Names

You can use the following names to specify collection of statistics only for specific functions. For C++/Versant methods, specify the C/Versant function that performs the same task.

How to use in a function

When statistic names are to be used in a function:

- Enter the name in uppercase.
- Delete the "o_" prefix.
- Prefix the name shown with "CAPI_".

For example:

shown — `o_releaseobjs`

use — `CAPI_RELEASEOBJS`

How to use in `vstats` or an environment variable

When statistic names are specified to the vstats Statistics Tool or used in the environment variable `VERSANT_STAT_FUNCS`:

- Enter the name in lowercase (as shown.)
- Enter the name with the leading "o_" prefix (as shown.)

For example:

shown — `o_releaseobjs`

use — `o_releaseobjs`

Names of function statistics

Statistics Functions	Description
o_beginpinregion	Statistics for o_beginpinregion()
o_deleteobj	Statistics for o_deleteobj()
o_endactivity	Statistics for o_endactivity()
o_endpinregion	Statistics for o_endpinregion()
o_gdeleteobjs	Statistics for o_gdeleteobjs()
o_getcod	Statistics for o_getcod()
o_greadobjs	Statistics for o_greadobjs()
o_gwriteobjs	Statistics for o_gwriteobjs()
o_isclassobj	Statistics for o_isclassobj()
o_pathselect	Statistics for o_pathselect()
o_preptochange	Statistics for o_preptochange()
o_refreshobj	Statistics for o_refreshobj()
o_refreshobjs	Statistics for o_refreshobjs()
o_releaseobj	Statistics for o_releaseobj()
o_resetoct	Statistics for o_resetoct()
o_savepoint	Statistics for o_savepoint()
o_select	Statistics for o_select()
o_undosavepoint	Statistics for o_undosavepoint()
o_xact	Statistics for o_xact()
o_xactwithvstr	Statistics for o_xactwithvstr()
o_zapcods	Statistics for o_zapcods()

Process Statistics Names

The following statistics can be collected per application process.

How to use in a function

When statistic names are to be used in a function:

- Enter the name in uppercase.
- Prefix the name shown with "STAT_".

For example:

shown — fe_net_bytes_read

use — STAT_FE_NET_BYTES_READ

How to use in vstats or an environment variable

When statistic names are specified to the vstats Statistics Tool or used in the environment variable VERSANT_STAT_FUNCS:

- Enter the name in lowercase (as shown.)

For example:

shown — fe_net_bytes_read

use — fe_net_bytes_read

Names of process statistics

Statistics	Description
fe_cpu_time	Derived statistic. fe_cpu_time = fe_system_time + fe_user_time Not all operating systems support this statistic.
fe_net_bytes_read	Bytes read from back end.
fe_net_bytes_written	Bytes written to back end.
fe_net_read_time	Seconds reading from back end.

fe_net_reads	Reads from back end.
fe_net_write_time	Seconds writing to back end.
fe_net_writes	Writes to back end.
fe_reads	Objects read into object cache.
fe_real_time	Seconds of real time.
fe_run_time	Derived statistic. `fe_run_time = fe_real_time - fe_net_read_time - fe_net_write_time - fe_latch_wait_time`
fe_swapped	Dirty objects swapped out of object cache.
fe_swapped_dirty	Objects written as a result of object swapping.
fe_system_time	Seconds in OS kernel functions.
fe_user_time	Seconds not in OS kernel functions.
fe_vm_maj_faults	Virtual memory major page faults.
fe_writes	Objects written from object cache.

Session Statistics Names

How to use in a function

When statistic names are to be used in a function:

- Enter the name in uppercase.
- Prefix the name shown with "STAT_".

For example:

shown — `se_cache_free`

use — `STAT_SE_CACHE_FREE`

How to use in `vstats` or an environment variable

When statistic names are specified to the vstats Statistics Tool or used in the environment variable `VERSANT_STAT_FUNCS`:

- Enter the name in lowercase (as shown.)

For example:

shown — `se_cache_free`

use — `se_cache_free`

Names of session statistics

Statistics	Description
`se_cache_free`	Derived statistic. `se_cache_free = se_heap_free`
`se_cache_used`	Derived statistic. `se_cache_used = se_heap_used`
`se_cods`	CODs in object cache.
`se_heap_allocates`	Number of "allocate" operations smaller than one page.
`se_heap_small_allocates_0`	Allocates in size class 0 (1-8 bytes).
`se_heap_small_allocates_1`	Allocates in size class 1 (9-12 bytes).
`se_heap_small_allocates_2`	Allocates in size class 2 (13-16 bytes).
`se_heap_small_allocates_3`	Allocates in size class 3 (17-20 bytes).
`se_heap_small_allocates_4`	Allocates in size class 4 (21-24 bytes).
`se_heap_small_allocates_5`	Allocates in size class 5 (25-32 bytes).
`se_heap_small_allocates_6`	Allocates in size class 6 (33-40 bytes).
`se_heap_small_allocates_7`	Allocates in size class 7 (41-48 bytes).
`se_heap_small_allocates_8`	Allocates in size class 8 (49-64 bytes).
`se_heap_small_allocates_9`	Allocates in size class 9 (65-80 bytes).
`se_heap_small_allocates_10`	Allocates in size class 10 (81-96 bytes).
`se_heap_small_allocates_11`	Allocates in size class 11 (97-128 bytes).

se_heap_small_allocates_12	Allocates in size class 12 (129-160 bytes).
se_heap_small_allocates_13	Allocates in size class 13 (161-192 bytes).
se_heap_small_allocates_14	Allocates in size class 14 (193-256 bytes).
se_heap_small_allocates_15	Allocates in size class 15 (257-336 bytes).
se_heap_small_allocates_16	Allocates in size class 16 (337-408 bytes).
se_heap_small_allocates_17	Allocates in size class 17 (409-512 bytes).
se_heap_small_allocates_18	Allocates in size class 18 (513-680 bytes).
se_heap_small_allocates_19	Allocates in size class 19 (681-816 bytes).
se_heap_small_allocates_20	Allocates in size class 20 (817-1024 bytes).
se_heap_small_allocates_21	Allocates in size class 21 (1025-1360 bytes).
se_heap_small_allocates_22	Allocates in size class 22 (1361-1632 bytes).
se_heap_small_allocates_23	Allocates in size class 23 (1633-2048 bytes).
se_heap_small_allocates_24	Allocates in size class 24 (2049-2720 bytes).
se_heap_small_allocates_25	Allocates in size class 25 (2721-3072 bytes).
se_heap_small_allocates_26	Allocates in size class 26 (3073-4096 bytes).
se_heap_small_allocates_27	Allocates in size class 27 (4097-5120 bytes).
se_heap_small_allocates_28	Allocates in size class 28 (5121-6144 bytes).
se_heap_small_allocates_29	Allocates in size class 29 (6145-10240 bytes).
se_heap_small_allocates_30	Allocates in size class 30 (10241-12288 bytes).
se_heap_small_allocates_31	Allocates in size class 31 (12289-20480 bytes).
se_heap_empty_segments	Number of empty segments in front-end heap.
se_heap_free	Bytes free in front-end heap.
se_heap_frees	Number of "free" operations on front-end heap.
se_heap_small_frees_0	Number of frees in size class 0 (1-8 bytes).
se_heap_small_frees_1	Number of frees in size class 1 (9-12 bytes).
se_heap_small_frees_2	Number of frees in size class 2 (13-16 bytes).
se_heap_small_frees_3	Number of frees in size class 3 (17-20 bytes).
se_heap_small_frees_4	Number of frees in size class 4 (21-24 bytes).
se_heap_small_frees_5	Number of frees in size class 5 (25-32 bytes).
se_heap_small_frees_6	Number of frees in size class 6 (33-40 bytes).

se_heap_small_frees_7	Number of frees in size class 7 (41-48 bytes).
se_heap_small_frees_8	Number of frees in size class 8 (49-64 bytes).
se_heap_small_frees_9	Number of frees in size class 9 (65-80 bytes).
se_heap_small_frees_10	Number of frees in size class 10 (81-96 bytes).
se_heap_small_frees_11	Number of frees in size class 11 (97-128 bytes).
se_heap_small_frees_12	Number of frees in size class 12 (129-160 bytes).
se_heap_small_frees_13	Number of frees in size class 13 (161-192 bytes).
se_heap_small_frees_14	Number of frees in size class 14 (193-256 bytes).
se_heap_small_frees_15	Number of frees in size class 15 (257-336 bytes).
se_heap_small_frees_16	Number of frees in size class 16 (337-408 bytes).
se_heap_small_frees_17	Number of frees in size class 17 (409-512 bytes).
se_heap_small_frees_18	Number of frees in size class 18 (513-680 bytes).
se_heap_small_frees_19	Number of frees in size class 19 (681-816 bytes).
se_heap_small_frees_20	Number of frees in size class 20 (817-1024 bytes).
se_heap_small_frees_21	Number of frees in size class 21 (1025-1360 bytes).
se_heap_small_frees_22	Number of frees in size class 22 (1361-1632 bytes).
se_heap_small_frees_23	Number of frees in size class 23 (1633-2048 bytes).
se_heap_small_frees_24	Number of frees in size class 24 (2049-2720 bytes).
se_heap_small_frees_25	Number of frees in size class 25 (2721-3072 bytes).

se_heap_small_frees_26	Number of frees in size class 26 (3073-4096 bytes).
se_heap_small_frees_27	Number of frees in size class 27 (4097-5120 bytes).
se_heap_small_frees_28	Number of frees in size class 28 (5121-6144 bytes).
se_heap_small_frees_29	Number of frees in size class 29 (6145-10240 bytes).
se_heap_small_frees_30	Number of frees in size class 30 (10241-12288 bytes).
se_heap_small_frees_31	Number of frees in size class 31 (12289-20480 bytes).
se_heap_max_gap	Size of largest free area in front-end heap in bytes.
se_heap_optimistic_free	Derived statistic. `se_heap_optimistic_free = se_heap_free + se_heap_small_free`
se_heap_optimistic_used	Derived statistic. `se_heap_optimistic_used = se_heap_used - se_heap_small_free`
se_heap_population	Derived statistic. se_heap_population = se_heap_allocates - se_heap_frees
se_heap_size	Derived statistic. `se_heap_size = se_heap_used + se_heap_free`
se_heap_small_allocates	Number of "allocate" operations smaller than 1 page.
se_heap_small_free	Bytes free for allocations smaller than 1 page.
se_heap_small_frees	Number of "free" operations smaller than 1 page.

se_heap_small_population_n	Derived statistic. For n, substitute a number from 0 to 31, which refers to a class size state. See the reference to the heap_size_class application profile parameter for an explanation of class size states. se_heap_small_population = se_heap_small_allocates - se_heap_small_frees
se_heap_small_size	Derived statistic. se_heap_small_size = se_heap_small_used + se_heap_small_free
se_heap_small_used	Bytes used for allocations smaller than 1 page.
se_heap_total_segments	Number of segments in front-end heap.
se_heap_used	Bytes used in front-end heap.
se_net_bytes_read	Bytes read from back end.
se_net_bytes_written	Bytes written to back end.
se_net_read_time	Seconds reading from back end.
se_net_reads	Reads from back end.
se_net_write_time	Seconds writing to back end.
se_net_writes	Writes to back end.
se_objs	Objects in object cache.
se_objs_dirty	Dirty objects in cache.
se_reads	Objects read into object cache.
se_swapped	Objects swapped out of object cache.
se_writes	Objects written from object cache.
se_swapped_dirty	Objects written as a result of object swapping.

Connection Statistics Names

The following statistics can be collected per server connection. This makes it possible to connect to a database and view database process statistics for connections other than your own.

How to use in a function

When statistic names are to be used in a function:

- Enter the name in uppercase.
- Prefix the name shown with "STAT_".

For example:

shown — `be_data_located`

use — `STAT_BE_DATA_LOCATED`

How to use in `vstats` or an environment variable

When statistic names are specified to the vstats Statistics Tool or used in the environment variable `VERSANT_STAT_FUNCS`:

- Enter the name in lowercase (as shown.)

For example:

shown — `be_data_located`

use — `be_data_located`

Names of server connection statistics

Statistics	Description
be_cache_hit_ratio db	Derived statistic, where db is the name of a database. be_cache_hit_ratio db = be_data_located db / (be_data_located db + be_data_reads db)
be_cpu_time db	Derived statistic, where db is the name of a database. be_cpu_time db = be_system_time db + be_user_time db Not all operating systems support this statistic.
be_data_located	Pages found in database page cache.
be_data_reads	Pages read from sysvol + added volumes.
be_data_writes	Pages written to sysvol + added volumes.
be_ev_defined	Events defined.
be_ev_sys_delivered	System events delivered.
be_ev_sys_raised	System events raised.
be_ev_user_delivered	User events delivered.
be_ev_user_raised	User events raised.
be_inst_cache_hit_ratio db	Derived statistic, where db is the name of a database. be_inst_cache_hit_ratio db = delta be_data_located db / (delta be_data_located db + delta be_data_reads db)
be_lock_deadlocks	Deadlocks occurred.
be_lock_timeouts	Timeouts waiting for locks.
be_lock_wait_time	Seconds clients spent waiting for locks.
be_lock_waits	Lock waits which occurred.
be_locks_granted	Locks requested and granted.
be_net_bytes_read	Bytes read from front end.

be_net_bytes_written	Bytes written to front end.
be_net_read_time	Seconds reading from front end.
be_net_reads	Reads from front end.
be_net_rpcs	Database RPCs received from clients.
be_net_write_time	Seconds writing to front end.
be_net_writes	Writes to front end.
be_obe_locks_waiting	1 if waiting for lock, 0 otherwise.
be_obj_received	Objects received from front end.
be_obj_sent	Objects sent to front end.
be_qry_btree_objs	Objects read during B-tree query.
be_qry_btree_time	Seconds spent in B-tree query.
be_qry_hash_objs	Objects read during hash query.
be_qry_hash_time	Seconds spent in hash query.
be_qry_scan_objs	Objects read during sequential scan query.
be_qry_scan_time	Seconds spent in sequential scan query.
be_real_time	Seconds elapsed.
be_run_time db	Derived statistic, where db is the name of a database. `be_run_time db = be_real_time db - be_net_read_time db - be_net_write_time db - be_latch_wait_time db`
be_system_time	Seconds in OS kernel functions.
be_user_time	Seconds not in OS kernel functions.
be_vm_maj_faults	Virtual memory major page faults.
be_xact_active	Active transactions.
be_xact_committed	Transactions committed.
be_xact_rolled_back	Transactions rolled back.
be_xact_started	Transactions started.

Database Statistics Names

The following statistics can be collected per database.

How to use in a function

When statistic names are to be used in a function:

- Enter the name in uppercase.
- Prefix the name shown with "STAT_".

For example:

shown — `db_bf_llog_flushes`

use — `STAT_DB_BF_LLOG_FLUSHES`

How to use in `vstats` or an environment variable

When statistic names are specified to the vstats Statistics Tool or used in the environment variable VERSANT_STAT_FUNCS:

- Enter the name in lowercase (as shown.)

For example:

shown — `db_bf_llog_flushes`

use — `db_bf_llog_flushes`

Names of database statistics

Statistics	Description
`db_bf_llog_bytes_written`	Bytes written to logical log.
`db_bf_llog_flushes`	Number of writes to logical log.
`db_bf_plog_bytes_written`	Bytes written to physical log
`db_bf_plog_flushes`	Number of writes to physical log.

db_cache_hit_ratio db	Derived statistic. db_cache_hit_ratio db = db_data_located db / (db_data_located db + db_data_reads db)
db_checkpoints	System checkpoints.
db_data_located	Pages found in database page cache.
db_data_reads	Pages read from sysvol + added volumes.
db_data_writes	Pages written to sysvol + added volumes.
db_disk_free	Bytes of storage available.
db_disk_reserved	Bytes of storage reserved by classes.
db_ev_defined	Events defined.
db_ev_sys_delivered	System events delivered.
db_ev_sys_raised	System events raised.
db_ev_user_delivered	User events delivered.
db_ev_user_raised	User events raised.
db_inst_cache_hit_ratio db	Derived statistic. db_inst_cache_hit_ratio db = delta db_data_located db / (delta db_data_located db + delta db_data_reads db)
db_lock_deadlocks	Deadlocks occurred.
db_lock_timeouts	Timeouts waiting for locks.
db_lock_wait_time	Seconds clients spent waiting for locks.
db_lock_waits	Lock waits which occurred.
db_locks_granted	Locks requested and granted.
db_net_bytes_read	Bytes read from front end.
db_net_bytes_written	Bytes written to front end.
db_net_read_time	Seconds reading from front end.
db_net_reads	Reads from front end.
db_net_rpcs	Database RPCs received from clients.
db_net_write_time	Seconds writing to front end.
db_net_writes	Writes to front end.

db_obe_locks_waiting	Connections waiting for locks.
db_obj_received	Objects received from front end.
db_obj_sent	Objects sent to front end.
db_qry_btree_objs	Objects read during B-tree query.
db_qry_btree_time	Seconds spent in B-tree query.
db_qry_hash_objs	Objects read during hash query.
db_qry_hash_time	Seconds spent in hash query.
db_qry_scan_objs	Objects read during sequential scan query.
db_qry_scan_time	Seconds spent in sequential scan query.
db_xact_active	Active transactions.
db_xact_committed	Transactions committed.
db_xact_rolled_back	Transactions rolled back.
db_xact_started	Transactions started.
db_at_root_located	Associative table root pages found in database page cache
db_at_root_read	Associative table root pages read from sysvol and added volumes.
db_at_leaf_located	Associative table leaf pages found in database page cache
db_at_leaf_located	Associative table leaf pages read from sysvol and added volumes.

Latch Statistics Names

There are four basic types of latch statistics:

- `latch_released`
- `latch_granted`
- `latch_waits`
- `latch_wait_time`

For each of these, there are both per database (`db_*`) and per connection (`be_*`) versions of each statistic.

In addition, each statistic is available for all latches or for an individual type of latch (e.g. `*_sda`, `*_heap`, `*_voldev`, etc.).

The exception to this is the latch_released statistic, which is not available for the individual latch types.

Also, there are two derived statistics: `be_latch_holds` and `db_latch_holds`. These are defined to be the number of latches granted minus the number released (`latch_granted - latch_released`) and give an indication of how many latches are currently being held. The values of these statistics are not absolute, but are relative to the number of latches held when the statistic is turned `ON`. The value will be `0` if the same number of latches, positive if more, and negative if less.

How to use in a function

When statistic names are to be used in a function:

- Enter the name in uppercase.
- Prefix the name shown with "`STAT_`".

For example:

shown — `be_latch_granted`

use — `STAT_BE_LATCH_GRANTED`

How to use in `vstats` or an environment variable

When statistic names are specified to the vstats Statistics Tool or used in the environment variable `VERSANT_STAT_FUNCS`:

- Enter the name in lowercase (as shown.)

For example:

shown — `be_latch_granted`

use — `be_latch_granted`

Names of latch statistics

Statistics	Description
`be_latch_granted`	Number of latches granted of any type.
`be_latch_granted_bf`	Number of BF latches granted.
`be_latch_granted_bf_bkt`	Number of BF_BKT latches granted.
`be_latch_granted_bf_dirty`	Number of BF_DIRTY latches granted.
`be_latch_granted_bf_free`	Number of BF_FREE latches granted.
`be_latch_granted_cp`	Number of CP latches granted.
`be_latch_granted_cp_wait`	Number of CP_WAIT latches granted.
`be_latch_granted_ev`	Number of EV latches granted.
`be_latch_granted_heap`	Number of HEAP latches granted.
`be_latch_granted_l2file`	Number of L2FILE latches granted.
`be_latch_granted_l2file_da`	Number of L2FILE_DA latches granted.
`be_latch_granted_llog`	Number of LLOG latches granted.
`be_latch_granted_lock`	Number of LOCK latches granted.
`be_latch_granted_log_unit`	Number of LOG_UNIT latches granted.
`be_latch_granted_phy`	Number of PHY latches granted.
`be_latch_granted_plog`	Number of PLOG latches granted.
`be_latch_granted_ps`	Number of PS latches granted.
`be_latch_granted_sc`	Number of SC latches granted.
`be_latch_granted_sce`	Number of SCE latches granted.
`be_latch_granted_sch`	Number of SCH latches granted.
`be_latch_granted_sd`	Number of SD latches granted.
`be_latch_granted_sda`	Number of SDA latches granted.

STATISTICS OPERATIONS

Following are some regular statistical operations which can be performed and the functions to be used to find out that particular statistics:

Here, `db` denotes the database name.

See the previous section, "Statistics Names," for the formula used by derived statistics.

To find out which operations to delve into

`fe_real_time`	Seconds of real time.

To find out whether an operation is application bound or database bound

`fe_run_time`	A derived statistic.
`be_run_time db`	A derived statistic.

To see if the CPU is the bottle neck

`fe_cpu_time`	A derived statistic.
`be_cpu_time db`	A derived statistic.

Not all operating systems support these statistics.

To see if there is virtual memory thrashing

`fe_vm_maj_faults`	Virtual memory major page faults.
`be_vm_maj_faults`	Virtual memory major page faults.

Not all operating systems support these statistics.

To see if there is lock contention

be_lock_waits	Lock waits which occurred.
db_lock_waits	Lock waits which occurred.
be_lock_wait_time	Seconds spent waiting for locks.
db_lock_wait_time	Seconds spent waiting for locks.

To see if there is latch contention

be_latch_waits	Number of waits for any latch.
db_latch_waits	Number of waits for any latch.
be_latch_wait_time	Seconds waiting for any latch.
db_latch_wait_time	Seconds waiting for any latch.

To see if there is an opportunity for group operations

be_net_rpsc	Database rpc's received from clients.
db_net_rpsc	Database rpc's received from clients.
fe_reads	Objects read into the object cache.
se_reads	Objects read into the object cache.
be_obj_sent	Objects sent to the application.
db_obj_sent	Objects sent to the application.
be_obj_received	Objects received from the application.
db_obj_received	Objects received from the application.

To see if the disk is the bottleneck

be_data_reads	Pages read from sysvol + added volumes.
db_data_reads	Pages read from sysvol + added volumes
be_data_writes	Pages written to sysvol + added volumes.
db_data_writes	Pages written to sysvol + added volumes.
db_bf_llog_bytes_written	Bytes written to logical log.
db_bf_plog_bytes_written	Bytes written to physical log.

To tune logging

db_checkpoints	System checkpoints.
db_bf_llog_flushes	Number of writes to logical log.
db_bf_plog_flushes	Number of writes to physical log.
db_bf_llog_bytes_written	Bytes written to logical log.
db_bf_plog_bytes_written	Bytes written to physical log.

To see if cached object descriptors need to be zapped or if objects need to be released

se_cods	CODs in object cache.
se_objs	Objects in object cache.

To see Associate Table page operations

db_at_root_located	Associative table root pages found in database page cache
db_at_root_read	Associative table root pages read from sysvol and added volumes.
db_at_leaf_located	Associative table leaf pages found in database page cache
db_at_leaf_located	Associative table leaf pages read from sysvol and added volumes

STATISTICS USAGE NOTES

General Notes

Turning Statistics collection ON/OFF

Only the user who created a database can turn statistics collection ON/OFF. After statistics collection has been enabled, other users can collect the statistics.

Possible Overhead

When statistics collection is turned ON, the profiling overhead will be two remote procedure calls and, possibly a potential file write each time a collection point is encountered. Since this could be a significant overhead, you can limit collection to specific databases and/or connections and change parameters for collection and write intervals.

Statistics Collection and the Fault Tolerant Server

Even when database replication has been turned ON, statistics collection tools and mechanisms operate only on named databases and not on replica databases.

To collect statistics for a replica database, you must apply statistics collection mechanisms specifically to the replica database.

Procedure for Collecting Statistics from an Application

To collect statistics from within an application,

1. First, turn ON statistics collection with the `turnOnCollectionOfStatistics` method in `VStatistics`.

   You can turn OFF collection of some or all statistics by using the `turnOffCollectionOfStatistics` method.

2. To retrieve statistics from within an application, use the `getStatisticsOf` method in `VStatistics`.
3. To turn `ON` automatic statistics collection, use the `beginAutoCollectionOfStatistics` method in `VStatistics`.

   To turn `OFF` automatic statistics collection, use the `endAutoCollectionOfStatistics` method in `VStatistics`.
4. To add collection points for your own methods, use the `autoStatsEnteringFunction` and `autoStatsExitingFunction` methods in `VStatistics`.
5. To find information about locks, transactions, and connections, use the `getActivityInfoOn` method in `VStatistics`.
   Support classes for the `getActivityInfoOn` method are the classes `VLockInfo, VXactInfo` and `VConnectInfo`.

For more information, refer the Reference manuals for usage of specific functions, methods, utilities and configuration parameters.

CHAPTER 11

Performance Tuning and Optimization

This Chapter describes various methods of improving the performance of the Versant Database.

Following topics are covered:

- Performance Statistics
- Data Modeling
- Memory Management
- Disk Management
- Message Management
- Multiple Applications
- Application Programming

BY STATISTICS COLLECTION

For best performance, you need to fine tune an application. If you know where it is spending time, you can reduce the same and increase its performance.

Following are some ways of collecting performance statistics:

Add a statistics collection routine to your application

In your program, make a call to a "write statistics" routine after the point where you want to gather statistics. Depending upon your interface, this routine is `o_autostatswrite()`, `autostatswrite()` or `autoStatsWriteOnDBs`:.

Disable low level collection points

To avoid collecting unwanted statistics about Versant routines, disable the low level collection points built into Versant by setting the `VERSANT_STAT_FUNCS` environment variable to an empty string.

Specify the statistics to be collected

Specify the statistics to be collected by setting the `VERSANT_STAT_STATS` environment variable.

For more information, refer to “Statistics Quick Start for Performance Monitoring” on page 218 in "Chapter 10 - Statistics Collection".

Commonly used statistics are:

- `fe_real_time` — Seconds elapsed since the fe_real_time statistic was turned on.
- `be_data_located` — The number of pages read from the server cache, including data, table, and index pages, but not log pages. This statistic is a measure of "cache hits," which are a lot less expensive than disk reads.
- `be_data_reads` — The number of pages not found in the server cache, including data, table, and index pages, but not log pages. This statistic is a measure of "cache misses," in which needed information had to be read from disk.
- `be_net_write_time` — Seconds spent sending data from the server process to the application. This statistic is expensive to collect.

Set statistics collection ON and send statistics to a file

Tell Versant to collect statistics by setting the environment variable VERSANT_STAT_FILE to the name of a file.

When doing initial performance tuning during development, writing statistics to file is usually preferable to viewing with a direct connection, because writing to file shows statistics for an application's operations rather than for time slices. Writing to file also allows you to preserve statistics so that they can be looked at in different ways.

Run your application and collect statistics

Run your application within the scope of the above environment variables and collect statistics for the activities of your application.

View statistics with the vstats utility

View your statistics with the vstats utility by invoking it as:

```
vstats -filename filename -stats "delta fe_real_time" "delta
be_data_reads db1"  "delta be_data_located db1"
```

where filename is the name of the file specified in VERSANT_STAT_FILE.

When you view your statistics, you should see some very useful information about where your application is spending its time.

For example, consider the following sample output.

```
VERSANT Utility VSTATS Version 7.0.1.3
Copyright (c) 1989-2006 VERSANT Corporation
    0 = delta fe_real_time
    1 = delta be_data_reads db1
    2 = delta be_data_located db1
    0         1         2
  =====     =====     =====
    -         -         -
  0.008       2         0      op2 on obj 130476
  1.873     132         2      op1 on obj 20384
  0.061       1         2      op4 on obj 710912
```

In the above, column `0` for `fe_real_time` shows which operations are taking the most time, and column `1` shows the time spent reading data.

In this case, the application is spending most of its time reading data pages during operation #1 on object #20384. With this knowledge, you can now investigate what is special about this particular object and operation. In this case, you might also want to immediately check the setting of the database profile parameter `max_page_buffs` (which sets the maximum number of pages used by the server process page buffer) and increase it.

Use a third party profiler

Another useful tool, particularly for tuning application code, is a profiler such as Pure Software's Quantify or the standard `prof` and `gprof`.

By running a profiled version of the application, you can see how much each function contributes to the total time which it took the application to run. Just seeing what percentage various Versant operations contribute to the total cost of a user operation can be worth the effort of using a profiler.

For more information on collecting statistics, refer to chapter “Statistics Collection” on page 215.

BY DATA MODELING

If your application is spending a lot of time reading data and/or passing data across a network, consider the following suggestions related to your data model.

Consider how data is used

During the design stage, you might want to use a "pure" object model to describe your situation. During the development stage, you may need to modify your object model to reflect how data is used rather than how it is structured.

The process of structuring data in terms of how it is used is sometimes called "finding your real objects." To find "real objects," walk your model with your tasks in mind, consider how your data is actually created, found, and updated and make adjustments based upon the real world of disk reads and network traffic.

Adjusting a model to reflect data use can have a major impact on performance, because pure object models seldom consider the physical limitations of hardware. For example, if you have tiny objects everywhere, you are going to pay large performance costs. Your data model will have minuscule performance implications once all needed objects are in memory (where they can be accessed in microseconds,) but it may have considerable impact on disk reads and network traffic.

Although you may look for "real objects" with performance in mind, almost certainly you will make changes that will also improve the reusability, extensibility, and scalability of your model. It is almost always a good idea to base your model on data usage rather than data structure.

Combine objects that are accessed together

If you look at your data model and see lots of tiny objects, consider combining the small objects into a larger object. Reading a single large object is cheaper than reading two smaller objects, because it reduces the number of disk find and network message operations required.

For example, an employee object could logically either embed a name or contain a link to a separate name object. In this case, if you will need the employee name whenever you need the employee object, it would be faster to embed the name.

In C and C++, you can combine objects with embedding.

In general, if two objects are always accessed together, consider combining them into a single object in order to improve access speed.

Use links to minimize embedding repetitious data

It is certainly possible to overdo embedding. If you look at your data model and see repetitious data in your objects, consider using links.

For example, if each employee has a department, you would not want to embed the same information about a department in each employee object associated with that department. (It would take too much disk space, be costly to change information about the department, and you may not need to look at department information each time you look at an employee.) In this case, it would be better to relate an employee to a department with a link.

Use links to minimize queries

If you walk your data model and see the need for queries to assemble needed information about a root object, then you should stop and re-evaluate your data model.

It is much faster to find an object with a link than to find an object with a query. By definition, a query traverses a database to find and evaluate a group of starting point objects, while a link contains all the information needed to go directly to a particular object, regardless of its current location (on disk or in memory.)

In general, once you have a root object, you should be able to follow links to all information related to that object. To do this, you need to build data relationships into your data model.

There is virtually no penalty for adding links to an object, even if they are rarely used. This is because a link takes negligible storage space (about 8 bytes per link on disk) and thus does not appreciably slow the retrieval of a root object.

Use direct links if possible

While finding objects with links is fast, it is possible to misuse links.

For example, suppose that several objects need to reference a particular object, but you want to be able to change the object to which they point. One way to model this is to have each object contain a link to a generic object that then points to the desired target object. Another way to structure this is for each object to contain an identical link to the desired object. If you use a generic object approach, it will be easier to change the references, but you will pay a cost each time you need to travel to the target object, because separate fetches are involved.

In general, consider creating data structures that have a root object containing direct links to all other objects needed. You can then read all your objects with two messages to the server: one to find and return the root object and a second to group read all the other objects.

Use shallow link structures if possible

As cheap as links are, it is not always possible to directly link root objects with target objects. Instead, you may need to create a graph or tree structure.

For example, suppose that a person has garages that can contain cars that have colors. If the only question you will ever ask is what colors belong to a person, then you could create a direct link from a person to a color, or even embed color literals in the person object. However, if at some other time you also need to know which garages have which cars, then you will want to create a tree structure.

In general, shallow, wide trees are better than deep trees, because they minimize the number of fetches required to move from a root object to a leaf object. Wide trees are created with link arrays. At each step downwards, you "cold traverse" a database to get an array of links, bring the array into memory, "hot traverse" the array in memory to get the next array object, and then cold traverse again to move to the next level. Wide trees are practical, because link arrays take relatively small amounts of memory (about 8 bytes per link on disk.)

Place access information in your root objects

Suppose that you have an object with links to many other objects. You are only really interested in one of these other objects, but in order to determine which one, you have to consult information in each of them. In this case, consider placing the decision making information in the initial object. This can be as simple as keeping a list of links sorted so you can just pick the first one, or it might involve using a dictionary that allows you to go directly to the object that you want.

Use bi-directional links with care

In a multi-user environment, avoid implementing inverse relationships with bi-directional links without understanding concurrency issues.

BY MEMORY MANAGEMENT

The applications performance can be improved greatly by doing some memory management tricks.

Memory Management Strategies

The basic memory management strategies are -

- To improve access times, get the objects you want into memory as fast as possible, and then keep them there until they are no longer needed.
- Let the objects which are no longer needed go out of memory.
- Balance your use of client and server resources.

Implementing Concepts

Following are concepts that are necessary while implementing the memory management strategies:

Memory caches

The following are the Memory caches:

Object cache

To improve access to objects used in a transaction, an object cache is created on the machine, which is running the application. An object cache will expand as needed and use both real and virtual memory.

Object cache table

To improve access to objects used in a session, a cached object descriptor table is created on the machine running the application. It contains information about the current location (memory or disk, application machine or database machine) of all objects referenced during a session.

Session tables

Various session information tables track your processes, connected databases, and transactions. These tables are created on the machine which is running the application.

Server page cache

To improve access to objects by all users of a database, a server page cache is created in shared memory on the machine containing the database. The page cache will have a fixed size and can use both real and virtual memory.

Server page cache tables

Associated with the server page cache are information tables that map objects to their physical locations, contain index information, and maintain other information about the state of the database.

Object locations

You can improve application performance dramatically by understanding the location (place) of an object and taking steps to control its location.

At any particular time, an object is usually in one of the following places:

- On the disk, on the database machine.
- In a server page cache, on the database machine.

  In an object cache, on the application machine.

Pinning behavior

If you have pinned an object, then it is in real or virtual memory of the application machine. If you try to pin more objects than can fit into real and virtual object cache memory, you will get an error.

If you unpin an object after it has been pinned, then it can be in any of several places. If ample space remains in the object cache, it will probably remain there.

If the object cache fills up and the object is clean, then it will be dropped from memory.

If the object cache fills up and the object is new or dirty, then it will be flushed to the machine containing its source database.

C++/Versant automatically pins objects whenever they are dereferenced. C/Versant does not automatically pin objects.

Hot, warm and cold traversals

A traversal is a navigation through a group of related objects with the purpose of accomplishing some task.

A "hot" traversal finds an object already in application cache memory (real or virtual.) Hot traversals are very fast.

A "warm" traversal finds an object that is not in application cache memory, but that is in server page cache memory.

A "cold" traversal finds an object in a database on disk. A cold traversal is slower than a warm or hot traversal, because more needs to be done to find the object. For example, if a database containing an object is remote, a cold traversal sends a request across the network to the database, and then the database server process finds the object, brings the data page containing the object into page cache memory, and sends the object back across the network to the application.

Tips for Better Memory Management

Object Cache Usage

The following object cache statistics are specially useful in managing the memory for better performance.

- `fe_vm_maj_faults` — Virtual memory major page faults (not supported by some operating systems)
- `se_cache_free` — Bytes free in the application heap, which include the object cache, cached object descriptor table and internal data structures. The heap grows when the number of free bytes run out.
- `se_cache_used` — Bytes used in the application heap, which include the object cache, cached object descriptor table and internal data structures.
- `se_objs` — Number of objects in the object cache.
- `se_objs_dirty` — Number of dirty objects in the object cache.
- `se_reads` — Number of objects read into the object cache.

- se_swapped — Number of objects swapped out of the object cache.
- se_writes — Number of objects written from the object cache.

Once you have determined your model for object cache usage, you should examine these statistics and make sure things are really working the way you intended.

Creating a realistic test environment

During development, test databases are often small, which can obscure issues that will appear when you deploy to a large database and cannot increase cache sizes proportionally. For example, operating systems will have their own layer of caching which you cannot turn OFF.

According, during development, consider using a raw device for your database system volumes, which will bypass operating system caching. This simulates the fact that the chances of getting a cache hit in a very large database is by definition, low.

When measuring scalability, it is important to avoid comparing a hot case with a cold case.

Using commit and rollback routines

To maintain the object cache

By default, the object cache is flushed whenever a transaction is committed or rolled back. Since this is often undesirable, several options are provided for retaining objects between transactions.

In C and C++, these options are used with the o_xact() and xact() routines.

- O_COMMIT_AND_RETAIN — Commit new and dirty objects and keep all objects in the object cache.
- O_CHECKPOINT_COMMIT — Commit new and dirty objects, keep all objects in the object cache, and retain locks.
- O_ROLLBACK_AND_RETAIN — Rollback all changes and keep all clean objects in the object cache.
- O_RETAIN_SCHEMA_OBJ — Retain schema objects in the object cache after the transaction ends. (This will improve performance the next time you get or select instance objects of the classes involved.)

On objects in an array

For even a finer control over what objects are affected by ending a transaction, consider using a commit routine that operates on objects in a `vstr` or `collection` rather than on the entire object cache.

In C and C++, the routines are `o_xactwithvstr()` and `xactwithvstr()`.

For example, suppose that you want to commit some set of objects and retain some other set.

Following is one way to do this:

1. Create a simple collection, such as a link vstr of objects to commit. Call it, say, "commitset". (If you want to commit all dirty objects, in C or C++, you can easily construct a set of modified objects by using a Versant routine that iterates over cached object descriptors marked dirty.)
2. Create a set of objects to retain after the commit. A set is needed if the number of objects is large. Call it, say, "retainset".
3. Perform a checkpoint commit on the objects in your commitset using `o_xactvstr()`, `xactwithvstr()`, or `checkpointCommit` in `SequenceableCollection`. This will commit all of the objects in `commitvstr` and retain all objects and locks.
4. Iterate through all of the objects in the cached object descriptor table and check each one. You should retain all class objects and all those in your `retainset`. Make a set, say "releaseset", from all other objects.
5. Use a "release objects" routine on the objects in your `releaseset`. (Use `o_releaseobjs(), releaseobjs(), or releaseAndSkipDirtyObject()`.)

Using resources appropriately

If you are dealing with more objects than can fit into object cache memory, what you should do depends upon your hardware and network resources.

Your first step should be to subdivide your operations.
Your basic strategy should be to keep all objects needed for a particular operation in object cache memory on your application machine. You do this by pinning needed objects and unpinning unneeded objects. Your rule of thumb should be to pin no more objects than those that will fit into real memory.

Your next step should be to decide what you want to do with the temporarily unneeded objects.

If your server connection is cheap and/or you are accessing objects in random order, using server resources will probably outperform using virtual memory. To encourage unneeded objects to be dropped or flushed, set the `swap_threshold` parameter in your application profile

to be slightly smaller than the size of your real memory (you may need to experiment to get it right.) Generally, setting `swap_threshold` is enough, but in drastic cases, you can force unneeded objects to be dropped by releasing them.

If your connection to the server is expensive and objects are accessed in sequential passes, then virtual memory may work best. To encourage unneeded objects to remain on the application machine, choose a high value for the `swap_threshold` parameter. In this case, Versant will expand and use virtual memory until resources are no longer available.

In most cases, the balancing of the use of client versus server resources needs to be performed empirically. (Of course, the best solution is to get more memory for your application machine.)

As a rule of thumb, you should use the object cache for both the current working set of objects and any frequently used objects. If recently accessed objects are likely to be accessed again, it may make sense to cache them as well.

If memory constraints prevent you from caching an object, it may still be useful to cache a link to it. Caching a link costs almost nothing. A good way to do this is to replace a query on the server with a dictionary lookup held in memory on the application machine. With a four-byte key, this only costs you a megabyte for each 43,690 objects. In this case, it is not a big deal to use virtual memory and allow the dictionary to be swapped out.

Keeping object cache clean

Keeping objects out of the object memory cache is also as important as keeping objects in the cache. If your object cache grows too big, it will trigger virtual memory page faults, which can dramatically decrease performance.

Keeping your cache clean is a very common issue. If you don't keep your cache clean, you will initially get good performance and then start to see bad performance when real memory runs out and you begin to thrash virtual memory.

To determine whether page faults are occurring, monitor the `fe_vm_maj_faults` statistic. You can also monitor `se_cache_used` and `se_objs` to determine when real memory is exhausted.

If you run out of real and virtual memory in the application process heap, you will see error number `4161, OM_HEAP_NOMEM, "Out of frontend heap memory"`.

Following are ways to keep your object cache clean:

- Unpin unneeded objects

  Unpin objects that are not needed. This will allow Versant to drop or flush unneeded objects, although it does not guarantee that the objects will be removed from memory.

- Release unneeded objects

  To force Versant to drop unneeded objects immediately, release them. However, be sure that you do not release a dirty object or a class object.

- Zap unneeded objects

  If you reference a very large number (say, a million or more) of objects in a session, the cached object descriptor (cod) table, which maintains information about all objects accessed in the session, may grow to a significant size. To both drop an object and clear its cod table entry, you can zap it with a "zap cods" interface routine. If you need to reclaim space in the application process heap and have not been zapping objects as you go, you can commit or rollback a transaction with a "zap cods" option that will both clear the entire table and drop all objects from cache memory.

  If you zap objects, be sure that you do not zap a dirty object, an object that is the target of a link in a dirty object, or a class object.

Setting server page cache size appropriately

It is faster for a database server process, to retrieve an object from a database than from virtual memory, because a server process has access to information about object locations and can be smarter about caching.

This means that you should always set the size of the server page cache to be as large as possible but still smaller than available real memory.

The size of the server page cache is set by the value of the `max_page_buffs` parameter in the server process profile for a database, where each page specified in `max_page_buffs` equals `16K` of memory.

If you are also simultaneously running other programs on the database machine, remember to allow for their needs, because the page cache is a fixed size. Also, remember to allow for the server page cache support tables, which have a major impact on performance.

For example, even if you have a database which is many orders of magnitude larger than your physical memory, you may still get a major improvement in performance by increasing your server cache size if it means that the support tables can remain in real memory.

Statistics for the server page cache

The statistics which are most relevant to the server page cache are `be_data_reads`, `be_data_located`, `be_data_writes` and `be_vm_maj_faults`.

If your server cache is working well, the values for `be_data_reads` and `be_data_writes` will be low, while `be_data_located` is likely to be high.

If values for `be_vm_maj_faults` starts increasing, this means that there is not enough physical memory. This could be because the `max_page_buffs` is set higher than physical memory or it could mean that another application on that machine is using more memory than you anticipated.

Running with a single process

If your application will be accessing only a single database that is located on the same machine as the application, consider linking your application with the single process library.

Running your application in a single process can improve performance dramatically (by 20-50%.)

However, if you run in a single process, it is possible for a runaway pointer bug in your application code, to corrupt data structures and crash the database.

BY DISK MANAGEMENT

Improving the disk input and output can increase the performance dramatically.

Particularly important can be your storage strategy which includes clustering. Indexes can also significantly improve disk access by queries.

For more information, refer to “Query Indexing” on page 425 in "Chapter 18 - Versant Queries".

Disk Management Suggestions

Some of the disk management strategies for better performance are given below:

Use faster, better and more disk devices

The faster the disk drive on which a database resides, the better.

Multiple disk drives can also increase performance. Ideally, the physical log, the logical log, and the system volumes should all be on different physical devices.

It is better to use raw devices for a database than operating system files. They are 5% to 10% faster, and they are more predictable, because they do not implement additional buffering outside of Versant.

In general, consider all available network and storage resources available to you and then place your databases where they can best take advantage of your resources.

Gather statistics about disk usage

Storing some related objects, which are physically close to each other can greatly improve the speed of accessing them together.

If two related pieces of information are stored on the same disk page, accessing one will bring both into the backend cache in a single operation.

While tuning clustering, it is generally a good idea to temporarily set the database `max_page_buffs` profile parameter, as low as possible. However, do not set it so low that the

pages necessary for a single operation don't fit in the cache and thrashing ensues, which would distort your statistics.

Using a small cache while tuning, minimizes the possibility of cache hits and allows you to see the true number of reads for each operation. You can then look at the `be_data_reads` statistic and try to get it as low as possible via clustering.

Use the following formula to determine a rough goal for the `avg delta be_data_reads` statistic:

```
(number of objects)(average object size + 20)/16K +
(number of objects)/809 + 2
```

Keep in mind that you may not be able to achieve this goal, particularly if multiple access patterns force you to trade off each specific case in order to optimize the average case.

In any event, without a cache hit you can never hope for better than two reads, one for the data page and one for the object location table page. Once you have tuned clustering to your satisfaction, don't forget to restore the `max_page_buffs` parameter to its original state.

Do not use NFS file systems

If you use an NFS partition to mount and access your databases, performance may suffer. Versant does not need NFS to access databases.

Cluster Classes

The Cluster classes can be used in following suggested ways to improve the performance.

Cluster classes if instances are frequently accessed together

Objects are stored in extents of contiguous pages.

The default Versant strategy places first priority on clustering all instances within a class, but does not group particular sets of classes.

With the default strategy, each class is in a separate "cluster," and “instances” are stored on a set of data pages within the cluster.

If instances of one or more class are usually accessed together, you can use an interface specific routine to suggest to Versant that instances of the desired classes be stored in the same cluster. This will place them in physical proximity on the storage media and thus reduce the time required to find and retrieve instances of the classes if they are being used as a group.

The clustering routines do not force a clustering. Instead, they suggest to a database that a clustering should be performed. If a class already has been assigned storage, no action will be taken for that class.

IMPORTANT:- It is important to run a clustering routine before instances are created, or else Versant will ignore the hint.

To cluster classes after they have been created:

You can migrate all objects of the classes to be clustered to an interim database, drop and then recreate the classes in the original database. Also specify clustering hints and then import instances from the interim database.

Routines that will cluster classes are:

C	`o_cluster()`
C++	`cluster()` in PDOM

Create objects in the order in which you will access them

Once you have asked Versant to cluster related classes, it will improve performance if you create your objects and then use a group write routine to store them in the same order in which you will access them. This will improve the chances that related objects will be stored on the same or contiguous pages.

When concurrently loading objects, turn the `single_latch` server process profile parameter temporarily `ON`. Although this reduces concurrency, it also ensures that the database server process will not release a latch on the database cache acquired by a group write routine until the routine has completely finished.

If you have multiple access patterns, then you will probably have to experiment with different orderings and clusterings.

If you have a root object that is always accessed first, it may be useful to put related objects on both sides of it, not just after it. On the average, any given object is more likely to be in the middle of a page than at the beginning.

This technique of creating objects in a useful order has an important side effect. Each time an object is stored, Versant updates an object location table that associates the object identifier with the object physical location. When you use a link to find an object, Versant first finds the object identifier in the location table, and then uses that information to find the object physically. Each page in the table holds 809 associations, and look ups will proceed more quickly if all needed object identifiers are found on the same page.

Cluster Objects

Cluster Objects can improve performance in the following ways:

Cluster Objects on a Page

You can cluster objects on a page by using a “group write isolated” routine.

A “group write isolated” operation works like a normal group write, but it suggests that the storage manager isolate the given objects on their own page. This has two effects:

- First, the Storage Manager always starts with a new page to store the objects, rather than trying to use the "current" page for the storage file or trying to refill existing pages.
- Second, the Storage Manager marks all pages used as "NO REFILL"; this means that they will not be used for normal refill.

However, the "group write clustered" operation is not considered refill and the page will be used if the parent object is on the page. A page will continue to be freed if it has no objects stored on it.

Routines that will cluster objects on a page are:

C	`o_groupwriteisolated()`
C++	`groupwriteisolated()` in PDOM

Cluster Objects Near a Parent

You can cluster objects near a parent object by using a "group write clustered" routine.

A "group write clustered” operation works like a normal group write, but the Storage Manager places the given objects near an existing parent object.

First, the parent object is looked up to find its physical location. The object must exist in the database and it must be in the same storage file as the child objects. If both these conditions are

met, the Storage Manager attempts to store the child objects on the same page as the parent object. If the page does not have enough space, new pages will be allocated and the objects will be stored there.

Routines that will cluster objects near a parent object are:

C	`o_groupwriteclustered()`
C++	`groupwriteclustered()` in PDOM

Configure Log Volumes

Set the size of your log volumes appropriately

Versant internally triggers a checkpoint when either its physical or logical log volume is filled. A checkpoint is an expensive operation in which all data is flushed to the system volumes and the log files are cleared. Depending on the amount of data in memory, checkpoints can take up to several seconds or more. The operation that triggers a checkpoint will take much longer than otherwise identical operations.

Small log file sizes lead to more frequent but less expensive checkpoints. Larger log files will reduce the number of checkpoints, but they will be more expensive when they do happen. Larger log files give you better throughput and average response time, but increase the worst case response time.

One option is to use huge log files, which won't fill up and then, at low usage times, force a checkpoint by stopping the database with the `stopdb` utility. However, since stopping a database clears the server page cache as well as the log files, this approach is useful only if there are times when the database is not in use, such as at the end of a day. When you restart the database, all traversals will be cold because the page cache will initially be empty.

If you want to force a checkpoint at each commit, set the server process parameter `commit_flush` to `ON`.

Put your log volumes on raw devices

Placing the logical and physical log volumes on raw devices can improve performance if you are triggering a large number of checkpoints.

If the log file is on a raw device, it must be big enough to hold information for the biggest transaction that will occur. If it's on a file system, it will temporarily grow if necessary (disk space permitting), so being big enough to hold the largest transaction. is not quite as crucial.

BY MESSAGE MANAGEMENT

In most cases, Versant runs in at least two process - an application process and a server process for each database connection, including the connection to the session database. (The only time that Versant runs in one process is if you are accessing only a personal database and have linked your application with the single process library.)

If you are accessing a database on a remote machine, communications between the application process and a server process will occur over a network, which will lead to potential performance problems if network traffic is heavy or otherwise slow. However, even when you are accessing a local database, you should try to reduce the number of messages that need to be exchanged between application and server processes.

Gather statistics about network traffic

To assess the number of network messages that are occurring, collect and examine the `be_net_rpcs` statistic.

Keep in mind that the number of network messages (or "RPCs") reported will tend to be one higher than if you were not collecting statistics. This is because each line of statistics output involves a single RPC to get the value of any server statistics being collected.

Use group operations to read and write objects

Use group operations (group read, group write, and group delete) whenever possible.

In addition to the normal benefit of reducing network traffic, group operations contain optimizations, which make them inherently more efficient than multiple iterations of their single object counterparts.

If you use a group write or group delete, you can specify options that will release objects from memory and/or clear the cached object descriptor table. Remember however, that these objects will still be a part of the current transaction and subject to roll back, so you should not downgrade their locks. If you perform a group write, downgrade locks, and then roll back, you will get unpredictable results. (If you want to save changes and release locks, instead use a "commit with vstr" or "commit with collection" routine.)

Cache frequently used operations

Rather than repeat an operation, save its value from the first time and reuse it.

Although storing objects in the object cache is the most obvious use of this technique, there are other additional possibilities. As appropriate, consider caching non-object information (such as numbers, links, and query results) in local variables or transient data structures. If it enables you to meaningfully reduce your number of network messages, you should also consider caching information in persistent objects in a session database on your local machine.

BY MANAGING MULTIPLE APPLICATIONS

If the database is serving multiple clients, there are multiple issues which need to be handled.

Some of these case studies are answered here and suggesstions are given to handle them.

Gather multiple client statistics

To gauge contention for shared resources, collect and examine the following statistics.

`be_user_time` — Seconds of CPU time not in operating system kernel functions

`be_system_time` — Seconds of CPU time in operating system kernel functions

`be_lock_wait_time` — Seconds clients spent waiting for locks

`be_latch_wait_time` — Seconds waiting for latch.

`be_real_time` — Seconds elapsed since the `be_real_time` statistic was turned on.

Add additional clients if server performance is the primary issue

Even when it is not strictly necessary, concurrent access to a single database may be desirable since it can improve performance on the database machine by reducing context switching and increasing utilization of disk, processor, and shared data structures.

When dealing with multiple client applications, it is important to distinguish between response time and throughput. In general, adding an additional client will increase throughput on the machine containing the database and decrease response time as experienced by the applications.

If you are adding clients to a database, reduce latch contention by turning the `single_latch` server process profile parameter to "off."

If disk, processor or data structure utilization approaches 100%, consider adding additional databases.

Add additional databases if application performance is the primary issue

If your objective is to maximize response time and if the nature of the data allows it, consider distributing objects over multiple databases.

Use Asynchronous replication to improve application performance

You can also replicate objects in several databases. This has the same advantage of eliminating bottlenecks and works particularly well in a widely distributed environment. Each client can then use the database to which it is closest. Asynchronous replication is best implemented with event notification and "move object" features. The only downside is that there will be a small lag time between when an update is made to one database and when the change is propagated to the other databases.

BY PROGRAMMING BETTER APPLICATIONS

Tips for better Performance

Develop a transaction, locking and server strategy

In a multi-user environment, it is important to develop and implement a transaction, locking, and server usage model as early as possible in your development process.

For example, if multiple applications will be updating objects, it is important to keep your transactions as short as possible to minimize lock contention. In your data model, try to make sure that all objects needed for a particular operation can be accessed as a group and then released quickly. If you are going to use optimistic locking, you must be sure that all applications involved use a consistent application protocol and access objects in ways and times that maximize concurrency.

In general, your transaction model, locking model, and client/server models should be designed in, as retrofitting rarely works well.

Use multiple processes and threads

Depending upon your platform and release, you may be able to use multiple processes or threads in a transaction. In brief, it makes sense to use multiple processes or threads if one operation, such as printing, is input or output bound while others, such as a group read, are not.

Turn off locking if only one application will access the database

If it is possible that one application will attempt to modify an object that is being used by another application, some form of locking is necessary.

If only one application will access a database, you can safely turn locking off.

If only a few applications will access a database and if the chances of them conflicting is low (say, for example, that most applications are just reading data,) then consider using optimistic locking.

Lock as few objects as possible

If you need to use strict locking, don't lock any more objects than necessary.

If a group of objects is always locked together, consider using an application protocol that avoid placing a separate lock on every object in the group. For example, you could use the convention that none of the objects in a container can be modified unless a write lock is obtained on the container object. When using a "single lock plus a protocol" technique, be careful not to implicitly lock a group of objects which should not always be locked together. The reduced concurrency typically will more than negate the advantage of reducing the number of locks.

If concurrent access to objects is an issue, gather and examine the following statistics.

`be_lock_wait_time` — The number of seconds that applications spent waiting for locks.

`db_obe_locks_waiting` — The number of server processes (applications) waiting for locks.

`be_lock_waits` — The number of lock waits that occurred.

`be_lock_timeouts` — The number of timeouts that occurred due to waiting for locks.

`be_lock_deadlocks` — The number of deadlocks that occurred.

Turn logging off if it is not needed

Logging should be "`ON`" if you want your database to be recoverable, if you want to do rollbacks, or if you want to use log Roll Forward. In other words, it should be "`ON`" for the vast majority of production applications. If you don't care about any of these things, turn logging "`OFF`" for a quick, easy and substantial performance increase.

Optimize queries

Using links to navigate to objects and phrasing of queries to use indexes properly can significantly improve query performance.

Optimize link dereferencing

Under some circumstances, you might consider the order in which you connect to databases.

When you dereference a link, databases are searched for the target object in the following order:

1. The database of the object containing the link.
2. Other databases in the order in which you connected to them, starting with your session database. (If you connect and disconnect a number of times, the order will change.)

This approach will improve performance only if all of the following are true:

- You are using C++.
- You are connected to a large number of databases.
- The connected databases are large.
- You are mainly dereferencing links from a particular database.

Preallocate identifiers if you are going to create a large number of objects

By default, for each application, Versant reserves `1024` logical object identifiers (`LOIDs`) to reduce amount of requests going to the server every time you create new persistent objects.

If you know ahead of time that you are going to create a large number of objects, use a "set loid capacity" routine to reserve a specific number of logical object identifiers. This can improve performance by reducing the number of server requests.

For example, suppose that you want to create `34` million new objects. If you reserve your logical object identifiers ahead of time, then you will make just one call to the server to reserve object identifiers, rather than `33,236` calls.

Place access methods in your class rather than your applications

Although not strictly a performance tip, you should recognize that the characteristics of objects will change over time; however, interfaces should persist. So that all applications in a system operate in consistent ways, avoid defining attributes (C++ data members) as public. Always use access methods (get methods) and modifier methods (set methods) to manage data.

CHAPTER 12

Versant Event Notification

This Chapter gives detailed explanation on Versants Event Notification mechanisum. The Event notification mechanisms allow you to monitor objects and classes for the C and C++ interfaces.

Following topics are covered:

- Event Notification Overview
- Event Notification for C and C++
- Usage Notes
- Examples in C++

EVENT NOTIFICATION OVERVIEW

Event notification mechanism allow you to monitor objects and classes in the database, and receive a message when a specified object is modified or deleted. It also allows you to monitor a change, when an instance of a specified class is created, modified or deleted.

You can also define your own events and receive a message if the event occurs on a specified object or an instance of a specified class.

Terms Used in Event Notification

Following are the key concepts in event notification mechanisum:

Event

An "event" is something that happens.

System event

A "system event" is an event pre-defined by Versant.

For example, system events occur when a class or instance object is created, modified, or deleted.

User event

A "user event" is an event defined in an application.

Event message

An "event message" is a message sent by Versant when an event occurs.

Event registration

To receive an event message when a particular event occurs, you must "register" interest in the event with the database involved.

Event notification

When a registered event occurs, Versant sends an event message to a database queue on the database where the event occurred. These messages stay in the queue until you explicitly ask for them.

Transient Mode

In transient mode, Versant keeps a list of event registrations and a list of generated events in the memory. When the database stops, the lists are lost. This is the default mode.

Persistent Mode

In persistent mode, Versant stores the list of event registrations and the list of generated events in the database. When the database stops, the lists remain as is.

To specify persistence, use the following parameters in your application server profile file `profile.be`.

`event_msg_mode`	Specify persistent event messages.
`event_registration_mode`	Specify persistent event registrations.

You should set these parameters to Transient Mode for compatibility with existing/legacy applications.

New applications can use either of the mode in any combination. For instance, you can set your new application to use persistent event registration with transient event messages. You should avoid using `old_transient` with new applications though as it will be phased out in later releases.

For more information on setting mode, please refer to the section “Event Notification Parameters” in the chapter "Database Profiles" in the *Versant Database Administration Manual*.

Number of Event notifications

Event registrations are dynamically allocated and kept in the server database. Theoretically, there is no limit to the number of events that you can register, but practically the number is limited by available shared memory.

As the number of registrations are increased against a specific object on an event, the time to search for those interested registrations will also increase. And when an event is raised, it will

trigger an event for each interested registration, which could potentially generate a large number of events.

If an extremely large number of events are raised, potentially some of them may not be able to be delivered due to limitations on the message queue.

Current Event notification status

You can use the `dbtool` utility to display current event notification status information.

You can see the following information:

- Event notification status - "active" or "not active".
- Event notification configuration information.
- Event message queue information, such as the number of events left in the event queue.
- A list of all event registration entries and their status - "active" or "not active".

For more information, please refer to the dbtool reference in the chapter "Database Utilities" in the *Versant Database Administration Manual*.

Event Notification Process

A database produces event messages and puts them on an event message queue, where they are stored. Applications access messages, using an event delivery daemon. The flow of messages is:

db --*server process*--> msg queue --*delivery daemon*--> application

There is one event message queue and one event delivery daemon per database.

Both the event message queue and the event delivery daemon must be running for you to use event notification functions.

Since event notification is tightly bound to interface languages, there are separate event notification functions for C and C++.

Event Notification Procedures

To use event notification, complete the following steps.

1. Create event notification entry in the database profile file profile.be.

   For each database on which you want to use event notification, open the database profile file profile.be and create an event delivery daemon entry.

   The event delivery daemon entry can have either of the following two formats:

   `event_daemon` *`code_path`*

   or

   `event_daemon MANUAL`

   In the first form, code_path is the full path name of object code that will start an event delivery daemon. You must separately write this code, which is operating system specific. This code must be on the same machine as the database.

   In the second form, you must manually start an event delivery daemon after the database starts. If there is no `event_daemon` entry, "`event_daemon MANUAL`" is the default.

2. Start event notification daemon.

   You must start the event daemon before doing anything else. Otherwise, event defining requests from an application will remain pending until the daemon is started.

   When a database starts, it will configure itself per the parameters in the file profile.be.

   If there is an `event_daemon` entry in `profile.be`, the server process will create an event message queue, if one does not already exist.

- **Automatic startup**

  If the database process sees the entry event_daemon *code_path*, the server process will invoke the code located at code_path, which will start an event delivery daemon.

- **Manual startup**

  If the database process sees the entry event_daemon MANUAL, no event delivery daemon will be started and you must manually start an event delivery daemon before proceeding.

- **Null startup**

  If the database process sees no `event_daemon` entry in `profile.be`, manual startup behavior occurs.

3. Enable event notification for a database.

   Follow normal procedures to start a session and then connect to the database on which you want to enable event notification.

Next, enable event notification by running the `o_initen()` function or `initen()` method with the `O_EV_DAEMON` option. This will identify you to the database server process as the owner of the event delivery daemon and enable event notification.

After event notification has been enabled, the database will create a database event message queue, if it does not already exist. Then the event daemon will issue a request to open the database message queue to get a handle, and start to read requests. The database will immediately begin monitoring itself for registered events and, as appropriate, produce messages and put them on the event message queue. At this point, all event notification functions are available both for you and for all other users. This means that you and others can now register events, read events, raise events, and so on.

Note that it is the responsibility of an application to activate event notification. You must take care that your application waits for notification from the event daemon that it has received a handle to the database message queue before proceeding to register, read, and raise events. Otherwise, the application may attempt to perform actions before event notification has been activated. Alternately, you can build in logic that causes the application to retry its action until event notification has been activated.

4. Register event.

   Register the events that you wish to monitor with `o_defineevent()` or `defineevent()`.

5. Get events.

   Read events from the event message queue with `o_readevent()` or `readevent()`.

6. Release event notification system.

   When you are finished with event notification, your application should disable its event notifications with `o_disableevent()` or `disableevent()` and optionally disable event notification for the database with `o_initen()` or `initen()` and kill the event notification daemon.

Event Daemon Notification to Clients

The database clients might want to know whether the event daemon is running or not running. Depending upon the application requirements the database can be configured to notify the clients/DBA about event daemon's existence.

For every commit, the database server checks for the existence of the event daemon. If the event daemon is down, it takes different actions based on the setting of the following database server process profile parameter:

```
event_daemon_notification off/on
```

off - Writes the error 6524, EV_DAEMON_NOT_RUN to the database LOGFILE but doesn't abort the commit. This is the default behavior.

on - Aborts the commit and throws the error 6524, EV_DAEMON_NOT_RUN to the application in addition to writing the error to the database LOGFILE.

EVENT NOTIFICATION ACTIONS

Following are the actions performed by the Event Notification mechanisum.

- Disable event notification for a database
- Enable event notification for a database
- Initialize event notification for a database

Also given below are the corresponding C and C++ functions which perform the same:

Remove event message queue for a database

```
c     o_initen()
c++   initen()
```

Depending on options, the `o_initen()` and `initen()` routines can:

- Enable event notification for a database.
- Disable event notification for a database.
- Reactivate event notification after it has been disabled.
- Remove the event message queue.

Disable event notification for a registered event

Temporarily disable monitoring on previously registered events.

```
c     o_disableevent()
c++   disableevent()
```

Drop event notification for a registered event

Drop monitoring on previously registered events.

```
c     o_dropevent()
c++   dropevent()
```

Enable event registration for a registered event

Reactivate monitoring on events previously registered then disabled.

```
c      o_enableevent()
c++    enableevent()
```

Get event notification message

Retrieve an event from the event message queue.

```
c      o_readevent()
c++    readevent()
```

Get event notification status

Get event notification status.

```
c      o_getevstatus()
c++    getevstatus()
```

Raise event to daemon

Send a message to the event delivery daemon.

```
c      o_sendeventtodaemon()
c++    sendeventtodaemon()
```

This function allows you to send an event message to the event delivery daemon:

- Without registering
- Without specifying a target object and
- Without waiting for a commit for your message to be processed.

Raise event on an object

Explicitly raise an event on an object.

```
c     o_raiseevent()
c++   raiseevent()
```

Register event

Register for an application the objects and events to be monitored by a database. Afterwards, you will receive an event message whenever a specified event occurs to a specified object.

```
c     o_defineevent()
c++   defineevent()
```

Set event options

Set event notification options.

```
c     o_eventsetoptions()
c++   eventsetoptions()
```

Clear event notification options

Clear event notification options.

```
c     o_eventclearoptions()
c++   eventclearoptions()
```

Start work on event message queue

Start an event message daemon and get a handle to the database event message queue.

```
c     o_openeventqueue()
c++   openeventqueue()
```

Before you can access your event messages, you must use this method to receive a handle to your event message queue.

You can then use the handle, returned to `eq_handle`, to read your messages with `o_readevent()` or `readevent()`.

Event Notification Performance Statistics

The following statistics related to the event notification can be collected per database connection.

Statistics	Description
`STAT_BE_EV_DEFINED`	Events defined.
`STAT_BE_EV_SYS_RAISED`	The number of system events raised (but not necessarily delivered).
`STAT_BE_EV_SYS_DELIVERED`	The number of system events delivered, not including the system events `O_EV_BEGIN_EVENT` and `O_EV_END_EVENT`.
`STAT_BE_EV_USER_RAISED`	The number of user defined events raised (but not necessarily delivered).
`STAT_BE_EV_USER_DELIVERED`	The number of user defined events delivered.

The following statistics can be collected per database:

Statistics	Description
`STAT_DB_DISK_FREE`	Bytes of storage space available for any use (space in free extents)
`STAT_DB_DISK_RESERVED`	Bytes of storage space reserved by classes (space in allocated extents and free bytes in data pages)
`STAT_DB_EV_DEFINED`	Events defined.
`STAT_DB_EV_SYS_RAISED`	The number of system events raised (but not necessarily delivered).

STAT_DB_EV_SYS_DELIVERED	The number of system events delivered, not including the system events O_EV_BEGIN_EVENT and O_EV_END_EVENT.
STAT_DB_EV_USER_RAISED	The number of user defined events raised (but not necessarily delivered).
STAT_DB_EV_USER_DELIVERED	The number of user defined events delivered.

USAGE NOTES

Event notification initialization

When invoked with the `O_EV_DAEMON` option, the `o_initen()` function or `initen()` method initializes and enables event notification and identifies you as the owner of the event delivery daemon.

If you have linked your application with the single process library libosc.a, then `o_initen()` or `initen()` must be invoked within a database session by the DBA user who created the database.

When invoked, `o_initen()` and `initen()` act immediately if the application is using the one process model. Otherwise, they will act at the next transaction commit.

If you want to suspend the operation of the event message daemon, you can call `o_initen()` or `initen()` with the `O_EV_DISABLE` option. After the event delivery daemon has been disabled, all event notification functions (except `o_initen()` and `initen()` using the `O_EV_ENABLE` option and `o_dropevent()` and `dropevent()`) will receive an error.

NOTE:- If you have used Versant mechanisms to start an event delivery daemon and it terminates while a database is still running, the database cleanup process will attempt to bring up another copy of the daemon. However, when the database starts a new daemon, it will not restart event monitoring until you again invoke `o_initen()` or `initen()` with the `O_EV_DAEMON` option.

Event notification registration

To receive event messages, an application must register with the required database to monitor particular objects and events. The `o_defineevent()` function and `defineevent()` method register objects and events to be monitored.

Before you can make a registration, the event notification daemon must be running and some application, not necessarily your application, must have called `o_initen()` or `initen()`.

The objects to be monitored can be either class objects or instance objects. You can more precisely specify objects to be monitored by specifying a predicate, which will narrow the group of instances to which a raised event can apply. A registration can optionally specify when to evaluate this predicate, at the time when event is raised or when the originating transaction of the event commits.

The events of interest defines a subset of all events that can happen to the target objects. Only a raised event falling into this subset will send an event message.

You make registrations at any time during a session. The server process for the specified database will begin event monitoring as soon as an application makes a registration, rather than waiting until the next commit, so that you can see changes before making a commit.

Only after all applications monitoring an object drop their interests or terminate does a database stop monitoring the object. Shutting the database down cancels all event registrations.

At any time during a session, you can suspend event monitoring with `o_disableevent()` or `disableevent()`, reactivate monitoring with `o_enableevent()` or `enableevent()`, or terminate monitoring with `o_dropevent()` or `dropevent()`. If your application or the database terminates abnormally, monitoring for the application or database will be terminated.

Information about event registrations is kept in shared memory managed by the database, which means that it is available to all applications connected to that database.

Database event message queues have a maximum size, which is defined by the operating system on which the database resides. If the queue is full when an event message is generated, it will be lost.

Event registering, releasing, deactivating, or re-activating has no impact on locks. Use of event notification does not require that locking be turned `ON`, which means that it can be used in an optimistic locking environment.

The objects you want to monitor need not be in your application object memory cache, as only logical object identifiers (their loids) are sent to a database.

Event notification parameters

The contents of the events parameter in `o_defineevent()` or `defineevent()` determines the events monitored.

The events parameter must be a `vstr` containing pairs of event numbers, with the first part of each pair representing the lower range and the second part representing the upper range of the event numbers for the events to be monitored. If you are interested in just one event, use the same number for both parts of the pair.

The order in which the pairs of event number are specified in the events parameter determines the order in which events are evaluated. This means that performance will be better if events with a higher frequency of occurrence are kept in front of the list.

For example, a `vstr` of {[10, 20], [4,4], [98,99]} covers event numbers 4, 98, 99, and 10 thru 20. In this case, there is an expectation that the frequency of occurrences of events 10 thru 20 will be higher than that of event 4, which in turn is expected to occur more often than events 98 and 99.

The "define event" routines take effect immediately without waiting till the end of the transaction. The registration will be saved in shared memory on the machine containing the database and will be accessible by all server processes on that machine.

The event notification recipients, defined in `definer_info`, will be notified of an event on a registered object if the event falls into the event ranges of interest and the predicate is evaluated to TRUE.

You can invoke the "define event" routines more than once with identical parameters, but this practice is not recommended. If you double register, on each qualified event you will receive only one event message, but to drop, disable or enable the registration you will need to invoke the appropriate function twice.

The following are the macros for Event Definer and Raiser:

- O_EV_DEFINER_INFO (event_msg)

  Use this macro to get definer supplied information specified at event registration where `event_msg` is a pointer to the event message received.
- O_EV_RAISER_INFO (event_msg)

  Use this macro to get raiser supplied information where `event_msg` is a pointer to the event message received.

System Events

A system event is an event pre-defined by Versant.

System events include status changes to a database such as modification and deletion of instances, creation, modification, or deletion to any instances under a class object, or a schema evolution of a class object.

Another category of system events are tags that indicate the start and end of events raised from a database within a transaction. The time a system event is raised is determined by the system.

In an events parameter, you can specify the following numbers recognized by Versant.

Numbers	System Events
0xC0000000	O_EV_OBJ_DELETED, Deletion of an object
0xC0000001	O_EV_OBJ_MODIFIED, Modification of an object
0xE0000002	O_EV_CLS_INST_DELETED, Deletion of an instance of a class object
0xE0000003	O_EV_CLS_INST_MODIFIED, Modification of an instance of a class object
0xE0000004	O_EV_CLS_INST_CREATED, Creation of an instance of a class object
0x80000001	O_EV_BEGIN_EVENT, The start of an event
0x80000002	O_EV_END_EVENT, The end of an event

Events are added to the database event message queue when the transaction raising the event is committed. Otherwise, all events related to the transaction will be discarded.

However, an event is added immediately if it is raised with an explicit use of the "send event to daemon routine," either `o_sendeventtodaemon()` or `sendeventtodaemon()`.

The system events are explained as follows:

O_EV_OBJ_DELETED

Send a message when an object is deleted.

This event will be applied to both instance and class objects contained in the objects parameter.

For an instance object, a message is sent when it is deleted. For a class object, a message is sent when the class is dropped.

This event is similar to an object modified event, except that a message is sent only on deletion.

O_EV_OBJ_MODIFIED

Send a message when an object is updated.

This event will be applied to both instance and class objects contained in the objects parameter.

For an instance object, a message is sent when it is modified. For a class object, a message is sent when schema evolution occurs, such as when an attribute is added or dropped or when a subclass is dropped.

O_EV_CLS_INST_DELETED

Send a message when an instance of a class is deleted.

This event will be applied only to class objects contained in the objects parameter.

If you delete all instances of a class, as with `o_dropclass()` or `o_dropinsts()`, you will receive a message for each deleted instance, which could be overwhelming if the class contained many instances. Also, the server for the originating transaction trying to drop all instances could conceivably run short of memory trying to store all deferred events for each deleted instance.

O_EV_CLS_INST_MODIFIED

Send a message when an instance of a class is modified.

This event will be applied only to class objects contained in the objects parameter.

O_EV_CLS_INST_CREATED

Send a message when an instance of a class is created.

This event will be applied only to class objects contained in the objects parameter.

O_EV_BEGIN_EVENT

Send a message when an event starts.

This event will be applied to both instance and class objects contained in the objects parameter.

O_EV_END_EVENT

Send a message when an event ends.

This event will be applied to both instance and class objects contained in the objects parameter.

Multiple Operations

In a transaction, multiple operations may be applied to the same object and cause multiple events to occur. The result depends on whether the object is an instance object or a class object.

For instance objects, the following shows the net effect of two system events on the same object. Only relevant sequences are shown.

First event	Second event	Message sent
object modified	object modified	One obj modified message.
object modified	object deleted	One obj deleted message.

For class objects, the following shows the net effect of two system events on the same object. Only relevant sequences are shown, and in the following the word "object" refers to a class object.

First event	Second event	Message(s) sent
object modified	object modified	One object modified message.
object modified	object deleted	One object deleted message.
object modified	class instance created	Both messages.
object modified	class instance modified	Both messages.
object modified	class instance deleted	Both messages.
class instance created	object modified	Both messages.
class instance created	object deleted	One object deleted message.
class instance created	class instance modified	Both messages.
class instance created	class instance deleted	None.
class instance modified	object modified	Both messages.
class instance modified	object deleted	One object deleted message.
class instance modified	class instance modified	One class instance modified message

class instance modified	class instance deleted	One class instance deleted message.
class instance deleted	object modified	Both messages.
class instance deleted	object deleted	One object deleted message.

Defined Events

A "defined event" is an event defined in an application. It is bound to an object when you register it with `o_defineevent()` or `defineevent()`.

A defined event may or may not involve changes to an object. Defined events are known only to the application, and Versant does not attempt to interpret defined events. For example, you can define an event to occur when a routine affecting an object completes.

A defined event is identified by a number. You can specify your own event number pairs using any non-negative number, 0 through 2147483647.

You can register the same event to more than one object. However, when registered to different objects, an event may have different meanings when applied to one object or another. For example, an event on class C1 could mean a temperature increase, while an event on class C2 could mean a temperature drop.

Event Notification Timing

For system events, the server process will monitor registered objects.

For defined events, the application process will monitor registered objects and notify the server when a defined event on an object occurs.

When an event occurs, the server first evaluates each registration for the object and determines whether to evaluate the predicate immediately or upon commit.

Regardless of when evaluation is performed, a registration qualifies a raised event only if the registration is active, event occurred on a registered object is of interest, and the evaluation of the predicate, if any, is satisfactory.

Event Notification Message Queue

Notifications are stored in event message queues maintained by Versant. Messages for an application remain in the queue until the application requests its messages with a "get event message" routine, `o_readevent()` or `readevent()`.

If the message queue runs out of space and a message is sent to it, by default the following will happen.

1. All event messages generated in the transaction that have already been delivered are not affected. (There is no "rollback" of message delivery once a message in a string of messages has been delivered.)
2. Each time an undeliverable message is encountered, the event daemon will retry sending it three times. After three tries, the database server will write the event message to the file `LOGFILE` in the database directory.
3. If the event message queue clears during the time the event message queue is delivering a string of messages, all subsequent messages will be delivered as usual.

To improve performance when the event message queue runs out of space, you can override the default behavior by invoking `o_initen()` or `initen()` with the `O_EV_NO_RETRY` option.

If you use `O_EV_NO_RETRY`, the following will happen when the message queue runs out of space:

1. The error `EV_EVENT_LOST` is returned. As events are delivered only when a transaction is committed, it is unnecessary to rollback the transaction upon receiving this error.
2. The first event undeliverable message generated by the transaction is written to the file LOGFILE in the database directory.
3. All other undeliverable event messages generated by the transaction are discarded.
4. All deliverable event messages generated by the transaction are delivered, but the database server will try to deliver the message only once.

To restore default behavior, you can invoke `o_initen()` or `initen()` again with the option `O_EV_RETRY_SEND`.

NOTE:- In single latch mode i.e. when mutli_latch is off, if the event daemon is slow and the events are generated rapidly and if the transient event message queue is small, then any other utilities e.g. dbtool, db2tty that try to begin a database session, may appear hung. The reason for this is that when the events are generated and the transient event message queue is full, the database server will acquire a latch and check the queue periodically for a certain

number of times, where retry option is the default property. Once it reaches the max count and if it is not able to place the event message in the event message queue, then it writes an error EV_EVENT_LOST into the database LOGFILE and releases the latch. Since the latch is held throughout this operation other processes cannot acquire it and appear hung. They will eventually proceed ahead. So if multi_latch is off, the transient event message queue needs to be set large enough to improve the performance and to avoid the loss of messages.

EXAMPLES IN C++

Following are some examples using event notification with C++/Versant.

Alarm.h

```
/*
 *  Alarm.h
 *
 * Alarm class declaration.
 */
#ifndef ALARM_H
#define ALARM_H
#include <cxxcls/pobject.h>
class Alarm: public PObject
{
   private:
      int   level;
   public:
      //
      // Constructors & destructors
      //
      Alarm(int j=0):level(j) {};
      ~Alarm() {};
      //
      // Accessors
      //
      int getLevel() {return level;};
      //
      // Misc
      //
      friend ostream &operator<<(ostream &o, Alarm &e);
};
#endif
```

Daemon.cxx

```
/*
```

```
 * Daemon.cxx
 *
 * A sample event dispatching daemon process.
 */
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <iostream.h>
#include <strstream.h>
#include "Alarm.h"
#define AUX_LEN 100
//
// This is a sample of event delivery daemon for Event
// Notification
//
int main()
{
   char     *DBNAME = getenv("DBNAME");
   if (DBNAME == NULL) {
      cout << "DBNAME environment variable not set" << endl;
      exit(1);
   }

   o_ptr    eventQueue;
   int      lengthEvent = 0;
   o_event_msg *receivedEvent = 0;
   int      myPid = getpid();
   int      definerPid;
   VSession * aSession = new VSession(DBNAME, NULL);
   //
   // Specify that I am an event delivery daemon
   //
   aSession->initen(DBNAME, O_EV_DAEMON | O_EV_ENABLE, 0, 0);
   aSession->openeventqueue(DBNAME, &eventQueue);
   //
   // Allocate buffers for received events
   //
   lengthEvent = sizeof(struct o_event_msg) + AUX_LEN;
   receivedEvent = (o_event_msg *) malloc(lengthEvent);
   //
   // Signal only not for me
```

```
   //
   while(TRUE)
   {
      aSession->readevent(eventQueue, lengthEvent, receivedEvent, 0);
      definerPid = atoi((char *)O_EV_DEFINER_INFO(receivedEvent));
      if (definerPid != myPid)
      kill(definerPid,SIGUSR2);
   }
}
```

Element.cxx

```
/*
 * Element.cxx
 *
 * Implementation of Element class.
 */
#include "Element.h"
LinkVstr<Element> Element::getBad(int aLevel)
{
   PAttribute alarmAttr =
    PAttribute("Element::history\tAlarm::level");
   LinkVstr<Element> matchingObj =
    PClassObject<Element>::Object().select(
      NULL,
      FALSE,
      alarmAttr >= (o_4b)aLevel
   );
   return matchingObj;
}
LinkVstr<Element> Element::getGood(int aLevel)
{
   PAttribute alarmAttr =
    PAttribute("Element::history\tAlarm::level");
   PPredTerm predicate = (alarmAttr < (o_4b)aLevel);
   //
   // None of them have received an alarm of level2
   // or no alarm at all.
   //
   predicate.set_flags(O_ALL_PATHS|O_EMPTY_TRUE);
```

```
   LinkVstr<Element> matchingObj =
    PClassObject<Element>::Object().select(
      NULL,
      FALSE,
      predicate
   );
   return matchingObj;
}
void Element::addHistory(Alarm *a)
{
   dirty();
   history.add(a);
}
ostream &operator<<(ostream &o, Element &e)
{
   return o << e.name << '@' << e.address;
}
```

Element.h

```
/*
 *  Element.h
 *
 * Element class declaration.
 */
#ifndef ELEMENT_H
#define ELEMENT_H
#include <iostream.h>
#include <iomanip.h>
#include <cxxcls/pobject.h>
#include <cxxcls/pstring.h>
class Alarm;
class Element: public PObject
{
   private:
      PString         name;
      int             address;
      LinkVstr<Alarm> history;
   public:
      //
      // Constructors & destructors
```

```
        //
        Element(char *aName,int anAddr):
         name(aName),address(anAddr) {}
        ~Element() {};
        //
        // Mutators
        //
        void addHistory(Alarm *a) ;
        //
        // Queries
        //
        static LinkVstr<Element> getGood(int aLevel);
        static LinkVstr<Element> getBad(int aLevel);
        //
        // Misc
        //
        friend ostream &operator<<(ostream &o, Element &e);
};
#endif
```

MakeAlarm.cxx

```
/*
 * MakeAlarm.cxx
 *
 * Modify Element class instances to raise event.
 */
#include <iostream.h>
#include <iomanip.h>
#include "Alarm.h"
#include "Element.h"
int main()
{
   char  *DBNAME = getenv("DBNAME");
   if (DBNAME == NULL) {
      cout << "DBNAME environment variable not set" << endl;
      exit(1);
   }
   //
   // Start a session on DBNAME
```

```
//
VSession * aSession = new VSession(DBNAME, NULL);
//
// Parameters for the alarm
//
int   aLevel,anAddr;
char  anEl[255];
int   goOn = 0;
do
{
   printf("Alarm level:");
   scanf("%i", &aLevel);
   printf("Element name:");
   scanf("%s", anEl);
   printf("Element address:");
   scanf("%i", &anAddr);
   //
   // Get the equipment
   //
   LinkVstr<Element> elements =
    PClassObject<Element>::Object().select(
      NULL,
      FALSE,
      (PAttribute("Element::name") == anEl) &&
      (PAttribute("Element::address") == (o_4b)anAddr)
   );
   if (elements.size() == 0)
   {
      cout << "Bad input" << endl << flush;
      return -1;
   }
   //
   // Create the Alarm and commit
   //
   elements[0]->addHistory(new(
      PClassObject<Alarm>::Pointer()) Alarm(aLevel));
   aSession->commit();
   printf("Continue:");
   scanf("%i", &goOn);
}
while (goOn == 1)
```

```
      ;
   //
   // Commit and end session
   //
   aSession->commit();
   delete aSession;
   return 0;
}
#   link for interpretation with ObjectCenter
#   Notice: you must either run "getocinit" and "source init"
#   or you must add steps here to load Versant libraries.
#   Otherwise you will get a long list of undefined symbols.
#   compile
```

Monitor.cxx

```
/*
 * Monitor.cxx
 *
 * Register and wait for events.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <iostream.h>
#include "Alarm.h"
#define PID_LEN 20
o_bool theEnd;

//
// Notification through signals.
//

static void sig_handler()
{
   printf("BIP BIP BIP BIP BIP\n");
   theEnd = TRUE;
}
int main()
```

```
{
   signal(SIGUSR2,(SIG_PF)sig_handler);
   char     *DBNAME = getenv("DBNAME");
   if (DBNAME == NULL) {
      cout << "DBNAME environment variable not set" << endl;
      exit(1);
   }
   Vstr<o_u4b> events;
   LinkVstrAny classes;
   loid        regID;
   o_4b        badObjIndex = -1;
   char        processID[PID_LEN];
   int         myPid = getpid();
   int         pidLen;
   //
   // Start a session on DBNAME
   //
   VSession * aSession = new VSession(DBNAME, NULL);
   theEnd = FALSE;
   sprintf(processID, "%i", myPid);
   pidLen = strlen(processID) + 1;
   //
   // Define the event
   //
   classes.add(aSession->locateclass("Element", DBNAME));
   PPredicate pred= (PAttribute("Element::history\tAlarm::level")
         == (o_4b)2) && (PAttribute("Element::name") == "WACSEM");
   events.add(O_EV_CLS_INST_MODIFIED);
   events.add(O_EV_CLS_INST_MODIFIED);
   aSession->defineevent(DBNAME, classes, events, pred, NOLOCK,
      EV_EVAL_AT_COMMIT, pidLen, (o_u1b *)processID,&regID,
      &badObjIndex);
   //
   // Wait for the SIGUSR2
   //
   printf("I am doing something very important\n");
   while(!theEnd)
      ;
   //
   // Only reached after the signal
   //
```

```
   delete aSession;
}
```

Populate.cxx

```
/*
 * Populate.cxx
 *
 * Create instances of Element class.
 */
#include <iostream.h>
#include <iomanip.h>
#include "Alarm.h"
#include "Element.h"
int main()
{
   char  *DBNAME = getenv("DBNAME");
   if (DBNAME == NULL) {
      cout << "DBNAME environment variable not set" << endl;
      exit(1);
   }
   //
   // Start a session on DBNAME
   //
   VSession *aSession = new VSession(DBNAME, NULL);
   PClass *pe = PClassObject<Element>::Pointer();
   PClass *pa = PClassObject<Alarm>::Pointer();
   //
   // Create 3 elements
   //
   Element *e1 = new(pe) Element("FETEX-150", 100);
   Element *e2 = new(pe) Element("FETEX-150", 200);
   Element *e3 = new(pe) Element("WACSEM", 300);
   //
   // Commit and end session
   //
   aSession->commit();
   delete aSession;
   return 0;
}
```

Statistics.cxx

```
/*
 * Statistics.cxx
 *
 * Gather simple statistics about Element objects.
 */
#include <iostream.h>
#include <iomanip.h>
#include "Alarm.h"
#include "Element.h"
int main(int argc, char *argv[])
{
   char  *DBNAME = getenv("DBNAME");
   if (DBNAME == NULL) {
      cout << "DBNAME environment variable not set" << endl;
      exit(1);
   }
   if (argc < 2) {
      cout << "Usage: Statistics <level>" << endl;
      exit(2);
   }
   //
   // Start a session on DBNAME
   //
   VSession * aSession = new VSession(DBNAME, NULL);
   int aLevel = atoi(argv[1]);
   //
   // Get the bad equipments
   //
   LinkVstr<Element> bads = Element::getBad(aLevel);
   cout << "Bad equipments" << endl << flush;
   for (int i = 0; i < bads.size(); i++)
      cout << *bads[i] << endl << flush;
   bads.release();
   //
   // Get the good equipments
   //
   LinkVstr<Element> goods = Element::getGood(aLevel);
```

```
    cout << "Good equipments" << endl << flush;
    for (i = 0; i < goods.size(); i++)
       cout << *goods[i] << endl << flush;
    goods.release();
    //
    // Commit and end session
    //
    aSession->commit();
    delete aSession;
    return 0;
}
```

CHAPTER 13

Versant XML Toolkit

This Chapter gives complete guidelines on Versants XML Toolkit.

Following topics are covered:

- VXML Overview
- XML Representation of Versant Database Objects
- View and Pruning the XML Output
- Export Considerations
- Import Considerations
- Command Line Utilities
- APIs for Export/Import
- Frequently Asked Questions

VERSANT XML TOOLKIT OVERVIEW

The Versant XML toolkit consists of two APIs, one for exporting object(s) as XML and one for importing XML into object(s), and some command line utilities for bulk export and import.

The toolkit has the following features:

- Multiple language support -- objects created using the Java Versant, C++/Versant or C/Versant language interfaces can be exported or imported.
- Choice of raw database representation or language structured view.
- Predicate selection of classes/instances to export.
- Control over attributes exported.
- Import (update) of existing database objects and attributes.
- Import of new database objects preserving their interrelationships.

XML REPRESENTATION OF VERSANT DATABASE OBJECTS

The XML toolkit provides two variant representations: one paralleling the database representation and one paralleling the language structure of the object.

The DTD for these variants will be described with examples given below. If a more "natural" (or specific) representation is desired, it is necessary to transform the XML data with XSLT.

Throughout this guide, the following class examples will be used.

Java

```
class human {
    String    name;
    int       age;
}
class employee extends human {
    int       salary;
    int       daysOff[2];
}
```

C++

```
class human {
    PString   name;
    int       age;
};
class employee : human {
    int       salary;
    int       daysOff[2];
};
```

Fundamental DTD

The root element of the fundamental DTD (`www.versant.com/developer/vxml/dtds/v1.0/vxml.dtd`) represents the selected contents of the Versant database.

The root element can have multiple Versant `inst` (object instance) elements.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT   vxml          (inst)* >
```

The `inst` element can have zero or more `composite` and `attr` elements.

```
<!ELEMENT inst          (composite|attr)* >
<!ATTLIST inst
          class         CDATA #REQUIRED
          id            CDATA #IMPLIED >
```

The `inst` element requires a `class` attribute (the name of this class in the Versant database). The optional `id` attribute is the instance's LOID.

The `composite` element indicates a set of values.

```
<!ELEMENT composite     (composite|attr)* >
<!ATTLIST composite
          name          CDATA #REQUIRED >
```

The `attr` element indicates a database value (scalar attribute or array member).

```
<!ELEMENT attr          (#PCDATA) >
<!ATTLIST attr
          name          CDATA #REQUIRED >
```

The XML representation for a single instance of Java class `employee` might appear like this:

```
<?xml version="1.0"?>
<!DOCTYPE vxml SYSTEM "http://www.versant.com/developer/vxml/dtds/
v1.0
/vxml.dtd">
<vxml>
  <inst class="employee" id="3.1.12345">
```

```
    <attr name="name">Fred</attr>
    <attr name="age">32</attr>
    <attr name="salary">50000</attr>
    <composite name="daysOff">
      <attr name="[0]">6</attr>
      <attr name="[1]">7</attr></composite></inst></vxml>
```

The XML representation for a single instance of C++ class `employee` might appear like this:

```
<?xml version="1.0"?>
<!DOCTYPE vxml SYSTEM "http://www.versant.com/developer/vxml/dtds/
v1.0/vxml.dtd">
<vxml>
  <inst class="employee" id="3.1.12345">
    <attr name="human::name">Fred</attr>
    <attr name="human::age">32</attr>
    <attr name="employee::salary">50000</attr>
    <composite name="employee::daysOff">
      <attr name="[0]">6</attr>
      <attr name="[1]">7</attr></composite></inst></vxml>
```

Language DTD

The root element of the language DTD (`www.versant.com/developer/vxml/dtds/v1.0/vxmllang.dtd`) represents the selected contents of the Versant database.

The root element can have multiple Versant `inst` (object instance) elements.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT   vxmllang      (inst)* >
  <!ELEMENT composite     (composite|attr)* >
```

The `inst` element can have zero or more `composite` and `attr` elements.

```
  <!ELEMENT inst          (composite|attr)* >
  <!ATTLIST inst
            class         CDATA #REQUIRED
            id            CDATA #IMPLIED

            hash          CDATA #IMPLIED >
```

The `inst` element requires a `class` attribute (the name of this class in the Versant database). The optional `id` attribute is the instance's LOID. The `hash` attribute is the hash code value for Java classes with persistent hashes. (In the fundamental view, this value would appear as a distinct attribute.)

The `composite` element indicates an embedded class (or superclass), structure or array.

```
<!ATTLIST composite
          name          CDATA #REQUIRED
          serialization CDATA #IMPLIED >
```

Each database value (scalar attribute or array member) has a corresponding `attr` element.

```
<!ELEMENT attr          (#PCDATA) >
<!ATTLIST attr
          name          CDATA #REQUIRED
          serialization CDATA #IMPLIED >
```

The `serialization` attribute is a byte string encoding of the Java serialization of the second class object referenced. (In the fundamental view, this value would appear as an additional attribute.)

The XML representation for a single instance of Java class employee might appear like this:

```
<?xml version="1.0"?>
<!DOCTYPE vxmllang SYSTEM "http://www.versant.com/developer/vxml/
dtds/v1.0
/vxmllang.dtd">
<vxmllang>
  <inst class="employee" id="3.1.12345">
    <composite name="human">
      <attr name="name">Fred</attr>
      <attr name="age">32</attr></composite>
    <attr name="salary">50000</attr>
    <composite name="daysOff">
      <attr name="[0]">6</attr>
      <attr name="[1]">7</attr></composite></inst></vxmllang>
```

The C++ representation would be the same as above.

The Java example is almost the same as for the fundamental view (since the Java binding names database attributes with member names), with the exception of the superclass human being called out. For the C++ example, the difference is more striking: the language view shows the declared member names whereas the fundamental view shows the database attribute names (which have class scoping).

VIEW AND PRUNING THE XML OUTPUT

The `vxmlview` defines a view (class/attribute selection) for generated XML output, more precisely, what classes or attributes should be "pruned" from the output. (There is no equivalent "pruning" for import; the user would simply omit those elements from the XML stream, possibly via an XSLT transformation.)

A view object can be created programmatically, or by providing the XML declaration of the view (with `new VXMLView(xmlString)`).

The DTD for a `vxmlview` is as shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT   vxmlview      (class)* >
  <!ELEMENT class         (attr)* >
  <!ATTLIST class
            name          CDATA #REQUIRED
            include       (yes|no) "yes" >
```

The `include` attribute for a `class` element decides whether or not the `class` object will be output in the XML stream.

The default behavior is "`yes`", meaning that objects of this `class` and all its attributes should be in the resulting XML stream.

Programmatically, one gets the effect of `<class name="human" include="no">` by `view.addClass("human")`.

```
  <!ELEMENT attr          EMPTY >
  <!ATTLIST attr
            name          CDATA #REQUIRED >
```

The presence of an `attr` element means that this attribute will not be present in the resulting XML stream.

Programmatically, one gets the effect of `<class name="employee"><attr name="daysOff"></attr></class>` by `view.addAttribute("human","daysOff")`.

The naming of attributes and their scoping for pruning depends on the DTD view chosen.

The following examples show how to prune the `age` and `daysOff` fields in the different DTD views (Fundamental and Language DTD).

Fundamental DTD

Within the fundamental view, pruning acts against the database representation and naming scheme.

The fact that the field `age` is declared in superclass `human` is irrelevant. (That is, saying `<class name="human"><attr name="age"></attr></class> would have no effect.`) However, the language's database attribute scoping rules are relevant.

To get the effect of pruning `age` and `daysOff` from the Java example, one would need the following `vxmlview`:

```
<vxmlview>
  <class name="employee">
    <attr name="daysOff"></attr>
    <attr name="age"></attr></class></vxmlview>
```

and the following for C++ `vxmlview`:

```
<vxmlview>
  <class name="employee">
    <attr name="employee::daysOff"></attr>
    <attr name="human::age"></attr></class></vxmlview>
```

Language DTD

Within the language view, pruning acts against the actual field names, relative to their class scoping.

To get the effect of pruning `age` and `daysOff` from the Java example, one would need the following `vxmlview`:

```
<vxmlview>
  <class name="employee">
    <attr name="daysOff"></attr></class>
  <class name="human">
    <attr name="age"></attr></class></vxmlview>
```

The `C++ vxmlview` would be the same.

EXPORT CONSIDERATIONS

The export operation generates a XML representation of a specified collection of objects, which is written to a specified output stream.

Only the objects specified are exported - if objects referenced by those objects are to be exported as well, they must be included in the export collection.

NOTE:- If the class of an object is excluded by the view, that object will not appear in the output stream.

Only persistent objects can be exported.

The toolkit exports the value of the objects as known by the database. If the objects have been modified in this transaction the results are unpredictable.

Valid Characters

VXML supports characters, which are valid according to XML specifications. (Refer to XML1.0 Second Edition).

- Valid XML characters are:
  - `#x9 |`
  - `#xA |`
  - `#xD |`
  - `[#x20-#xD7FF] |`
  - `[#xE000-#xFFFD] |`
  - `[#x10000-#x10FFFF]`
- Characters in the compatibility area (i.e. with character code greater than `#xF900` and less than `#xFFFE`) are not allowed in XML names.
- Characters `#x20DD-#x20E0` are excluded (in accordance with Unicode 2.0, section 5.14).
- Signed C, C++ characters only between range 0x00 to 0x79 are valid.

Invalid Characters

If an invalid character is found while exporting data, VXML throws an exception. To ignore invalid characters in export use `'-ignore'` option, which will remove invalid characters from the output XML file.

IMPORT CONSIDERATIONS

The import operation reads a specified input stream, extracting the contents of a collection of objects from their XML representation, which are used to update existing database objects or to populate new ones.

Database Schema

The database must already have the necessary schema.

If the input data schema is different from the schema in the database, loose schema mapping will be used (as is consistent with the XML philosophy), and the database schema will not evolve. There is no warning generated.

- If the XML instance contains attributes not present in the database schema, the import utility will ignore the extra attribute.
- If the XML instance lacks one or more attributes that are present in the database schema, the import utility will:
  - set the missing attributes to their default values if a new database object is created.
  - leave the missing attributes unchanged if an existing database object is updated.

Preserve Mode

If the preserve flag is set: The importer will attempt to name the resultant instances with the ID values from the import stream. If a database object is found that has the matching ID of an input element (and the database object is of the correct type), that object is used. If the matching database object does not exist, a new one will be created with the ID value from the import stream.

If the preserve flag is not set: The new database objects with new IDs will be created. The ID fields of the XML instances are used to maintain object relationships between the imported objects; any imported object that contains a reference targeting one of these ID fields will be replaced by the system assigned ID for the corresponding reference target.

NOTE:- The Java constructors are not executed for newly created (Java) objects.

COMMAND LINE UTILITIES

Export

```
java com.versant.vxml.vdb2xml -d database {-q query |-qf queryfile}
[-p
 vxmlview] [-v userURI] [-f filename] [-s  schema] [-n] [-r]
[-c charEncoding] [-config configFile] [-ignore]
```

Parameters are:

`-d database`

Database to query. If the objects being referred in one database spans across the other database, then the name of the database containing the referencing objects should be passed as first parameter and the name of the other databases containing the referenced objects can be passed after that separated by ';'.

`-q query`

One or more VQL query strings separated by the delimiter ';'. Do not end the last query with ';'.

`-qf queryFile`

A file containing VQL queries separated by the delimiter ';'.

`-p vxmlview`

File (in the `vxmlview` schema) specifying pruning to be done.

`-v user DTD URI`

Location of DTD to be used, such as "`file:/abc/`".

`-f filename`

Export to a file instead of standard output.

`-s schema`

Export schema format: `vxml | vxmllang` [default is `vxml`]

-n

Nolock mode. By default, the tool applies a read lock to each object as it is accessed, which is held until the entire object collection is exported (so as to ensure referential consistency). This option disables locking.

-r

Referential mode. By default, only the objects selected by the query are exported. This option causes those objects and all objects they reference (recursively) to be exported.

-c charEncoding

User defined character encoding to be used in XML document.

-config configFile

Configuration file to pass the appropriate options.

-i

Ignore invalid XML characters.

The exported data is written to the standard output.

By default, the tool applies a read lock to each object as it is accessed, which is held until the entire object collection is exported (so as to ensure referential consistency). The nolock option disables locking.

By default, only the objects selected by the query are exported. The referential option causes those objects and all objects they reference (recursively) to be exported.

```
java com.versant.vxml.vdb2xml -d db1 -q "select selfoid from Person"
 -s vxmllang >  Person.xml
```

This above command selects all the `Person` objects in database `db1` and generates a XML representation, which is directed to file `Person.xml`.

Import

```
java com.versant.vxml.vxml2db -d database [-f filename] [-p] [-n num]
 [-v userDtdURI]
```

Parameters are:

`-d database`

Database to update.

`-f filename`

Read from a file instead of standard input.

`-p`

Preserve ID values from import stream [default is no]

`-n num`

Number of objects to import per commit unit [default is all]

`-v userDtdURI`

URI name used in export.

The imported data is read from standard input.

Objects are fetched from the database (or created) with WLOCK mode and continue to be locked until commit. By default, a single commit is performed after all objects have been imported. The user can specify smaller transaction units via the number option.

```
java com.versant.vxml.vxml2db -d db2 -c < Person.xml
```

This above command would recreate the set of `Person` objects (from `db1` above) in database `db2`.

Version

```
java com.versant.vxml.Version
```

This tool displays the version information of VXML.

Config file parameters

`dtd`

Location of DTD to be used, such as "`http://localhost:8080/`".

`vxmlview`

File (in the vxmlview schema) specifying pruning to be done.

`schema`

Export schema format: `{vxml | vxmllang}`. Default is vxml schema.

`querymode`

Whether to do query recursively or not. Set it to true for recursive behavior.

`lockmode`

The tool applies a read lock to each object as it is accessed, which is held until the entire object collection is exported (so as to ensure referential consistency). This option disables locking. Set it to false if you don't want the tool to apply the lock while reading objects for export. Set it to true if you want the lock to be applied while reading objects.

`outfile`

The name of the output file where the content of XML output is stored.

`encoding`

character encoding to be used for XML document.

`ignore`

Ignore invalid XML characters

Note that if the same options are passed through the command line as well as the config file, the options given at command line will override the options given in config file.

EXPORT / IMPORT APIS

The export / import process is driven by an instance of the `VXMLExporter`/ `VXMLImporter` class.

To start an export / import process, one creates an instance of the appropriate class, specifying the session to which it is bound. Other aspects of the operation (what `VXMLView` object to use, if any and whether to preserve input object IDs etc.) are specified by calling appropriate `setProperty` methods. Although these properties can be subsequently changed, it is intended that an exporter / importer is given a set of properties that will span its usage.

The actual work is done by invoking an `export` / `import` method, which identifies the I/O stream to use and the objects to be affected.

Once set up, the `export` / `import` methods may be invoked multiple times (within the same session), even concurrently by multiple threads.

Refer to the supplied programs for usage examples for the APIs.

Export Process

The export process is driven by a `VXMLExporter` object.

The supported properties are the target `schema` type (the default is `vxml`) and a view object (The default is none, meaning that no classes or attributes are pruned).

The exporter actually uses the interface Pruner for this purpose; the supplied `VXMLView` class implements that interface.

The exporter has two `export` methods: one which takes a collection of object `Handles` (suitable for use with the JVI fundamental binding) and one which takes a collection of Java Object references (which can only be used with the JVI transparent binding).

Import Process

The import process is driven by a `VXMLImporter` object.

The supported properties are create mode, update mode and an import "listener".

In the normal case, objects are fetched from the database (or created) with `WLOCK` mode. (If the user is operating in an optimistic session, no locks are applied.) After modification, the object will continue to be write locked. It is up to the user to release any locks, if desired.

NOTE:- The `VXMLImporter` performs no implicit commit.

By default, as each `inst` element is processed an object is instantiated. It is quite likely that the user would wish to perform special processing at this time (such as adding the object to a list or committing the update). For this purpose, the importer supports an import event listener (interface `VXMLImportListener`). The user can supply an object that implements this interface. When the importer has processed an object, the `endImport` method of the listener is invoked, passing a `Handle` to the new (or updated) object.

FREQUENTLY ASKED QUESTIONS

This topic addresses the most commonly asked questions about XML and Versant's XML Toolkit.

XML Specific Questions

What is XML?

Extensible Markup Language, or XML for short, is a new technology for data representation and transmission for Internet applications. XML is a World Wide Web Consortium (W3C) standard targeted at the emerging content management and business-to-business e-commerce application market. The standard supports the creation of custom tags (markup elements) to define data as well as being able to package data. An example of using XML would be to describe and deliver data for business-to-business e-commerce, such as RFQs/RFPs, order processing, etc.

Why is XML important?

Because different organizations (or even different parts of the same organization) rarely standardize on a single set of applications, and therefore data formats, it takes a significant amount of work for two groups to communicate. XML provides a standardized mechanisms for the exchange of structured data across the Internet between disparate sources while preserving the meaning of the data.

Where does persistence fit in XML?

Persistence for XML is important for several reasons. Without persistence, there is no integrity for all the important information being put into XML. A single failure anywhere on the network and everything is lost - including the customer's shopping cart. On an e-commerce site, that is equivalent to showing a customer to the parking lot and slamming the door behind them.

Persistence for XML is also important when data is aggregated from disparate data sources in information portals. The data may originate in legacy systems and databases, and once aggregated, would need to be maintained persistently to allow users to query the aggregate content and publish the content to Web-applications.

Persistence is also needed when managing the meta-model of XML content (i.e., for managing DTDs - Document Type Definition that describe the XML content).

What is a Document Type Definition (DTD)?

A Document Type Definition (DTD) is a set of syntax rules for tags. It tells what tags can be used in a document, in what order they should appear, which tags can appear inside what other ones, which tags have attributes and so on. Originally developed for use with SGML, a DTD can be part of a XML document, but it's usually a separate document or series of documents.

Because XML is not a language itself, but rather a system for defining languages, it doesn't have a universal DTD the way HTML does. Instead, each industry or organization that wants to use XML for data exchange can define its own DTDs.

The Document Type Definition is the schema of the data in a XML document.

The provisioning of the data in a document is, or should be, based on the DTD. In the case of B2B commerce, the DTD would be an agreed upon format between two enterprises for a particular set of data.

In XML, a document can reference the DTD in two ways: internally and externally.

Internal DTD

When a DTD is referenced internally, it is contained in the XML document and the document becomes self-describing. In theory, a document such as this can be delivered to a XML-enabled application and be parsed and understood. The challenge with internal DTD's is if one partner implements the DTD incorrectly or it changes and does not get updated by all users, there could be a problem.

External DTD

External DTD's can be referenced by a XML document by its URL. The XML document would still contain the data elements but the definition would be outside the document. The DTD would be stored on a server somewhere accessible by both parties. When a document is created, the DTD would be referenced for the schema and when the document is received, it would be validated against the DTD. This is a good way of centralizing the access to a DTD and ensuring that if the schema changes, all would be able to use the changed format.

Well-Formed and Valid XML

XML documents do not necessarily require DTD's. In the case of not having a DTD but being formatted correctly, a XML document is considered well-formed but not validated.

If a XML document has a reference to a DTD, either internal or external, it is considered well-formed and validated, providing it passed validation.

What is XSLT?

XSLT is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of XSL, which is a style sheet language for XML.

XSL provides a mechanism for formatting and transforming XML, either at the browser or on the server. It allows the developer to take the abstract data semantics of a XML instance and transform it into a presentation language such as HTML.

Versant XML Toolkit Specific Questions

How can the Versant ODBMS be used to provide persistence for XML?

There are fundamentally two different ways of viewing persistence for XML:

- Store XML documents as XML documents in a Versant database. This can be done using JVI and storing the DOM (Document Object Model) representation of the XML document in the database.
- Map XML content to existing database classes and objects. The Versant XML toolkit makes this possible.

Each form of persistence is valid and suitable for different application categories.

What are the components of the toolkit?

The Versant XML toolkit consists of two APIs for import and export.

For exporting object(s) as XML, use - `com.versant.vxml.VXMLExporter`

For importing XML into object(s), use - `com.versant.vxml.VXMLImporter`

Command line utility for bulk export

`com.versant.vxml.vdb2xml`

Command line utility for bulk import

```
com.versant.vxml.vxml2db
```

The ancillary class `com.versant.vxml.VXMLView`, supports views (selections) of attributes and classes.

What software is needed to use the toolkit?

The Versant XML toolkit is written in pure Java® and operates in any Versant environment with the following components:

- Versant ODBMS 6.0.0 or higher
- Java Versant Interface (JVI) 6.0 or higher
- JDK 1.2 or higher

What language interfaces are supported?

The toolkit itself is written in Java and therefore can only be used within the JVI environment (either a fundamental or transparent session).

The toolkit supports database objects created using the Versant Java, C++ or C language interfaces.

What is the difference between the Versant DTDs?

The two object representation DTDs, `vxml` and `vxmllang`, are effectively the same relative to the definition of XML elements and attributes. The key difference is in their usage.

The `vxml` DTD "flattens" a database object, reflecting its database representation. The `vxmlview` DTD is used by the end users of the Java APIs to define views (selections) for their XML documents.

The `vxmllang` DTD makes extensive use of the composite element to show language structure. Aside from providing a more natural view (from the language's point of view) of the object structure, it permits attributes to be named by their language names rather than their database names.

How does the toolkit handle schema changes?

If the input data schema is different from the schema in the database, loose schema mapping will be used (as is consistent with the XML philosophy) and the database schema will not evolve. There is no warning generated.

- If the XML instance contains attributes not present in the database schema, the import utility will ignore the extra attributes.
- If the XML instance lacks one or more attributes that are present in the database schema, the import utility will set the missing attributes to their default values.

What concurrency considerations affect toolkit usage?

- In general, the user of the APIs is responsible for managing transactions.
- For export, the set of objects is passed to the export API and the user should lock the objects as appropriate before passage.
- For import, objects are fetched from the database (for update), or created, with WLOCK mode. If the user is operating in an optimistic session, no locks are applied. After modification, the object will continue to be write locked. It is up to the user to release any locks, if desired, by registering an import event listener. No implicit commit is performed by the import API.
- The command line utilities operate within a single session.
- For export, all located objects are fetched with RLOCK mode unless the nolock command line argument is given.
- For import, objects are fetched from the database (for update), or created, with WLOCK mode. An explicit commit is performed after every N (user settable) objects, and at the completion of the process.

CHAPTER 14

Versant Backup and Restore

This Chapter gives details about the Database backup and Restore procedures.

Following topics are covered:

- Backup and Restore
- Using vbackup
- Using Roll Forward Archiving
- Roll Forward Archiving
- Roll Forward Management
- Roll Forward Procedure

BACKUP AND RESTORE

Overview

Data backup and restore is an essential and important need of any mission critical application.

You should ideally always plan for your data backup in event of an unavoidable system crash.

The purpose of a backup is to get the database in a consistent state. The purpose of restore is to restore the database with the backed up data.

Versant provides a very efficient and strong backup and restore mechanism.

When you create a backup and recovery strategy, depending on the importance of the data, there are many considerations and decisions to be made, like -

- the method of back up,
- the size of the backup device,
- the frequency and time with which the data back up,
- the trade-off between online and offline backups
- the place of the backup, archive and other files

You can backup multiple databases to a single tape drive or to a single file.

Database backups can be "online" or “offline”

Online - which means that a database can be backed up while it is being used. This is the default mode.

Offline - which means the backup can be taken when the database is not in use.

Database Restores are only “offline” and are generally faster than the backup.

Backups can also be "incremental" or “full”.

Incremental - which means, saving only those changes made since the last backup.

Full - which means saving the entire database.

If multiple databases are backed up with a single command, the first one will be stored in the position specified in the position parameter (see the option parameters below) and any additional ones will be appended after it.

A database backup saves the results of all transactions committed at the time `vbackup` is invoked.

For additional safety, you may also want to use the `-rollforward` option in conjunction with the `-backup` option. This will ensure that no log records are discarded unless they have been archived, using the `vbackup` option `-log` on a seperate command prompt.

Versant uses the `vbackup` utility for backup, restore, Roll Forward archiving and info operations.

Versant also uses Incremental restore procedure.

Backup and Restore Methods

Using Vbackup

The purpose of the `vbackup` utility is to allow recovery, either from a device failure or from accidental deletion of data.

The `vbackup` utility is used for the following operations:

- Backup
- Restore
- Roll Forward

Versant incremental backup (**vbackup**) strategy consists of three different backup levels, level 0, 1 and 2.

As database size increases, it becomes more important to set the level in a way, which minimizes backup size and time. It usually takes less time to create a level 1 or level 2 backup compared to a level 0 (full) backup.

Level 0 performs a full backup. Level 1 backs up all changes made since the last level 0 backup. Level 2 backs all changes made since the last level 0 or level 1 backup, whichever was most recent.

Roll Forward (RF) archiving is another feature that preserves logical log records generated by a database during normal operation in a log archive. These records can be replayed on the database during recovery. Thus, RF makes it possible to recover a database to its state just prior to the crash.

If you use just the backup and restore features of `vbackup` utility, you can recover data to the point of the last backup.

But if you want to recover the data, to the point of the last committed transaction, then use the Roll Forward features of `vbackup` utility.

NOTE:- The multi_latch parameter in the backend profile should be set to "`ON`" when backing up the database. If it is `OFF`, `vbackup` may hang.

For more information, please refer to the `vbackup` utility in the *Database Administration Manual.*

Using Roll Forward Archiving

If a severe database problem like a application crash or disk crash occurs, you can generally recover/restore your database using your last database backup. However this database backup would only be in the state it was, at the time when the last (proper) backup was taken.

This means that you will lose all database transactions, that have been committed after that “backup time”.

Roll Forward archiving helps to recover and to restore a damaged database to the most recent state before a failure occurred. Roll Forward archiving makes it possible to recover the database to the most recent state as it journals all database transactions.

If you want to keep a history of database states, then you probably want to use a combination of database backups plus Roll Forward archiving. In this way, you can always restore to a previous or last historical state.

If performance is an issue, then you also probably want to use Roll Forward archiving to a local tape or disk drive, because this avoids network traffic without compromising the ability to recover from a destroyed machine.

To ensure that you do not loose that data (post backup and till the restore is done), you should either use some form of Database replication or Roll Forward archiving.

Roll Forward archiving is described in detail in the later section of this chapter.

Versant provides additional Add-Ons modules to suppress or minimize the latency time to recover from a hardware or software failure, you will need any of those Add-Ons if continuous operation is your first priority.

For an overview of each module, please refer to Chapter 15 “Versant Add-ons” on page 349.

ROLL FORWARD ARCHIVING

Overview

Using just the regular backup makes it possible to restore a database to the state it was in when the backup was taken. This implies that you lose all database transactions, that have been committed after that “backup time”.

To ensure that you do not loose that data, you should either use some form of Database replication or Roll Forward (RF) archiving (which journals all database transactions).

The Roll Forward (RF) archiving makes it possible to recover the database to the most recent state.

Restoring a database with rollforward archiving turned on, first does the backup recovery from the regular level 0 backup, then from the incremental (level 1 or level 2) backup and then applies all the transaction log records form the archives. At the end the current logical.log is restored. This brings the database to its most recent state.

The steps needed to perform Roll Forward archiving are:

- The last full backup
- The incremental backups
- The transaction logs post last backup

Roll Forward Management

To use Roll Forward, you do the following for each database on which you want to use Roll Forwarding.

1. Create a level 0 backup and enable Roll Forward.

   The default setting for a newly created database is, Roll Forward not enabled. Roll Forward can only be enabled while performing a full backup (level 0, level 1, or level 2). A full backup is the prerequisite for running with Roll Forward.

   To backup a database and to turn on Roll Forward, you can use the command line `vbackup` utility. Once the full backup has been completed and Roll Forward has been enabled, Versant will not delete any log records out of the logical.log.

2. Start Roll Forward archiving.

   After you created the full backup and enabled Roll Forward, you need to turn on Roll Forward archiving (`vbackup -log`), using the command line `vbackup` utility. After Roll Forward archiving is turned on, Versant will start archiving log records created since the last backup. You can simultaneously backup multiple database logs to a single tape/file, or use multiple tape drives or files up to one per database.

   Only running an instance of the archiver process (`vbackup -log`) will remove the log entries out of the logical.log into an archive device (file, tape, etc.) It is absolutely important to run an archiver process (`vbackup -log`) at all times after rollforward is enabled, as else the logical.log will expand until it reaches its maximum size (OS dependent) upon which the database will have to shut down.

3. Temporarily stop Roll Forward archiving, if desired.

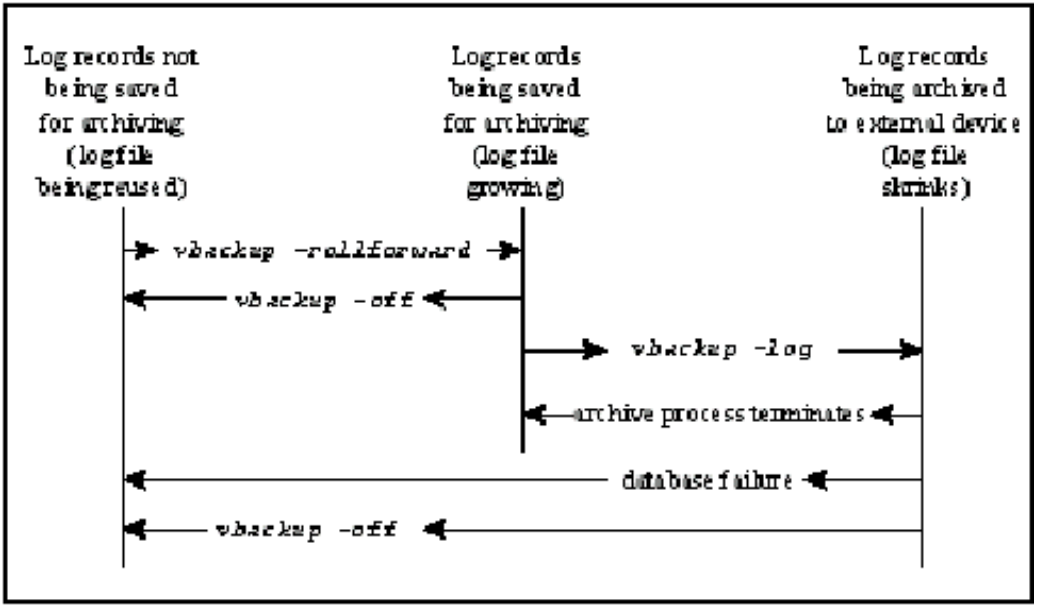
   Once Roll Forward has been enabled, you can safely stop and start archiving temporarily, as long as Roll Forwarding remains `ON`. As long as Roll Forwarding is `ON`, you are always guaranteed that no log records will be discarded until they have been archived.

   There are many reasons why you may need to suspend Roll Forward. For example, you may need to change a tape or file or use the Roll Forward tape drive to make an online backup. Or, Roll Forward may be interrupted by a tape device problem or network crash.

Following are possible database states:

The `vbackup` command options that will move you from one state to another are indicated in italics.

For more information on the usage of vbackup, refer to Chapter “Database Utilities” of the *Versant Database Administration manual.*


Log records not being saved for archiving (log file being reused)
Log records being saved for archiving (log file growing)
Log records being archived to external device (log file shrinks)
vbackup -rollforward
vbackup -off
vbackup -log
archive process terminates
database failure
vbackup -off

Roll Forward Procedure

If you need to Roll Forward, use the `vbackup` command line utility and first restore a level 0 backup of the database.

After you have restored from your last level 0 backup, you will be given the opportunity first to restore from any incremental backups (level 1 or level 2) and second for your Roll Forward archives and third you will use the entries in the current logical.log which did not make it to an Roll Forward archive yet.

You will not have to invoke `vbackup` repeatedly as all restore steps are performed in a single invocation.

Typical Sequences of Events

Archiving with one device

If you have only one tape drive or external drive, a typical sequence of events to backup two databases at the same time is the following (the procedure would be the same for just one database db1):

1. Perform a level 0 backup and enable Roll Forwarding.

   For example:

   ```
   vbackup -level 0 -device tape1 -rollforward -backup db1 db2
   ```

   After the backup, the command prompt will return. The backup does not have to be a full backup, just whatever is appropriate.
2. Begin Roll Forward archiving.

   For example:

   ```
   vbackup -device tape1 -log db1 db2
   ```

   After this command, the command prompt will not return and Roll Forward archiving will continue until you end the archiving process by closing the window or pressing <Enter> to exit `vbackup`.
3. Temporarily stop Roll Forward archiving (by pressing <Enter>) to change a tape. During this time, all log records are building up in the logical.log until you start archiving again. Depending on client activities, you might not want to interrupt Roll Forward archiving for too long.
4. Temporarily stop Roll Forward archiving to make an incremental backup. Again the log records will build up in the logical.log while Roll Forward archiving is stopped. Depending on client activity the temporary stop of Roll Forward archiving should not last too long.

For example:

```
vbackup -level 1 -device tape1 -backup db1 db2
```

5. Resume Roll Forward archiving.

   For example:

   ```
   vbackup -device tape1 -log db1 db2
   ```

Archiving with multiple devices

Suppose that you have three tape drives and two databases.

In this case, a typical sequence of events for databases db1 and db2 is the following.

1. Perform a level 0 backup and enable Roll Forwarding.

   For example:

   ```
   vbackup -level 0 -device tape3 -rollforward -backup db1 db2
   ```

   After the backup, the command prompt will return. The backup does not have to be a full backup, just whatever is appropriate.

2. Begin Roll Forward archiving on database db1 using tape1:

   ```
   vbackup -device tape1 -log db1
   ```

3. Open another window and begin Roll Forward archiving on database db2 using tape2:

   ```
   vbackup -device tape2 -log db2
   ```

4. Make an online incremental backup using tape3 while Roll Forward archiving is active on both databases:

   ```
   vbackup -level 1 -device tape3 -backup db1 db2
   ```

Restoring after a crash with Roll Forward enabled

For the above examples, to restore after a crash, do the following:

1. If the databases were completely demolished, first recreate the database directories and support files(.sharemem, .lock, etc.). The user who recreates the databases must be the same user who created the original databases. In this case:

   ```
   makedb -g db1
   makedb -g db2
   ```

2. Insert the level 0 backup tape and restore:

   ```
   vbackup -device tape3 -restore db1 db2
   ```

If additional tapes are needed for the full backup, you will be prompted for them. Also, you will be asked for incremental backups (level 1 and level 2), the Roll Forward tapes as well as the current logical.log as each step is completed.

Usage Notes

Following are usage rules related to Roll Forward archiving.

- To perform Backups, Roll Forward archiving, and Restores on a database, you must be the DBA user who created the database.
- You do not have to suspend Roll Forward archiving to perform an online backup if you use a device for the backup that is different from the Roll Forward device.
- If you need to spread Roll Forward archiving over several tapes or devices, the tape or device files will be marked with a sequence number. When you use the files to restore a database, the sequence number will be checked to ensure that they are applied in the proper order.
- If a database goes down, Roll Forward archiving for that database will be stopped, but Roll Forward archiving for other databases will continue unless they are writing to the same output media.
- If multiple databases are being archived to the same output media, a failure to any of the databases will cause log archiving to be stopped for all of the databases.
- If a database goes down, when it returns, you will have to restart Roll Forward archiving for that database.
- If Roll Forward archiving is stopped, either explicitly or by a process termination, and then it is restarted, nothing will be lost, and there will be no reason to do a fresh incremental backup. However, if Roll Forward archiving is stopped, you should restart the Roll Forward archiving process as soon as possible. If the database runs out of space in its log files while Roll Forward archiving is suspended, you will get an error message, and the database will shut down.
- Roll Forward archiving has no effect on the normal writing of log files to a database.
- If a database must be restored from backup and log archives, only log archives created after the latest incremental backup are utilitized.
- If you turn off Roll Forward with the `-off` option of vbackup, Versant will stop saving log records for archiving, and they will be discarded after a transaction ends.

CHAPTER 15

Versant Add-ons

This Chapter gives information about the Versant Add-on components.

Following topics are covered:

- For Database backup
- Warm Standby
- Habackup
- For Database replication
- Versant Vedding (FTS)
- Versant Asynchronous Replication (VAR)
- For Database reorganization
- Versant Vorkout
- For Structured Query Language Interface
- Versant ReVind(VSQL)
- V/ODBC

FOR DATABASE BACKUP

Added to the standard backup functionality described in Chapter 3 - Database Utilities, of the *Versant Database Administration Manual*, Versant provides some backup add-on components.

All the Backup add-on components require a separate license.

Versant Warm Standby

The Versant “Warm Standby” feature is used as an incremental Roll Forward recovery. It is designed to minimize the downtime in an emergency event, which requires a database recovery.

To achieve this, an up-to-date copy of the primary database needs to be maintained - this is the Warm Standby database. In case of an emergency, this Warm Standby database can be updated very quickly to the state of the primary database - just by applying the last Roll Forward archive plus the logical.log of the primary database (Rather than starting a full database restore that may take a considerable time with large databases).

NOTE:- This is a licensed component. Please contact Versant Support for License related information.

For more information on Warm Standby usage, please refer to the *Versant Warm Standby Usage Guide.*

Versant Habackup

Versant High Availability Backup (habackup) solution is based on a specific storage devices enabling full online Database backups to be performed within seconds. It exploits the device mirroring functionality of the modern disk arrays and the capabilities of special storage devices.

It allows the user to execute certain operations such as, splitting a mirrored device, after bringing the database to a consistent state. Thus, an instant copy of the database can be made at a consistent point without jeopardizing server availability, data integrity and performance. Subsequently, this copy can be used for various purposes including, continuous online backup and stand-by database for Decision Support System (DSS) applications.

NOTE:- This is a licensed component. Please contact Versant Support for License related information.

For information about Habackup, please refer to the *Versant High Availability Backup Usage Manual*.

FOR DATABASE REPLICATION

In many applications, there is a need to replicate data, typically to improve availability, to improve performance by geographically co-locating databases with the applications that access the databases, to isolate decision support systems from on-line production systems, and to help in recovery from failures using Hot Standby systems.

With Versant, replication can be achieved in two ways: Synchronous and Asynchronous.

Synchronous replication increases the duration of application transactions as a transaction cannot be completed until all the replicas have been brought to the same state.

Asynchronous replication does not prolong the transaction much, as it updates the replicas in a separate transaction from the application transaction. Therefore, asynchronous replication is only suitable where certain latency before achieving consistency among replicas is permissible.

Versant Vedding (FTS)

Vedding is used for synchronous database replication.

The purpose of Vedding (FTS) is to keep two databases in synchronization so that the applications can continue operating even if one of the databases fails.

Synchronous database replication mirrors the contents of one database in another, in a predictable and orderly manner. This provides either local or geographically remote redundancy, which protects against the unavailability of a database in the case of a system failure.

When you use synchronous database replication, the two databases kept in synchronization are called a “replica pair”. If a crash occurs to one database, Versant will continue to use the other remaining database. When the crashed database comes back, Versant will automatically re synchronize the database with the database that did not crash and return to fully replicated operation. The re synchronization will occur while the database that did not crash continues to be updated.

NOTE:- This is a licensed component. Please contact Versant Support for License related information.

For information on Fault Tolerant Server (FTS), please refer to the *Versant Vedding Usage Guide* and to the `ftstool` utility, in the Versant *Database Administration Manual.*

Versant Asynchronous Replication - VAR

This is for asynchronous database replication.

Versant Asynchronous replication is useful for achieving load balancing in a distributed environment.

For example, suppose that you have numerous regional offices accessing a common database. If you used asynchronous replication to maintain ten replicated local databases, users in each regional office would be making local database connections and the number of users per database would be reduced. In some situations, this approach would dramatically improve performance.

Versant Asynchronous replication provides two modes of replication: Default and Advanced.

The Default mode is ID based, the replication messages contain the ID of the objects to be asynchronously replicated and supports only Master-Slave configuration. This mode is suitable for LAN based replication where network is fast and reliable.

The Advanced mode provides a snapshot-based replication (the modified objects are serialized within the massage) and supports peer-to-peer and Master-Slave configurations. This mode is suitable for WAN based replication where the network is slow and unreliable. The Advanced mode provides customizes messaging, transport mechanism and conflict detection and resolution.

NOTE:- This is a licensed component. Please contact Versant Support for License related information.

For more details on Asynchronous Database Replication (VAR), please refer to the *Versant Asynchronous Replication Manual*.

FOR DATABASE REORGANIZATION

The data needs to be re-organized optimum usage, specially when some data has been added or removed. The `vcompactdb` utility helps us to organize the database.

Versant Vorkout

Vorkout is Versant's Online Database Reorganization Tool.

In a freshly populated database the objects are efficiently packed on disk, in storage units called pages.

A compact database enjoys performance advantages caused by better allocation of data on disk. A compact database also utilizes its backend cache to the fullest extent because each page in cache contains the maximum possible number of objects.

Over time as objects grow or are deleted, empty holes are created in the tightly packed database resulting in fragmentation of data segments. Thus, performance starts degrading and disk usage is also increased.

Versant has addressed this issue by introducing "Vorkout", the Versant Online Database Reorganizer. Its utility name is `vcompactdb`.

The enhanced `dbtool` utility provides the user the ability to analyse a database for wasted space and Vorkout reorganizes the data for reduced fragmentation and restored performance.

NOTE:- This is a licensed component. Please contact Versant Support for License related information.

For information about Vorkout, please refer to the *Versant Vorkout Usage Guide*.

For information about dbtool, please refer to the dbtool utility, in the *Versant Database Administration Guide.*

FOR STRUCTURED QUERY LANGUAGE INTERFACE

The Versant ReVind(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.

The Versant ReVind(VSQL) product suite addresses the need to obtain additional connectivity and openness.

The Versant ReVind Suite includes the following modules:

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.

The SQL language conformance is extended SQL grammar.

NOTE:- This is a licensed component. Please contact Versant Support for License related information.

For information, please refer to the *Versant ReVind Reference Manual*.

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 executes on the client machine and is available on windows only. It is built as a DLL (dynamically linked library). 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.

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

For information about VODBC, please refer to the *Versant ODBC Reference Manual*.

CHAPTER 16

Versant Internationalization

This Chapter explains Versant Internationalization framework.

Following topics are covered in this chapter:

- Globalization
- Versant Internationalization
- Versant Localization
- Usage Notes
- Examples Using I18N

GLOBALIZATION

Concepts

Most of the software products are developed with English as the language to store and retrieve data. In today's global market these products do not work as products have different requirement for storing and interpreting the data.

The process of enabling a software product for global market is called Globalization.

Our current global economy requires global computing solutions. These global computing solutions should be in harmony with user's cultural and linguistic rules. Mostly users expect the software to run in their native language.

The Versant ODBMS product consists of the database management system as well as utilities and administration tools. Versant acknowledges the use of the IBM ICU library to implement the locale dependent string comparisons.

Globalization consists of two distinct steps:

- Internationalization
- Localization

The first step in globalizing a product is to Internationalize it.
Versant internationalization will allow user to store and access data in their locale.

From this release Versant has internationalized the product so that you can develop Internationalized application. It will also help in localizing it.

Internationalization

Internationalization is the process of designing a software that can be adapted to various languages and regions without changing executables.

The term internationalization is also abbreviated as I18N, because there are 18 letters between the first "I" and the last "N".

Internationalization has the following characteristics:

- Textual elements, such as status messages and the GUI component labels, are not hard coded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages does not require recompilation.
- Culturally dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- The product can be localized quickly.

Localization

Localization is the process of adapting an internationalized product to meet the requirements of one particular language or region.

Steps are provided to help you in localizing Versant error messages. Information about the files which are used to generate messages and localization file details are also given.

VERSANT INTERNATIONALIZATION

While developing Versant's "Internationalized" applications, the essence of developing Versant applications remains largely unchanged.

Versant's "Internationalized" applications, offer the following core features which are described in details in the sub-sections that follow:

- Pass-through certification guarantee (that ensures "what you store, is what you get back")
- Searching and sorting of string-data is possible in Non-English languages
- Locale sensitive pattern matching at various collation strengths will be supported
- Error messages can be localized

Versant supports string valued datatype in users locale.

The localization of the error messages is described in the section "Versant Localization" on page 367.

UNICODE Support

Unicode is a standard that precisely defines a character set as well as number of encodings for it. It enables you to handle text in any language efficiently and allows a single application to work for a global audience.

UNICODE characters are stored in the attributes of "string" type i.e., `Array or vstr of char, o_1b or o_u1b` in Versant.

In Versant/C++, the internationalized character data can be stored in any of the Unicode encoded forms such as UTF-8, UTF-16 or UTF-32.

In Versant/Java, the character data is converted and stored in UTF-8 encoded form.

In order to query on unicode characters that are stored, virtual attributes built on "national" VAT (Virtual Attribute Template) should be used.

In Versant/Java interface, virtual attributes need to be defined to query data if non-ascii characters are stored in the database.

Pass-through (8-bit clean) Certification

Versant guarantees that the left most bit in a data-byte will not be used for internal purpose.

Following components are ensured to be pass-through certified:

- Character attribute values of object instances (i.e. java strings, C++ string types etc.).
- Output of command line utilities.
- Names of volumes and log files, trace file names in profile.be.
- Database names and directories involved.
- User names.
- VQL queries.
- Error messages.
- Environment variables.

Storing Internalized string data

The unicode character data that are stored in Versant attributes of string type can be queried. Attributes of string type - in Versant/C++, array or vstr of char or o_[u]1b; in Versant/Java, string or char[].

Virtual attributes based on `/national` need to be defined to query data if non-ascii character data is stored in the database and must specify the correct encoding in which it is stored.

Searching and Sorting of string-data

Versant introduced the concept of Virtual Attributes to support query on internationalized character data.

Virtual attributes are derived attributes, built at run-time on one or more schema attributes of a class; the data type of schema attributes can be augmented/overridden at run-time when a virtual attribute is defined on them. Virtual Attribute Template (VAT) is the “class” of virtual attributes that categorizes the virtual attributes. The implementation of VATs are plug-ins provided as shared objects that gets loaded into the Versant server process' address space. VAT implementation provides methods used in queries to get the value or compare values of virtual attributes.

The virtual attributes defined using /national VAT can be used to query on unicode encoded character data.

/national

To query and sort on Internationalized Strings, Versant uses the national virtual attribute.

`/national LOCALE ENCODING [STRENGTH] attrname`

Virtual attribute can be defined on attribute of Versant string type i.e attrname can be array or `vstr of char or o_[u]1b`.

Locale provides a means of identifying a specific region (user community who have similar culture and language expectations) for the purposes of internationalization.

LOCALE is an identifier that has fields for language, country and it can be represented as a string with fields separated by an underscore. The collation service that provides the string comparison capability is instantiated by referencing the locale and it maps the requested locale to the localized collation data available to ICU. (e.g en_US)

ENCODING specifies the encoding scheme used to store the unicode character data in the database. (e.g, UTF8)

STRENGTH specifies the requested level of comparison. Character strings have properties, some of which take precedence over others. There is more than one way to prioritize the properties. A common approach is to distinguish characters by their unadorned base letter (for example, without accents, vowels) then by accents and then by the case of the letter.

For locale-sensitive string comparison, the collation service provides four strength levels to allow different levels of differences to be considered significant:

Primary: A letter difference.

For example, 'a' and 'b' are considered as different, but not ‘a’, ‘A’ and ‘à’.

Secondary: An accent difference.

For example, 'a' and 'à' are considered as different, but not ‘a’ and ‘A’.

Tertiary: A case difference.

For example, 'a' and 'A' are considered different.

Identical: No difference.

For example, 'a' and 'a'.

The default value is tertiary.

Applications can specify the collation strengths per predicate term. Note that, this collation strength will impact not only the behavior of operators `O_MATCHES` (like, in VQL) and `O_NOT_MATCHES` (not_like, in VQL), but also on the equality operators `O_EQ, O_NOT_EQ`. For example in comparison at primary strength with operator `O_EQ`, strings with less equality as compared to the secondary strength and `O_EQ` will be returned.

NOTE:- The difference between Tertiary and Identical types is that some strings can be spelled with more than one different sequence of numeric Unicode values. Tertiary strength will compare two variant representations of the same string as equal, but Identical strength will compare them as different.

Valid examples

Following examples are valid:

`/national de_DE UTF8 first_name == "Andre" ` // valid

`/national en_US LATIN_1 last_name == "try" ` // valid

`/national BRIGHT LATIN_1 last_name == "lll" `// will be done on en_US locale.

Invalid examples:

Following examples are invalid:

`/national UTF8 de_DE first_name == "Paul"`

`/national de_DE bogus_encode last_name == "Yulin"`

If the locale specified by you does not exist then a default, en_US locale will be used. If the encoding specified by you does not exist then an error will be returned.

Back-end profile parameter- locale

If you want a particular locale and encoding to be applied to all string attributes in your defined classes then the "*locale*" parameter needs to be set in profile.be

Syntax for `` `locale` `` is - `<locale_name>:<encoding>`

For German locale and UTF8, `` `locale` _parameter `` will be as follows:

```
localede_DE:UTF8
Firstname = "Andrè"
```

is transformed to

```
`/national de_DE utf8 Firstname` == “Andrè”
```

This removes the overhead of specifying `/national` for all queries in the application.

A typical profile.be setting for I18N in profile.be is as follows:

```
locale              de_DE:UTF_8
custom_be_plugins   /versant/lib/libnational.so$/versant
```

You will have to specify a virtual attribute template to indicate encoding and collating sequence specifier.

Back-end profile parameter- virtual_locale

The syntax for the virtual locale:

```
virtual_locale “/national  $locale_name$encoding$attr”
```

This parameter should be used when you need nested virtual attributes involving /national. Default values for “$locale_name”, “$encoding” will be used from “locale” profile.be parameter.

“$attr” should be used for specifying nested virtual attributes.

```
virtual_locale  "/national de_DE utf8 $attr"
$attr - specifies that you can specify a single string
attribute to an index.
```

Locale Specific Pattern Matching

This Unicode pattern matcher allows users to run pattern queries in their native locale languages.

It will support all features similar to existing English pattern matcher. For example, the wild card “*”, ‘?’ and “[-]“.

In locale-language, the comparison is not at character level.

Applications can specify the collation strengths per predicate term. Note that, this collation strength will impact not only the behavior of operators O_MATCHES (like, in VQL) and O_NOT_MATCHES (not_like, in VQL), but also on the equality operators O_EQ, O_NOT_EQ. For example in comparison at primary strength with operator O_EQ, strings with less equality as compared to the secondary strength and O_EQ will be returned.

NOTE:- The difference between Tertiary and Identical types is that some strings can be spelled with more than one different sequence of numeric Unicode values. Tertiary strength will compare two variant representations of the same string as equal, but Identical strength will compare them as different.

Application Support

For Java / C++

You can use Java application to insert data into the database. Ensure that you are using the same encoding as specified in the profile.be to enter the data. Versant will ensure that data entered is not lost.

For Database Utilities

All the regular Versant utilities like `vstream`, `vbackup` etc. will work with internationalization.

Application Impact

Developer impact

To internationalize an application, the developer must be aware of:

- virtual attribute - /national
- profile.be parameter locale
- Review queries and indices on string valued attributes.
- Display issues if any

Database Administrator (DBA) impact

Must know locale, encodings application needs and set the parameters accordingly. Once the data has been loaded you cannot modify the locale.

Error Message files

Error message file will be managed according to national code set.

- `lib/en_us_L1.msg`

  This is an example. The new error message file named by locale name is suffixed with "`.msg`" if in national code set. The same format is applied to `de_ch_L1.msg`.

  `lib/error.msg` file still exists for backward compatibility. If the error message file named by the Environment variable is not found, then “lib/error.msg” is used.

- `lib/en_us_L1.msi`

  Above file is generated and used internally by Versant. `Verrindx` needs to be run before using Versant if user has copied his error files using a particular locale.

- In `db` directory:

  `db/dbname/profile.be` file contains database's national code set information.

Deployment Issues

As an application developer you need to ensure that your application (at client side) displays the data correctly. Versant will not do any code conversion.

Appropriate `VERSANT_LOCALE` value needs to be set to get an error message in a particular locale.

See also “Versant Localization” on page 367.

VERSANT LOCALIZATION

VERSANT_LOCALE

The environment variable will indicate if you want to choose a locale specific error message or not. Default error messages will be in Latin-1.

For example:

`../lib/en_US_latin1.msg` (using Latin1 encoding for US English)

`../lib/ja_JP_jis.msg` (using JIS encoding For `ja_JP`)

If a you set `VERSANT_LOCALE` to "`ja_JP`" then all Versant error messages will be displayed in Japanese language.

Localizing Interfaces and Tools

Versant interfaces and tools use a common mechanism called *Standard Localization Facility* to generate all natural language messages, both error messages and non-error messages.

For example, progress messages, prompts, and names seen in graphical interfaces are all generated by the Standard Localization Facility.

Following are instructions for localizing the database kernel, database utilities, C interface, and C++ interface.

Files Used to Generate Messages

Versant localization is achieved using the environment variable `VERSANT_LOCALE`. You will have to copy the modified files like error.txt and error.msg as `$VERSANT_LOCALE.msg`, and `$VERSANT_LOCALE.txt`. To generate messages, the Standard Localization Facility uses the following files. These files are all located in the lib directory for a particular release.

error.msg

This text file is the source of all natural language messages for all interfaces and tools that use the standard facility.

error.msi

This binary file is an index of the locations of the error messages in the error.msg file. The last letter of that file has been changed to “i” to mean “index”.

When this file exists and is accurate, message retrieval is accelerated. If it does not exist or is inaccurate, messages will still be retrieved, but slowly. Thus any time a change is made to error.msg, the error.msi file should be rebuilt by running the verrindx command.

error.txt

This text file is only used by the `verr` command.

The lines in the error.msg file are a subset of the lines in this file. The additional lines in this file provide elaboration on the error messages.

Standard Character Set

The following ASCII characters are treated specially by some components of C++/Versant. You may have problems if they are given new meanings as letters.

Description	Example	ASCII #
white space		32
Tab	`\t`	9
Newline	`\n`	Usually 10, a standard word and line separator.
Comma	`,`	44, Delimiter in error.msg and error.txt
Colon	`:`	58, Delimiter in error.msg and error.txt.
Bang	`!`	33, Used as a special token in .sch files
Percent	`%`	37, Used for substitution marks in error.msg.
Backslash	`\`	92, Used for escape sequences in `error.msg`.
angle-bracket	`<`	60
close-angle-bracket	`>`	62, Used in formatted error messages.
forward-quote	`\'`	39
backward-quote	`` ` ``	96, Used in formatted error messages.
Zero	`0`	48

One	1	49
Two	2	50
Three	3	51
Four	4	52
Five	5	53
Six	6	54
Seven	7	55
Eight	8	56
Nine	9	57, Used in error.msg and error.txt

Localization File Details

Details of the following files used in localization is given below:

- `error.txt`
- `error.msg`

error.txt

There are two types of lines in the `error.txt` file: primary lines and secondary lines.

Primary lines :

Primary lines have the following syntax:

```
nnnn, error_name: message \n
```

Elements are:

nnnn

Primary lines begin with a digit, which is the error number. The error number can be one or more digits, 0-9.

,

A comma delimiter.

space

A space delimiter.

error_name

The error name, which can be one or more characters. You can use any character except a colon.

:

A colon delimiter.

space

A space delimiter

message

The natural language message, which can be any characters.

\n

normal line termination

Within message, certain two-characters sequences have special meaning:

%s

Substitute a string

%d

Substitute a number, print in decimal.

%x

Substitute a number, print in hexadecimal.

\n

Substitute a newline character, the \n of C.

\t

Substitute a tab character, the \t of C.

\b

Substitute a backspace character, the \b of C.

\f

Substitute a formfeed character, the \f of C.

****

Substitute a backspace character, for escaping.

\%

Substitute a percent character, for escaping.

Secondary lines :

Zero or more secondary lines follow each primary line and elaborate on the error in the primary line.

The syntax for secondary lines is:

```
\t message \n
```

Elements are:

\t

Secondary lines begin with a tab character.

message

The message can contain any characters except for escape characters.

\n

normal line termination

Some secondary lines are of this form:

```
[ used by `  class Link (vnih)']
```

These lines are not used consistently, and could be left out rather than translated.

error.msg

The syntax of the `error.msg` file is the same as the error.txt file, except that secondary lines are not allowed.

Following is a fragment from the error.txt file with the components described above.

Each primary line must be one long line in the file. For example, the lines beginning with 8052 and 8054 are shown here as two lines, but in the file they must each be a single line.

The lines that do not start with a digit must begin with a tab character.

```
8045, CXX_HASH_UNDEFINED: hashing method undefined
      It is the responsibility of a subclass of PVirtual to define
      a hash method. This error is thrown by the default
      PVirtual::hash().
8050, CXX_ZERO_F_ARG: bad zero argument for %s in function %s
8051, CXX_ZERO_M_ARG: bad zero argument for %s in method %s
8052, CXX_OM_PANIC: internal object manager panic: %s (line %d file
      %s)
      This is a serious error. No other database object manager
      operations will be possible for this process after this error.
      This is cause for terminating your program and trying
      again. It may indicate database corruption.
8054, CXX_CLASS_SIGNATURE: for class %s, compiled class
      signature %x does not match schema signature %x
      in database %s.
      A 32-bit signature is calculated for each class during schema
     capture using a CRC function across the class and its attributes
    This 32-bit signature is compiled into your application and is
     also deposited in the .sch file. sch2db will read it from the
    .sch file, and put it into the database. When your application
     runs, the signatures in your program are compared against the
     signatures in the database whenever database classes are used.
     If the signatures do not match, this exception is throw. This
    can happen when you have a different defintion of the classes in
     your application than there are in the database, because you
     installed the wrong .sch files into the database.
     To correct the problem, use sch2db to install the correct .sch
     files into the databases.
```

In the `error.msg` file, all of the secondary lines in the above fragment would be left out and only the following would remain (again, each element must be a single long line in the file):

```
8045, CXX_HASH_UNDEFINED: hashing method undefined
8050, CXX_ZERO_F_ARG: bad zero argument for %s in function %s
8051, CXX_ZERO_M_ARG: bad zero argument for %s in method %s
8052, CXX_OM_PANIC: internal object manager panic: %s (line %d file
                                %s)
8054, CXX_CLASS_SIGNATURE: for class %s, compiled class signature %x
      does not match schema signature %x in database %s
```

Localizing the “Standard Localization Facility” Files

To localize the files used by the Standard Localization Facility:

1. Edit `error.txt`.

   Edit the `error.txt` file to translate the English messages into the local language.

   - You must retain the syntax of this file, as documented below.
   - The "%" substitutions must also be retained, in exactly the order they appear here.

   For example, the following error message must be translated so that the attribute name is still substituted first, and the class name is still substituted second, even if that is inconvenient in the local language:

   ```
   8544, SCAP_NEWATTRAT: Cannot add new attribute %s to class %d
   ```

2. Build `error.msg`.

   Build the `error.msg` file by copying the new `error.txt` file, but retaining only the lines that begin with a digit.

   The following UNIX command will do it for you:

   ```
   cd .../lib ; grep "^[0-9]" < error.txt > error.msg
   ```

   In the above, for .../lib substitute the path to the VERSANT lib directory.

3. Run `verrindx`.

   Run the Versant `verrindx` command. This should be done any time the `error.msg` file is altered.

Localizing Versant View

LOCALE

LOCALE represents only the collating sequence and is not used for date and time attributes. A partial list of the locales supported by Versant is as follows:

German

Country	Language (German=de)
Austria	de_AT
Germany	de_DE
Luxembourg	de_LU
Switzerland	de_CH

French

Country	Language (French=fr)
Belgium	fr_BE
Canada	fr_CA
France	fr_FR
Luxembourg	fr_LU
Switzerland	fr_CH

Chinese

Country	Language (Chinese=zh)
China	zh_CN
Hong Kong	zh_HK
Taiwan	zh_TW

Spanish

Country	Language (Spanish=es)
Argentina	es_AR
Bolivia	es_BO
Chile	es_CL
Colombia	es_CO
Costa Rica	es_CR

Dominican Republic	es_DO
Ecuador	es_EC
El Salvador	es_SV
Guatemala	es_GT
Honduras	es_HN
Mexico	es_MX
Nicaragua	es_NI
Panama	es_PA
Paraguay	es_PY
Peru	es_PE
Puerto Rico	es_PR
Spain	es_ES
Uruguay	es_UY
Venezuela	es_VE

English

Country	Language (English=en)
Australia	en_AU
Belgium	en_BE
Canada	en_CA
Ireland	en_IE
New Zealand	en_NZ
South Africa	en_ZA
United Kingdom	en_GB
United States	en_US

ENCODING

This specifies the encoding used in this attribute.

A partial list of encodings supported by Versant is as follows:

8859-1[LATIN_1]	ascii[LATIN_1]	iso-8859-1[LATIN_1]
8859-15 [923]	asmo-708 [1089]	iso-8859-15 [923]
8859-2 [912]	big-5 [1370]	iso-8859-2 [912]
8859-3 [913]	big-5 [1370]	iso-8859-3 [913]
8859-4 [914]	chinese [1386]	iso-8859-4 [914]
8859-5 [915]	cp037 [1140]	iso-8859-5 [915]
8859-6 [916]	cp1008 [5104]	iso-8859-6 [1089]
8859-7 [917]	cp1025 [1154]	iso-8859-7 [4909]
8859-8 [918]	cp1026 [1155]	iso-8859-8 [916]
8859-9 [919]	cp1027 [5123]	iso-8859-8i [916]

Versant will support locales and encodings supported by ICU 1.8.1.

For more details go to `www.alphaworks.ibm.com`**.**

If a wrong `LOCALE` is specified, a default locale, `en_US` will be used. For example: if you misspell a locale and the corresponding locale does not exist, then `en_US` will be used.

USAGE NOTES

Debugging Messages

Debugging messages are not localized at the present time. Instead, they are compiled out of production releases. In the future, debugging messages will be localized.

Shell Scripts

Some shell scripts do not use the Standard Localization Facility. Their messages are in the shell scripts, and can be easily edited. Other shell scripts do use the Standard Facility. These scripts can be identified because they call the undocumented command `perrfilt`.

Restrictions

Versant internationalization has the following restrictions:

Class names and attribute names

Class names and attribute names are not certified to be Unicode/multi-byte compliant.

Conversion between different locales

Client-side environment variable `VERSANT_LOCALE` and Server-side variable, `virtual_locale` for Locale does not indicate any code conversion between Client-side and Server-side.

Pass through certification

- ReVind(VSQL) is not pass through certified.
- `vexport`, `vimport` and `vstream` is pass-through certified for UTF-8, Latin-1 only.
- X-based utilities are not pass-through certified
- Vedding(FTS), Event Notification in C++, Java are not be tested in lab-to-lab release

OS paths and file name size

The OS path size and the file name sizes will not be increased. The length of these variables will remain the same. Some of the Operating Systems do not allow non-ASCII file names.

It is advisable to check these restrictions.

Current limits for Versant are as follows:

Database names, User names	31	bytes
Node name	223	bytes

Modification of Profile.be

Internationalization parameters like 'locale' that can be specified in the database server profile file (`profile.be`) should not be modified after the data has been loaded in the database. If you change the profile parameters, the locale specific string comparisons may yield incorrect results as the data may be in a different encoding.

Locale specific data comparisons cannot be specified in path-queries.

Operator [] returns nth byte instead of character

`Vstr<char>` will be treated as `Vstr<o_ulb>`, that is [] operator will return nth byte rather than a character.

Supports encoding that do not use NULL

Versant will support encodings, which do not use NULL as part of any character. For example: UTF-8, Latin-1.

Java strings not converted to encoding

Java strings will be returned as `ByteArray`. It will not be converted to encoding as done currently.

Pattern matching query with accent character

A pattern matching query with accent character before the wildcard may not work correctly under the following conditions:

- Accent character before the wildcard in the pattern string

- There is a corresponding index
- Primary strength level

For example, the following query doesn't return the object that contains "Frederic"

```
select selfoid from BasicEmployee where `/national fr_FR utf8
                                                    PRIMARY
     BasicEmployee::emp_name` like 'fr?*'
```

The workaround for this problem is to avoid using the accent character in the pattern string:

```
select selfoid from BasicEmployee where `/national fr_FR utf8
                                                     PRIMARY
     BasicEmployee::emp_name` like 'fre*'
```

Syntax for Virtual Attribute

A Virtual Attribute is used in certain operations where a normal attribute would be used. To indicate that it is virtual, it begins with a slash (/) character. Parameters that contain SPACE or TAB characters are quoted within curly (braces{{...}}({...}). This is important, as parameters could be other Virtual Attribute with parameters.

Following is the Virtual Attribute grammar that uses an informal BNF notation.

```
SLASH"/"
BRACE-START "{"
BRACE-END"}"

virtual-attribute ::= SLASH<virtual-attribute-template>
   { <attribute> | BRACE-START <virtual-attribute> <BRACE-END }
   [ { <attribute> | BRACE-START <virtual-attribute> <BRACE-END }+ ]
virtual-attribute-template ::= nocase <encoding> | tuple | national
                                                    <locale>
<encoding> | …
```

- { symbol } means a mandatory symbol
- [symbol] means an optional symbol
- * represents 0 or more symbols
- + represents 1 or more symbols

- keywords are presented in bolded font, and are case-insensitive
- terminal symbols are represented in uppercase
- non-terminal symbols are represented in *italic* font for example: *virtual-attribute*

EXAMPLES USING I18N

VQL

VQL queries can be written to use I18N feature as under:

Suppose a class ‘Book’ has a attribute ‘*Title*’. There are instances of German and English books. A VQL query on books with titles beginning with letter “U” might be:

```
int run_query()
{

d_VQL_query query("select OID from Book  where  title >=
"U" and title <= "V");
   d_oql_excute(query,structVstr);
...

}
```

But, the given query will not retrieve the titles beginning with “Ü”. In order to achieve the required result, you will have to modify the query as follows:

```
int run_query()
{

d_VQL_query query("select OID from Book where `/national
de_DE utf8 title` >= "U"
and `/national de_DE utf8 title` <= "V" " );
  d_oql_excute(query,structVstr);
........
}
```

The following locale specific pattern matching queries are allowed:

```
select * from BasicEmployee where `/national fr_FR utf8 Tertiary
BasicEmployee::emp_name` like ‘Frèd’
```

C++

In C++, `PPredicate` needs to be changed as follows:

```
PClassObject<Book>::Object().select(NULL,FALSE,
PPredicate(PAttribute("/national de_DE utf8 Book::title") > "U"
  " &&
PAttribute("/national de_DE utf8 Book::title") ==
 " "V" "));
```

RESULT: will include books beginning with "U" or "Ü" upto “V”

CHAPTER 17

Versant Open Transactions

This Chapter gives detailed explanation about the “Open Transactions”.

The following are explained in detail:

- Versant Open Transaction
- X/Open Distributed Transaction Processing Model
- Versant X/Open Support

OVERVIEW

An open transaction is a transaction controlled by an external process, called a Transaction Manager.

In a Versant open transaction, each phase of a commit or rollback to a Versant database is controlled explicitly by a non-Versant Transaction Manager.

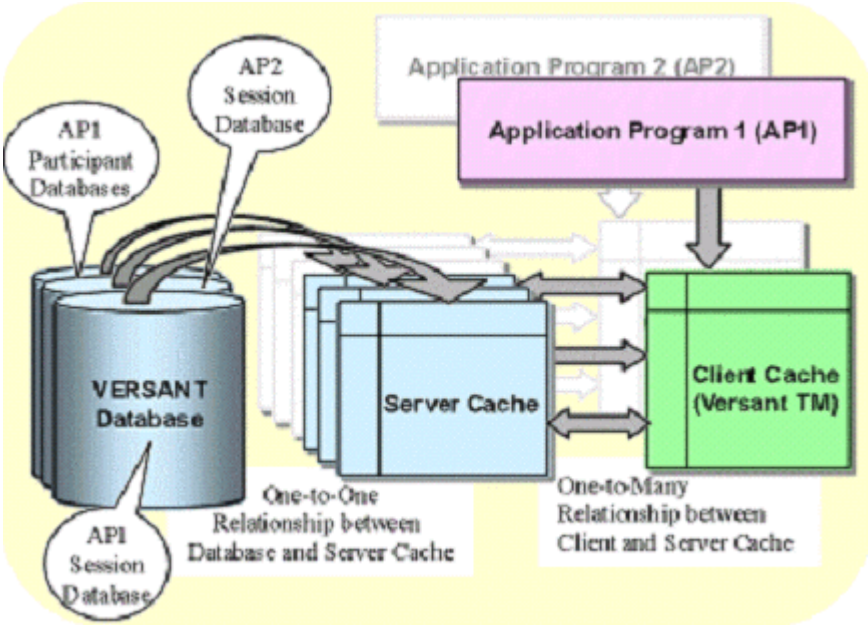
Versant system supports a number of different transaction protocol standards, including X/Open and its subset XA, with numerous C/Versant functions.

NOTE:- The Versant routines that support the X/Open Distributed Transaction Process (DTP) Model should not be used in conjunction with the standard Versant commit routines, such as `o_xact()` and `o_xactwithvstr()`. In other words, the XA open transaction model is incompatible with the Versant standard transaction model.

VERSANT TRANSACTION MODEL

The default Versant transaction model performs commits using a strict two phase commit protocol. This protocol uses:

- **Transaction manager** — a process running on the machine containing the session database
- **Coordinator transaction log** — a log containing coordinator transaction entries which is maintained by the session database
- **Participant transaction logs** — a log containing participant transaction entries, which is maintained by each participant database.



Commit and Rollback

The procedures for commits and rollbacks are similar:

1. The transaction manager sends create, update and delete information and an "are you ready?" message to the session database and to each connected database. This state is called "**Phase 1**".

2. When every database responds that it is ready, the transaction manager sends a "commit" message to all databases and writes to its coordinator transaction log. This is state is called "**Phase 2**".
3. Each database performs its part of the commit or rollback, writes to its participant transaction log, and sends a success or failure message to the transaction manager.
4. If the transaction manger receives a success message from each database, it deletes the transaction entry from its log and sends a message to each participating database to delete its transaction entry.

Recovery

If any database involved in a transaction crashes during a commit or rollback, the transaction entry will remain in the coordinator transaction log. When the crashed database restarts, the following will happen as part of the startup procedure:

1. The database will ask that the coordinator transaction log be examined.
2. For each Phase 2 entry in the transaction log, the transaction manager will attempt to finish the commit or rollback.
3. For each Phase 1 entry, the transaction manager will declare that the transaction involved is a "**zombie** ".

OPEN TRANSACTION CONCEPTS

An open transaction is a transaction controlled by an external process, called a Transaction Manager. In an open transaction, each phase of a commit or rollback to a Versant database can be controlled explicitly by a non-Versant application.

Versant system supports a number of different transaction protocol standards, including X/Open and its subset XA, with numerous C/Versant functions.

The flow of control in a generic transaction protocol system is something like the following (details differ from one protocol standard to another):

1. A `Begin_Work()` function starts the transaction. It registers the transaction with a Transaction Manager and creates a unique transaction identification number.
2. The application now invokes Resource Managers, reading and writing from the client machine or from databases, and sending requests to local and remote services.
3. When the Resource Managers get their first requests from a Transaction Manager, they join the associated transaction, telling the Transaction Manager that they want to participate.
4. After the system answers all requests, the transaction calls a `Commit_Work()` function. This causes the Transaction Manager to begin a two-phase commit protocol. In Phase 1, the Transaction Manager asks all of the Resource Managers that have joined the transaction if they think the transaction is consistent and complete.
5. If all the Resource Managers respond "yes," the transaction is a correct transformation. The Resource Managers record this fact in their individual transaction logs. The Resource Managers then release the locks on the messages, finish any other necessary functioning, and the transaction is complete.
6. If any Resource Manager votes "no" the commit fails, causing the Transaction Manager to orchestrate a rollback. The Transaction Manager reads the transaction's log, and for each log record, calls the Resource Manager that wrote the record, asking the Resource Manager to undo the transformation. Once all the undos are finished, the Transaction Manager calls all the Resource Managers that had joined the transaction to tell them that the transaction was aborted.
7. The Transaction Manager also handles transaction recoveries if a site or node fails. If a site fails, the transaction protocol system restarts all the resource managers. As part of the Transaction Managers' restart code, the Transaction Manager tells Resource Managers the outcome of all transactions that were in progress at the time of the failure. The Resource Manager updates the logs and reconstructs its state.
8. Typically, each site has its own Transaction Manager. This allows each site to operate independently from each other. If the transaction deals with remote sites during its execution, remote Transaction Managers handle their own part of the process.

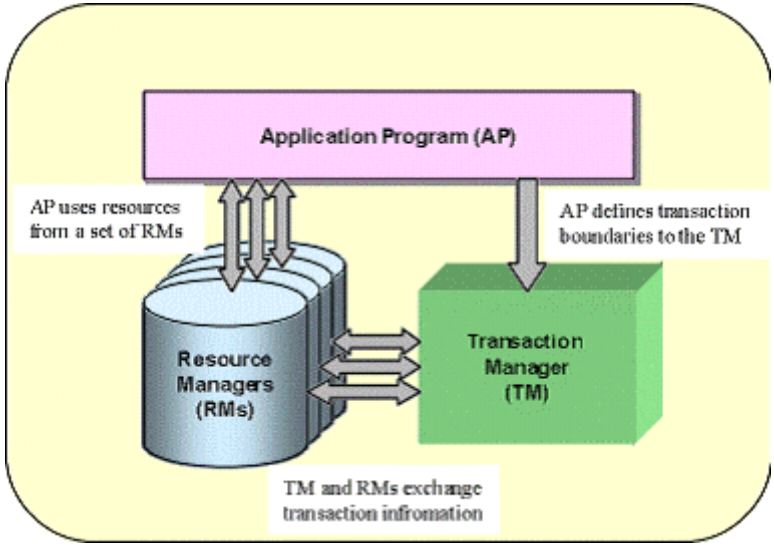
Versant provides the following functions that support open transactions:

Functions	Description
`o_beginxact()`	Start a transaction and assign a transaction identifier.
`o_commitp1()`	Commit phase 1.
`o_commitp2()`	Commit phase 2.
`o_deleteZombieList()`	Delete list of zombie transactions.
`o_flush()`	Flush specified objects.
`o_getparticipants()`	Get databases participating in transaction.
`o_getxid()`	Get transaction identifier.
`o_getZombieCount()`	Get zombie count.
`o_getZombieIndex()`	Get zombie index.
`o_getZombieList()`	Get zombie list.
`o_incZombieIndex()`	Increment zombie index.
`o_recover()`	Get list of zombie transactions.
`o_rollback()`	Rollback transaction.
`o_setflushobjects()`	Set flush options.
`o_setxid()`	Set transaction identifier.
`o_setZombieCount()`	Set zombie count.
`o_setZombieIndex()`	Set zombie index.
`o_unsetxid()`	Remove transaction identifier.
`o_xastatetransition()`	Is XA call consistent.

X/OPEN DISTRIBUTED TRANSACTION PROCESSING MODEL

The X/Open organization is a standards group that has developed a standard dealing with portability, called X/Open Distributed Transaction Processing (X/OpenDTP). This standard defines the concept of Resource Managers and Transaction Managers. Part of this definition is an interface for transaction management and resource management communications. This interface is called XA, and is a library of subroutines that allow Resource Managers to register with local Transaction Managers, Transaction Mangers to invoke Resource Managers at restart, and for transactions to begin, commit and abort.

The X/Open DTP Model as shown in the XA Specification looks something like this:



Please refer to the XA Specification and the X/Open Guide for more information.

The key concept is that the XA and X/Open transaction models open up the two-phase commit process so that each phase can be controlled explicitly by an external Transaction Manager.

VERSANT X/OPEN SUPPORT

To allow Versant databases to function as a Resource Manager in compliance with the XA Specification, new interface routines have been created. Following is description of Versant support of X/Open and XA:

Resource Manager Identifier

The Resource Manager Identifier is an integer assigned by the Transaction Manager to uniquely identify the called Resource Manager instance within the thread of control. The Transaction Manager passes the Resource Manager Identifier on calls to XA routines to identify the Resource Manager. This identifier remains constant until the Transaction Manager in this thread closes the Resource Manager.

Transaction Identifier

Since XA allows multiple processes within a server group to share the same transaction, Versant provides routines that can associate a Transaction Identifier (XID) to a process. The Transaction Identifier is used to define the scope of the transaction for commits and rollbacks and to aid record keeping in the event of a recovery operation.

The Resource Manager must be able to map the Transaction Identifier to the recoverable work it did for the corresponding branch. The Resource Manager may perform bitwise comparisons on the data components of an Transaction Identifier for the lengths specified in the Transaction Identifier structure. Most XA routines pass a pointer to the Transaction Identifier. These pointers are valid only for the duration of the call. If the Resource Manager needs to refer to the Transaction Identifier after it returns from the call, it must make a local copy before returning.

The `xa.h` header defines a public structure called a Transaction Identifier, `xid_t`, to identify a transaction branch. Resource Managers and Transaction Managers both use the Transaction Identifier structure.

Resource Manager Switch

The Transaction Manager administrator can add or remove a Resource Manager from the Distributed Transaction Processing system by controlling the set of Resource Managers linked to executable modules.

Each Resource Manager must provide a switch that gives the Transaction Manager access to the Resource Manager's XA routines. This lets the administrator change the set of Resource Managers linked with an executable module without having to recompile the application.

A different set of Resource Managers and their switches may be linked into each separate application executable module in the Distributed Transaction Processing system. Several instances of a Resource Manager can share the Resource Manager's switch structure.

A Resource Manager's switch is defined as a structure called `xa_switch_t`.

Resource Manager Identifier and Versant Sessions

There is a one-to-one mapping between Resource Manager Identifiers and Versant database sessions. The following shows how to translate a Resource Manager Identifier into a Versant session name.

```
sprintf(sessionName, "XA.Resource Manager Identifier:%0d", rmid);
```

Based on the assumption that Resource Manager Identifiers are unique, the session names should be unique also.

In the multi-process model, there is not much advantage to have this Resource Manager Identifier to Versant session binding. It may be useful, however, in the multi-thread model.

Transaction Identifiers and Versant Transactions

Internal transactions are created whenever a new external transaction is initiated, except for the first external transaction, which will attach to the transaction started when you begin a session.

When an external transaction terminates, all transaction states will be removed from the client even if there are objects in the cache. With this change, you must still be in a database session to be in a transaction, but you can now be in a session but not in a transaction. In the standard Versant transaction model, once a session started, you were always in a transaction.

To allow you to associate an external transaction with a Versant transaction, use the set transaction function, `o_setxid()`. Once the association is done, the Versant transaction will be considered an external transaction, and it will give up its role as a coordinator on deciding how to terminate the transaction. In other words, it will become passive and wait for an external request before recovering zombie transactions.

Only one Versant transaction is allowed to associate with an external transaction. If the server finds that the external transaction with the given Transaction Identifier is already associated with a Versant transaction, the Versant server will return the Versant transaction to the client. This allows an external transaction to attach to a Versant transaction multiple times providing that they are serialized (i.e., no more than one attachment at any instance).

To avoid inconsistencies, every external transaction will be allowed to use only one front end cache at all times.

Recovery

If there is an external Transaction Manager, the Versant transaction manager will not terminate a transaction from a coordinator transaction entry when it is in the zombie state. Instead, it will rely on the external Transaction Manager's decision. There will be no change in any other states.

Structures and Functions that Support X/Open

The following Versant structures and functions directly support X/Open transaction. They are available when you include the header file `xa.h`.

Data Structures	Description
`vsnt_xa_switch`	An instance of `xa_switch_t` that maps XA functions to Versant functions.
`xa_switch_t`	Information about function mapping.
`xid_t`	Information about a transaction.

Functions	Description
`vsnt_xa_close()`	Close a Resource Manager.
`vsnt_xa_commit()`	Commit a transaction.
`vsnt_xa_complete()`	Complete asynchronous operation.
`vsnt_xa_end()`	Notify when a thread of control ends.
`vsnt_xa_forget()`	Remove external transaction entry.
`vsnt_xa_open()`	Open a Resource Manager.
`vsnt_xa_prepare()`	Prepare for a commit.
`vsnt_xa_recover()`	Get list of prepared transactions.
`vsnt_xa_rollback()`	Rollback transaction.
`vsnt_xa_start()`	Notify when application is ready.

Versant/XA function mappings

vsnt_xa_open	xa_open()
vsnt_xa_close	xa_close()
vsnt_xa_start	xa_start()
vsnt_xa_end	xa_end()
vsnt_xa_rollback	xa_rollback()
vsnt_xa_prepare	xa_prepare()
vsnt_xa_commit	xa_commit()
vsnt_xa_recover	xa_recover()
vsnt_xa_forget	xa_forget()
vsnt_xa_complete	xa_complete()

For a more detailed description of Versant Structures and Functions that support X/Open, please refer to the chapter "Versant Open Transactions Reference" in the *C Versant Reference Manual*.

CHAPTER 18

Versant Queries

This Chapter explains the Versant query engine.

Query processing with VQL 6.0 and VQL7.0 are explained in separate chapters.

Following topics are covered:

- Search Queries
- Query Indexing
- Cursor Queries
- Advanced Queries

OVERVIEW

Versant Object Database provides an ability to query the database using a query language called VQL.

The Versant Query Language (VQL) is a "light-weight" language which enables you to find and manipulate objects in Versant databases using statements expressed as a string. You can then parse and execute the string using C/Versant or C++/Versant routines.

In some ways, VQL looks like SQL with some object extensions. For example, in VQL you can perform a path query and query on a `vstr`.

SEARCH QUERIES

This section gives details about the Search Queries.

Concepts

Query definition: A "query" examines a group of objects and returns those that satisfy the search condition.

Query types: The general term "query" conceptually refers both the routines that evaluate object values ("find") and routines that evaluate object identities using links ("get").

Routines	Details
find	Functions and methods that use search conditions to find objects and return links are called "search" or "find" routines. The term "query" is often used to refer to just "find" routines.
get	Functions and methods that use links to bring objects in memory are called "dereference," "retrieval," or "get" routines.

Versant evaluates "find" queries on the server machine containing the database to be searched and then sends links to the found objects to the application.

Versant evaluates "get" queries on the application machine and then retrieves the desired objects from any connected database.

Due to the client/server architecture for search queries, objects to be found by a "find" query must be located in a single database. However, you can send the same search query to numerous databases, store the object references (links) to objects in multiple databases in a collection, and then use a dereference method to retrieve them as desired.

Mechanisms

Find Object

The following functions find objects by using a search condition.

Function	Description
`o_defineevent()`	define event to find objects
`o_fetchcursor()`	find objects with cursor
`o_findarchivedobj()`	find objects in an archive
`o_pathselect()`	find objects of a class or vstr
`o_pathselectcursor()`	find objects with cursor
`o_select()`	find objects of a class

Logical database: If you want to perform a query in parallel over a number of databases, you can use the following functions to create and manage a logical database:

Function	Description
`o_addtologicaldb()`	Add a database to a logical database.
`o_deletelogicaldb()`	Delete a logical database.
`o_newlogicaldb()`	Create a logical database.
`o_removefromlogicaldb()`	Remove a database from a logical database.

Get Object

The following functions use handles, links, or pointers to retrieve objects. In a sense, they are not query functions, but rather they are "get" functions.

Function	Description
`o_getclosure()`	Get a group of objects
`o_greadobjs()`	Get a group of objects
`o_locateclass()`	Get a class object

`o_locateobj()`	Get an object
`o_refreshobj()`	Get a fresh object
`o_refreshobjs()`	Get fresh objects

Structure

Parameters

The functions that search for objects require specifications of various parameters. For example, the syntax of `o_select()` is:

```
o_vstr o_select(
 o_clsname    classname,
 o_dbname     dbname,
 o_bool       iflag,
 o_lockmode   lockmode,
 o_vstr       predicate );
```

Let's examine this function conceptually.

Return value

A query returns links to objects that satisfy the query: the objects themselves are not passed from the server database to the client application. To actually use a returned object, you must use a link to bring it into memory with a "get" function, such as `o_locateobj()` or `o_greadobj()`.

Starting point objects

A query evaluates a group of objects known as the "starting point objects." The starting point objects may be a class of objects, a class and its subclasses, or a specific group of objects. After evaluating the starting point objects, a query can expand outwards to examine attribute values in embedded objects and linked objects.

In the above, the starting point objects are specified by the following parameters:

Parameter	Description
`classname`	Name of a class. Other functions, such as o_pathselect(), allow you to query over a specific group of objects.
`dbname`	Name of a database. Versant evaluates "find" queries on the server machine containing the database to be searched, so the starting point objects must be located in a single database. However, you can send the same search query to numerous databases, store the returned links in a vstr, and then use a "get" function to retrieve them as desired. ("Get" functions are evaluated on a client machine and can fetch objects from any connected database.)
`iflag`	Inheritance specifier: specify TRUE to evaluate and return instances of both the class and its subclasses or specify FALSE to evaluate and return only instances of the class.

Lock mode

By definition, a select operation returns a vstr of links to the objects that satisfy the query conditions. In the above, the lock that you specify in the lockmode parameter will be placed on the class object for the returned objects.

In a query, you will usually want to set an intention read lock on a class, which prevents the definition of the class from changing while still allowing other users to use instances of the class in a normal manner.

For more information refer to “Locks and Transactions” on page 92, in "Chapter 5 - Locks and Optimistic Locking".

Predicate

A predicate is a vstr of pointers to search conditions called "predicate terms". To express how predicate terms are to be collectively applied, they are concatenated into a predicate using logical, boolean operators. Allowed boolean operators are logical AND, OR, and NOT.

The real action of a find function happens in the predicate term. The remainder of this section discusses predicates and predicate terms.

Predicate Term

In C/Versant, predicate terms are structures of type `o_predterm`. The syntax of the `o_predterm` data type is:

```
typedef struct o_predterm {
    o_attrname      attribute;
    o_opertype      oper;
    o_bufdesc       key;
    o_typetype      keytype;
    o_u4b           flags;
}   o_predterm;
```

A predicate term specifies:

- An evaluation attribute
- A comparison operator
- A key value
- An optional logical path statement.

Let's examine this structure conceptually.

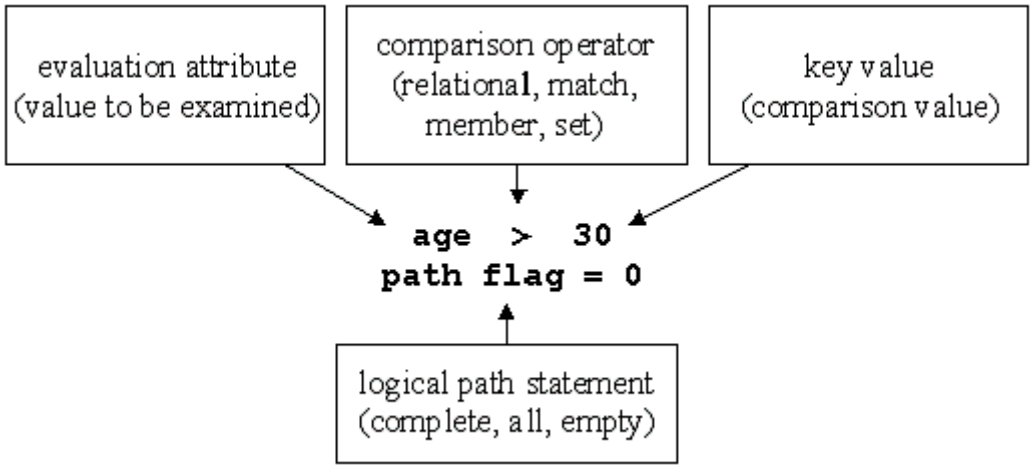
Query evaluation attribute (the `attribute` parameter): The "evaluation attribute" is the attribute whose value is to be examined. Evaluation attributes must be specified in a precise, non-ambiguous way that is understandable to a database.

Query comparison operator (the `oper` parameter): A "comparison operator" determines how an attribute value is compared to a key value. A comparison operator can be a relational operator for comparing scalar values, a pattern matching operator for comparing strings, an "is-a" operator for determining class membership, or a set operator.

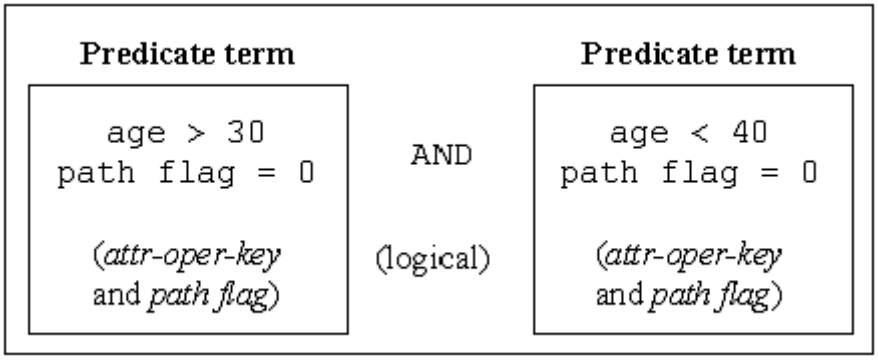
Query key value (the `key` and `keytype` parameters): A "key value" is the value to which a specified attribute is compared. A key value can be a link, links, or a value for any type of attribute allowed in a query.

Query logical path statement (the `flags` parameter): A "logical path statement" determines how to handle cases where leaf attributes (the attribute being evaluated) are null or where there are multiple leaf attributes for a particular starting point object. By default, a starting point object is returned only if a valid leaf attribute is found and the value found in the leaf attribute satisfies the specified search condition.

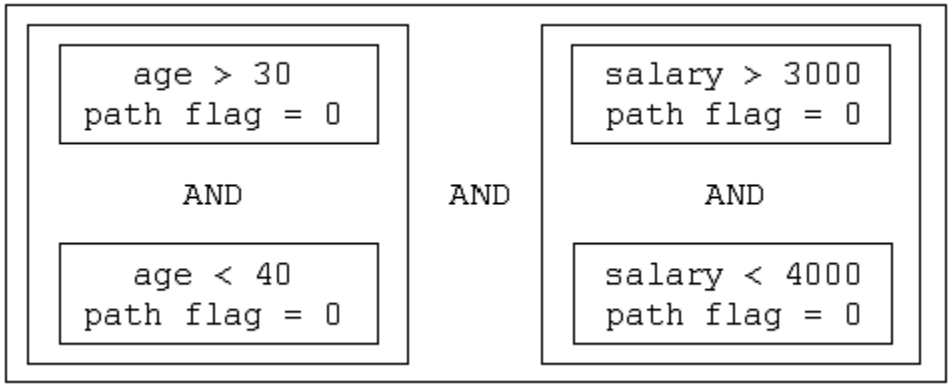
Predicate Term: Conceptually, a predicate term looks like this:



Predicates: Predicate terms can be used alone as a predicate or concatenated to form a multi-part predicate.



Also, predicate terms can be concatenated with other predicates.



The following discusses predicate term attributes, comparison operators, and path statements in detail.

Evaluation and Key Attributes

An "evaluation attribute" is the attribute whose value is to be examined. A "key value" is the value to which a specified attribute is compared.

The following explains the attribute types allowed and how to specify an evaluation attribute.

Attribute Types Allowed

Query attribute types allowed: In search functions, you can use the following types of attributes when specifying a search condition:

Description	Attribute

Elementary data types with a repeat count of 1.	`char`
	`o_1b`
	`o_2b`
	`o_4b`
	`o_8b`
	`o_double`
	`o_float`
	`o_u1b`
	`o_u2b`
	`o_u4b`
	`o_u8b`

Fixed size arrays of elementary Versant data type.
For example: `char firstname[20]`
	`fixed array of char`
	`fixed array of o_2b`
	`fixed array of o_u2b`

Vstrs of elementary Versant data type.	`vstr of char`
	`vstr of o_1b`
	`vstr of o_u1b`

Links of type `o_object`.	`o_object`
Vstrs containing links.	`vstr of o_object`

Fixed size arrays of links. For example: `o_object names[20]`	`Fixed array of o_object`

Date and time types.	`o_date`
	`o_interval`
	`o_time`
	`o_timestamp`

Attribute Types NOT Allowed

You **cannot** use the following types of attributes in a search method:

Attribute	Description
Do **not** use enum	You should not use enum, as it is not portable: some machines implement enum as `o_2b`, while others implement enum as `o_4b`.
Do **not** use an embedded class or struct with a fixed length	Do not use an embedded class or struct with a fixed length that embeds another fixed array, class or struct.
Do **not** use vstrs of class or struct	Do not use vstrs of class or struct types. For example, you cannot use `Vstr<Employee>`.

Attribute Specification by Name

You can specify an evaluation attribute by name or by path. If you specify an attribute by name, you must use a special, non-ambiguous, database syntax for the name.

Precise attribute names are needed, because a derived class could conceivably inherit attributes with the same name from numerous superclasses along one or more inheritance paths. Even if there is no initial ambiguity, classes may later be created which do cause ambiguity.

Following are attribute name rules required by Versant for each type of attribute that may be used to query or index.

Name of an elemental attribute

The attribute name in the database should be used. For example, in a C++ application, the following class is defined:

```
Class_name public PObject {
   o_4b attribute_name;
};
```

In the database, the name of the attribute becomes:

```
Class_name::attribute_name
```

This form should be used even when an attribute is used in a derived class.

For example, if attributes id and code are in class Base and attribute value is in class Derived, then the database names for these attributes are:

```
Base::id
Base::code
Derived::value
```

In the following sections, we assume the schema is generated for a C++ application.

Name of a link attribute

The database name for a link attribute is:

```
classname::attribute_name
```

For example, suppose that Employee has a `worksIn` attribute that points to a Department object. The database name for the attribute is:

```
Employee::worksIn
```

This form should be used even when an attribute is used in a derived class.

Name of a fixed array attribute

Fixed arrays of `char, o_1b, o_u1b,` and `o_object` with two or more fixed length dimensions are flattened into discrete attributes, except for the last dimension, which is given to the database as a repeat count for each database attribute. The attribute names used in the database include the subscript inside of brackets.

The general form is:

```
class::attribute_name[0]
class::attribute_name[1]
class::attribute_name[...]
```

```
class::attribute_name[n-1]
```

This form should be used even when an attribute is used in a derived class.

Name of a vstr attribute

The database name of an attribute that is a vstr of `char, o_1b, o_u1b` or `o_object` is:

```
classname::attribute_name
```

For example, if Department has an attribute named employees to hold department members that is a vstr of `o_object`, then the database name for the attribute is:

```
Department::employees
```

Name of a class or struct attribute

Embedded attributes of struct or class data types are flattened into simpler individual attributes.

For example, suppose that `embeddedObject` is an attribute of `ClassA` and is of type `ClassB`. Also suppose that `ClassB` has an attribute named `attributeB`. Then the database name of `attributeB` in `embeddedObject` is:

```
ClassA::embeddedObject.ClassB::attributeB
```

The name `ClassA::embeddedObject.ClassB::attributeB` must be used to refer to the `embeddedObject` attribute wherever it appears, whether in `ClassA` or in a class derived from `ClassA`.

The attribute `attributeB` can itself be of any type, including an embedded class or embedded struct type.

For example, if a struct Date and struct Base are specified as:

```
struct Date {
   int  day;
   Base b;
};
struct Base {
   int  id;
   int  code;
};
```

Then the database attribute names are:

```
Date::day
Date::b.Base::id
Date::b.Base::code
```

Query Attribute Names Not Allowed

You cannot query on an attribute whose name contains an ASCII TAB character (decimal value 9).

Attribute Specification by Path

You can specify an evaluation attribute with a path name rather than an attribute name. As with names, you can use relational, match, membership, and set operators to make comparisons between a key value and an attribute specified with a path.

In ODMG, this is called a "path expression." By definition, a "path expression" specifies a path through several interrelated objects to access a destination object.

The general syntax for a path expression is a listing of the attribute names on the path to the attribute to be examined, with the names on the path delimited with \t.

For example, suppose that a class Person has a links attribute garages whose domain is class Garage. Suppose that class Garage also has a link attribute cars whose domain is class Car, and that class Car has an attribute colors.

Now suppose that we want to search over the class Person for people with cars of a certain color. Then, to specify the attribute colors in a predicate, we can specify it with the path from Person to colors using \t:

```
Person::garages\tGarage::cars\tCar::colors
```

This is conceptually similar to specifying the colors attribute by name as:

```
Person::garages->Garage::cars->Car::colors
```

The last name in the attribute path, in this case colors, is called the "leaf name." The intermediate names, in this case garages and cars, are called "non-leaf" names.

When you use a path name for an attribute to be examined:

- All objects involved must be in the same database
- All non-leaf attributes must be a link, link array, or link vstr.
- The comparison operator must be a relational operator or a set operator.
- The attribute path can include any number of steps to the final attribute.

- The starting point objects can be either the objects in a class or the objects in a link vstr or link array.
- Once the starting point objects have been returned, you can also return objects linked to the starting point objects to a specified level of closure. The following of links to a particular level of closure is unrelated to the following of a path to a leaf attribute.

Comparison Operators

A "comparison operator" determines how an attribute value is compared to a key value. A comparison operator can be a relational operator for comparing scalar values, a pattern matching operator for comparing strings, an "`is-a`" operator for determining class membership, or a set operator for comparing sets.

Relational operators

Compare scalar values

Attribute	Description
Evaluation attribute	a scalar value
Key attribute	a scalar value

Operator	Evaluation attribute
`O_EQ`	Equal to key value
`O_NE`	Not equal to key value
`O_LT`	Less than key value
`O_LE`	Less than or equal to key value
`O_GT`	Greater than key value
`O_GE`	Greater than or equal to key value

Behavior of Relational Operators on Special Values

There are some special key values for certain Versant data types on which queries using relational operators can exhibit a certain specific behavior. These special keys can be used to denote null values and can therefore be used as markers to indicate a complete lack of value for that particular

attribute. The following table summarizes the special values and the supported data types that can take on these values:

Data Type	Special Value
o_float	IEEE single precision NaN
o_double	IEEE double precision NaN
o_date	NULL_DATE
o_time	NULL_TIME
o_timestamp	NULL_TIMESTAMP

IEEE NaN (Not a Number) can be represented by multiple bit patterns. Versant will not distinguish between these bit-patters and will treat them all as NaN representations.

When an attribute of a persistent class belonging to one of the above data types is initialized with its corresponding special value (as indicated by the table above), Versant will treat these attributes as not initialized and the query result depends on the relational operator / key value combination being used.

The following table explains the behavior when special key values are used in predicate terms or when objects having special key values for the evaluation attribute are encountered during a particular query evaluation.

Key Value	Relational Operator	Expected Behavior
Normal	O_EQ / O_NE / O_LT / O_LE / O_GT / O_GE	Objects having a special value for that particular attribute will be ignored and will never be a part of the result set.
Special	O_EQ	This will return only objects whose evaluation attribute contains a special value. This could be used to find out all objects in which the evaluation attribute is not initialized.

Special	O_NE	Objects having a special value for the evaluation attribute in question will be ignored during the query evaluation and will never be a part of the result set. This could be used to find out all objects whose evaluation attribute is initialized.
Special	O_LT / O_LE / O_GT / O_GE	Will throw error "5423 - QRY_OP_NOT_ALLOWED" as this relation is not defined

This behavior of queries and special values is applicable only when using Versant 6.0 APIs and is currently not supported for VQL 7.0 APIs

Indexes on class attributes that can contain these special values are allowed.

NOTE:- When a null predicate is used to select all objects of a class then there will be no filtering performed and the result set will even include objects whose attributes contain special values (if such objects exist for the class).

String operators

Compare strings

Attribute	Description
Evaluation attribute	a string
Key attribute	a string, possibly containing wildcards

Operator	Evaluation attribute:
O_MATCHES	Matches key string
O_NOT_MATCHES	Does not match key string

Wildcards	Match
*	Any sequence, including an empty sequence.
?	Any single character
[chars]	Any character in set.

`[x-y]`	Any character between x and y, inclusive.
`\x`	Single character x (avoid wildcard interpretation of * ? [ ] in a pattern)

Class operators

Compare class memberships

These operators test whether an object is an instance of a given class, but they do not test whether the object is an instance of a subclass.

Attribute	Description
Evaluation attribute	an instance object
Key attribute	class name of a class object

Operator	Evaluation attribute
`O_ISA_EXACT`	Is a member of key class. A null link returns false if flags is set to 0 and true if flags is set to `O_EMPTY_TRUE`.
`O_NOT_ISA_EXACT`	Is not a member of key class. A null link returns true if flags is set to 0 and false if flags is set to `O_EMPTY_TRUE`.

Set operators

Compare sets of objects or elementary data types

Attribute	Description
Evaluation attribute	One or a set of links or elements of elementary data types.
Key attribute	One or a set of links or elements of elementary data types.

Operator	Evaluation attribute:
O_INTERSECT	Contains at least one link that is in the key attribute (intersection of links is not empty).
O_NOT_INTERSECT	Contains no link that is in the key links (intersection of links is empty).
O_SUBSET_OF	Contains links that are all in the key attribute (subset is true).
O_NOT_SUBSET_OF	Contains at least one link that is not in the key attribute (subset is false).
O_SUPERSET_OF	Contains all links that are in the key attribute (superset is true).
O_NOT_SUPERSET_OF	Does not contains all links that are in the key attribute (superset is false).
O_EQUIVALENT_SETS	Contains the same set of links that are in the key attribute (equivalent is true).
O_NOT_EQUIVALENT_SETS	Does not contain the same set of links that are in the key attribute (equivalent is false).

Usage Notes

- By definition, a "set query" compares sets of objects.
- In a set query, the evaluation operator and the key attribute can be either a link or a set of links. For example, operator `O_INTERSECT` performs inquiries on those objects linking to some specific objects. In reverse, operator `O_NOT_INTERSECT` performs inquiries those not linking to any of the given objects.There is no restriction on the number of links in the key attribute, except available heap space on both the client and server machines at the time the query is executed.
- A sequence of queries provides the means of eliminating incongruent search outcomes as a result of ambiguous expressions caused by a multiple path query. For example, operator `O_INTERSECT` checks whether found objects link to a set of specific objects.
- To improve the performance of a query involving links, you can create a `btree` index on an attribute containing links.

  A query with path expressions uses a Depth-First-Fetch approach to evaluate each path expression. Its performance can becomes an issue for a large starting class with complex

relationship among classes along the navigation path as it requires an exhaustive traversal defined on the links from the given path expressions.

Example

For the following example, assume that we have the following three classes and attributes:

Classes	Attributes
Broker	dealers(linkvstr)
Dealer	vehicles(linkvstr)
Vehicle	color, make

Each broker works for some dealers. Each dealer has certain inventory of vehicles. Each vehicle has attributes of color and make.

Assume the following objects in these classes in a database:

Broker	Dealer	Vehicle
s1:{d1}	d1:{v1, v4}	v1: red Ford
s2:{d3, d2}	d2:{v2, v3, v6}	v2: blue VW
s3:{}	d3:{v2, v4, v5}	v3: white GM
s4:{d2}		v4: blue Honda
		v5: red Honda
		v6: red Honda

Suppose that we want to select those brokers who sell a "red Honda".

At first, it seems as if this request can be described in the Versant Query Language as:

```
SELECT  selfoid
FROM    Broker
WHERE   dealers->vehicles->color = "red" AND
        dealers->vehicles->make  = "Honda"
```

But the above query returns a broker who does not sell a red Honda:

Broker	Description

s1	This broker does not sell a red Honda, but is returned because she sells a red Ford and a blue Honda ("a red car" and a "Honda").
s2	This broker does sell a red Honda.
s4	This broker does sell a red Honda.

We can easily construct a sequence of queries that will correctly get only brokers who sell red Hondas by using set operators. It will become quite intricate if no set operators are used.

Here is one way to get only brokers who sell red Hondas, using reverse traversal and reverse joins:

1. First, get all red Hondas.

   Let Vehicles_red_Honda be the result of the following query, which is a collection of the logical object identifiers of all red Hondas in inventory from all dealers:

   ```
   SELECT  selfoid
   FROM    Vehicle
   WHERE   color = "red" AND make = "Honda"
   ```

2. Next, get all dealers who have a red Honda.

   Let `Dealers_red_Honda` be the result of the following query, which is a collection of the logical object identifiers of all dealers who have "red Honda" in their inventory:

   ```
   SELECT  selfoid
   FROM    Dealer
   WHERE   vehicles INTERSECT Vehicles_red_Honda
   ```

   Note that we use operator INTERSECT to find all dealers having a vehicle in Vehicles_red_Honda in their inventory.

3. Finally, get the brokers who represent dealers with red Hondas.

   ```
   SELECT  selfoid
   FROM    Broker
   WHERE   dealers INTERSECT Dealers_red_Honda
   ```

Logical operators

When multiple predicate terms and/or predicates are used in a search query, they are concatenated with logical operators. The following boolean operators can be used to logically join predicate terms and predicates:

Operators	Description

O_AND	logical AND
O_OR	logical OR
O_NOT	logical NOT

Per precedence rules, operator O_NOT binds tightest and O_OR binds weakest.

Logical Paths

When you specify an attribute path, the number of object paths searched using intermediate arrays or vstrs of links can expand to any number. This means that a search with numerous link arrays as intermediate steps can involve a large number of paths and a large number of objects.

A particular path is said to be "complete" when all of the following are true:

- The path can be followed all the way from a starting point object to an evaluation attribute.
- The data type of the evaluation attribute is such that it can be compared with the key value specified in the predicate.
- The comparison operator specified is appropriate for comparing the data types of the evaluation attribute and the key value.

There are many reasons a particular path can be "incomplete."

For example, the following things can go wrong when a query attempts to follow a particular actual object path.

- A link in a non-leaf attribute may be NULL.
- A link may point to an object in another database.
- A link may point to an object that has been deleted.
- A leaf or non-leaf object might not have the attribute named.
- The operator may not be valid for the leaf attribute.
- The attribute may not be the same type as the key value.

Logical path statements are a way of being more precise about search conditions. When you consider paths, a query can represent both a value test and a path test.

A query's value test occurs in its predicate. A predicate is said to be "true" if an attribute has a value that meets the search condition expressed by the comparison operator and the key value. For example, suppose that Henry has a red Porsche. If you ask for all persons with a red Porsche ("Porsche = red"), then for Henry the predicate evaluates true.

A query's path test occurs in its logical path statement. Most of the time you can ignore the path test, but when you are phrasing some kinds of queries, you may need to consider paths. For example, if you have no Porsches, are all of your Porsches red? In this case, the answer depends upon your logical path statement.

Versant allows you to specify what happens when paths are complete or incomplete. The following principles form the basis of the logical evaluation of paths:

- Incomplete paths are ignored.
- If complete paths = `0`, then you get the starting point object only if you also ask for "empty succeeds."
- If complete paths = `1`, then you get the starting point object only if your predicate is true.
- If complete paths > `1`, then, by default, you get the starting point object if at least one predicate is true.
- If complete paths > `1` and you also ask that "all paths" be evaluated, then you get the starting point object only if all predicates are evaluated as true.

In C/Versant, logical path conditions are specified as a parameter.

Following are options for logical path conditions:

Logical Path Options	Description
`0 ("exists")`	Return a starting point object if there exists one complete path for which the predicate evaluates true. This option is the default. Notice that this option ignores incomplete paths. Also ignored are complete paths whose predicate evaluates false. The only test is whether there is one complete path whose predicate is true.
`0_ALL_PATHS`	Return a starting point object if there is at least one complete path and if all complete paths have a predicate that evaluates true. This option ignores incomplete paths. It does not ignore complete paths whose predicate is false.

O_EMPTY_TRUE	Return a starting point object if there exists one complete path with a true predicate OR if no paths are complete. This option is the same as "exists" except that you also get the starting point object if there are no complete paths. For example, if you ask for people with red Porsches, you also get people with no cars.

You can combine the above flag options with the | operator. A useful combination is the following.

Logical Path Option	Description
O_ALL_PATHS | O_EMPTY_TRUE	Return a starting point object if there are no complete paths ("empty succeeds") OR if all complete paths have a predicate which evaluates true.

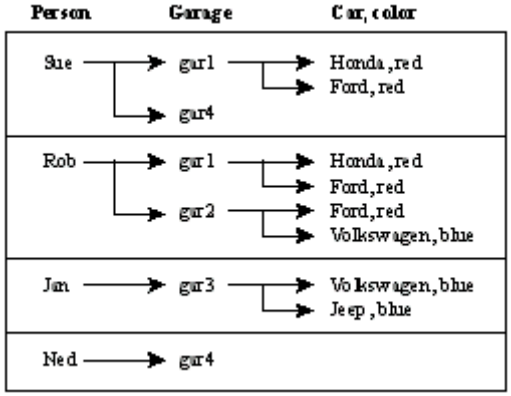
Exceptions

- If recursion has been turned off, then the first attribute in the attribute path must exist on the class which is selected on, or else the query returns an error code regardless of option selected. If the first attribute in the path exists, then the remainder of the query is performed per the option selected.
- If there is only one attribute name in the attribute path, then in all cases the comparison operator must apply to the data type found there, or else the query returns an error code.

Example

For example, consider the example of persons who have garages which have cars which have colors. In this example, suppose that a person can be associated with more than one garage, and that a garage can be associated with more than one car.

Suppose the following cases.



The cases are:

Person	Colors	Complete paths
Sue	red	Some (2 out of 3)
Rob	red, blue	All
Jan	blue	All
Ned	none	None

In the following, we will test whether a person has a red car.

Test for red

If you evaluate for red (= "red"), the results are:

Test — There is at least one complete path to red.

Path condition = 0

Results:

Person	Result	Description
Sue	Returned	One path to red.
Rob	Returned	One path to red.
Jan	Not returned	No path to red.

Ned	Not returned	No path to red.

Test — There is at least one complete path, and all complete paths are red.

Path condition = `O_ALL_PATHS`

Results:

Person	Result	Description
Sue	Returned	Two complete paths, and all complete paths are red.
Rob	Not returned	Four complete paths, but one complete path is not red.
Jan	Not returned	Two complete paths, but all complete paths are not red.
Ned	Not returned	There is no complete path.

Test — There is at least one path to red OR all paths are not complete.

Path condition = `O_EMPTY_TRUE`

Results:

Person	Result	Description
Sue	Returned	One path to red.
Rob	Returned	One path to red.
Jan	Not returned	No path to red and at least one path is complete.
Ned	Returned	There are no complete paths.

Test — All complete paths are red OR there are no complete paths.

Path condition = `O_ALL_PATHS | O_EMPTY_TRUE`

Results:

Person	Result	Description
Sue	Returned	All complete paths are red.

Rob	Not returned	One complete path is not red.
Jan	Not returned	One complete path is not red.
Ned	Returned	There are no complete paths.

Test for NOT red

If you evaluate for not red (!= "red"), the results are:

Test — There is at least one complete path to not red.

Path condition = `0`

Results:

Person	Result	Description
Sue	Not returned	No path to not red.
Rob	Returned	One path to blue.
Jan	Returned	One path to blue.
Ned	Not returned	No path to not red.

Test — There is a least one complete path, and all complete paths are not red.

Path condition = `0_ALL_PATHS`

Results:

Person	Result	Description
Sue	Not returned	All complete paths are red.
Rob	Not returned	One path to red.
Jan	Returned	Two complete paths, and all complete paths are not red.
Ned	Not returned	There is no complete path.

Test — There is at least one path to not red OR all paths are not complete.

Path condition = `0_EMPTY_TRUE`

Results:

Person	Result	Description
Sue	Not returned	Two complete paths, but no path to not red.
Rob	Returned	One complete path to not red.
Jan	Returned	One complete path to not red.
Ned	Returned	All paths are not complete.

Test — All complete paths are not red OR there are no complete paths.

Path condition = `O_ALL_PATHS | O_EMPTY_TRUE`

Results:

Person	Result	Description
Sue	Not returned	One complete path is red.
Rob	Not returned	One complete path is red.
Jan	Returned	All complete paths are not red.
Ned	Returned	There are no complete paths.

QUERY INDEXING

Concepts

Index purpose

Indexes allow routines that query, create, and update objects to have quick access to the values of a particular attribute of a class.

Query routines can use indexes to filter objects so that only objects of interest within a specified range are fetched from disk. This can improve query performance in many situations.

Create and update routines can use indexes to enforce uniqueness constraints on attribute values.

Index structure

An index is set on a single attribute of a class and affects all instances of the class. It is a list of pairs consisting of key values and associated object identifiers.

There are two kinds of index structures: b-tree and hash table.

Both types of structures maintain a separate storage area containing attribute values, only their organization is different. The type of structure that you want to use depends upon the types of queries you will be making.

Index constraints

There are two kinds of index constraints: "normal" and "unique."

A "normal" index places no constraints on attribute values. A "unique" index requires that each instance of a class has a value in the indexed attribute that is unique with respect to the values in that attribute in all other instances of the class.

If you want to create a unique index, you can use either a b-tree or hash table structure. The only difference is that when you create the index, you specify that you want unique values.

General Index Rules

Indexes do not have names.

Indexes are maintained automatically.

Once created, indexes are maintained automatically, which means that they may somewhat slow the creation, updating, and deletion of instances.

Indexes must be committed.

The actual creation of an index does not occur until the current transaction is committed.

Attributes can have two indexes.

An attribute can have up to two indexes, a b-tree index (normal or unique) and a hash table index (normal or unique).

Each database has its own indexes.

Each database has its own set of indexes.

When you migrate an object to a database in which its class is not defined, the class definition, index definition, and index behavior are also migrated.

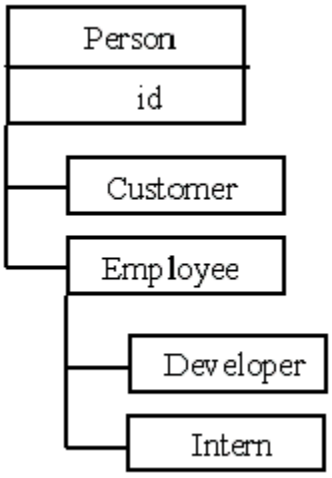
When you migrate an object to a database in which its class is already defined, the migration will fail if the target database does not have the same class definition, index definition, and index behavior as the source database.

An index is set on one attribute in one class.

When you create an index, that index is created on an attribute of a class and not on an attribute in any other class. In other words, indexes are not inherited. To index subclass attributes, you need to specifically set indexes on each subclass.

This is important to remember if you are using multiple inheritance and/or are querying over members of a class and its subclasses. The general caution is that if you use multiple inheritance but treat inheriting classes separately, then you may get anomalies.

For example, suppose that you have the following schema.



In the above, `class Person` has an `id` attribute that is inherited by several subclasses. If you set an index on `id` in `Person` and then perform a query on `Employee`, then the index on `Person` is not used. Similarly, if you set a unique index on `id` in `Employee`, then when you create a new `Developer`, a uniqueness check on `id` is not made.

Indexes should be consistent.

Since indexes are set only on one attribute in one class, you should set corresponding indexes on all classes that have an inheritance relationship. For example, in the above, if you set an index on `Person`, then set the same index on `Customer`, `Employee`, `Developer`, and `Intern`.

Cursor queries use indexes in the same way that other queries do.

Attribute Rules

Attributes that Can be Indexed

You can index on the following types of attributes:

- Elementary types with a repeat count of 1: `char, o_u1b, o_1b, o_u2b, o_2b, o_u4b, o_4b, o_u8b, o_8b, o_float, o_double, enum, o_object`

NOTE:- A unique BTree index cannot be created on `o_object` data types.

- In B-Tree Index, the following elementary types with a repeat count: char, o_u1b, o_1b, o_u2b, o_2b, o_u4b, o_4b, o_u8b, o_8b, o_float, o_double, enum and o_object

  For example: char firstname[20]
- In B-Tree index, the Vstrs of the following elementary types : char, o_u1b, o_1b, o_u2b, o_2b, o_u4b, o_4b, o_u8b, o_8b, o_float, o_double, enum and o_object
- Vstrs of all fundamental elementary data types except “char” cannot be Hash indexed
- Date and time types: `o_date, o_interval, o_time`, and `o_timestamp`.

Attributes that Cannot be Indexed

You cannot index on the following types of attributes:

- In Hash Index, the following elementary fixed arrays with a repeat count, except for o_1b, o_u1b, and char.

  For example: o_4b pixel[5]
- Embedded struct with a fixed length.
- Vstrs of elemental types except `o_1b`, `o_u1b`, and `char`.
- Vstrs of objects of struct types.
- Lists of type `o_list`.

The indexed attribute can have a maximum size of 2k bytes.

Mechanisms

Index Functions

Functions	Description
`o_createindex()`	Create a b-tree or hash index, either normal or unique.
`o_deleteindex()`	Delete a b-tree or hash index, either normal or unique.

Index Utility

The `dbtool` utility has the following index options:

Index Options	Description
`-index -create -btree`	Create b-tree.
`-index -create -btree -unique`	Create unique b-tree.
`-index -create -hash`	Create hash table.
`-index -create -hash -unique`	Create unique hash table.
`-index -delete -hash`	Delete hash table.
`-index -delete -btree`	Delete b-tree.
`-index -info -list [C[A]]`	Print index information.

Index Options

In routines that create and delete indexes, following are options for the type of index.

Index Options	Description
`O_IT_BTREE`	b-tree
`O_IT_UNIQUE_BTREE`	unique b-tree
`O_IT_HASH`	hash
`O_IT_UNIQUE_HASH`	unique hash

Query Costs

One of the costs of a query is fetching data pages from disk so that the server process can evaluate objects. Once objects are in memory, applying the predicate happens quickly.

If you use no indexes, a query will fetch all data pages for a class and evaluate all objects in a single pass through the data pages. This approach optimizes the disk fetch and is appropriate if you want to return a relatively large proportion of the instances of a class.

If you have a large number of objects and want to return, say, a quarter or less of all instances, then an index can improve query performance. The tradeoff is improved query speed versus index overhead, as indexes are automatically updated when objects are added, changed, or dropped.

An index might logically be set on attributes related to domains, subclasses, and links for queries that evaluate substructures. Alternately, they may be set on a first-level attribute used frequently as a search condition or on logical object identifiers.

A b-tree index is useful for value-range comparisons, and some “set query”. The hash index is useful for direct comparisons of values.

A b-tree index will return objects in ascending order; to access objects in descending order, access the returned vstr or array from its end rather than beginning. However, if a query evaluates a class and its subclasses, objects are returned in ascending order in each subclass, which means that the set of all objects are not necessarily in ascending order.

If you will only be making equality comparisons, generally a hash table is better; otherwise, use a b-tree index.

Following is additional detail on how indexes are used.

Indexable Predicate Term

The system optimizer will automatically use an index in a query if a predicate term is "indexable."

A predicate term is indexable if all of the following are true.

- The evaluation attribute has an index on it.
- The evaluation attribute is not expressed in terms of a path.
- The comparison operator is appropriate to the type of index.

The following types of comparison operators can use the following types of indexes:

Comparison Operator	Usage	Description
O_EQ	Scalar equal to	Can use either a hash or b-tree index. If there are both kinds of indexes on the evaluation operator, the hash index will be used.
O_NE	Scalar not equal to	Does not use an index.
O_GT,O_GE,O_LT,O_LE	Scalar relative to	Can use a b-tree index.
O_MATCHES	String equal to	Can use a b-tree index, if the key string to be compared does not start with "*" or "?".
O_NOT_MATCHES	String not equal to	Does not use an index.
O_INTERSECT	Set intersection	Can use a b-tree index.
O_SUPERSET_OF	Set superset	Can use a b-tree index.
O_EQUIVALENT_SET	Set equal to	Can use a b-tree index.
O_NOT_INTERSECT	Set not intersection	Does not use an index.
O_SUBSET_OF	Set subset	Does not use an index.
O_NOT_SUPERSET_OF	Set not superset	Does not use an index.
O_NOT_SUBSET_OF	Set not subset	Does not use an index.
O_NOT_EQUIVALENT_SET	Set not equal to	Does not use an index.
O_ISA_EXACT	Class is same as	Not relevant.
O_NOT_ISA_EXACT	Class is not same as	Not relevant.
O_IS_EMPTY		Can use a b-tree index
O_IS_NOT_EMPTY		Does not use an index

Query Evaluation and B-tree Indexes

A query can use a b-tree index in an exact match or range predicate. When a b-tree index exists, each object has an entry on a leaf page of the b-tree. These leaf pages are sorted on the b-tree attribute and doubly linked.

Exact match predicate

By definition, an exact match predicate has the form (`attribute == key_value`).

A query with an exact match predicate will traverse a b-tree index on an attribute starting from the root page down through interior node pages, if any, to a leaf page with an entry for the key value.

If the key value cannot be located, then no object satisfies the predicate, and the query is finished.

If an entry for the key value is found, then the object is retrieved into server memory from the corresponding data page whose location is stored inside the entry for the key value. If there are other predicate terms, the retrieved object is further evaluated and returned to the application if it satisfies all predicate terms.

If the index is a unique index, the query is finished. If the index is not unique, then all objects with the key value are retrieved, evaluated, and returned if they satisfy all predicate terms.

Range predicate

By definition, a range predicate has the form (`attribute >= lower_bound and attribute <= upper_bound`). An exact match predicate is a special form of a range predicate where the `lower_bound` is the same as the `upper_bound`.

Like the exact match case, a query with a range predicate traverses a b-tree index searching for an entry on a leaf page whose key value is greater than or equal to the lower bound. As in the exact match case, as each entry that matches the range predicate is found, its corresponding object is retrieved, further evaluated, and returned if it satisfies all predicate terms. If the last entry of a leaf page is processed and the upper bound has not been reached, the query follows the next pointer of the doubly linked entries to the right-hand neighbor and resumes sequential processing with the first entry. This continues until either the upper boundary, `u`, is reached, or the last leaf page has been processed.

Predicate using a set operator

A set predicate has the form:

```
attribute set_operator key_links
```

where attribute has the type: link, array of link, or link vstrs.

The operator options for a `set_operator` are:

```
O_INTERSECT
O_NOT_INTERSECT
O_SUBSET_OF
O_NOT_SUBSET_OF
O_SUPERSET_OF
O_NOT_SUPERSET_OF
O_EQUIVALENT_SETS
O_NOT_EQUIVALENT_SETS
```

If the operators `O_INTERSECT, O_SUPERSET,` or `O_EQUIVALENT_SETS` are used, a B-tree index can be created on the attribute. The B-tree index can then be used to drive the predicate evaluation. Instead of scanning the entire class, the targets to be examined can be reduced by searching the B-tree index from `key_links`, reducing the time and resource costs because the number of links in `key_links` is generally much smaller than the number of links in the class.

When a query traverses a B-tree index, frequently referenced pages, such as the root page and its directly descendant interior node pages, are likely to remain in the buffer cache. As a result, the cost of b-tree overhead for a query includes a couple of node pages down the traversed path, and a list of leaf pages covered by the key values.

Query Usage of Indexes

Various types of queries use indexes differently.

Query with a single predicate term

Suppose that your predicate consists of a single predicate term, and that it is indexable.

In this case, index pages are brought into memory and read sequentially on its leaf pages. If an index value satisfies the predicate term, then the data page for the corresponding object is fetched.

Query with terms concatenated only with AND

Suppose you have a predicate such as:

```
( A and B and C and ... )
```

where A, B, and C... are "terms," such as `color=="Red"` or `age>=65`.

In this case, the query terms will be read from left to right.

If none of the terms are indexable, then all objects will be fetched and evaluated in a single pass through the data pages.

If there is only one indexable term, then the index pages are brought into memory and read sequentially from its leaf pages. If an index value satisfies the indexable term, then the data page for the corresponding object is fetched, and the object is evaluated further using the rest of the predicate terms.

If there are multiple indexable terms, then the terms will be evaluated to determine which one to use. If any of the indexable terms uses the equals operator, then the first term using the equals operator, reading from left to right, is used. If none of the indexable terms use the equals operator, then the first term found, reading from left to right, will be used. (The reasoning for preferring the equals operator is that it will likely produce fewer objects for evaluation.)

Once an indexable term is chosen, index pages for that term will be brought into memory and read sequentially. If an index value satisfies the predicate term, then the data page for the corresponding object is fetched, and the object is evaluated further.

Knowing that access paths are chosen with left-to-right, equality-preference logic, you can phrase your query to use the indexable term that you want. For example, suppose that you have an indexable term with an equality operator, such as (`sex=='M'`). To avoid having it used as the access path, you can rewrite this term as a range predicate, such as (sex >= 'M' and sex <= 'M'), and then move it to the right end of your predicate.

Query with terms concatenated with OR

If your query uses terms concatenated with the OR operator, then what happens is more complex. First your query is converted to disjunctive normal form, which means that mathematical conversions are done so that the query is expressed in a left-to-right format such as:

```
( A and B and C ) or ( D and E ) or ( F ) ...
```

where A, B, C... are "terms" and groups like (A and B and C) are called "orterms".

If you write your query in this form to begin with, its components will be evaluated left-to-right per the following discussion. If your query has to be converted to disjunctive normal form, DeMorgan's rules and the distributive rule of boolean algebras will be applied, but the basic left-to-right ordering of elements in your query is preserved as much as possible.

If any orterm lacks an indexable term, then the entire class extent will be walked, one object at a time, evaluating the query on each object (with shortcut optimization when an orterm is true or a term inside an orterm is false). In this case, all objects are traversed, because if all objects need to be evaluated, it is faster to do it in one pass.

If each orterm has at least one indexable term, then the entire query is indexable, and an index is used to traverse each orterm in succession. This means that objects in each orterm are returned in ascending order with respect to the index attribute, and that all objects in the result set are not necessarily sorted in a particular order.

Within each orterm, there may be multiple indexable terms concatenated with AND. If so, the terms are evaluated to determine which one to use. This evaluation occurs as described above for a query with terms concatenated with AND. In each index traversal of each orterm, data pages for objects whose indexes satisfy the query are fetched for further evaluation.

Although the automatic index choices are made in a way that optimizes most queries, you can control the choice by phrasing your query so that your first choice of index appears in the left-most term of each "orterm." If you have a range expressed in two terms, such as x>=42 and x<=54, put these two terms on the left.

Knowing these behaviors, you can customize your indexes and queries to your needs. To evaluate the impact of various query strategies, you can monitor disk activity with performance monitoring statistics.

For more information refer to “Statistics Collection” on page 215, in "Chapter 10 - Statistics Collection".

Overriding the default index selection behavior

If the class fields associated with a query predicate are indexed, the Versant database will select from the indexes to find the best to use for the query. The rules used to select an index for term evaluation will, in most cases provide the best choice.

However, there are situations where it is desirable to be able to guide the default index selection.

For example, consider a query term containing a “like” condition on one particular indexed class field and an “equals” condition on another indexed field.

In this case the default, preferred choice would be to use the index for the class field associated with the “equals” condition, independent of the actual data distribution.

Example: `name LIKE "M*" AND firstname = "John"` (index of firstname is chosen)

This could lead to poor query performance if the “equals” condition has a very low selectivity (i.e., many matches) and the “like” condition a very high selectivity (i.e., few matches).

For such situations, VQL 7.0 provides a “Hint” feature that allows you to override the default index selection.

NOTE:- The Hint feature is also provided for JDOQL. The syntax for the same can be found in the *JDO Versant Manual.*

Example: `name = "Miller" or name LIKE "M*"`

The syntax to specify that the index defined for the field of the predicate P should be selected for evaluation of the larger term is as follows:

[INDEX P]

The hint keyword, INDEX, is case-sensitive.

In the example above, the predicate associated with the indexed field name may be rephrased as:

[INDEX name LIKE "Miller*"] AND firstname = "John"

Now the index associated with the class field name, i.e., the "like" condition, will be used.

For the rare scenario where a hash index and a b-tree index are defined for the same field, you may specify which to use with `[INDEX.HASH P]` or `[INDEX.BTREE P]` as appropriate. This could, for example, be used to override the preferred hash index and select the b-tree index for “equals” conditions. (The hint keywords `INDEX.HASH` and `INDEX.BTREE` are case-sensitive.)

For complex query statements with two or more “or” conditions, if none of the predicates of one or more of the “or” conditions have an associated indexes, then no indexes are used for the entire query statement.

Consider the following complex query statement:

P && Q || R && S

where P, Q, R, and S are simple predicates (“like” or “equals” conditions). If there is no index associated with the predicates in either of the “or” conditions (i.e, no index for the fields in P

and Q or no index for the fields in R and S) then no indexes at all will be used to evaluate the entire query.

Sorting Query Results

Only b-tree indexes can return sorted results.

A b-tree index will be used if the predicate term contains operators `>, >=, <, <=, ==,` or `O_MATCHES`. A b-tree index will not be used if the predicate term contains `!=` or `O_NOT_MATCHES` or if the predicate term contains a set operator with the prefix `O_NOT_`.

If there are multiple predicate terms concatenated with `AND`, the leftmost predicate term with a b-tree index determines the ordering, except when an index is applied to an equality operator.

See also "Query Usage of Indexes" on page 433.

Since indexes are set and evaluated on a class basis, if a query traverses subclasses, then each subclass will be a separately sorted sub-set.

When you change the definition of a class, instances are not updated to the new definition as they are accessed. This prevents paralyzing a database whenever a schema change is made. If a class definition has been changed but all instances have not yet been evolved to the new definition, then a query will return the evolved and non-evolved sets as separate sorted subsets. Once all instances have been evolved, queries will return a single set of sorted objects. (One way to evolve instances is to mark them dirty and then perform a commit.)

The following examples illustrate how b-tree indexes sort query terms. In the following, the notation (btree) indicates that a b-tree index has been set on the attribute used in the query term.

```
predterm_A(btree)
```

Results will be sorted on predterm_A.

```
predterm_A(btree) O_AND predterm_B
```

Results will be sorted on predterm_A.

```
predterm_A(btree) O_AND predterm_B(btree)
```

Results will be sorted on predterm_A.

```
predterm_A O_AND predterm_B(btree)
```

Results will be sorted on predterm_B.

```
predterm_A(btree) O_AND predterm_B O_OR predterm_C(btree)
```

Results will be sorted into 2 sets, one on predterm_A and one on predterm_C.

```
predterm_A(btree) O_AND predterm_B O_OR predterm_C O_AND
predterm_D(btree)
```

Results will be sorted into 2 sets, one on predterm_A and one on predterm_D.

```
predterm_A(btree) O_AND predterm_B O_OR
predterm_C(btree) O_AND predterm_D(btree)
```

Results will be sorted into 2 sets, one on predterm_A and one on predterm_C.

```
predterm_A(btree) O_AND predterm_B(btree) O_OR predterm_C
 O_AND predterm_D
```

Results will not be sorted, because the non-indexed second OR term requires a full pass of the class instances, therefore the entire predicate can be efficiently resolved in this single pass.

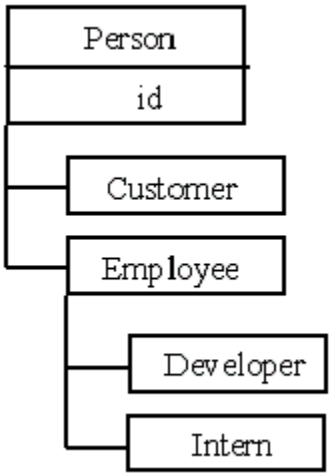
```
predterm_A(btree) O_AND predterm_B O_OR predterm_A(btree)
O_AND predterm_D
```

Results will not be sorted, because using the index on both OR terms might result in duplicates in the result set. A workaround is to perform two queries and then eliminate duplicates.

Indexes and Unique Attribute Values Usage Notes

If you create an index with either of the unique index options (O_IT_UNIQUE_BTREE or O_IT_UNIQUE_HASH), then the class can have no instances with identical values in the designated attribute.

Following are usage notes related to unique indexes. For these notes, suppose that you have the following schema.



You cannot set a unique index if duplicate values already exist.

If a class has duplicate values in an attribute and you try to create a unique index on that attribute, you will get the error `SM_E_DUPLICATEKEY` and the index will not be created.

Once set, the uniqueness of an index cannot be changed.

Once an attribute has an index of a particular kind (either B-tree or hash), you cannot change its uniqueness. If you try to do so, you will get the error `SCH_INDEX_CONFLICT`. Instead, you must delete the existing index and then create a new one. For example, if an attribute has a b-tree index and you try to create a unique b-tree index, you will get an error.

Remember that indexes should be consistent.

If you place both a hash index and a btree index on an attribute, you should make sure both are unique or non-unique.

Remember that an index is set on one attribute in one class.

While using unique indexes, it is especially important to remember that indexes are not inherited.

In the above, class Person has an id attribute that is inherited by several subclasses. If you set a unique index on id in Employee, then when you create a new Developer, a uniqueness check on id is not made.

A database checks for uniqueness whenever it receives a value.

If a unique index has been set, the uniqueness of an attribute will be checked by the database server process as soon as a new value is sent to it. Values will be sent when you perform a commit, checkpoint commit, query, or group write. They will also be sent if Versant needs to swap an object, and the object is not pinned. If a duplicate value is found, you will get the error `SM_E_SAMEKEYFOUND`.

If you decide to delay the cost of checking for uniqueness until a commit, you will get better performance, but you may have problems with exception handling since you will not know which object caused the exception.

If you explicitly write each new and changed object at the time they are created or changed, your performance will be slower but exceptions will be easier to handle.

To check uniqueness, perform a query on the top-most class or else keep track of unique values.

If you want to test whether a given value exists for an attribute without waiting until a commit, you can query the class using a predicate that compares the attribute with the test value. If you have set a unique index, this query will be very fast, and an empty result will probably mean that your test value will be a unique value. However, this test is not infallible, because another user could conceivably add a duplicate value during the time between your test query and your commit. Therefore, if you are using a unique index, you should always have an exception handler for a duplicate value error.

For example, in the above, before creating a new instance of Developer, perform a query on the id attribute of Person that specifies a search of its subclasses. This query will return a value only if a conflicting attribute value is found.

If a large number of classes and/or instances are involved, you might want to create a separate class whose only purpose is to keep track of unique values in a particular attribute.

If you want to rollback a transaction, be aware of database write behavior.

When you create or update objects with unique indexes, remember that a group write, query, or object swap will return an object to its database while still leaving it subject to a roll back. This can cause a problem if you rollback your transaction and another user has created or updated an object with a value that would conflict with the roll back value.

For example, if you want to check for uniqueness with a group value, you may want to either reserve your rollback value or set a lock on the entire class.

Suppose a class has an attribute "a", and a unique index has been defined on it. The following sequence of events will cause a problem.

The WRONG Way

	Application 1	Application 2	Database sees
time 0			obj1, a = 0
time 1	get obj1, a = 0		obj1, a = 0
time 2	set obj1, a = 1		obj1, a = 0
time 3	write obj1		obj1, a = 1
time 4		create obj2, a = 0	obj1, a = 1
time 5		commit	obj1, a = 1 obj2, a = 0 No uniqueness problem
time 6	try to roll back to obj1, a = 0		Exception raised: obj2 already has a = 0

In the above case, an exception was raised, because a second user had a window of time in which to create an object with an attribute value that conflicted with the roll back value of an object written by user 1.

A similar problem might arise if Application 1 had unpinned obj1 (which allowed it to be flushed to its database) or had performed a query on its class (which flushes new and dirty objects before operating.)

To avoid rollback problems when using objects with unique attributes, you can do any of the following:

- Do not unpin, query, or group write objects in high concurrency situations. However, as mentioned above, this will make your exception handling harder, because you will learn of uniqueness conflicts only at commit time.
- Set a read lock on the class object, which will prevent other users from making any changes at all. However, this may not be feasible, as it would cause serious concurrency problems in a multiple user environment.
- Reserve your roll back values with a dummy object.

The following repeats the above example, except in this case Application 1 protects a rollback value with a dummy object.

Example of reserving a rollback value.....

	Application 1	Application 2	Database sees
time 0			obj1, a = 0
time 1	get obj1, a = 0		obj1, a = 0
time 2	set obj1, a = 1		obj1, a = 0
time 3	create obj2, a = 0		obj1, a = 1
time 4	write obj1, obj2		obj1, a = 1 obj2, a = 0
time 5		create obj3, a = 0	obj1, a = 1 obj2, a = 0
time 6		commit	Exception raised: obj2 already has a = 0
time 7	delete obj2, then rollback to obj1, a = 0		obj1, a = 0

Do not create a unique index on a time stamp attribute.

You should not create a unique index on a time stamp attribute used for optimistic locking, since it is likely that there will be duplicate values for that attribute. In other words, it is not a good idea to overload an attribute to play dual roles for both unique constraint and optimistic locking. However, Versant does not restrict such a practice.

Indexes and Set Queries

A B-tree index can be created on attributes of links to speed up execution for the set query operators. This is important, because the current implementation of query with path expressions uses a "Depth-First-Fetch" approach to evaluate each path expression. Its performance therefore becomes an issue when, say there is a large starting class with complex relationship among classes along the navigation path (as it requires an exhaustive traversal defined on the links from the given path expressions, and each navigation via a link is likely to involve a random i/o operation.)

Because of the ability to create indexes, there are many advantages of using the strategy of reverse traversal and reverse joins to execute a path query. Each stage works like regular query. Then indexes, if any, can be used in each stage. In consequence, the selectivity in general is lower than the normal path query. Thus, one can expect a better query performance. However, you should also consider a test of network traffic from each stage as the result needs to be sent back to the client.

For example, consider again the example of brokers who represent dealers who have vehicles that was presented in the section "Set Queries". In this example, we have the following three classes and attributes:

Classes	Attributes
`Broker`	`dealers(linkvstr)`
`Dealer`	`vehicles(linkvstr)`
`Vehicle`	`color, make`

Each broker works for some dealers. Each dealer has certain inventory of vehicles. Each vehicle has attributes of color and make. We assumed the following objects in these classes in a database:

Broker	Dealer	Vehicle
`s1:{d1}`	`d1:{v1, v4}`	`v1: red Ford`
`s2:{d3, d2}`	`d2:{v2, v3, v6}`	`v2: blue VW`
`s3:{}`	`d3:{v2, v4, v5}`	`v3: white GM`
`s4:{d2}`		`v4: blue Honda`
		`v5: red Honda`
		`v6: red Honda`

In this example, one of the predicates was "vehicles INTERSECT Vehicles_red_Honda". Since vehicle is a vstr of links, without index support, this would be an expensive operation as the predicate needs to compare each dealer's vehicles with the given linkvstr of "red Honda". Since it compares two variable length of links, even one comparison may be costly, not to mention repeating it for each dealer.

However, if we create an index on the link vstr attribute, the cost can be much lower. For example, an index on vehicles on Dealer will have this content among others:

Loid of Vehicles	Poid of Dealers
v1	d1
v2	d2
v2	d3
v3	d2
v4	d1
v4	d3
v5	d3
v6	d2

This is a normal B-tree index with a new meaning, because it is constructed with a special method. For example, a dealer such as d1:{v1, v4} produces two entries to the B-tree index:

Loid of Vehicles	Poid of Dealers
v1	d1
v4	d1

In other words, the index stores the reverse links from vehicles to dealers given a dealer class. From this index, we can easily find all the dealers who have a specific vehicle in their inventory. For example, given Vehicles_red_Honda={v5, v6}, we can quickly locate two entries, and return {d3, d2} as the answer.

Note that this requires two iterations: the first iteration is for vehicle v5, and the next iteration is for v6. Each iteration will work the same way as the normal speedy searching capability from a B-tree index. By contrast, if there is no B-tree index defined on links_attr and a predicate has an O_INTERSECT operator, then each instance under the class is visited once and each link under links_attr has to compare with those under links_value until a match or exhaustion. In most cases, this would be very expensive.

Search Query Usage Notes

Queries and Locks

There are numerous routines that perform queries, and there are numerous locking options to the various query routines. Locking behavior during a query depends on which routine you use and which options you select.

Following is a general description of query lock behavior, but for a detailed information on locks, refer chapter “Locks and Optimistic Locking” .

In general, a query will return a vstr or array (depending on the interface) containing links to all objects that satisfy the query predicate, but the objects themselves are not brought into memory unless you specifically use a "pin in memory" or "fetch" option.

If you do specify that you want to fetch or pin returned objects, then a lock will be set on them as part of the query operation. In this case, all normal locking rules apply. For example, if you want to fetch an object that has been write locked by another user, your query will wait (until timed out) for the lock to be released before it will finish.

If you do not specify that you want to fetch or pin returned objects, which is the usual case, then you will receive links to all objects that satisfy the query regardless of locks placed by other users. This is the default behavior. However, there is a way to place a lock on each object that satisfies the query.

In query routines, you can specify a "class lock" and/or an "instance lock". Your specified class lock will be placed on the class objects for the instances returned. Your specified instance lock will be placed the instances returned. The defaults are to place an intention read lock on the class objects (so that the class definition cannot be changed while you are looking at objects) and no lock on the instances (instead, locks will be set on individual objects only when you bring them into memory).

Just as there are various forms of queries, there are various ways to bring an object into memory. You can use an explicit routine, such as a "locate object" or "group read" routine, or, in C++, you can de-reference an object using its link. If you use an explicit routine, you will need to specify a lock mode. If you de-reference it, the default lock will be set. In either case, all normal lock rules will apply.

When you perform a query, there is a chance that another user will change the class definition during your query period. Accordingly, many query routines allow you to specify a lock on the class objects for the instances returned by the query. You can specify the following kinds of locks in many query routines.

Lock Types	Description
intention lock	An intention lock affects only class objects and has no effect on instance objects. The primary purpose of an intention lock is to prevent the class definition from being changed during the time in which you are looking at instance objects. In most cases, you will want to specify an "intention read lock" on the class objects involved in a query (which is the default for query routines that do not require a lock parameter.) The only effect of an intention read lock on a class is to prevent it from being changed, and it allows other users to set normal read, update, and write locks on instances of the class.
normal lock	Specifying a normal read, update, or write lock on a class object conceptually places the corresponding lock on all instances of the class. For example, if you specify in your query that you want to place a read lock on the class objects involved, then, conceptually, you will get a read lock on all instances of the classes involved, even though the actual instances are not brought into memory. This means that all normal lock rules will apply, and the evaluation of your query will not start until all instances can be successfully read locked. Similarly, if you specify a write lock, your query will not be evaluated until all locks held on instances by other transactions are released. Specifying a normal read, update, or write lock on an instance object places the corresponding lock on all instances of the class immediately, rather than waiting until the objects are retrieved from their databases.
no lock	You can explicitly request that no lock be placed on the class objects for the instances involved in a query. Although this will improve performance, because no lock checking will be done, you should do this only if you know for sure that there will be no changes to the class objects involved during the query period.

Your choice of locks will depend upon what you want to do. **Following are some sample combinations of class and instance locks.**

Class lock: intention read Instance lock: no lock	In this case, you will be guaranteed that the class definition will not change until your transaction ends, but instance locks will not be set until you retrieve an object.
Class lock: intention read Instance lock: read	In this case, you will be guaranteed that neither class definitions or instance objects will change.
Class lock: intention write Instance lock: write	In this case, you are guaranteed to be the exclusive user of both the class and instance objects.
Class lock: read Instance lock: ...	Setting a read lock on class objects has the effect of setting read locks on all instance objects.
Class lock: write Instance lock: ...	Setting a write lock on class objects has the effect of setting write locks on all instance objects. In this case, you are guaranteed to be the exclusive user of both the class and instance objects.

You can set an inheritance flag in a select statement. If set to `true`, the inheritance flag will cause the query to evaluate instances of both the specified class and also its subclasses. If you do set the inheritance flag, your choice of short lock will be set on both the class object of the query class and on the class object of each subclass instance returned by the query.

Queries and Dirty Objects

Queries Mark Objects as Clean: Except in unusual cases, applications run with a client process for the application and a server process for each connected database. To reduce network traffic when a database is on a different machine than the application, queries are performed in the database server process and only results are sent to the application process.

Queries are performed in the server process on the server machine. Before a query is performed, all objects that you have marked as dirty in the current transaction are flushed from the object cache associated with the application to the server caches associated with the connected databases. This ensures that when the query is performed, the latest state of each object is sent from the database to the application.

To improve performance in the case of multiple queries in the same transaction, when objects are flushed to server caches prior to a query, they are marked as "clean" in the application object

cache (since the updates have already been logged in the server cache, where they will be saved or abandoned at the next commit or rollback). This means that if you perform multiple queries in the same transaction, the same dirty objects do not have to be repeatedly flushed to the server, only the objects that have been marked dirty since the last query.

Since queries mark all objects in the object cache as clean, if you change an object after performing a query, you must mark it as dirty even if earlier in the transaction you already marked it as dirty.

Performance

To improve the performance of queries, you can do any or all of the following:

- Cluster instances of a class.
- Cluster objects about a parent object.
- Specify raw devices or files for database volumes.
- Set indexes.
- Tune database operating parameters.
- Turn locking and logging on and off.
- Specify link based navigation.
- Reduce context switching by running an application with a single process.

CURSOR QUERIES

Cursors Queries allow you to perform a query in stages.

Concepts

When you perform a query, you send a message from your application to a database, the database finds the objects satisfying your search conditions, and then the database returns links to the found objects. You can then use the returned links to bring objects into memory one at a time or in a group.

The default behavior for queries is fine for most situations, but if your query involves an extremely large number of objects, the following kinds of problems can occur:

- You may run out of memory for the returned objects;
- Your application may pause for an unacceptably long time while you wait for all objects to be returned;
- Your chances of encountering a locked object increases, which might time-out the entire query;
- Any read or write lock you place directly or indirectly on the objects returned might hinder access by other users;
- You cannot sample just a few objects to confirm that the objects being returned are those that you actually want;
- Your interface language is oriented to handling one object at a time rather than sets of objects.

A cursor allows you to perform a query in stages. A cursor query is still performed by the database server process, but now you can control how many objects are found and sent to your application at a time.

In some situations, there are numerous advantages to a cursor query. If you choose a small batch size, the time to get your first objects will be short. If your class contains, say, a million or more objects and you only need the first few, then you can close the cursor after finding and fetching only the objects you need. And, while you are looking at a batch of objects, the objects not in your batch can remain unlocked and available for other users.

Mechanisms

The following functions create cursors, fetch objects from cursors and release cursors.

Functions	Description
`o_pathselectcursor()`	Create cursor.
`o_fetchcursor()`	Get object with cursor.
`o_releasecursor()`	Release cursor.
	Also: all open cursors are closed when a transaction ends with a commit, checkpoint commit, or rollback.

Elements

The following are key elements of a query performed with a cursor.

Cursor handle

Each cursor has an identifier, called a "cursor handle," so that you can separately control numerous cursors at the same time.

Cursor result set

The "cursor result set" contains all objects that satisfy the query. The difference between the cursor result set and the links returned by a normal query is that the cursor result set is fetched in batches rather than all at once.

Cursor fetch request

A "cursor fetch request" is a request to the query to fetch another batch of objects.

Cursor fetch count

The "cursor fetch count" is the number of objects that you ask to be fetched in the next batch of objects.

Cursor batch

A "cursor batch" is a set of objects returned in a cursor fetch request. The number of objects in a cursor batch may be less than the cursor fetch count if there are no more objects satisfying the query predicate or if the request encountered a lock conflict. In this release, you can move sequentially forward through the batches that comprise the cursor result set.

Cursor position

The "cursor position" is a pointer to the current cursor batch. Once you have a cursor batch, you can read, update, or delete any object in the batch. Once the cursor moves forward, the objects previously touched by the cursor will not be selected again.

Cursor creation

A cursor is created by calling a cursor allocation method, which can also optionally fetch the first cursor batch and set the cursor position. A cursor remains active until it is released.

Cursor release

You can close a cursor and release its resources either explicitly with a function call or implicitly by ending a session or disconnecting from the database associated with the cursor query. Cursors are also released implicitly at the end of a transaction.

Result Set Consistency

Since a query performed with a cursor steps through the result set, you can encounter inconsistencies among objects as you move from batch to batch or try to repeat a cursor query. These inconsistencies can occur due to activities performed by other concurrent transactions.

The following are some key concepts and terms related to the consistency of objects returned with a cursor query.

Dirty read

In the context of a query performed with a cursor, a "dirty read" means fetching an object that is being modified by another open transaction. For example, suppose another user is modifying an object, has written it to a database, but has not committed their transaction. While this is

happening, suppose that you fetch the object without a lock (before the other user commits their transaction.) If the other user then rolls back their transaction, the object you have can be considered to have never existed.

Cursor stability

Placing a lock on the objects in a cursor batch is called "protecting the cursor batch" in order to provide "cursor stability." If locks are placed only on the cursor batch and then released when the next batch is fetched, you can still get "non-repeatable reads" and "phantom updates," but you are guaranteed that the objects in the cursor batch are stable while you are looking at them.

Non-repeatable read

A "non-repeatable" read is when you read the same object twice in the same transaction and get different object values each time. This can happen if you use a dirty read to fetch an object that is later modified by another transaction, or the lock placed on the object is not held until the end of the transaction.

Phantom updates

A "phantom update" is when you perform the same query twice in the same transaction and get different objects each time. This can happen after a read if another user creates or deletes an object that satisfies your search conditions, or if another user changes an object so that it newly falls within your search conditions.

Serializable execution

"Serializable execution" means that separate, concurrent transactions have the same effect as performing the same transactions one after another. This is a relevant concept when you are using cursor queries to step through batches of objects, because it means that each concurrent transaction is seeing a consistent set of objects and changing them in ways that does not compromise the work of all other transactions. As a result, no inconsistencies will be seen from a serializable execution of transactions.

Cursors and Locks

Cursor result set consistency

The consistency of the cursor result set depends upon the lock options you specify when you create the cursor.

The key parameters are:

Class lock mode

The short lock to be placed on the class object for the class on which the cursor query will operate.

Instance lock mode

The short lock to be placed on the current set of cursor batch objects.

Lock release option

Whether to release read locks (but not other locks) on a batch when the next batch of objects is fetched. To release read locks (while holding update and write locks,) specify the `O_CURSOR_RELEASE_RLOCK` option.

Your choices for these three parameters determine the "isolation level" of the cursor operation. The "isolation level" of a transaction is a measure of how operations on a cursor result set can affect and be affected by operations in other concurrent transactions.

Transaction isolation levels

The following are the isolation levels that result from various choices of lock parameters.

Isolation level 0, "read uncommitted"

Class lock mode — No lock

Instance lock mode — No lock

Lock release option — (Not relevant)

Cursor batch objects are unstable, and non-repeatable reads and phantom updates may occur.

Isolation level 1, "read committed"

Class lock mode — Intention read lock or stronger

Instance lock mode — Read lock or stronger

Lock release option — `O_CURSOR_RELEASE_RLOCK`

Cursor batch objects are stable, but non-repeatable reads and phantom updates may occur.

Isolation level 2, "repeatable read"

Class lock mode — Intention read lock or stronger

Instance lock mode — Read lock or stronger

Lock release option — None

Cursor batch objects are stable and reads are repeatable, but phantom updates may occur.

Isolation level 3, "serializable"

Class lock mode — Read lock or stronger

Instance lock mode — No lock or stronger

Lock release option — None

Cursor batch objects are stable, reads are repeatable, and phantom updates will not occur.

Note that as the isolation level increases, objects and reads become more stable, but concurrency is reduced. For example, at isolation level 3, the read lock on the class object has the effect of placing a read lock on all objects of the class, which prevents all other users from updating, deleting, or creating an object of that class. Also, for each isolation level, a stronger class or instance lock mode than the minimum lock mode shown above will further reduce concurrency with other transactions.

Whatever your isolation level, you can still acquire write locks on cursor batch objects in order to update or delete them, and your transaction will always commit or roll back as an atomic unit.

Anomalies

Isolation level 0 : As objects are fetched without a lock, your application needs to handle the possibility of error `5006, OB_NO_SUCH_OBJECT`, which can occur if an object is deleted by another transaction after the "fetch" but before the "get."

Isolation level 0 and 1 : In this release, an object is physically moved to the end of the storage space for its class if it outgrows its current page(s) or if it is deleted and then restored by a transaction rollback.

When an object is in a cursor result set and then gets moved, it can be returned more than once. For example, suppose a cursor fetches an object with a read lock, but then the read lock is released when the cursor position is moved. In this case, if another transaction causes the object to move physically, then the object will be seen again by the cursor.

Example

Following is a C/Versant program that illustrates the creation, use, and release of a cursor. It assumes a database named engineering.

```
#include <stdio.h>
#include <string.h>
#include "osc.h"
#include "oscerr.h"
#include "omapi.h"
#define NULL_OBJECT ((o_object)NULL)
o_clsname   emp_class = "Employee";
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* define_emp() function that creates Employee class object */
/*                                                          */
static void define_emp()
{
  o_vstr    new_attrs;
  o_object  new_Empclass;
  /* static description of every Employee attribute. in an array */
  static o_attrdesc emp_attrs[]=
  {
   {"emp_name", {"",""}, "char", O_DYNA_VECTOR, NULL, 0, NULL},
   {"emp_department", {"",""}, "char",O_DYNA_VECTOR, NULL,0, NULL},
   {"emp_salary", {"",""}, "o_float", O_SINGLE, NULL, 0, NULL},
   {"emp_number", {"",""}, "o_u4b",   O_SINGLE, NULL, 0, NULL},
   {"emp_job_description",{"",""},"char",O_DYNA_VECTOR, NULL,0,NULL}
  };
  /* static array of POINTERS TO Employee attributes */
  static o_attrdesc *emp_attrs_ptr[]=
  {
    &emp_attrs[0],
```

```
    &emp_attrs[1],
    &emp_attrs[2],
    &emp_attrs[3],
    &emp_attrs[4]
  };
  /* create vstr of Employee attributes */
  if (o_newvstr(&new_attrs, sizeof(emp_attrs_ptr),
       (o_u1b *)emp_attrs_ptr) == NULL_VSTR)
  {
    printf ("ERROR: cannot create new emp vstrs (err %d)\n", o_errno);
    o_exit(1);
  }
  /* create Employee class object  */
  new_Empclass = o_defineclass(emp_class, NULL,
           NULL_VSTR, new_attrs,NULL_VSTR);
  if (new_Empclass != NULL)
    printf ("Created Employee class\n");
  else if (o_errno == SCH_CLASS_DEFINED)
    printf ("ERROR: Employee class already exists\n");
  else
  {
    printf ("ERROR: could not define Employee class
     (err %d)\n",o_errno);
    o_exit(1);
  }
  o_deletevstr(&new_attrs);
} /* define_emp */
/*                                                              */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* define_dep() function that creates Department class object */
/*                                                              */
static void define_dep()
{
  o_vstr    new_attrs;
  o_object  new_Depclass;
  o_clsname new_clsname = "Department";
 /* static description of every Department attribute. in
  * an array */
  static o_attrdesc dep_attrs[]=
  {
```

```
   {"dep_name", {"",""}, "char", O_DYNA_VECTOR, NULL, 0, NULL},
   {"dep_manager", {"",""}, "Employee", O_SINGLE, NULL, 0, NULL},
   {"dep_employees",{"",""},"Employee",O_DYNA_VECTOR, NULL,0,NULL},
  };
/* static array of POINTERS TO Department attributes */
  static o_attrdesc *dep_attrs_ptr[]=
  {
    &dep_attrs[0],
    &dep_attrs[1],
    &dep_attrs[2],
  };
  /* create vstr of Department attributes */
  if (o_newvstr( &new_attrs, sizeof(dep_attrs_ptr),
       (o_u1b *)dep_attrs_ptr) == NULL_VSTR)
  {
    printf ("ERROR: cannot create new dep vstrs (err %d)\n", o_errno);
    o_exit(1);
  }
  /* create Department class object  */
  new_Depclass = o_defineclass(new_clsname, NULL,
   NULL_VSTR,new_attrs, NULL_VSTR);
  if (new_Depclass != NULL)
     printf ("Created Department class\n");
  else if (o_errno == SCH_CLASS_DEFINED)
printf ("ERROR: Department class already exists\n");
  else
  {
     printf ("ERROR: could not define Department class
            (err %d)\n",o_errno);
     o_exit(1);
  }
  o_deletevstr(&new_attrs);
}
 /* define_dep */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* get_emp_name() function to get Employee name value */
/*   caller must delete returned vstr            */
static o_vstr get_emp_name(emp_object)
  o_object  emp_object;
{
  o_vstr    ename;
```

```
  o_bufdesc attr_buf;
  attr_buf.data = (o_u1b *)&ename;
  attr_buf.length = sizeof(o_vstr);
  if (o_getattr(emp_object,"emp_name",&attr_buf) !=
 O_OK)
  {
    printf ("ERROR: getting employee name (err %d)\n", o_errno);
    o_exit(1);
  }
  return ename;
}    /* get_employee_name() */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - -   */
/* get_dep_employees() function to get Department employees value */
/*                 caller must delete returned vstr               */
static o_vstr get_dep_employees(dep_object)
  o_object  dep_object;
{
  o_vstr    emps_linkvstr;
  o_bufdesc attr_buf;
  attr_buf.data = (o_u1b *)&emps_linkvstr;
  attr_buf.length = sizeof(o_vstr);
  if (o_getattr(dep_object,"dep_employees",&attr_buf) != O_OK)
  {
    printf ("ERROR: getting department employees (err %d)\n",o_errno);
    o_exit(1);
  }
  return emps_linkvstr;
}
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* check_dep_employee() test if Employee link is in  */
/*               dep_employees vstr                  */
/*                                                   */

static o_u1b check_dep_employee(dep_object,emp_object)
  o_object dep_object;
  o_object emp_object;
{
  o_u1b checkresult;
  o_vstr depempvstr;
  o_vstr empvstr;
```

```
  depempvstr = get_dep_employees(dep_object);
  if (o_newvstr( &empvstr, sizeof(o_object),
     (o_u1b *)&emp_object) == NULL_VSTR)
  {
    printf("ERROR: cannot create emp vstr (err %d)\n",o_errno);
    o_exit(1);
  }
  if (o_intersectvstrobjs( &depempvstr, &empvstr) == NULL_VSTR)
      checkresult = 0;
  else  checkresult = 1;
  o_deletevstr(&depempvstr);
  o_deletevstr(&empvstr);
  return checkresult;
}
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* set_dep_employees() replace dep_employees vstr with a new vstr*/
/*                                                                    */
static void set_dep_employees (dep_object, newemps_linkvstr)
  o_object dep_object;
  o_vstr   newemps_linkvstr;
{
  o_bufdesc attr_buf;
  attr_buf.data = (o_u1b *)&newemps_linkvstr;
  attr_buf.length = sizeof(o_vstr);
if (o_setattr(dep_object,"dep_employees",&attr_buf)!= O_OK)
  {
    printf("ERROR: dep_employees setattr failed
          (err %d)\n",o_errno);
    o_exit(1);
  }
}    /* set_dep_employees() */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* add_dep_employee() add Employee link to dep_employees vstr */
/*                                                                    */
static void add_dep_employee(dep_object,emp_link)
  o_object dep_object;
  o_object emp_link;
{
  o_vstr deptemps_linkvstr;
  if (check_dep_employee(dep_object,emp_link))
    return;
```

```
  deptemps_linkvstr = get_dep_employees(dep_object);
  if (o_appendvstr( &deptemps_linkvstr,
 sizeof(o_object),
     (o_u1b *)&emp_link) == NULL_VSTR)
  {
    printf ("ERROR: cannot append employee to dept vstr
             err(%d)\n", o_errno);
    o_exit(1);
  }
  set_dep_employees(dep_object,deptemps_linkvstr);
  o_deletevstr(&deptemps_linkvstr);
}
/* - - - - - - - - - - - - - - - - - - - - - - - - - -  */
/* make_emp() create an instance of Employee class  */
/*                                                  */
static o_object make_emp(aname,adep, dept,
 asal,anum,ajob)
  char*   aname;
  char*   adep;
  o_object dept;
  o_u4b   anum;
  o_float asal;
  char *  ajob;
 {
  o_object  newEmployee;
  o_bufdesc attr_buf;
  struct employeeRec
  {
   o_vstr  emp_name;
   o_vstr  emp_department;
   o_float emp_salary;
   o_u4b   emp_number;
   o_vstr  emp_job_description;
  };
  struct employeeRec empstruct;
  if (o_newvstr( &empstruct.emp_name, strlen(aname) + 1,
              (o_u1b *)aname) == NULL_VSTR)
  {
    printf("ERROR: cannot create new emp name vstr
          (err %d)\n",o_errno);
```

```
    o_exit(1);
  }
if (o_newvstr( &empstruct.emp_department, strlen(adep)+ 1,
                (o_u1b *)adep) == NULL_VSTR)
  {
    printf("ERROR: cannot create new emp dept vstr
           (err %d)\n",o_errno);
    o_exit(1);
  }
  empstruct.emp_salary = asal;
  empstruct.emp_number = anum;
  if (o_newvstr( &empstruct.emp_job_description, strlen(ajob) + 1,
              (o_u1b *)ajob) == NULL_VSTR)
  {
    printf("ERROR: cannot create new emp job vstr
          (err %d)\n",o_errno);
    o_exit(1);
  }
  attr_buf.length = sizeof(empstruct);
  attr_buf.data = (o_u1b *)&empstruct;
  newEmployee = o_createobj( emp_class, &attr_buf, FALSE);
  if (newEmployee == NULL_OBJECT)
  {
    printf("ERROR: cannot create new employee instance
          (err %d)\n",o_errno);
    o_exit(1);
  }
  if ( dept != NULL_OBJECT )
      add_dep_employee(dept, newEmployee);
  return newEmployee;
}
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* make_dep() create an instance of Department class */
/*                                                           */
static o_object make_dep(deptname,mgr_object)
  char*    deptname;
  o_object mgr_object;
{
  o_object  newDepartment;
  o_clsname depclass = "Department";
  o_bufdesc attr_buf;
```

```
  struct departmentRec
  {
   o_vstr   dep_name;
   o_object dep_manager;
   o_vstr   dep_employees;
  };
  struct departmentRec depstruct;
if (o_newvstr( &depstruct.dep_name, strlen(deptname) + 1,
              (o_u1b *)deptname) == NULL_VSTR)
  {
    printf("ERROR: cannot create new dep name vstr
          (err %d)\n",o_errno);
    o_exit(1);
  }
  depstruct.dep_manager = mgr_object;
  depstruct.dep_employees = NULL_VSTR;
  attr_buf.length = sizeof(depstruct);
  attr_buf.data = (o_u1b *)&depstruct;
  newDepartment = o_createobj(depclass, &attr_buf, FALSE);
  if (newDepartment == NULL_OBJECT)
  {
    printf("ERROR: cannot create new department instance
           (err %d)\n",o_errno);
    o_exit(1);
  }
  return newDepartment;
}
/* - - - -  - - - - - - - - - - - - - - - - - - - - - - - */
/*print_emp() function to print one Employee instance */
/*                                                      */
static void print_emp(emp_vstr)
  o_vstr  emp_vstr;
{
  o_4b       i, num_objs;
  o_object  *objp;
  o_vstr    ename, edep, ejob;
  o_u4b     enumber;
  o_float   esal;
  o_bufdesc attr_buf;
  num_objs = o_sizeofvstr(&emp_vstr)/sizeof(o_object);
```

```
  for (i = 0, objp = (o_object*)emp_vstr; i < num_objs; objp++, i++)
  {
      ename = get_emp_name(*objp);
      attr_buf.data = (o_u1b *)&edep;
      attr_buf.length = sizeof(o_vstr);
      if (o_getattr(*objp,"emp_department",&attr_buf) != O_OK)
      {
        printf ("ERROR: getting department name (err
        %d)\n", o_errno);
        o_exit(1);
      }
      attr_buf.data = (o_u1b *)&esal;
      attr_buf.length = sizeof(o_float);
if (o_getattr(*objp,"emp_salary",&attr_buf) != O_OK)
      {
        printf ("ERROR: getting employee salary (err
         %d)\n", o_errno);
        o_exit(1);
      }
      attr_buf.data = (o_u1b *)&enumber;
      attr_buf.length = sizeof(o_u4b);
if (o_getattr(*objp,"emp_number",&attr_buf) != O_OK)
      {
        printf ("ERROR: getting employee number (err
        %d)\n", o_errno);
        o_exit(1);
      }
      attr_buf.data = (o_u1b *)&ejob;
      attr_buf.length = sizeof(o_vstr);
if (o_getattr(*objp,"emp_job_description",&attr_buf) != O_OK)
      {
        printf ("ERROR: getting job description (err
        %d)\n", o_errno);
        o_exit(1);
      }
      printf ("%s %s $%.2f #%d %s \n",
              (char*)ename,(char*)edep,esal,enumber,
              (char*)ejob);
      o_deletevstr(&ename);
      o_deletevstr(&edep);
      o_deletevstr(&ejob);
```

```
  }
}
main()
{
    o_4b     i, j;
    o_object dept1, dept2, dept3, dept4;
    o_object mgr1, mgr2, mgr3, mgr4;
    o_object emp1, emp2, emp3, emp4,emp5,emp6,emp7,
    emp8,emp9,emp10;
    o_err    err;
    o_cursor emp_cursor;
    o_vstr   emp_objs;
    char*    dbname = "engineering";
    char*    depart = "Research & Development";
    o_vstr   mypredvstr;
    o_predterm   mypred;
    o_predterm*  mypredptr = &mypred;
    o_predblock  mypredblk;
    o_u4b    numobjects;
    o_u4b    retry_count = 0;
    o_u4b    options = O_CURSOR_RELEASE_RLOCK;
    if (o_beginsession("",dbname,"",0) != O_OK)
    {
printf("ERROR: begin session (err %d)\n", o_errno);
      return 1;
    }
    { /* begin database session */
      /* Create employee class object */
      o_dropclass("Employee", dbname);
      define_emp();
      if (o_xact(O_COMMIT, NULL) != O_OK)
      {
          printf("ERROR: commit (err %d)\n",o_errno);
          o_exit(1);
      }
      /* Create department class object */
      o_dropclass("Department", dbname);
      define_dep();
      if (o_xact(O_COMMIT, NULL) != O_OK)
      {
```

```
    printf("ERROR: commit (err %d)\n",o_errno);
    o_exit(1);
}
printf("=>Create new Employee managers...\n");
mgr1 = make_emp( "Bob Black","Production", NULL_OBJECT,
               (o_float)35000,(o_u4b)100022,"Manager");
mgr2 = make_emp( "Bill Green","Research & Development",
                NULL_OBJECT, (o_float)35000,
               (o_u4b)100022,"Manager");
mgr3 = make_emp( "Tom White","Quality Assurance",
               NULL_OBJECT, (o_float)35000,
               (o_u4b)100022,"Manager");
mgr4 = make_emp( "Mary Jones","Administration",
               NULL_OBJECT, (o_float)35000,
               (o_u4b)100022,"Manager");
printf("=>Create new Departments...\n");
dept1 = make_dep("Production",mgr1);
dept2 = make_dep("Research & Development", mgr2);
dept3 = make_dep("Quality Assurance",mgr3);
dept4 = make_dep("Administration",mgr4);
printf("=>Create new Employees...\n");
emp1 = make_emp("Larry Link","Production", dept1,
               (o_float)35000,(o_u4b)100022,"Engineer");
emp2 = make_emp("Alex Loids","Production", dept1,
               (o_float)35000,(o_u4b)100022,"Engineer");
emp3 = make_emp("Oliver Object","Research & Development",
               dept2, (o_float)35000,
               (o_u4b)100022,"Engineer");
emp4 = make_emp("Ethel Murtz","Production", dept1,
               (o_float)35000,(o_u4b)100022,"Engineer");
emp5 = make_emp("Lance Lee","Production", dept1,
               (o_float)35000,(o_u4b)100022,"Engineer");
emp6 = make_emp("Kate Krammer","Research & Development",
               dept2, (o_float)35000,
               (o_u4b)100022,"Engineer");
emp7 = make_emp("Liz Lucky","Administration", dept4,
               (o_float)35000,(o_u4b)100022,"Office admin");
emp8 = make_emp("Sam Skywalker","Administration", dept4,
               (o_float)35000,(o_u4b)100022,"Payroll admin");
emp9 = make_emp("Jill Jennings","Administration", dept4,
               (o_float)35000,(o_u4b)100022,"HR admin");
```

```
        emp10 = make_emp("Tammy Tanner","Administration", dept4,
                        (o_float)35000,(o_u4b)100022,"Receptionist");
        if (o_xact(O_COMMIT, NULL) != O_OK)
        {
            printf("ERROR: commit (err %d)\n", o_errno);
            o_exit(1);
        }
        /* Set up predicate for the search condition */
        mypredptr->attribute = "emp_department";
        mypredptr->oper = O_EQ;
        mypredptr->key.length = strlen(depart) + 1;
        mypredptr->key.data = (o_u1b *)depart;
        if (o_newvstr(&mypredvstr, sizeof(mypred), (o_u1b *)&mypred)
             == NULL_VSTR)
        {
          printf("ERROR: creating vstr mypredvstr
                 (err %d)\n",o_errno);
          o_exit (1);
        }
        /* Set up a pred block for o_pathselectcursor() */
        mypredblk.conj = O_AND;
        mypredblk.flags = 0;
        mypredblk.more_predblocks = NULL_VSTR;
        mypredblk.leaf_predterms = mypredvstr;
/*
* Create cursor on employee class to find employees
* in the Technical Support department.   */
  emp_objs = o_pathselectcursor(emp_class, dbname,
             options, IRLOCK, RLOCK, 1, mypredblk, &emp_cursor, 0, 0);
        o_deletevstr(&mypredvstr);
        if (emp_objs == NULL_VSTR)
        {
            if (o_errno == O_OK)
printf("No objects qualified for the cursor.\n");
            else
            {
              printf("ERROR:failed to create cursor, (err%d)\n",
                     o_errno);
              o_exit(o_errno);
            }
```

```
        }
        /* Fetch 2 employees at a time and print */
        while (emp_objs != NULL_VSTR)
        {
            printf("\nfetch %d employee\n",
              o_sizeofvstr(&emp_objs)/sizeof(o_object*));
            print_emp(emp_objs);
            o_deletevstr(&emp_objs);
            emp_objs =   o_fetchcursor(&emp_cursor, 2,0,0);
            numobjects = o_sizeofvstr(&emp_objs)/sizeof(o_object *);
            if (numobjects != 2)
            {
              err = o_errno;
              if (numobjects > 0)
              {
       if (err == O_OK) /* reach the end of cursor */
                  {
    printf("\nfetch %d employees\n", numobjects);
                    print_emp(emp_objs);
                    o_deletevstr(&emp_objs);
                    break;
                  }
                  else
                  { /* case: SM_LOCK_TIMEDOUT and
                     SM_LOCK_WOULDBLOCK */
                    printf("ERROR:failed to acquired lock,
                           (err %d)\n",err);
                    if (++retry_count > 1)
                    {   /* retry once only */
                        if(o_releasecursor(&emp_cursor,0,0))
            printf("ERROR:release cursor failed,
                                  (err %d)\n", o_errno);
                        o_exit(err);
                    }
                    else
                        continue;      /* retry */
                  }
              }
       if (err == O_OK) /* reach the end of cursor */
                    break;
  /* Severe error: release cursor on employee class */
```

```
printf("ERROR:failed to fetch objects from the cursor,
                                        (err =%d)\n", err);
            if (o_releasecursor(&emp_cursor,0,0) != O_OK)
                printf("ERROR:release cursor failed,
                       (err %d)\n", o_errno);
            o_exit(err);
          }
          /* reset lock timedout per new o_fetchcursor */
          retry_count = 0;
        }
      err = o_releasecursor(&emp_cursor,0,0);
      if (o_endsession("",NULL) != O_OK)
      {
 printf("ERROR: end session (err %d)\n", o_errno);
          o_exit(1);
      }
    }  /* session */
    o_exit(0);
}
```

ADVANCED QUERIES

This section explains in detail the concept of query enhancing capabilities of Versant ODBMS. To improve the query capability, a new concept of Virtual Attributes has been introduced. The enhanced query capability of Versant allows you to query using Virtual Attributes and create index on Virtual Attributes.

Virtual Attribute

Virtual Attribute is a derived attribute, built on one or more normal attributes of a class. As the Virtual Attributes are computed at runtime, they are not part of the schema of a class, but can be indexed. The indexes on Virtual Attributes are maintained during normal operations like insert, delete and update and used during query processing. This release will support only b-tree index on Virtual Attribute of a Class.

The implementation of Virtual Attribute Templates (the "class" of Virtual Attributes) are plugins provided as shared objects that gets loaded into the Versant server process' address space.

NOTE:- Virtual Attributes are currently not supported by VQL 7.0.

Virtual Attribute Syntax

A Virtual Attribute is used in certain operations where a normal attribute would be used.

To indicate that it is virtual, it begins with a slash (/) character. Next follow parameters separated by spaces.

Parameters that contain SPACE or TAB characters are quoted within curly braces {{...}}. This is important as parameters could be other Virtual Attribute with parameters.

Following is the Virtual Attribute grammar that uses an informal BNF notation:

```
SLASH"/"
BRACE-START "{"
BRACE-END"}"
virtual-attribute   ::= SLASH<virtual-attribute-template>
  { <attribute> | BRACE-START <virtual-attribute> <BRACE-END }
  [ { <attribute> | BRACE-START <virtual-attribute> <BRACE-END }+ ]
virtual-attribute-template ::= nocase <encoding> |
       tuple | national <locale>
```

<encoding> | …

- symbol } means a mandatory symbol
- [symbol] means an optional symbol
- * represents 0 or more symbols
- + represents 1 or more symbols
- keywords are presented in bolded font, and are case-insensitive
- terminal symbols are represented in uppercase
- non-terminal symbols are represented in *italic* font e.g *virtual-attribute*

For more information refer to “Versant Localization” on page 367, in "Chapter 16 - Versant Internationalization".

Virtual Attribute Template (VAT) Implementation

The implementation of Virtual Attribute Templates (VAT) are packaged as shared objects. The backend profile parameter `custom_be_plugins` is used to specify the VAT plug-ins, that the server loads at system initialization. An application must specify the VAT plug-ins for the Virtual Attributes that it would like to use, before starting up the database.

The syntax of profile.be entry to load a VAT plug-in is:

on UNIX:

```
custom_be_plugins <VERSANT_RUN_ROOT>/lib/lib<plugin-name>.so$
<VERSANT_RUN_ROOT>
```

on Windows:

```
custom_be_plugins <VERSANT_RUN_ROOT>/bin/<plugin-name>.dll$
<VERSANT_RUN_ROOT>
```

where, <VERSANT_RUN_ROOT> is path to the bin and lib folders in your Versant installation and <plugin-name> is one of the following:

```
tuple
nocase
national
```

If you want to load more than one plug-in, separate them by comma without space.

Example:

In order to load the nocase and the national plug-ins on UNIX specify:

```
custom_be_plugins <VERSANT_RUN_ROOT>/lib/libnocase.so$
<VERSANT_RUN_ROOT>,<VERSANT_RUN_ROOT>/lib/libnational.so$
<VERSANT_RUN_ROOT>
```

Virtual Attributes Usage

Virtual Attributes can be used in predicate terms of search queries and cursor queries, and in index operations.

The evaluation attribute specified in predicate terms can be a Virtual Attribute. The key value for predicate terms which has Virtual Attribute can be specified in a string buffer quoted with curly braces. If a Virtual Attribute is built on multiple attributes, a composite key value can be specified by separating the key values by space.

Virtual Attributes in Predicate Term of Search Queries and Cursors

In C/Versant, predicate terms are structures of type o_predterm.

```
typedef struct o_predterm {
    o_attrname   attribute;
    o_opertype   oper;
    o_bufdesc    key;
    o_typetype   keytype;
    o_u4b        flags;
} o_predterm;
```

The attribute in o_predterm is the evaluation attribute, for which the value is to be examined. This could be a Virtual Attribute. This allows the following C/Versant APIs to be used with Virtual Attributes.

```
o_vstr o_select (
    o_clsname    classname,
    o_dbname     dbname,
    o_bool       iflag,
    o_lockmode   lockmode,
    o_vstr       predicate );
```

```
o_vstr o_pathselect (
    o_clsname    classname,
    o_dbname     dbname,
    o_4b         levels,
    o_vstr       linktypes,
    o_u4b        options,
    o_lockmode   this_cls_lockmode,
    o_lockmode   instance_lockmode,
    o_predblock  *predicate,
    o_object     vstr_object,
    o_attrname   vstr_attribute );

 o_vstr o_pathselectcursor
    o_clsname    classname,
    o_dbname     dbname,
    o_u4b        options,
    o_lockmode   classlock,
    o_lockmode   instlock,
    o_u4b        fetchcount,
    o_predblock* predicate,
    o_cursor*    cursor,
    o_u4b        flags1,
    o_u4b        flags2 );
```

Virtual Attributes in Index Operation

In the following C/Versant APIs, `attrname` is the name of an attribute or a *Virtual Attribute* on which the index will be built.

```
 o_err o_createindex (
    o_dbname     dbname,
    o_clsname    classname,
    o_attrname   attrname,
    o_indextype  indextype );

o_err o_deleteindex (
    o_dbname     dbname,
    o_classname  classname,
    o_attrname   attrname,
    o_indextype  indextype );
```

The database utility, `dbtool` can be used to create or delete index, defined on Virtual Attributes. The Virtual Attribute should be specified within quotes.

- `dbtool -index -create -btree <C> {A | virtual-attribute}`
  Creates a b-tree index for the specified attribute or a Virtual Attribute of a class.
  For example: To create an nocase index on the first name use the following statement:
  `dbtool -index -create -btree Person "/nocase ascii firstName" <dbname>`

- `dbtool -index -create -btree -unique <C> {A | virtual-attribute}`
  Creates a b-tree index and enforces unique values for the specified attribute or Virtual Attribute.
- `dbtool -index -delete -btree <C> {A | virtual-attribute}`

  Deletes a b-tree index for the specified attribute or a Virtual Attributes of the specified class.
- `dbtool -index -info -list [-short] <C> {A | virtual-attribute}`
  Prints index information on Virtual Attributes of a class.
  - `-index -info -list -short` to display index information in tabular form.
  - `-index -info -list` to display index information in a verbose form.

Virtual Attributes in VQL Query

The Virtual Attributes are specified within back quotes {`...`} in a VQL query. The query key value is specified within quotes. The key value that contain special characters are quoted within curly braces {{...}}. A composite key value can be specified by separating the key values by whitespace.

For example: to select the objects from a class Employee whose LastName ignoring the case is "samuel" and FirstName is "Mohan M", the VQL query would be :

```
select * from Employee where `/tuple { /nocase
ascii LastName } Firstname` == "{samuel} {Mohan  M}";
```

Virtual Attributes Templates

From the 7.0 release, Versant supports the following Virtual Attribute Templates.

Virtual Attribute Template name	Description	Synopsis of virtual attribute	Plug-ins
`nocase`	Basis for creating virtual attribute on string valued attributes for case independent string matching. This converts strings to lowercase before any operation	`/nocase encoding {attribute|virtual-attribute}`	`libnocase.so (Solaris)` `nocase.dll (Windows)`
`tuple`	Basis for creating virtual attribute that are used in multi-attribute queries. The one or more attributes involved in forming this virtual attribute could be heterogeneous.	`/tuple {attribute|virtual-attribute} {attribute|virtualattribute}+`	`libtuple.so (Solaris)` `tuple.dll (Windows)`
`national`	national collating sequences for querying on string-valued attributes.	`/national locale encoding {attribute|virtual-attribute}`	`libnational.so (Solaris)` `national.dll (Windows)`
`real`	Basis for the "real" virtual attribute. The attribute on which the "real" virtual attribute is built, cannot be virtual. For details on how to use this refer Chapter-21 Internationalization.	`/real attribute`	`Built-in`

Examples

Examples of Virtual Attributes used in index creation/deletion and query operations.

- create unique B-TREE index on virtual attribute "/nocase ascii Title" of class `MyClass` (case-independent index on attribute Title of class MyClass)

  ```
  o_createindex ("MyDB", "MyClass",  "/nocase ascii Title",
  O_IT_UNIQUE_BTREE);
  ```

- to query objects from a class MyClass whose case-insensitive Title is "the diamond age".

  ```
  select * from MyClass where `/nocase ascii Title` = "the diamond age";
  ```

- create unique B-TREE index on Virtual Attributes "/tuple {/nocase ascii LastName} FirstName" of class Employee (composite index on Virtual Attribute "/nocase ascii LastName" and attribute FirstName of class Employee)

  ```
  o_createindex ("EmpDB",  "Employee",  "/tuple { /nocase ascii LastName
  } FirstName",  O_IT_BTREE);
  ```

- to select the objects from a class Employee whose LastName ignoring the case is "samuel" and FirstName is "Mohan M".

  ```
  select * from Employee where `/tuple { /nocase ascii LastName }
  Firstname` = "samuel {Mohan M}";
  ```

- drop an index on Virtual Attributes "/nocase ascii Title" of class MyClass

  ```
  o_deleteindex ("MyDB", "MyClass",  "/nocase ascii Title",
  O_IT_UNIQUE_BTREE);
  ```

Restrictions

- The size of Virtual Attributes is limited to 512 bytes.
- Virtual Attributes can only be used in a predicate term of a query and not in projection, that is Virtual Attributes cannot be used in front end operations like `o_getattr()`.
- Virtual Attributes cannot be specified on a path attribute. For example, the following is not supported :

  ```
  /nocase ascii Person::garages\tGarage::cars\tCar::color
  ```

- Virtual Attributes can not be specified in a query which uses `O_SELECT_WITH_VSTR` option. The starting point objects should be objects of a class or a class and its subclasses if `O_SELECT_DEEP` option is specified.
- Renaming an attribute requires indexes defined on Virtual Attributes for that specific attribute to be dropped.

- Dropping all instances in a class will also require indexes defined on Virtual Attributes of that class to be dropped. .
- Usage of online index creation on Virtual Attributes with FTS is not supported.
- Indexes defined on Virtual Attributes needs to be created explicitly on the target database after `vcopydb`.
- Creating index on Virtual Attributes can be performed only on an non-empty class. After dropinst of all instances of a class, the indexes on Virtual Attributes needs to be created explicitly only after creating at least an instance.
- Only B-TREE index structure is supported on Virtual Attribute of a class.
- Virtual Attributes can not be part of predicates which are used in event registration.

Additional points to remember while using the Virtual Attributes:

- Querying with different encoding scheme than the scheme used to store the data would cause unexpected results.
- If an index is defined on a Class for a Virtual Attribute, queries that do not use the Virtual Attributes may not use the index defined on that Class. For example: if an index is defined on a Virtual Attribute "/nocase ascii Title", querying on the attribute Title will not make use of the index defined on the Virtual Attribute.
- Restrictions on the indexability of normal attributes is also applicable on Virtual Attributes. Apart from this, different Virtual Attribute implementation may impose restriction on the data type of the underlying attribute on which it is built. For e.g, /nocase or /national will work only with string data types.

CHAPTER 19

Query Processing with VQL 6.0

This Chapter explains the Query Processing using the Versant Query Language 6.0.

Following topics are covered:

- VQL 6.0 Overview
- VQL 6.0 Procedures
- Mechanisms
- Statements
- Predicate term
- Example

VERSANT QUERY LANGUAGE 6.0

Overview

The Versant Query Language (VQL) is a "light-weight" language which enables you to find and manipulate objects in Versant databases using statements expressed as a string. You can then parse and execute the string using C/Versant or C++/Versant routines.

In some ways, VQL looks like SQL with some object extensions. For example, in VQL you can perform a path query and query on a vstr.

VQL Procedures

Parsing and executing a VQL statement are separate operations.

To perform a VQL operation with C/Versant, do the following.

1. Write a VQL query or update statement and place it in a string of data type `char*`.
2. Parse the VQL statement using functions provided in the C/VQL interface.
3. Pass the parse results to a standard C/Versant function for execution.

VQL 6.0 MECHANISMS

OQL Statements

Statements	Description
COMMIT	Commit transaction.
DELETE	Delete objects.
INSERT	Insert objects.
QUIT	Terminate application.
ROLLBACK	Roll back transaction.
SELECT	Find objects.
UPDATE	Update objects.

VQL Functions

The following C/Versant functions parse VQL statements:

Function	Description
o_parsequery()	Parse VQL statement and return results.
o_freequery()	Free memory used by a VQL parse result.

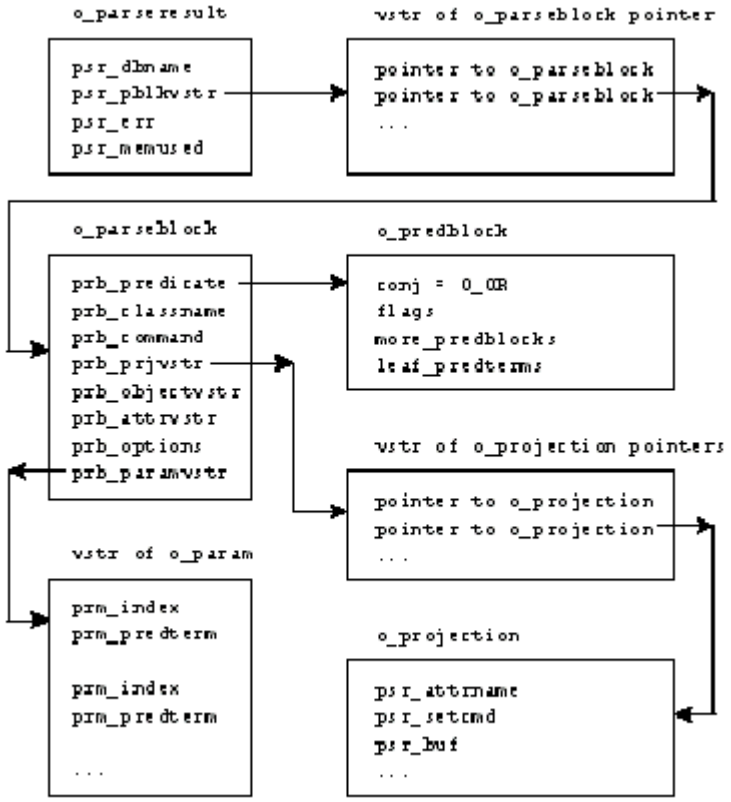
VQL Data Structures

The following C/Versant data structures support statement parsing:

Data Structures	Description
o_parseresult	VQL parse result structure.
o_cmdtype	VQL statement type.
o_parseblock	VQL parse block.
o_setcmd	VQL update type.
o_projection	VQL function parameters.
o_param	VQL substitution parameters structure.

`o_parerr`	VQL parser error information.
`o_psrmem`	VQL memory for parse result.

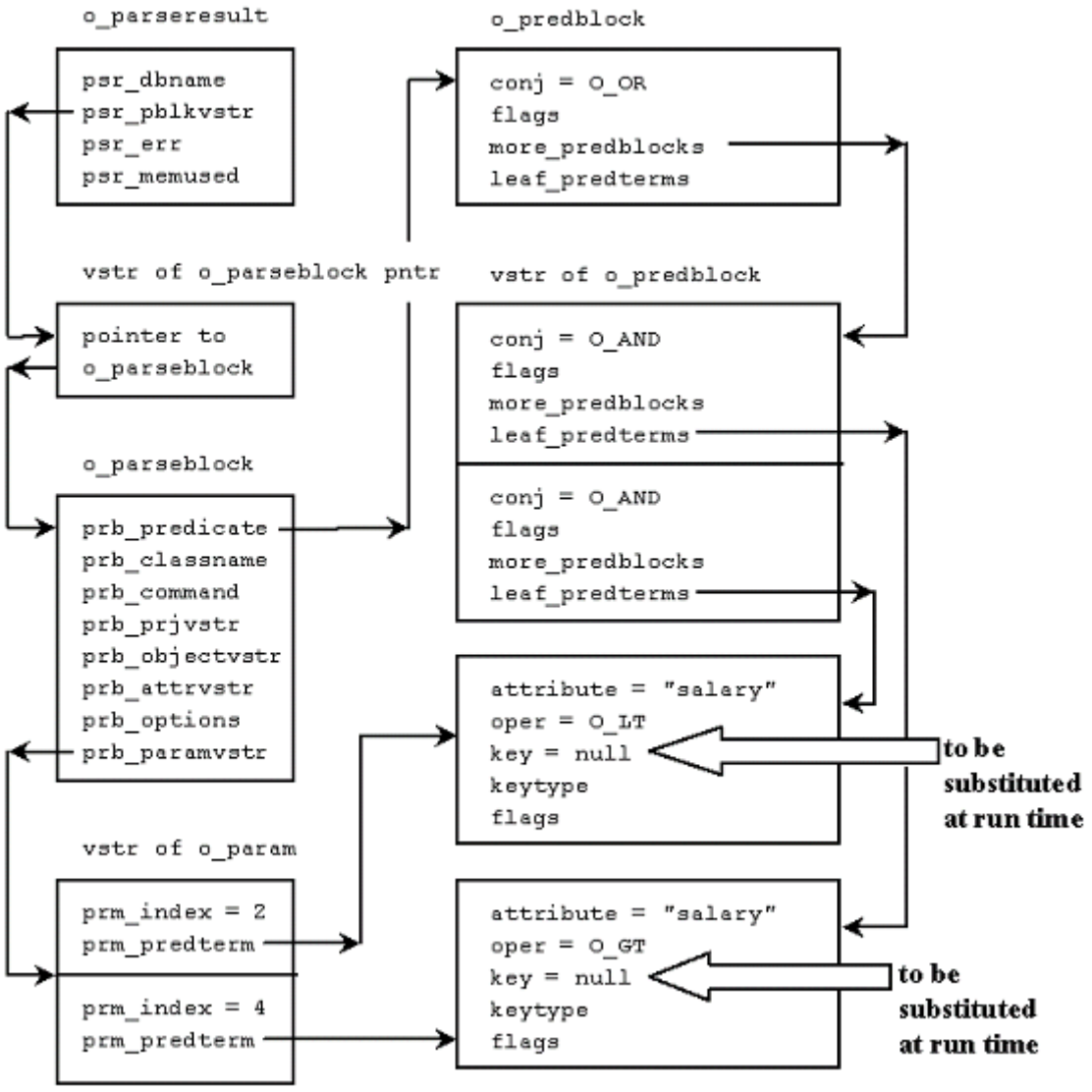
Following are the relationships of data structures for the result of `o_parsequery()`.



For example, consider the following query:

```
select name from Employee where salary > $4 or salary < $2
```

The following shows the parse results of the above statement:


o_parseresult
psr_dbname
psr_pblkvstr
psr_err
psr_memused
o_predblock
conj = O_OR
flags
more_predblocks
leaf_predterms
vstr of o_parseblock pntr
pointer to
o_parseblock
vstr of o_predblock
conj = O_AND
flags
more_predblocks
leaf_predterms
conj = O_AND
flags
more_predblocks
leaf_predterms
o_parseblock
prb_predicate
prb_classname
prb_command
prb_prjvstr
prb_objectvstr
prb_attrvstr
prb_options
prb_paramvstr
attribute = "salary"
oper = O_LT
key = null
keytype
flags
to be
substituted
at run time
vstr of o_param
prm_index = 2
prm_predterm
prm_index = 4
prm_predterm
attribute = "salary"
oper = O_GT
key = null
keytype
flags
to be
substituted
at run time

VQL STATEMENT

Usage Notes

Statement Strings

You can express one or more operations in a VQL statement string by concatenating the individual operations with a semicolon. The general syntax is:

```
stmts ::= stmt [; stmt ... ]
```

For example:

```
select * from Employee; select * from Department
```

Not Supported

In this release, VQL is a limited subset of OQL.

The VQL SELECT statement does not support:

- an attribute reference in an intermediate node of a path expression, either in a projection list or in a where clause
- functional query language support
- multi-class access (joins) other than navigation
- queries over collections
- universal or existential quantification
- method or function invocation
- named objects
- string expressions
- subquery
- the following set features: order by, group by, having, distinct, aggregate, union, intersection

Case Sensitivity

VQL statements (such as SELECT) and key words (such as SELFOID) are not case sensitive. However, class names, attribute names, and data values are case sensitive.

Specifying Attributes

The SELECT, INSERT, and UPDATE statements require specification of a list of attributes. Following are notes related to specifying an attribute in an attribute list.

General syntax

The general syntax for an attribute name is:

```
name | `name`
```

Reserved words or special characters

If the attribute name contains no reserved words or special characters, such as spaces, then you can simply supply the name.

If the attribute name does contain a reserved word or special character, then use backquotes (ASCII 96 `) rather than single or double quotes (ASCII 39 ' or ASCII 34 ") around it. Single and double quotes are reserved for specifying strings.

For example, assuming that "Math Department" is a subclass of Department:

```
select * from Employee
```

where department is_a `Math Department`

Abbreviation examples

An attribute name can be either the full name of an attribute, which is defined in the database schema, or an abbreviation of the full name with the class name and type names being stripped.

For example, the full name of an attribute employees of class Department that has been defined in C++ and stored in a database is:

```
Department::employees
```

You can abbreviate this attribute name to:

```
employees
```

Suppose that the full name of an attribute named links is:

```
Department::dep_employees.VESet<Link<PVirtual>>::links"
```

You can abbreviate links as:

```
dep_employees.links
```

For more information, on how to express an attribute name in database format, please refer to the *C++/Versant Reference Manual*.

Multiple inheritance

If there is multiple inheritance, the attribute name abbreviation in a query may lead to ambiguity in terms of telling which attribute the query is to reference.

For example, suppose that both Class A and Class B have an attribute named number and suppose that Class C inherits from both A and B. This means that there are two attributes with the same name in C: `A::number` and `B::number`. For a query such as "`select *` from C where `number > 8`", there is no way to tell whether query references `B::number` or `A::number`. In this case, the parser will pick the first attribute it finds in the Versant database schema that matches the shortname number, which may or may not lead to the intended result.

In general, if you query over a class that uses multiple inheritance, we recommend that you use the full, unambiguous attribute name in the query.

Attribute values of type float

Literal values of data type float must be in either decimal form or exponential form.

The valid form of type float is only a subset of ANSI C decimal form and exponential form. For example,

For example, the following are acceptable values for a value of type `float`: `-134.789, 7.8976e-308, 8.9654E-876, -9.876e89, 9.654E70, 8.754e+204, 9.8765E+70`.

`8.754e+78F` is not valid in VQL because no character (in this case F) is allowed to be appended at the end of exponential form.

"Raw" attribute values

VQL can only refer to the "raw" attribute which is stored in the database schema. Some of them may be different from the ones that are presented in the applications.

For example, the attribute start_date of class Employee might be of type VDATE in a C++ application. However, the actual attribute which contains data about start_date in the database schema is:

```
Employee::start_date.VDate::absolute_day_number".
```

VQL can only refer to this attribute either using it's full name (given above) or using the following abbreviation:

```
start_data.absolute_day_number
```

You cannot use methods to access attribute values.

Keyword SELFOID

A logical object identifier or loid is the unique, never reused identifier number associated with each persistent object. This number is guaranteed to be unique among all objects in a system of databases.

In a SELECT statement, you can get the loid for an object returned by a query by including the keyword SELFOID in the list of attribute names. However, if you use the wildcard character * as your attribute list for a query, the object loid will not be returned.

Specifying an object with a loid

When you specify a logical object identifier as a parameter oidval or get a loid from a SELECT query that uses SELFOID in an attribute list, the syntax for the loid will be a decimal value triplet of the following form:

```
nn.nn.nn
```

For more information, on routines that get and use LOIDS, please refer to the chapter "Versant Query Language" in the *C++/Versant Reference Manual*.

Data type conversion

Data type conversion of most integers is automatic

The parser will convert constant values of integer types in a predicate into the corresponding database type.

The following types are converted:

```
o_1b
o_u1b
o_2b
o_u2b
o_4b
o_u4b
o_8b
o_u8b
o_float
o_double
```

If the value return overflows or underflows the database type, an error will be returned. For example, if you perform a `SELECT` query where `attr_u2b = -30`, the conversion from integer -30 to type `o_u2b` will fail due to the underflow of type `o_u2b`.

Data type conversion of a float depends on the attribute type

The parser will convert a constant value of type float to either type o_float or o_double, depending on the attribute type. Conversion from float to any other type is invalid and cause an error. If a float value overflows or underflows the corresponding type of the database attribute, the parser will raise error.

Creating a Predicate

Predicates are used in `SELECT, UPDATE`, and `DELETE` statements to specify selection criteria for objects.

Not specifying a predicate will return all instances.

The general syntax for a predicate clause is:

```
WHERE query_expr
```

Individual query expressions can be concatenated with the following boolean operators:

Boolen Operators	Description
`AND`	`query_expr AND query_expr` `Boolean AND.`

OR	query_expr OR query_expr Boolean OR.
NOT	NOT query_expr Boolean NOT.

Parentheses may be used to specify the precedence of boolean operators.

The syntax for an individual query expression is:

```
pathattr  relop      const     |
pathattr  relop      param     |
pathattr  isa_relop  class     |
pathattr  isa_relop  param     |
pathattr  relop ( type ) const |
pathattr  relop ( type ) param |
```

Elements are:

pathattr

Attribute path and name, specified with the following syntax:

```
attr[->attr..]
```

Specify attribute names in `attr` and indicate the path with the arrow operator `->`. In a query expression, an attribute without a path means that an attribute of the class is being queried.

For more information, on path queries, please refer to the *C/Versant* or *C++/Versant Reference Manuals*.

For example, the following shows path attribute names using full name syntax:

```
Person::garages->Garagage::cars->Car::color
```

The following show path attribute names using abbreviations:

```
garages->cars->color
```

See also “Specifying Attributes ” on page 483.

relop

A relational operator, one of the following:

Relational Operators	Description
=	Equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
LIKE	String pattern matches
NOT_LIKE	String pattern does not match

const

A literal value, one of the following:

Literal Value	Description
float	A number of type float
int	A number of type int
string	A string (use backquotes if special characters)
oidval	An object loid expressed with syntax nn.nn.nn
constaggr	A list of constant values with comma delimiters

When comparing strings, you can use the following wildcards:

Wildcards	Description
*	Match any sequence of characters
?	Match any single character
[chars]	Match any character in given set.
[x-y]	Match any character in given range.
\x	Match single character x.

isa_relop

A class membership operator, one of the following:

Operator	Description
ISA	Is member of class.
NOT_ISA	Is not member of class.

These operators test whether an object is an instance of a given class, but they do not test whether the object is an instance of a subclass.

class

Class name, specified with the following syntax:

```
name | `name`
```

param

A substitution parameter, specified in the following format:

$n

where n represents a non-negative integer.

During the VQL parsing phase, a substitution parameter is recognized as a place holder which is to be filled after the parsing and before the query execution. In this way, the same query can be executed with different parameters without the need of parsing the query again and again.

For more information, on using substitutions parameters in C and C++ applications, please refer to the VQL sections in the *C/Versant* and *C++/Versant Reference Manuals* .

type

Following are valid values for type when casting const and param values:

```
char
o_1b
o_u1b
o_2b
o_u2b
o_4b
o_u4b
o_8b
o_u8b
```

```
o_float
o_double
o_object
```

Handling of types

During parsing, the parser will try to find the exact type information based on the pathattr. However, if the parser cannot determine the attribute type (for example, if there is a C++ `LinkAny` type in the path expression), it will do the conversion for the constant as follows.

Cast type is given

If the cast type is given, the parser converts the constant value based on the cast type.

Cast type is not given

If the cast type is not given, the parser converts the constant value based on the constant type in VQL language terms:

```
Integer ==>  o_4b
String  ==>  char
Loid    ==>  o_object
Float   ==>  o_float
```

For a parameter, the value type is determined at runtime.

For example:

```
Class Author: public PObject
{  ...
   public:
   LinkAny lastBook;
   ...
}
Class Book : public PObject
{  ...
   public:
   Vstr(char)   title;
   o_u4b      numberOfPages;
   ...
}
```

The following query is valid:

```
select *
```

```
   from Author
   where lastBook ->
      Book::numberOfPages < (o_u4b) 1000
```

If there is a C++ LinkAny Type in a path expression, the fullname of the attribute (such as `Book::numberOfPages`) must be specified instead of a short name: otherwise, the database won't be able to recognize the name of attribute at runtime.

Attribute name resolution for select with vstr

If you select over a vstr, the parser will not verify an attribute in the select list during the parsing, because if the select vstr is a vstr of C++ type `VstrLinkAny`, it is not possible to verify the attribute types.

When the parser sees a query over a vstr, the parser places all attribute names into the parser result, and the attribute is resolved at run time by the application. If you use the wildcard "*" in select with vstr query and the vstr is of type `VstrLinkAny`, the parser will create an `o_projection` structure with a vstr of `0` elements for attributes.

For example:

```
Class Publisher: public PObject
{  public:
   ...
   VIList<Book> topSellerList;
   ...
}
select * from (4.0.7777)->topSellerList.elements
```

The above should return objects. However, no attribute will be displayed since the parser can not resolve the attribute list for the VstrLinkAny type during parsing.

The following is a better way is to rewrite the above query and will allow you to dereference returned objects in your application:

```
select selfOid from (4.0.7777)->topSellerList.elements
```

VQL Syntax and BNF notation

VQL grammar uses a rather informal BNF notation.

- { symbol } means a sequence of 0 or n symbol(s).
- [symbol] means an optional symbol.
- keywords are presented in uppercase. (Keywords are case-insensitive for the VQL parser).
- `xxx_literal` is self explanatory, e.g., "`Hello`" is a `string_literal`.

```
statements ::= statement {; statement }
statement  ::=
SELECT select_attr_list
   FROM [ ONLY ] target [ predicate ] |
SELECT select_attr_list FROM object   |
UPDATE [ ONLY ] class SET update_list [ <predicate> ] |
UPDATE object SET update_list         |
DELETE FROM [ ONLY] class  [ <predicate> ]  |
DELETE ( loid )                       |
INSERT INTO class attr_list VALUES ( constant_list ) |
COMMIT                                |
ROLLBACK                              |
QUIT
select_attr_list ::= select_attr { , select_attr }
target           ::= class | object->attribute
predicate        ::= WHERE predicate_expression
object           ::= ( loid )
update_list      ::= update_attr {, update_attr }
attr_list        ::= attribute {, attribute }
constant_list    ::= constant {, constant }
predicate_expression ::=
predicate_term                                |
predicate_expression AND predicate_expression |
predicate_expression OR predicate_expression  |
NOT predicate_expression                      |
( predicate_expression )
predicate_term ::=
path_attribute relation_operator constant         |
path_attribute relation_operator (type) constant  |
path_attribute isa_operator identifier             |
path_attribute relation_operator parameter        |
```

```
path_attribute relation_operator (type) parameter |
path_attribute isa_operator parameter
select_attr    ::= SELFOID | * | attribute
path_attribute ::= attribute { arrow attribute }
update_attr    ::= attribute update_operator constant
class          ::= identifier
attribute      ::= identifier
constant       ::= float_literal | integer_literal |
                   string |  loid | constant_collection
constant_collection ::= open_brace constant_list close_brace
parameter      ::= dollar_sign integer_literal
loid           ::= integer_literal.integer_literal.interger_literal
identifier     ::= string_literal | `string_literal`
string         ::= "string_literal" | 'string_literal'
update_operator   ::= += | =
relation_operator ::= < | > | <> | <= | >= | = | LIKE | NOT_LIKE
isa_operator      ::= ISA | NOT_ISA
open_brace  ::= {
close_brace ::= }
dollar_sign  ::= $
arrow       ::= ->
type        ::= string_literal
```

PREDICATE TERM

In C/Versant, predicate terms are structures of type `o_predterm`. The syntax of the `o_predterm` data type is:

```
typedef struct o_predterm {
    o_attrname      attribute;
    o_opertype      oper;
    o_bufdesc       key;
    o_typetype      keytype;
    o_u4b           flags;
}   o_predterm;
```

A predicate term specifies:

- An evaluation attribute
- A comparison operator
- A key value
- An optional logical path statement.

Let's examine this structure conceptually.

Query evaluation attribute (the attribute parameter)

The "evaluation attribute" is the attribute whose value is to be examined. Evaluation attributes must be specified in a precise, non-ambiguous way that is understandable to a database.

Query comparison operator (the oper parameter)

A "comparison operator" determines how an attribute value is compared to a key value. A comparison operator can be a relational operator for comparing scalar values, a pattern matching operator for comparing strings, an "is-a" operator for determining class membership, or a set operator.

Query key value (the key and keytype parameters)

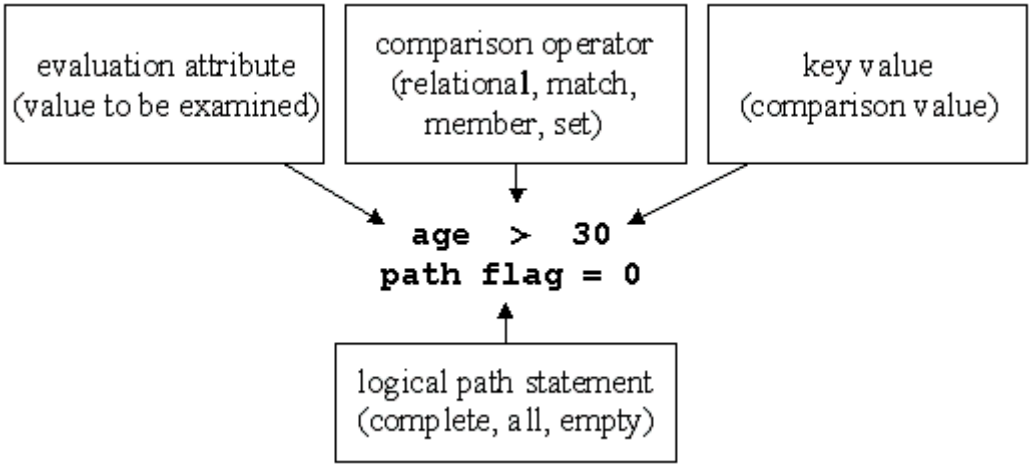
A "key value" is the value to which a specified attribute is compared. A key value can be a link, links, or a value for any type of attribute allowed in a query.

Query logical path statement (the flags parameter)

A "logical path statement" determines how to handle cases where leaf attributes (the attribute being evaluated) are null or where there are multiple leaf attributes for a particular starting point object. By default, a starting point object is returned only if a valid leaf attribute is found and the value found in the leaf attribute satisfies the specified search condition.

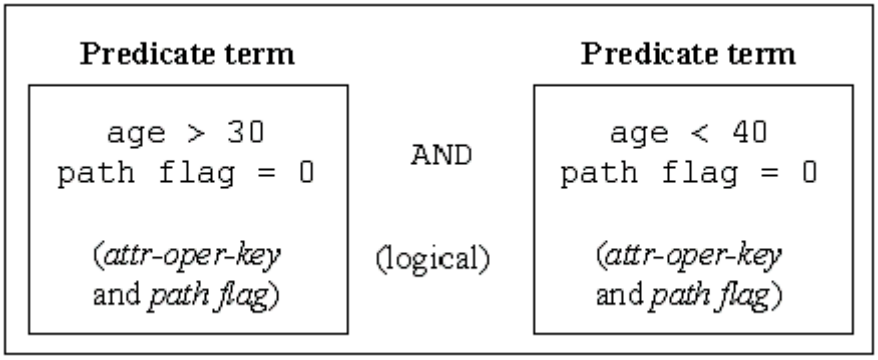
Predicate Term

Conceptually, a predicate term looks like this:

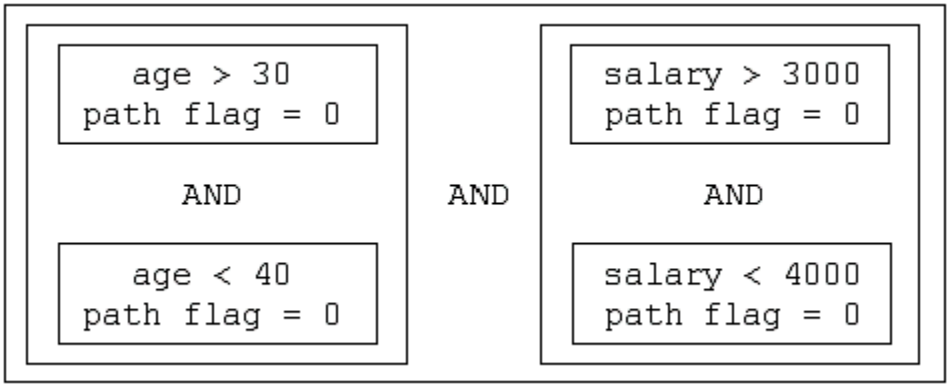


Predicates

Predicate terms can be used alone as a predicate or concatenated to form a multi-part predicate.



Also, predicate terms can be concatenated with other predicates.



The following discusses predicate term attributes, comparison operators, and path statements in detail.

Predicate Term Concatenation

When multiple predicate terms and/or predicates are used in a search query, they are concatenated with logical operators. The following boolean operators can be used to logically join predicate terms and predicates:

Boolen Operators	Description
O_AND	logical AND
O_OR	logical OR
O_NOT	logical NOT

Per precedence rules, operator O_NOT binds tightest and O_OR binds weakest.

Indexable Predicate Term

The system optimizer will automatically use an index in a query if a predicate term is "indexable."

A predicate term is indexable if all of the following are true.

- The evaluation attribute has an index on it.
- The evaluation attribute is not expressed in terms of a path.
- The comparison operator is appropriate to the type of index.

The following types of comparison operators can use the following types of indexes.

Operators	Usage	Index
O_EQ	Scalar equal to	Can use either a hash or b-tree index. If there are both kinds of indexes on the evaluation operator, the hash index will be used.
O_NE	Scalar not equal to	Does not use an index.
O_GT, O_GE, O_LT,O_LE	Scalar relative to	Can use a b-tree index.
O_MATCHES	String equal to	Can use a b-tree index, if the key string to be compared does not start with "*" or "?".
O_NOT_MATCHES	String not equal to	Does not use an index.

O_INTERSECT	Set intersection	Can use a b-tree index.
O_SUPERSET_OF	Set superset	Can use a b-tree index.
O_EQUIVALENT_SET	Set equal to	Can use a b-tree index.
O_NOT_INTERSECT	Set not intersection	Does not use an index.
O_SUBSET_OF	Set subset	Does not use an index.
O_NOT_SUPERSET_OF	Set not superset	Does not use an index.
O_NOT_SUBSET_OF	Set not subset	Does not use an index.
O_NOT_EQUIVALENT_SET	Set not equal to	Does not use an index.
O_ISA_EXACT	Class is same as	Not relevant.
O_NOT_ISA_EXACT	Class is not same as	Not relevant.

EXAMPLE

The following program illustrates how to use C/Versant to parse a VQL query statement and execute the query based on the result generated by the parser.

For this code and other examples, see your installation directory `$VERSANT_ROOT/demo/c/ivql`.

This program takes several VQL statements, parses them, and executes the query. The VQL statements involved include `SELECT, UPDATE, INSERT, DELETE, COMMIT, ROLLBACK` and `QUIT`. This program also shows how to perform parameter substitution.

```
#include <stdio.h>
#include <ctype.h>
#include <oscerr.h>
#include <omapi.h>
#include <omsch.h>
main (argc, argv)
   char **argv;
{
   o_parseblock *pblkp;      /* Parse block pointer */
   o_vstr objectvstr;        /* vstr contains objects that
                             ** are requested in a VQL
                             ** statement. */
   o_projection *projection; /* Projection structure */
   o_dbname dbname;
   o_vstr missing, *vstr_pointer;
   char bufdata[256];
   char new_bufdata[256];
   o_bufdesc new_buffer, buffer;
   o_4b num_statements, num_objs, num_params, num_attr;
   o_object object;
   o_opts options;
   o_param *param_pointer;
   o_4b i, j, k, *number;
   o_bool quit = FALSE;
   /* VQL Query Statements to process */
   char querystring[] =
      "select name, number from Employee where number > 20; \
       select name, number from (1467.0.20484)->employees \
          where number > 20; \
       select name, number from (1467.0.20485); \
```

```
    update Employee set name = 'Bill' where number = 100022; \
    update Department set employees += 1467.0.20485; \
    update (1467.0.20485) set name = 'Bill'; \
    insert into Department (name) values ('Marketing');\
    select name, number from Employee where number = $1; \
    delete from Employee where number = 100023; \
    delete (1467.0.20485); \
    rollback; \
    commit; \
    quit  ";
/* Allocate space to hold the result */
o_parseresult *parse_result =
    (o_parseresult *) malloc(sizeof(o_parseresult));
if (!parse_result)
{
   printf("Failed to allocate space for parse result.\n");
   exit(-1);
}
if (argc != 2)
{
   printf("Usage vqlsample <dbname>\n");
   o_exit(-1);
}
strcpy(dbname, argv[1]);
/* Open database */
options[0]=0;
if (o_beginsession("VQL_TRAN", dbname, "VQL_SESSION", options)
   != O_OK)
{
   printf("Failed to open database %s\n", dbname);
   exit(-1);
}
/* Parse VQL Statements */
if (o_parsequery(querystring, parse_result, dbname) != O_OK)
{
   printf("%s\n", querystring);
   printf("%s\n",1+parse_result->prs_err.prer_tokenoffset, "^");
   printf("%s\n", parse_result->prs_err.prer_errmsg);
   exit(-1);
}
```

```
num_statements = o_sizeofvstr(&parse_result->prs_pblkvstr) /
          sizeof(o_parseblock *);
/* Processing each statement */
for (i = 0; i < num_statements; i++)
{
   /* Get the handle of parse block. */
  pblkp = ((o_parseblock **)
  parse_result->prs_pblkvstr)[i];
   /* Substitute parameters if there is any */
   if (pblkp->prb_paramvstr)
   {
      number = (o_4b *) malloc(sizeof(o_4b));
      num_params = o_sizeofvstr(pblkp->prb_paramvstr) /
            sizeof(o_param);
      /* For this sample, there is only one parameter to
      ** substitute.*/
      if (num_params != 1)
      {
         printf("Unexpected number of parameters. \n");
         exit(-1);
      }
       param_pointer = ((o_param *)(*pblkp->prb_paramvstr));
      /* Set the new value buffer */
      *number = 100023;
      new_buffer.data = (o_u1b *) number;
      new_buffer.length = sizeof(o_4b);
      memcpy(&param_pointer->prm_predterm->key,
      (o_bufdesc *) &new_buffer,sizeof(o_bufdesc));
   }

/* Retrieve objects which the operation will apply. */
      switch (pblkp->prb_command)
   {
      /* Query a selection of objects */
      case O_QRY_SELECT:
      case O_QRY_UPDATE:
      case O_QRY_DELETE:
      /* Apply query */
      objectvstr =
         o_pathselect(pblkp->prb_classname,
            dbname, 0, NULL,
```

```
            pblkp->prb_options, IRLOCK,
            IRLOCK, pblkp->prb_predicate,
            pblkp->prb_vstrobj,
            pblkp->prb_vstrattr);
      if (objectvstr == (o_vstr) NULL)
      {
         if (o_errno != O_OK)

            printf ("Query execution error %d\n",o_errno);
            exit(-1);
         }
      }
      break;
      /* Get the single object */
      case O_QRY_SELOBJ:
      case O_QRY_UPDOBJ:
      case O_QRY_DELOBJ:
      /* Place the single object into objectvstr */
      if (!o_newvstr(&objectvstr, sizeof(o_object),
         (o_u1b *) &pblkp->prb_vstrobj))
      {
         printf("Fail to create vstr (err %d)\n",
            o_errno);
         o_exit(-1);
      }
   }
   /* Apply operation to the objects that have been
    *  retrieved */
   switch (pblkp->prb_command)
   {
      case O_QRY_SELECT:
      case O_QRY_SELOBJ:
         /* Get the  number of attributes */
         num_attr = o_sizeofvstr(&pblkp->prb_prjvstr) /
                    sizeof(o_projection *);
         /*
** For SELECT query of this operation, there is
** only two attributes: name, number; therefore
         ** we only expect two attributes
         */
```

```
    if (num_attr != 2)
    {
printf("Unexpected number of attributes. \n");
       exit(-1);
    }
    /* Prepare the result buffer */
    buffer.data = (o_u1b *) bufdata;
    buffer.length = 256;
    num_objs = o_sizeofvstr(&objectvstr) /
                sizeof(o_object);
    /* process each objects in objectvstr  */
    for (j=0; j < num_objs; j++)
    {
       char *attrname;
       object = ((o_object *) objectvstr)[j];
       /* Get first attribute: name */
       attrname =
          ((  (o_projection **)
          pblkp->prb_prjvstr)[0])->prj_attrname;
       if (o_getattr(object, attrname, &buffer)
             != O_OK)
       {
          printf("Error %d getting attribute.",
                      o_errno);
          exit(-1);
       }
       printf("value of attribute name: %s \n",
              *((char **)buffer.data));
       fflush(stdout);
       /* Get second attribute: number */
       attrname = (((o_projection **) pblkp->prb_prjvstr)
                     [1]) ->prj_attrname;
       if (o_getattr(object, attrname, &buffer)
             != O_OK)
       {
          printf("Error %d getting attribute.",
                o_errno);
          exit(-1);
       }
       printf("value of attribute number: %d \n \n",
              *((o_4b *)buffer.data));
```

```
            fflush(stdout);
          }
          break;
       case O_QRY_INSERT:
   /* Execute insert statement could consists of two
          ** steps:
          ** 1. Create a new object
        ** 2. Assign values on attributes. This is the
          **    same operation as UPDATE.
          */
          /* Step 1: Create a new object */
            object = o_createobj
          ((char*)pblkp->prb_classname,NULL,FALSE);
/* Step 2: Assign values on attributes using UPDATE */
          /*
   ** Once the object is created, it is placed into
   ** result vstr and apply update on the attributes
          ** based on value given in VQL.
          */
          if (!o_newvstr(&objectvstr, sizeof(object),
              (o_u1b *)&object))
          {
              printf("doInsert(error): creating vstr \
              (err %d)\n",o_errno);
          }
/* No "break" because values need to be assigned. */
       case O_QRY_UPDATE:
       case O_QRY_UPDOBJ:
          /* update objects */
      num_attr = o_sizeofvstr(&pblkp->prb_prjvstr) /
                          sizeof(o_projection *);
          /* Get number of object to update */
          num_objs = o_sizeofvstr(&objectvstr) /
                          sizeof(o_object);
            for (j=0; j < num_objs; j++)
            {
               object = (((o_object *)objectvstr)[j]);
               for (k=0; k < num_attr; k++)
               {
                  projection = ((o_projection **)
```

```
            pblkp->prb_prjvstr)[k];
            switch (projection->prj_setcmd)
            {
               /* Assign operator */
     case VQL_ASSIGN :
    /* Set new value to attribute */
      if (o_setattr(object,
      projection->prj_attrname,
      &projection->prj_buf) != O_OK)
      {
      printf("Error: o_setattr failed
      (err %d)\n", o_errno);
         }
       break;
/*
 ** Add new elements to a multi-values
 ** attributes.
 */
 case VQL_APPEND:
 /* Prepare the result buffer */
      buffer.data = (o_u1b *) bufdata;
      buffer.length = 256;
      /* Get the original collection */
      if (o_getattr(object,
      projection->prj_attrname,
      &buffer) != O_OK)
      {
       printf("Error: o_setattr \
       failed (err %d)\n", o_errno);
       }

       /* Prepare the new value buffer */
       new_buffer.data =
       (o_u1b *) new_bufdata;
       new_buffer.length = sizeof(o_vstr);
       /* Add new elements to collection.*/
       vstr_pointer =
       (o_vstr *) new_buffer.data;
       (*vstr_pointer) =
       o_concatvstr((o_vstr *)
       (buffer.data),
```

```
                (o_vstr *)
                (projection->prj_buf.data));

                /* Set new value to attribute */
                  if (o_setattr(object,
                  projection->prj_attrname,
                  &new_buffer) != O_OK)
                  {
                  printf("Error: o_setattr \
                  failed (err %d)\n",
                  o_errno);
                  }

                /* clean up */
                o_deletevstr((o_vstr *)(buffer.data));
  o_deletevstr((o_vstr *)(new_buffer.data));
                break;
                    }
                 }
              }
              break;
         case O_QRY_DELETE:
         case O_QRY_DELOBJ:
            missing = (o_vstr) 0;
            /* delete objects */
if (o_gdeleteobjs(objectvstr, NULL, &missing) != O_OK)
            {
               printf("Delete fails. \n");
               exit(-1);
            }
            break;
         case O_QRY_COMMIT:
            /* Commit transactions */
            o_xact(O_COMMIT, "VQL_TRAN");
            break;
         case O_QRY_ROLLBACK:
            /* Rollback transaction */
            o_xact(O_ROLLBACK, "VQL_TRAN");
            break;
         case O_QRY_QUIT:
```

```
            quit = TRUE;
            break;
         default:
            printf("Unrecognized command.");
            exit(-1);
            }
      if (quit)
         break;
   }
   if (number)
      free(number);
   o_freequery(parse_result);
   o_endsession("VQL_SESSION", options);
```

CHAPTER 20

Query Processing with VQL 7.0

This Chapter gives details about Versant Query Processing using the Versant Query Language 7.0.

Following topics are covered:

- Introduction
- Query Architecture
- Usage Scenario
- Evaluation and Key Attributes
- Query Language
- Compilation
- Execution
- Query Result Set
- Performance Considerations
- Differences between VQL 6.0 and VQL 7.0

INTRODUCTION

Versant Object Database provides an ability to query the database using a query language called VQL 7.0. This query language is a successor to the previous VQL version 6.0. It provides more expressive power, better efficiency, and support for features like quantification operators and server side sorting. The API offers easier and transparent functions to specify queries and iterate over the result set.

The Versant Object Database 6.0 query-related APIs and data structures are still available for use. However, users are encouraged to use the new query APIs, as Versant will deprecate the old query APIs in the future.

QUERY ARCHITECTURE

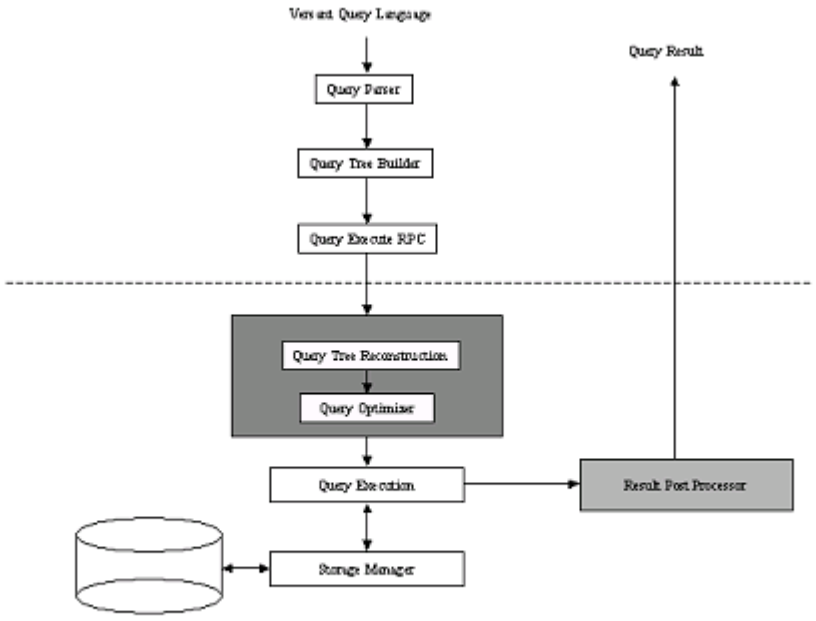
The new VQL 7.0 query interface is based on two main concepts: Query and Result.

The Query describes various aspects that determine the outcome of the query when it is executed. The Result represents the result of execution of a Query, which is a selection of objects or projection result.

The VQL 7.0 string received from the user is first compiled. The process of compilation results in the construction of an internal representation of the query. A handle to the query is returned to the user so that the user can refer to this query later in the application. This intermediate query structure is processed and sent over the network to the back end.

At the backend this query tree is rebuilt. Further processing is done on the query tree and this leads to a structure that specifies a plan for executing the query. The query is then executed based on the execution plan, and this generates the query result. This result is sent back over the network to the front end.

An identifier, referred to as a result handle, representing the query result is returned to the application. This handle can be used with the new APIs to extract the values in the query result.

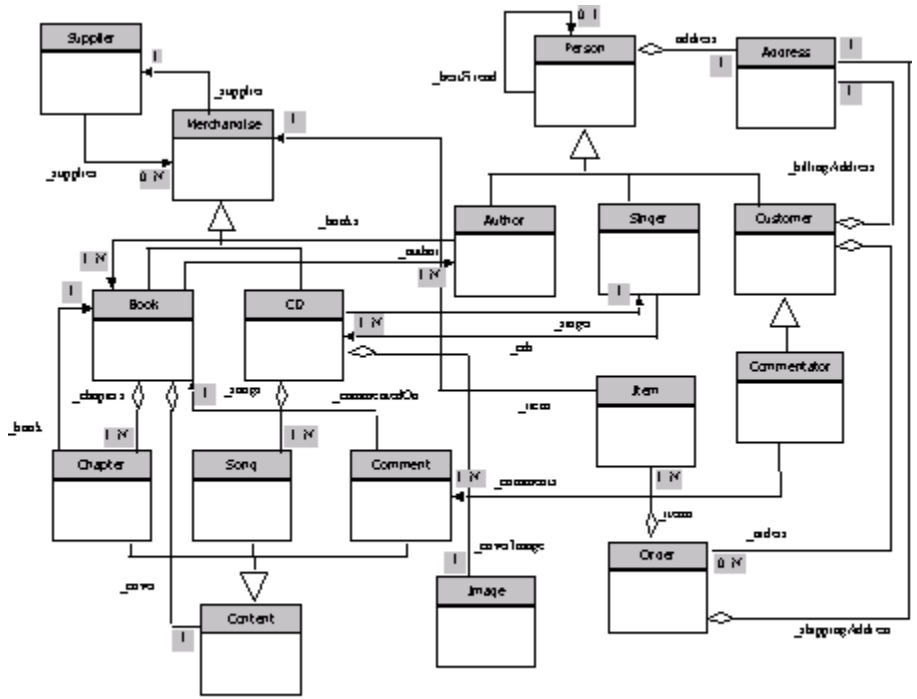


Query Processing Architecture

USAGE SCENARIO

In this chapter we will use examples to describe query features using usage scenarios. These scenarios will be described using the following data model.

Class Diagram



Model Description

The model describes a store that sells Book and CDs. These are expressed as general Merchandise.

The Merchandise has a price and is supplied by a `Supplier` and is either in stock or not. Specific merchandises are Book and CD.

`ISBN` number, an alphanumeric string, identifies each Book. An ID that the store assigns identifies a CD.

Book contains two types of Content: (a) a cover page and (b) multiple Chapter.

Content of a CD is lyrics of multiple Songs. CD may also contain an Image as a nice cover.

Book is written by one or more Author, CDs are associated with a Singer.

Both Author and Singer are specialized form of Person.

Person carries SSN as his/her primary identification and has personal details such as name, date of birth, gender and Address.

Customer is also a Person who places an Order. Order contains one or many Items.

Books are rated by Commentator. Customer becomes Commentator once he/she has made an Comment. Each Comment carries a rating and average rating for a particular Book can be evaluated.

Supplier supplies one or more Merchandise (can be either Book or CD).

EVALUATION AND KEY ATTRIBUTES

An "evaluation attribute" is the attribute whose value is to be examined. A "key value" is the value to which a specified attribute is compared.

The following explains the attribute types allowed and how to specify an evaluation attribute.

Attribute Types Allowed

Query attribute types allowed: In search functions, you can use the following types of attributes when specifying a search condition:

Description	Attributes
Elementary data types with a repeat count of 1.	`char,o_1b,o_2b,o_4b,o_8b,o_double,o_float,o_u1b,o_u2b,o_u4b,o_u8b`
Fixed size arrays of elementary Versant data type.	`fixed array of char, fixed array of o_2b, fixed array of o_u2b`
For example: `char first-name[20]`	
Some `vstrs` of one byte elemental types.	`vstr of char, vstr of o_1b,vstr of o_u1b`
Links of type `o_object`.	`o_object`
Vstrs containing links.	`vstr of o_object`
Fixed size arrays of links.For example: `o_object names[20]`	`Fixed array of o_object`
Date and time types.	`o_date, o_interval, o_time, o_timestamp`

Attribute Types NOT Allowed

You cannot use the following types of attributes in a search method:

Attributes	Description
Do not use `enum`	You should not use enum, as it is not portable: some machines implement enum as o_2b, while others implement enum as o_4b.
Do not use other types of vstrs.	Do not use vstrs of types other than char, o_1b, o_u1b, and o_object.
Do not use an embedded class or struct with a fixed length	Do not use an embedded class or struct with a fixed length that embeds another fixed array, class or struct.
Do not use vstrs of class or struct	Do not use vstrs of class or struct types. For example, you cannot use `Vstr<Employee>`.

Data type support and conversion in VQL

This section explains the data types supported and data type conversion in Versant Query Language (VQL).

Type conversion of Data

Type conversion of data is required when the attributes in an expression are not of same type.

The rule of type conversion is to convert the lower data type to higher data type.

For example:

```
Select selfoid from <class name> where attr1 >= attr2
```

The data types are grouped and represented in different form at the backend, as shown in the table given below:

Data types	Backend representation
O_2b,o_u2b,o_4b,o_u4b,o_8b,o_u8b	LONGINT
O_double	DOUBLE
O_1b/o_u1b	SIGNED/UNSIGNED DATA

The current implementation supports the type conversion of following data types. The table also shows the backend representation of these data type.

Attr1's data type	Attr2's data type	Backend Representation
o_2b/o_u2b/ o_4b/o_u4b/ o_8b/o_u8b	o_2b/o_u2b/o_4b/ o_u4b/ o_8b/o_u8b	LONGINT
LONGINT	o _1b/o_u1b (single, not an array)	LONGINT
LONGINT	o _double	DOUBLE
O_double	o _1b/o_u1b (single, not an array)	DOUBLE
String	o _1b/o_u1b	STRING

Data type supported in Predicate

The following table shows the combinations of the data types supported in the VQL expression.

In the table only those combinations are mentioned where the data type of attr1 and attr2 is different. For the same data type of attr1 and attr2 no conversion is required. This table also shows the resulting data type in case of type conversion.

For example consider the following VQL statement:

```
select selfoid from <class_name> where attr1 >/=/</!=
/>=/<= attr2
```

Attr1	Attr2	Resulting data type
LONGINT/DOUBLE/ SIGNED/UNSIGNED Data with length = 1	LONGINT/DOUBLE/ SIGNED/UNSIGNED Data with length = 1	DOUBLE
SIGNED/UNSIGNED DATA/STRING	SIGNED/UNSIGNED DATA/ STRING	STRING

Any other combinations of data types in an expression will result in data type compilation error.

Attribute Specification by Name

You can specify an evaluation attribute by name or by path. If you specify an attribute by name, you must use a special, non-ambiguous, database syntax for the name.

Precise attribute names are needed, because a derived class could conceivably inherit attributes with the same name from numerous superclasses along one or more inheritance paths. Even if there is no initial ambiguity, classes may later be created which do cause ambiguity.

Following are attribute name rules required by Versant for each type of attribute that may be used to query or index.

Name of an elemental attribute

The attribute name in the database should be used. For example, in a C++ application, the following class is defined:

```
Class_name public PObject {
   o_4b attribute_name;
};
```

In the database, the name of the attribute becomes:

```
Class_name::attribute_name
```

This form should be used even when an attribute is used in a derived class.

For example, if attributes id and code are in class Base and attribute value is in class Derived, then the database names for these attributes are:

```
Base::id
Base::code
Derived::value
```

In the following sections, we assume the schema is generated for a C++ application.

Name of a link attribute

The database name for a link attribute is:

```
classname::attribute_name
```

For example, suppose that Employee has a `worksIn` attribute that points to a Department object. The database name for the attribute is:

```
Employee::worksIn
```

This form should be used even when an attribute is used in a derived class.

Name of a fixed array attribute

Fixed arrays of char, o_1b, o_u1b, and o_object with two or more fixed length dimensions are flattened into discrete attributes, except for the last dimension, which is given to the database as a repeat count for each database attribute. The attribute names used in the database include the subscript inside of brackets.

The general form is:

```
class::attribute_name[0]
class::attribute_name[1]
class::attribute_name[...]
class::attribute_name[n-1]
```

This form should be used even when an attribute is used in a derived class.

For example consider a class in database with these attributes:

```
Class employee
{
o_1b address[2][10];
/* the address can be stored as a two line address */
o_1b name[20];
}
```

Now to write a query to search for objects which have "USA" in the second line of address field:

```
select selfoid from employee where `employee::address[1]` like
"*USA*"
```

NOTE:- Attributes with special characters like [,] etc. have to be enclosed in back quotes in VQL queries.

Now, suppose we want to search for the employees who have name starting with 'P', write the query as:

```
select selfoid from employee where employee::name like "P*"
```

The following query will give an error for the above said condition:

```
select selfoid from employee where `employee::name[0]` = 'P'
```

This is not a valid query, as the attribute "name", is an one dimension character array. The User cannot access it's value using array subscript. It will result in Query Compilation Error.

Name of a vstr attribute

The database name of an attribute that is a vstr of `char`, `o_1b, o_u1b` or `o_object` is:

```
classname::attribute_name
```

For example, if Department has an attribute named employees to hold department members that is a vstr of `o_object`, then the database name for the attribute is:

```
Department::employees
```

Name of a class or struct attribute

Embedded attributes of struct or class data types are flattened into simpler individual attributes.

For example, suppose that `embeddedObject` is an attribute of `ClassA` and is of type `ClassB`. Also suppose that ClassB has an attribute named `attributeB`. Then the database name of attributeB in embeddedObject is:

```
ClassA::embeddedObject.ClassB::attributeB
```

The name ClassA::embeddedObject.ClassB::attributeB must be used to refer to the embeddedObject attribute wherever it appears, whether in ClassA or in a class derived from ClassA.

The attribute attributeB can itself be of any type, including an embedded class or embedded struct type.

Consider the case where a class definition is like:

```
Class employee {
Char name[20];
Struct date joining_leaving_date[2];
/* to store the joining and the leaving dates */
}
```

where the struct date is:

```
Struct date {
int day;
int month;
};
```

To search for the employees who joined the company in the month of april (4), the query should be like:

```
select selfoid from employee where `employee::joining
_leaving_date[0].date::month` = 4
```

If a class name or an attribute in the class contains some special chars as: space, '[', ']' etc, then it can't be used in a query directly. Such class name or attribute name must be enclosed in back quotes ("`") in the query.

For example:

```
Select selfoid from `Tech Assistant` where `Tech Assis
tant::joining_leaving_date[0].date::month` = 4
```

If not enclosed in back quotes, it will result in Query Compilation Error.

Query Attribute Names Not Allowed

You cannot query on an attribute whose name contains an ASCII TAB character (decimal value 9).

Attribute Specification by Path

You can specify an evaluation attribute with a path name rather than an attribute name. As with names, you can use relational, match, membership, and set operators to make comparisons between a key value and an attribute specified with a path.

In ODMG, this is called a "path expression." By definition, a "path expression" specifies a path through several interrelated objects to access a destination object.

The general syntax for a path expression is a listing of the attribute names on the path to the attribute to be examined, with the names on the path delimited with \t.

For example, suppose that a class Person has a links attribute garages whose domain is class Garage. Suppose also that class Garage also has a links attribute cars whose domain is class Car, and that class Car has an attribute colors.

Now suppose that we want to search over the Person class for people with cars of a certain color. Then, to specify the attribute colors in a predicate, we can specify it with the path from Person to colors using \t:

```
Person::garages\tGarage::cars\tCar::colors
```

This is conceptually similar to specifying the colors attribute by name as:

```
Person::garages->Garage::cars->Car::colors
```

The last name in the attribute path, in this case colors, is called the "leaf name." The intermediate names, in this case garages and cars, are called "non-leaf" names.

When you use a path name for an attribute to be examined:

- All objects involved must be in the same database
- All non-leaf attributes must be a link, link array, or link vstr.
- The comparison operator must be a relational operator or a set operator.
- The attribute path can include any number of steps to the final attribute.
- The starting point objects can be either the objects in a class or the objects in a link vstr or link array.
- Once the starting point objects have been returned, you can also return objects linked to the starting point objects to a specified level of closure. The following of links to a particular level of closure is unrelated to the following of a path to a leaf attribute.

QUERY LANGUAGE

VQL 7.0 is a language that enables the user to define search queries to the database in the form of a string. The query string is compiled and the server can execute the resulting compiled query.

VQL allows the user to describe queries to search objects in the database that satisfy a certain set of search criteria.

This section describes the query syntax to specify a database query using VQL. We begin by understanding the building blocks that are used to write a VQL query. First we will look at the keywords, constants and accepted operators, followed by the query clauses which are described in detail.

VQL search queries are of the form: `select_clause from_clause [where_clause][order_by_clause] [';']`

The `where_clause`, `order_by_clause`, and `';'` are optional.

The details of how these clauses are specified are described in detail through the rest of this section.

SELECT Clause

The `SELECT` clause indicates that it is a search query. A `SELECT` clause begins with the keyword `SELECT` followed by the list projections of the query.

Selfoid

When `SELFOID` is specified the query returns the list of `LOIDs` of the objects that satisfy the search criteria.

Example

An example of a valid `SELECT` clause is as follows:

```
SELECT SELFOID
```

To find all the `LOIDs` of the objects of class `Author`, the following query is used:

```
SELECT SELFOID FROM Author;
```

Selection expressions

A selection may be a comma-separated list of expressions.

Each expression may be a path expression or a arithmetic expression.

Example

Assume there is a class `Person` and a class `Address`:

```
Person {
   String name;
   Person spouse;
   Address address;
};

and a class Address

Address {
int zip;
String city
String street;
};
```

Then, the following selection clause would select a collection of columns, where each column consists of a string, a persistent object of type `Person` and another string.

```
SELECT name, spouse, address.street from Person
```

VQL Grammar BNF

```
select_clause -> SELECT selection_list
              -> SELECT *

selection_list-> selection
              -> selection_list, selection

selection     -> SELFOID
              -> add_expr
```

FROM Clause

The FROM clause defines the source, which is to be searched to retrieve the objects that satisfy the search condition.

The FROM clause is specified by the keyword FROM followed by the source.

```
FROM source
```

There are three types of sources.

1. From class

The search query may be run on a class of objects to return qualifying objects of the class. The class is specified by the class name defined in the database. Objects of the class and all its subclasses are evaluated by default.

The query can be restricted to evaluate only objects of the class specified. The keyword ONLY is used to specify this before the class name.

An example of a query to select objects from the Class Person and all its subclasses is as follows:

```
SELECT SELFOID FROM Person
```

To restrict the query to select only objects of the Class Person one would specify:

```
SELECT SELFOID FROM ONLY Person ;
```

2. From Candidate Collection

A search query may be used to select objects in a candidate collection. The selection criteria are applied to the objects in the collection.

A candidate collection is a set of objects that belong to a particular class or its subclasses. The objects are specified using their LOIDs.

For example, a search query on objects 10.0.9876 and 10.0.9875 would be specified as follows:

```
SELECT SELFOID FROM {10.0.9876, 10.0.9875} AS Person ;
```

The class to which the objects belong to must be specified by its name using the “AS” keyword.

If an object in the collection does not belong to the specified class, it will not be evaluated, thus will not be selected.

3. From Vstr attribute of an object

A search query can be run on an attribute of an object that holds a list of links to objects. The search criteria are applied to the collection of objects that are in the list.

This collection is specified using the LOID of the object whose attribute holds the collection of objects to be queried and the name of the attribute holding the collection. The candidate list is specified as follows:

```
(LOID_CONSTANT)->attribute_name AS class_name
```

For example if we were to select all the authors of a Book with LOID 10.0.12354, the query would be as follows:

```
SELECT SELFOID FROM (10.0.12345)->Book::authors AS Author ;
```

WHERE Clause

The WHERE clause is used to specify the search predicate for the query.

The objects from the source objects that satisfy the conditions in the predicate are returned by the query.

Predicates

A predicate is an expression that evaluates to a logical value of true or false.

Predicates may be combined with other predicates using the AND, OR and NOT operators to create complex predicates.

A simple predicate can be a relational expression, set expression, string comparison expression, class membership expression, existential quantification or universal quantification expression. These expressions evaluate to true or false.

In the remaining part of this section each type of expression is described in further detail.

Relational expressions

Two arithmetic expressions A1 and A2 that evaluate to a numeric value can be compared using a relational operator R. Then,

```
A1 R A2
```

is a relational expression. The result of the comparison is a logical value, either true or false.

The relational expression is one in which two arithmetic expression values are compared. VQL provides six relational operators, `<, >, <=, >=, !=` and `=`. If an expression is true it evaluates to a value true, else false. The following expression will result in true for all Person objects with a best friend less than half her age

```
Person::age / 2 > Person::bestFriend->Person::age
```

Arithmetic expressions

There are two types of arithmetic expressions, unary expressions and binary expressions.

A unary arithmetic expression is one in which a unary operator is applied to an operand. The unary operators supported by VQL are `-, +, abs()`, and `~`. A unary arithmetic expression always evaluates to a numeric value. The operand must be a numeric type.

A binary arithmetic expression is one in which a binary operator is applied to two operands. A binary arithmetic expression always evaluates to a numeric value. The operands must be a numeric type.

Arithmetic operator Precedence

The following table defines the operator precedence for VQL arithmetic expressions. The operators are listed in the order of precedence beginning with the highest at the top of the table.

Operator	Description
`(type_name)`	Casting of an attribute to a particular type specified by type_name.
`->`	Path dereference operator
`~`	Bit wise complement operator
`- +`	Unary minus and plus operator
`* / %`	Multiply, divide, and modulo operator
`- +`	Binary subtract and add operator

String comparison and expressions

Literals, attributes or variables of type string may be compared in expressions that result in true or false.

An exact string match can be done using the = operator. If the strings are identical the expression evaluates to true, else it evaluates to false.

String values may be concatenated using the + operator. The result of concatenation of two string values is a string value.

To convert a string to uppercase, the toupper() function may be used. To convert to lowercase, the `tolower()` function may be applied. Both functions return a string values.

String values may be compared with regular expressions. For pattern matching and to check if a string matches a regular expression and with range expressions (like "[a-z]"), the `LIKE` operator is used in combination with the Wildcard Characters. The "`LIKE`" operator is used in the query because the user does not provide the exact value but a range of values or wildcard characters ('?' or '*') and thus other operators ("=", ">" etc) can't be used. If the string value matches the regular expression, it evaluates to true. If it doesn't it evaluates to false.

NOTE:- For pattern matching and with range expressions(like "[a-z]") the "`LIKE`" operator must be used. The predicate in which "`LIKE`" operator is used should have the following format "`attribute like pattern`".

For e.g.:

```
select selfoid from employee where name like "george*"
```

VQL support to Wildcard Characters

The following Wildcard Characters are supported by VQL:

·? - This is used to search for values, which contain at least one character

·* - This is used where user wants to match one or more characters

Consider the query:

```
select selfoid from Type where Type::char like <value>
```

where `Type` is the name of a user class which has an attribute as `Type::char` of `char[]` type.

Following are the sample values which can be stored in the field `Type::char`:

a
d
c
ab
bc

```
cd
de
abcd
bcde
cdef
defg
[a-z]
```

Wildcard Character - ?

```
select selfoid from Type where Type::char like "?";
```

The ? Wildcard Character will match only those objects, which has only one char (it can be any) stored in the Type::char attribute.

The results will display the loids which have these values in the Type::char field:

```
a
b
c
```

Wildcard Character - *

```
select selfoid from Type where Type::char like "*";
```

The * Wildcard Character will match all objects, which has any number of characters stored in the Type::char attribute. Thus it will display all the objects.

VQL support to Range Expression

The range expression is used to find the matching attributes characters between the given range.

For example if the range expression is '[a-z]', it will contain the characters in the range ([a-z]) (a to z in this case, inclusive of a and z).

We will consider the same query to show the syntax.

[a-z] expression

```
select selfoid from Type where Type::char like "[a-z]"
```

This will return only those objects which have only one character in the `Type::char` attribute and fits in the range of [a b c d .....z].

The results will display the loids which have these values in the `Type::char` field:

```
a
b
c
```

?[a-z] expression

```
select selfoid from Type where Type::char like "?[a-d]"
```

This will return those objects which has two characters in `Type::char` attribute, where the first character can be any character but the second must be in the range of [a-z]

The results will display the loids which have these values in the `Type::char` field:

```
ab
bc
cd
```

[a-z]* expression

```
select selfoid from Type where Type::char like "[a-z]*"
```

This will return those objects, which has any number of characters (1 or more than one) in the `Type::char` attribute starting with the character in the range [a-z].

The results will display the loids which have these values in the `Type::char` field:

```
a
d
c
ab
bc
cd
de
abcd
bcde
cdef
defg
```

\[a-z\]* expression

```
select selfoid from Type where Type::char like "\[a-z\]*"
```

This query selects only those objects, whose values include '[' and ']' characters. As these characters are used in range expression, to indicate the range e.g. [a-z], they cannot be directly used. To disable the special meaning of these characters, we have to use '\' before these characters. Now the above query will return those objects, which has data as starting from "[a-z]…" in the `Type::char` attribute.

The results will display the `loid` which has this values in the `Type::char` field:

```
[a-z]
```

Universal Quantification

Consider a collection C and a variable X and a predicate P, then

```
for all X in C : ( P )
```

is an expression. It returns true if all elements in collection C satisfy the predicate P and returns false otherwise.

For example:

```
FOR ALL a IN Book::authors AS Author : ( a->Person::age > 35 )
```

This returns true if all Author objects in collection Book::authors has Person::age > 35.

If the collection is empty the universal quantifier expression evaluates to true.

Specifying the collection type

The type of the elements of the collection can be specified using the AS clause. This is useful when the attribute is of null domain type or the LOID constant collection is specified. The type of the variable used in the expression is set to the type of the collection.

Variables and their scope

Variables are identifiers that are used in for all or exists expressions. They are used to iterate through a collection in quantification expressions.

The variable name must not have the same name as an attribute name of the class or its super classes being queried. If so, the variable name will mask the attribute name. Therefore, their names must be chosen carefully.

A variable is used to iterate the collection and the predicate is applied to it. The variable can be used in the predicate associated with the “for all” or “exists” expression. A variable is not accessible outside the scope of the predicate it is first used in.

Existential Quantification

Consider a collection C and a variable X and a predicate P, then the following is an expression:

```
exists X in C : ( P )
```

It returns true if at least one element in collection C satisfies the predicate P and returns false otherwise.

For example:

```
EXISTS a IN Book::authors AS Author : ( a->Person::name = “Sam” )
```

This returns true if at least one Author object in collection `Book::authors` has a `Person::name “Sam”`.

If the collection is empty the existential quantifier expression evaluates to false.

Collection Membership Testing

If C is a collection of a certain type and X is an element of the same type, then

```
X in C
```

is an expression. It returns true if X belongs to the collection C.

For example,

```
Person::age IN { 9, 16, 17 }
```

Results to true if `Person::age` is either 9, 16, or 17, else results in false.

Set expressions

Set operators may be applied to collections or sets. A singleton object or value is considered a set of cardinality of one. The set expression evaluates to a value of true or false. If S1 and S2 are two sets then the expression, then the following is a set expression.

```
S1 intersect S2
```

It returns true if `S1` and `S2` have at least one common element.

The set operators of this type are described in the following table:

Set Operators	Description
`intersect`	Evaluates to true if there is at least one element common to both operand sets, results in false otherwise
`not_intersect`	Evaluates to true if there are no elements common in both operand sets, results in false other wise
`subset_of`	Evaluates to true if all elements of the left operand set are elements of the right operand set, returns false otherwise
`not_subset_of`	Evaluates to true if at least one element of the left operand set is not an element of the right operand set, returns false otherwise
`superset_of`	Evaluates to true if all elements of the right operand set are elements of the left operand set, results in false otherwise
`not_superset_of`	Evaluates to true if at least one element of the right operand set is not an element of the left operand set, results in false otherwise
`equivalent_to`	Evaluates to true if all elements of the left operand set are elements of the right operand set, and all the elements of the right operand set are elements of the left operand set, results to false otherwise
`not_equivalent_to`	Evaluates to true if there is at least one element in either the left or right operand set that is not the element of the other, results in false if the sets are equivalent

To find if the Person's telephone number is either 2800016 or 2800021, the predicate will be written as follows:

```
Person::tel_nos INTERSECT { 2800016, 2800021 }
```

Class membership testing

An object O can be tested to be of a certain class type C.

The expression below, is an expression that evaluates to true if the type of the object O is exactly class C, else it evaluates to false.

```
O ISA C
```

Identifiers

Identifiers are names of variables, attributes and classes. C++, Java and C identifiers are generally accepted as valid VQL identifiers.

So the following - `::MyProduct::MyApplication::MyClass` is an accepted identifier that could be a class defined in the database. Similarly, `com.MyCompany.MyApplication.MyClass` is a Java class and is accepted as an identifier.

C++ template class names are also accepted. Thus, `VISet<Person>` is an accepted class name.

VQL accepts identifiers that satisfy the following regular expression:

```
("::")?[a-zA-Z_][0-9a-zA-Z_<>]*(("::"|".")[a-zA-Z_][0-9a-zA-Z_<>]*)*
```

If an identifier is a valid class name in the database it is recognized as a type name. Variables are identifiers that are used in for all or exists expressions.

There can be a class name/attribute name which could be a VQL reserved word or has spaces or characters '[', ']'. These are not accepted as part of an identifier by the parser. In such cases, they can still be used in the query, by specifying the identifier within back quotes ("`").

For example:

```
Select selfoid from `Tech Assistant` where
`Tech Assistant::joining_leaving_date[0].date::month` = 4
```

So `Order` is an identifier of name Order. Ordinarily an attribute with the name Order would be recognized as a VQL keyword. Similarly, `Last Name` can be a valid identifier with a space within.

If they are not specified within back quotes in the query, it results in a query compilation error.

Class names

Class names are identifiers that are names of classes that must be defined in the database schema. A valid class name is defined as an identifier that is a database schema class. Therefore, if `Person` is specified as a class name, there must be a class in the database schema by the name `Person`. If a correct class name is not specified the compilation process fails due to a syntax error.

Attribute names

Attribute names are identifiers that must have a database attribute object of that name. Therefore, if `Person::age` is specified as a attribute name, there must be an attribute in the database schema by the name `Person::age`. Specifying an invalid attribute name will result in a query compilation error.

Parameters

A query predicate may be parameterized using parameters. They provide an ability to set a constant value in the predicate without recompiling the query.

If we wanted to write a query to find the objects of the `Author` class whose best friend is older than 30. The predicate would look like the following:

```
SELECT SELFOID FROM Author
WHERE Author::bestFriend->Person::age > 30 ;
```

If this query needed to be run to search for best friends whose ages were greater than 45, the query would need to be recompiled. The query recompilation can be avoided by using parameters. We can write the above query using a parameter to hold the constant 45 as follows:

```
SELECT SELFOID FROM Author
WHERE Author::bestFriend->Person::age > $bestfriendsage ;
```

The query can be executed many times by setting different values for parameter `$bestfriendsage`.

The value of the parameter must be set after query compilation and before executing the query. The C API `o_querysetparam()` is used to set the parameter value.

For more information, please refer to "o_querysetparam()" parameter in the Chapter "Functions and Macros Reference" in the *C/Versant Reference Manual*.

Parameters begin with a $ followed by a series of alphabets, digits or underscore characters. $Name is a valid parameter name. So is `$1969` and `$my_best_friend`.

A parameter is a token that satisfies the regular expression:

```
$ [ a - z A - Z 0 - 9 _ ] +
```

Using Constants, Literals and Attributes

The operands in an expression may be of different kinds. This section defines the different kinds that may be used in the expressions.

LOID constants

LOID or Logical Object Identifier constants tokens specify the identity of an object.

A LOID has three parts, separated by dots. The first part is the database identifier part and is a 16-bit value. The second and third parts form the object identifier. The first part of the object identifier is a 16-bit value and the second part is a 32-bit value.

`256.123.8789876` is a valid LOID constant.

A LOID constant is defined by the following regular expression:

```
{digit}+"."{digit}+"."{digit}+
```

where digit is a decimal digit between 0 and 9.

Integer constants

Integer constants may be represented in hexadecimal, octal or decimal format.

1234, `0xAB345`, 0Xabcd and 012367 are examples of valid integer constants.

A valid decimal integer constants is defined by the following regular expression:

```
{digit}+ (u|U|l|L)?
```

where digit is a digit between 0 and 9.

A valid hexadecimal integer constants is defined by the following regular expression:

```
0[xX]{hexdigit}+ (u|U|l|L)?
```

where hex digit is a digit between 0 and 9 or an alphabet between a and f or A and F.

A valid octal integer constants is defined by the following regular expression:

```
0{octdigit}+ (u|U|l|L)?
```

where octdigit is a digit between 0 and 7.

Integer constants values are interpreted as values of type `o_8b`.

Floating point constants

Some examples of floating point number constants are 7.1, 8., .67, 0.52e-5, 6.34e10f and 89.3E4L.

Floating point constants may be defined by one of the following regular expressions:

```
{digit}+ {[Ee][+-]?{digit}+} (f|F|l|L)?
{digit}* "." {digit}+ ([Ee][+-]?{digit}+)? (f|F|l|L)?
{digit}+ "." {digit}* ([Ee][+-]?{digit}+)? (f|F|l|L)?
```

where digit is a digit between 0 and 9.

Floating-point constants values are interpreted as double precision values or of type o_double.

Character constants

A character constant is one, or more, characters enclosed within single quotes (‘). The back-slash (\) character can be used as an escape character just as it is in C language.

Examples of character constants are 'a' and '\n'.

A character constant is defined by the following regular expression:

```
'(\\.|[^\\'])+'
```

String constants

A string constant is zero, or more, characters enclosed within double quotes (").

An example of a string constant is "string foo".

An empty string is specified by two double quotes with no character in between them, such as "". The back-slash (\) character can be used as an escape character just as it is in C language.

String constants are defined as characters enclosed within double quote characters. The following regular expression defines string constants:

```
\"(\\.|[^\\"])*\"
```

Boolean constants

The keywords TRUE and FALSE define the values for boolean constants true and false respectively. Attribute path expressions of boolean type many be compared with these constants.

For example,

```
Person::statement = TRUE
```

NOTE:- VQL currently does not support casting of elementary types.

For example:

`Person::age = (o_2b)30` is not supported.

The only supported type for casting is the path expression.

See also "Casting" on page 539.

Path expressions and Attributes

Attributes of objects of the source class or collection can be used in the predicate. The attribute name defined in the schema must be used to refer to it.

For example, the SSN of a `Person` object would be referred to as `Person::ssn` in a C++ application.

To find a Person object whose SSN is 540293 the query would be:

```
SELECT SELFOID FROM Person WHERE Person::ssn =  540293 ;
```

Path expression

An attribute of the candidate object may be a link to another object. If the search condition tests the value of an attribute of the referenced object, then an attribute path is used.

For example, if we wish to search the object of the `Book` class for which the Author has an SSN of 534223. The query would look like the following:

```
SELECT SELFOID FROM Book WHERE Book::author-> Person::ssn =  534223 ;
```

Restrictions

The -> operator is called the dereference operator.

The dereference operator may be applied only to class reference types and not primitive types. During execution of a query, if a path is broken due to a link that is NULL, the particular object is ignored.

Fan-out

An attribute may be dereferenced only if it has a cardinality of one. Object link VStr or array attributes cannot be dereferenced.

Null domain types

An attribute may not have any defined type. It is said to have “null domain”. An attribute that is link and has null domain cannot be dereferenced without specifying its type. Attributes in the path with null domain must be cast to a valid type. This is necessary to verify that dereferenced attribute is a valid attribute of the class.

Casting

Attributes in a path expression with null domain must be set to a valid type. The type of the attribute being dereferenced can be set using a cast operator.

An attribute is cast to a type by preceding the attribute name with the type name enclosed in round brackets. For example, to cast the Author of a Book to Person the expression would look as follows:

```
SELECT SELFOID FROM Book
WHERE (Person)Book::author->Person::ssn = 1;
```

For a path expression with greater depth you have to set the cast operator immediately before the link attribute. For example, to cast the address link attribute of the boss to be a link to class Address the expression would be as follows:

```
SELECT SELFOID FROM Person
WHERE (Person)boss->(Address)address->street;
```

ORDER BY Clause

The `ORDER BY` clause is used to specify the sort order on the objects that satisfy the query search criteria. The sort order of the result set can be defined on a simple expression involving an attribute of the object. The default sort order is ascending; this may be overridden using the `DESC` keyword to specify descending order. The ascending sort order may be explicitly specified using the `ASC` keyword.

The general form of an `ORDER BY` clause is as follows:

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC]]*
```

If the objects of `Person` class were to be sorted according to their social security number in ascending order, the query would look like:

```
SELECT SELFOID FROM Person ORDER BY Person::ssn ASC ;
```

Additional expressions to define sort order may be specified. The objects are sorted according to the first expression and then the next and so forth.

In the ORDER BY clause the Arithmetic operators like +, -, *, / can be used. However the comparison operators like <, >, >=, <=, !=, = should not be used.

VQL Reserved words

VQL has a set of reserved words that are case insensitive. They are listed below:

`select`	`selfoid`	`from`	`only`	`as`
`where`	`order`	`by`	`asc`	`desc`
`not`	`and`	`or`	`for`	`all`
`exists`	`in`	`isa`	`not_isa`	`like`
`not_like`	`subset_of`	`not_subset_of`	`superset_of`	`not_in`
`not_superset_of`	`intersect`	`not_intersect`	`equivalent_to`	
`not_equivalent_to`	`toupper`	`tolower`	`abs`	`true`
`not_in`	`false`			

VQL Grammar BNF

The VQL grammar is described in an informal BNF notation.

{ symbol }	is 0 or n symbol(s)
[symbol]	is an optional symbol
KEYWORD	is a terminal of the grammar. VQL keywords are case insensitive.
CHARACTER	is a literal of the mentioned type

query → select_statement [**;**]

select_statement → select_clause from_clause where_clause order_by_clause

select_clause → **SELECT** select_criteria

select_criteria → select_expr

→ select_criteria, select_expr

select_expr → **SELFOID**

→ add_expr

from_clause → **FROM** class_name

→ **FROM** candidate_collection

→ **FROM** candidate_vstr

class_name → **IDENTIFIER**

candiate_collection → { candidate_list } as_type_name

candidate_list → candidate

→ candidate_list , candidate

candidate → **LOID**

as_type_name → **AS** class_name

candidate_vstr → **(** candidate **)** arrow_operator attribute [as_type_name]

arrow_operator → **->**

attribute → **IDENTIFIER**

where_clause → **WHERE** predicate

predicate → or_expression

or_expression → and_expression

→ or_expression or or_expression

or → **OR**

→ **||**

and_expression → not_expression

→ and_expression and and_expression

and → **AND**

→ **&&**

not_expression → conditional_expr

→ not not_expression

not → **NOT**

→ **!**

conditional_expr → for_all_expression

→ exists_expression

→ in_expression

→ set_expression

→ isa_expression

→ like_expression

→ relational_expr

→ **(** predicate **)**

for_all_expression → **FOR ALL** variable IN in_collection **: (** predicate **)**

exists_expression → **EXISTS** variable in_operator in_collection **: (** predicate **)**

in_operator → **IN** or **NOT_IN**

in_condition → path_attribute **IN** collection

→ constant_value **IN** collection

in_collection → path_attribute [as_typename]
→ constant_collection
→ candidate_collection
constant_collection → **{** constant_list **}** as_typename
constant_list → constant
→ constant_list , constant
constant → **CHARACTER**
→ **INTEGER**
→ **FLOAT**
→ **STRING**
collection → { candidate_list **}**
→ { constant_list **}**
set_expression → set set_operator set
→ unary_set_oper set
set_operator → **INTERSECT**
→ **NOT_INTERSECT**
→ **SUBSET_OF**
→ **NOT_SUBSET_OF**
→ **SUPERSET_OF**
→ **NOT_SUPERSET_OF**
→ **EQUIVALENT_TO**
→ **NOT EQUIVALENT_TO**
set → collection
→ constant
→ candidate
→ path_attribute
isa_expression → path_attribute_or_loid isa_operator class

```
path_attribute_or_loid → path_attribute
                       → LOID
isa_operator           → ISA
                       → NOT_ISA
like_expression        → string_expr   like_operator   string_expr
like_operator          → LIKE
                       → NOT_LIKE
string_expr            → STRING
                       → TOLOWER ( STRING )
                       → TOUPPER ( STRING )
                       → string_expr  +  STRING
relational_expr        → add_expr
                       → add_expr   relational_oper   add_expr
relational_oper        → <=
                       → >=
                       → !=
                       → =
                       → <
                       → >
add_expr               → mult_expr
                       → add_expr   +   add_expr
                       → add_expr   -   add_expr
mult_expr              → unary_expr
                       → mult_expr   *   mult_expr
                       → mult_expr   /   mult_expr
                       → mult_expr   %   mult_expr
unary_expr             → basic_expr
```

→ **-** unary_expr

→ **+** unary_expr

→ **ABS (** add_expr **)**

→ ~ unary_expr

basic_expr → constant

→ path_attribute

→ **PARAMETER**

→ **(** or_expression **)**

path_attribute → [type_cast] attribute

→ path_attribute arrow_operator [type_cast] attribute

attribute → **IDENTIFIER**

type_cast → **(** class_name **)**

order_by_clause → **ORDER BY** sort_criteria

sort_criteria → sort_expr

→ sort_criteria **,** sort_expr

sort_expr → add_expr [asc_or_desc]

asc_or_desc → **ASC**

→ **DESC**

COMPILATION

To run a query, it must first be compiled so that the server can execute it. The compilation process parses the query specified as a string and generates an in-memory intermediate format that can be understood by the database server.

The exact composition and layout of this intermediate structure is internal to Versant.

The compilation process generates a handle to this intermediate structure and is referred to as a query handle.

The user can use this query handle to execute the query to obtain the query result. The result is returned in the form of a result handle. The process of accessing the query result is described in further detail later in this chapter.

Query Handle

The compiled query handle can be used at any time during the life of the session. If the query handle is not going to be used by the application, the resources can be released and destroyed using the appropriate API. Once it is destroyed, it can no longer be used.

Ending the session implicitly destroys the query handle, i.e. it need not be explicitly destroyed. But it is safer and better programming practice to explicitly destroy a query handle than to depend on the session destruction to take care of query handle destruction.

Error Handling

The compilation process may fail due to an invalid query string or other reason. If the query string is invalid the appropriate error code is returned and the location of the failure within the query string is returned.

Usage Notes

C/Versant API

o_querycompile()

Compiles a VQL 7.0 query specified by a string. On success, a handle to a compiled query structure is returned. If there is an error in the input parameters or query string, the appropriate error code is returned and the error position within the string is set accordingly.

o_queryhandle()

If a query compiles successfully, a handle to a structure representing the compiled query is returned. This handle is of type `o_queryhandle`. The query may be executed many times during the session using this handle. The structure represented by the handle contains query settings such as lock modes and parameter values that are used while executing the query to obtain the query result set. These settings and parameters may be set or altered before execution of the query to suite the application purposes.

o_querydestroy()

The o_queryhandle passed to `o_querydestroy()` should be valid and obtained using the `o_querycompile()` API. Passing an invalid query handle can result in undefined behavior.

For more information, on C/Versant APIs, please refer to the chapter, “Functions and Macros Reference” in the *C/Versant Reference Manual*.

C++/Versant Classes

In the C++/Versant binding, a new class `VQuery` is introduced. The VQuery class contains methods that construct, compile and execute query statements written in the Versant Query Language 7.0.

For more information, please refer to Section “VQuery” in the chapter “Query Reference” in the *C++/Versant Reference Manual.*

Java/Versant Classes

In the JAVA/Versant interface, a new class Query is introduced. The Query class contains methods that construct, compile and execute query statements written in the Versant Query Language 7.0.

For more information, please refer to chapters “Fundamental JVI” and “Transparent JVI”, in the *JVI Reference Manual*.

EXECUTION

Overview

After a query has been successfully compiled, the query can be executed against the “candidate objects” specified in the query.

The candidate objects can be:

- objects of a entire class with or without subclasses
- a set of given objects
- objects within an attribute of type “vstr” of a given object.

The execution takes place at the back end. The results will be sent to the front-end once the query gets executed.

Options and parameters need to be specified to control:

- Whether objects that have been modified and are being cached by the front-end need to be flushed to the back-end first so that they can participate the query.
- The lock modes that need to be placed on objects during the execution to have a given “isolation level”.
- The batch size indicating the number of objects to be retrieved in each batch.
- Send loids or also the qualified objects to the front-end. By default only the loids are sent to the front-end. If qualified objects and their inter-connected objects need to be retrieved, then the query options need to be set accordingly.

The interface to execute a query varies depending on the programming language being used.

Usage Notes

C/Versant APIs

In C/Versant, there are two ways to execute a query.

o_queryexecute()

A query statement must first be compiled, then the compiled query can be executed. When a query is compiled, a handle of type `o_queryhandle` for the query is returned. This handle can be used to actually perform the query against the database. Once compiled, the query handle remains valid within the transaction and can be reused to perform the same query again and again without the need to re-compile.

o_querydirect()

In this API, the query statement is executed directly without going through the compilation step. Internally, the query statement will be compiled and executed.

Once the query is executed successfully, qualified objects will be returned to the front-end through a result set handle of type `o_resulthandle`. This handle should then be used to access and iterate through the results.

For more information, about the C/Versant APIs, please refer to the chapter "Functions and Macros Reference" in the *C/Versant Reference Manual*.

C++/Versant Classes

In C++/Versant a query is encapsulated by instance of `VQuery` class.
To execute the query, the method `VQuery::execute()` of the `VQuery` class needs to be invoked. This method returns an instance of class `VQueryResult`. This `VQueryResult` instance should then be used to iterate through the selected objects.

For more information, about VQuery and VQueryResult, please refer to the chapter "Query Language" in *C++/Versant Reference Manual* and the chapter "Queries" in the *C++/Versant Usage Manual*.

Java/Versant Classes

In Java/Versant, a query is represented by an instance of `Query` or `FundQuery` class.
To execute a query, the "`execute()`" method from respective class needs to be invoked. This method returns an instance of class `QueryResult` or `FundQueryResult`. This resulting instance can then be used to iterate through the selected objects/handles.

For more information on Query classes, please refer to the *JVI Reference Manual*.

Setting “Candidate Objects”

Before actually executing a query, the “candidate objects” must be specified.

The common usage of a query is to select objects from the objects of a given class, eventually including the subclasses. In this case, the query statement should be specified as against a given class. After compilation, nothing special needs to be done before execution.

Other than select objects from a given class, a query can be used to select objects from a given set of objects.

There are two cases:

- The original query statement is itself a candidate collection query i.e. query string had a list of loids specified, then the candidate collection does not need to be set again. In fact, setting a candidate collection on that query has no effect in this case.
- The original query is against some class, in this case the target candidate collection must be set.

The API to be used to set the candidate collection depends on the programming language used.

API Notes

The following APIs can be used to set or get the candidate collection for a query:

Component	API
C/Versant	`o_querysetcandidatecollection()`
	`o_querygetcandidatecollection()`
C++/Versant	`VQuery::set_candidatecollection()`
	`VQuery::get_candidatecollection()`
Java/Versant	`Query setCandidateCollection()`
	`Query getCandidateCollection()`
	`FundQuery setCandidateCollection()`
	`FundQuery getCandidateCollection()`

Setting Parameters

If a parameter is specified in the query, the parameter should be bound before the execution. If the query is executed without assigning a value to any parameter, an error

QRY_PARAM_NOT_SUBSTITUTED is thrown. Any previously assigned value to the same parameter will be overwritten by the new value.

The API to bind the actual value to a parameter differs depending on the programming language used.

Component	API
C/Versant	`o_querysetparam()`
	`o_querygetparam()`
C++/Versant	`VQuery::set_param()`
	`VQuery::get_param()`
Java/Versant	`Query bind()`
	`FundQuery bind()`

Setting Options

Various options can be specified before executing a query in order to control the following:

- Whether objects that have been modified and are being cached by the front-end need to be flushed to the back-end first so that they can participate the query.
- How the resulting objects will be sent back to the front-end, e.g. should the selected objects, eventually with interconnected objects, be retrieved on the fly or just loids of these objects need to be sent back.

The valid values of options are:

O_QRY_PIN_OBJS

Pin fetched objects that are selected by the query into memory. To fetch objects when the query is executed, the user must specify the O_QRY_FETCH_OBJS option. Fetched objects are not pinned by default.

Pinned objects can be unpinned using `unpinobj()` or they can be released by a commit, rollback, or undosavepoint operation.

O_QRY_FLUSH_NONE

Prevent flushing of the object cache before performing the query. This option cannot be used in conjunction with the O_QRY_FLUSH_ALL option.

O_QRY_FLUSH_ALL

Flush the object cache before performing the query. O_QRY_FLUSH_ALL cannot be used in conjunction with the O_QRY_FLUSH_NONE.

If neither the O_QRY_FLUSH_NONE nor the O_QRY_FLUSH_ALL option is set, the default behavior, before executing the query on the class, is to flush all objects from the object cache belonging to it. If the query is a deep query, i.e. it needs to evaluate all subclasses of the given class then all the dirty objects of the subclasses are also flushed to the server.

If the query is a candidate collection query in which the candidate collection is specified in the query string as set of loids, then only objects identified by these loids are flushed to the server.

If the query is over a class and a candidate collection vstr is specified using VQuery::set_candidatecollection() , then the default behavior is to flush the objects contained in the candidate collection vstr.

If the query is of select with vstr kind, then the default behavior is to flush only the object corresponding to the starting loid. In general, an object is flushed when it is dirty and also belongs to the target query database.

O_QRY_FETCH_OBJS

Fetch the objects that satisfy the query. The objects that satisfy the query are not fetched to the object cache by default.

O_QRY_DEFAULT

Default value for query options.

These options can be set through the following APIs:

Component	API
C/Versant	o_querysetoptions()
	o_querygetoptions()
C++/Versant	VQuery::set_options()
	VQuery::get_options()

Java/Versant	`Query setQueryExecutionOptions()`
	`Query getQueryExecutionOptions()`
	`FundQuery setQueryExecutionOptions()`
	`FundQuery getQueryExecutionOptions()`

Setting Lock Mode

In order to have an isolation level, short lock will be set on the class object and the objects being selected.

Class lock mode: The short lock to be placed on the class object on which query is to be executed.

Instance lock mode: The short lock to be placed on the selected objects.

The choices for these lock modes determine the isolation level of the query.

The isolation level of a transaction is a measure of how query can affect and be affected by operations in other concurrent transactions.

For more information refer to “Locks and Transactions” on page 92, in "Chapter 5 - Locks and Optimistic Locking".

The following APIs can be used to set the lock to be placed on the class object and instances that qualify the query:

Component	API
C/Versant	`o_querysetclslockmode()`
	`o_querysetinstlockmode()`
C++/Versant	`VQuery::set_cls_lockmode()`
	`Vquery::set_inst_lockmode()`
Java/Versant	`Query setClassLockMode()`
	`Query setInstanceLockMode()`
	`FundQuery setClassLockMode()`
	`FundQuery setInstanceLockMode()`

QUERY RESULT SET

Once a query has been successfully executed, results of the query will be returned to the user through a data structure, called a result handle. This handle can be used to iterate over the result set.

In C++/Versant, a `VQueryResult` object will be created. In JVI, `QueryResult` or `FundQueryResult` will be created.

Access the result set

The result handle can be used to retrieve the qualified objects, either one at a time or in a batch.

When the query is executed, the record position is set before the first record. If the record position is set at before first record, a call to `o_resultgetobject()` will throw the error

`QRY_RESULT_CURRENT_POSITION_AT_BEFORE_FIRST`. To move the record pointer forward, use `o_resultnext()`.

Various operations can be performed on the result set, like retrieving the number of objects that satisfied the query, advancing the current record position etc. These are listed here for reference. For details, please refer to the respective language binding manuals.

Component	API
C/Versant	`o_resultgetobject()`
	`o_resultgetobjects()`
	`o_resultgetrowinfo()`
	`o_resultrowinfofree()`
	`o_resultgetcolvalue()`
C++/Versant	`VQueryResult<T>::get_object()`
	`VQueryResult<T>::get_objects()`
	`VQueryResultAny::get_object()`
	`VQueryResultAny::get_objects()`
	`VQueryResultAny::get_column_value()`

Java/Versant

```
QueryResult next()
QueryResult nextAll()

FundQueryResult next()
FundQueryResult nextAll()
FundQueryResult nextRow()
FundQueryResult nextRows()
FundQueryResult allRows()
```

Fetch size

The result object encapsulates the actual query result.
What happens when the user retrieves results through the result object varies depending on the setting of “fetch size” parameter of the query.

When the fetch size is set to a non positive number, the backend will execute the query in one shot. It populates the result set with the loids of all the qualified objects, locks them with the specified lock mode and sends the result set back to the front-end. The front-end caches the select result.

If the fetch size is set to a positive number, there are two cases:

The “`ORDER BY`” clause is specified in the query statement;

The “`ORDER BY`” is not specified in the query statement.

If “`ORDER BY`” is specified, the backend will execute the query in one shot, populate the result set by all qualified objects, sort the selected objects according to the “`ORDER BY`” clause and lock them according to the specified lock mode. After that, it sends only about “fetch size” number of objects to the front-end as the first batch trough the query result handle. The front end uses the returned handle to browse the selected objects. When the end of the current batch is about to be reached, another batch of about “fetch size” objects will be fetched transparently from the server.

The “fetch size” is only a hint for the front-end to automatically fetch next batch of objects from the server. The actual number of objects fetched is decided by the front-end itself by an internal logic. The application has no control over when and the number of objects being fetched.

If "ORDER BY" is not specified, the back-end will execute the query against the candidate objects. It stops when it reaches the number of objects asked by the front-end. Locks and populate the result set by the selected objects and sends the results set back to the front-end. The front end uses the returned handle to browse the selected objects. When the end of the first batch is about to be reached, another batch of about "fetch size" objects will be fetched transparently from the server. The server then resumes evaluation from where it stops and stops again when the number of objects being asked by the front-end is reached, or when there are no more objects to be evaluated.

Similarly to the previous case, the number of objects asked by the front-end is about the specified fetch size but the actual number is calculated through the front-end's own internal logic. The application has no control over when and the number of objects being fetched.

Set a batch size hint specifying how many records should be brought to object cache in a single request. This setting would be effective for any subsequent requests to bring records to front-end. To bring all remaining records to the cache, the `fetch_size` can be set as –1 in the result handle. Setting the value of the fetch_size is meaningless if the original query was run with a default value of fetch_size as –1 because there would not be any more records to be brought to the front-end.

To set the fetch size parameter, the following APIs can be used:

Component	API
C/Versant	`o_querysetfetchsize ()`
	`o_resultgetfetchsize()`
	`o_resultsetfetchsize()`
C++/Versant	`VQuery::set_fetch_size()`
	`VQuery::get_fetch_size()`
	`VQueryResult::set_fetch_size()`
	`VQueryResult::get_fetch_size()`

Operations on result set

Candidate Collection

The candidate collection on which the query was executed can be retrieved from the result set:

Component	API
C/Versant	`o_resultgetcandidatecollection()`
C++/Versant	`VQueryResult<T>::get_candidatecollection()`

Parameter Substitution

The parameter values set during query execution can be retrieved from the result set:

Component	API
C/Versant	`o_resultgetparam()`

Query Options

The query options set during query execution can be retrieved from the result set.

Component	API
C/Versant	`o_resultgetoptions()`
C++/Versant	`VQueryResult<T>::get_options()`

Java/Versant	`QueryResult isFetchObjects()`
	`QueryResult isFlushAll()`
	`QueryResult isFlushNone()`
	`QueryResult isPinObjects()`
	`FundQueryResult isFetchObjects()`
	`FundQueryResult isFlushAll()`
	`FundQueryResult isFlushNone()`
	`FundQueryResult isPinObjects()`

Lock Modes

The result handle contains information about the class and instance lock modes that were set at the time the query was executed.

Component	API
C/Versant	`o_resultgetclslockmode()`
	`o_resultgetinstlockmode()`
C++/Versant	`VQueryResult<T>::get_cls_lockmode()`
	`VQueryResult<T>::get_inst_lockmode()`

PERFORMANCE CONSIDERATIONS

In general, query is an expensive and resource consuming operation. Therefore, query execution performance is critical for the application.

The following factors have impact on the performance:

- Memory usage
- Disk access
- Concurrency

Memory usage for queries with “ORDER BY” clause

The server will sort the query result at the back end if “order by” clause is specified or if projection of attributes is specified.

For Versant 7.0 release, the sorting is done in memory. Depending on the number of qualified objects for the query, the quantity of memory consumed for sorting varies. If a large number of objects are selected, the memory used for sorting can be very large. A heavy consumption of memory for sorting can have negative impact on the overall performance of the system. To limit the memory usage for sorting, parameters can be set for the server in profile.be file.

There are two parameters related directly to sorting for “ORDER BY”:

```
# The maximum size to which the sorting memory for each thread can
# grow.
max_sorting_memory_per_query      5M
# The maximum size to which the sorting memory areas of all threads put
# together can grow.
max_sorting_memory_total        10M
```

If the memory used for sorting on behalf of a particular query exceeds the amount of memory specified by max_sorting_memory_per_query in profile.be, error QRY_SORT_MEMORY_EXCEEDED_QUERY_MAX will be returned and the query will be aborted.

Similarly, if the total memory used for sorting exceeds the amount specified in by parameter max_sorting_memory_total in profile.be, the query, which is trying to allocate memory for sorting, will be aborted and error `QRY_SORT_MEMORY_EXCEEDED_TOTAL_MAX` will be returned.

If none of these parameters is specified, there will be no memory limit for sorting.

If only `max_sorting_memory_total` is specified, there will be no limit for individual query. But the total memory consumed for sorting cannot exceed the amount specified.

If only `max_sorting_memory_per_query` is specified, there will be no limit for total memory used for sorting. But individual query cannot use more than the amount specified for sorting.

If max_sorting_memory_total is smaller than `max_sorting_memory_per_query`, value of the max_sorting_memory_per_query will be ignored.

Locking

As the isolation level increases, objects and reads become more stable, but concurrency is reduced. For example, at isolation level 3, the read lock on the class object has the effect of placing a read lock on all objects of the class, which prevents all other users from updating, deleting, or creating an object of that class.

Also, for each isolation level, a stronger class or instance lock mode than the minimum lock mode shown above, will further reduce concurrency with other transactions.

For more information refer to “Locks and Transactions” on page 92, in "Chapter 5 - Locks and Optimistic Locking".

Indexes

Indexes allow query to filter objects so that only objects of interest within a specified range are fetched from disk. This can improve query performance in many situations. But it may have negative impact on update operations. If an index exists, the query will use the index when executing the query.

DIFFERENCES BETWEEN VQL 6.0 AND VQL 7.0

Virtual attributes not supported

The use of virtual attributes is not supported in VQL 7.0. The capabilities available with virtual attributes will be made available through another means in a later release of VQL 7.0.

Internationalization not currently supported

The VQL 7.0 language and API do not support internationalization. Later versions will enable features that will enable internationalization to be used in some form.

No explicit cursor support

There is no support for explicit cursors on the result of a query. The result handle can be used to iterate the result set. The fetch behavior may be tuned using the fetch size but this is more a hint than a means to control the behavior.

Behavior in case of incomplete path queries is different

While evaluating the predicate for an object if there is a null link, the predicate is evaluated to false. This happens while traversing the path expression.

CHAPTER 21

Embedding Versant Utilities in Applications

This Chapter gives detailed explanation on Embedding Versant Utilities in Applications.

The Chapter covers the following topics:

- Overview
- APIs for Direct Use
- Usage of o_nvlist
- Password Authentication for Utility APIs
- SS daemon enhancements

OVERVIEW

Most applications use Versant ODBMS as an embedded database.

The term "embedded" implies the database is not visible to the user of the application directly. Furthermore, it also implies the typical administration of the database can be done by the application transparently, without much involvement by DBAs.

For the traditional "Telco" application, Versant ODBMS has been embedded in switching and network management systems. In more recent applications, the trend is to use Versant ODBMS as a metadata or enterprise data cache for Java based e-biz applications. The latter set of applications also tends to use Java based application servers.

The Versant Object Database supports the embedding of utilities within user applications.

This has been achieved in the following ways:

- Support of utility functionality through API interface
- Remove interactive nature of utilities
- Returning of proper return codes to aid embedding
- Support of database authentication using “-password” option for embedded utility APIs
- Support for remote database cases

LIST OF APIS FOR DIRECT USE

The following APIs (in C, C++ and Java) are available for use directly through application programs:

Utility	C API	C++ API	Java API
addvol	o_addvol()	PDOM::addvol()	DBUtility.addVol
comparedb	o_comparedb()	PDOM::comparedb()	-
createdb	o_createdb()	PDOM::createdb()	DBUtility.createDB
			DBUtility.createAndInitializeDB
createreplica	o_createreplica()	PDOM::createreplica()	DBUtility.createReplica
			DBUtility.replicateToExistingDB
dbuser	o_dbuser()	PDOM::dbuser()	DBUtility.getDBUsers
ftstool	o_ftstool()	PDOM::ftstool()	DBUtility.ftstoolDisablePolling
			DBUtility.ftstoolEnablePolling
			DBUtility.ftstoolStopSync
makedb	o_makedb()	PDOM::makedb()	DBUtility.makeDB
			DBUtility.makePDB
			DBUtility.makeGDB
makeprofile	o_makeprofile()	PDOM::makeprofile()	-
readprofile	o_readprofile()	PDOM::readprofile()	BEProfile.getProfile

removedb	o_removedb()	PDOM::removedb()	DBUtility.removeDB
			DBUtil-ity.removeDBDirec-tory
			DBUtility.killAnd-RemoveDB
			DBUtility.killAnd-RemoveDBDirectory
removerep-lica	o_removereplica()	PDOM::removerep-lica()	DBUtility.remov-eReplica
			DBUtility.remov-eReplicaForce
setdbid	o_setdbid()	PDOM::setdbid()	DBUtility.setDBID
Sch2db	o_sch2db()	PDOM::sch2db()	-
startdb	o_startdb()	PDOM::startdb()	DBUtility.startDB
stopdb	o_stopdb()	PDOM::stopdb()	DBUtility.stopDB
			DBUtility.killDB
vcopydb	o_vcopydb()	PDOM::vcopydb()	DBUtility.copydb-ToExistingDB
			DBUtility.copydb
vmovedb	o_vmovedb()	PDOM::vmovedb()	-
writepro-file	o_writeprofile()	PDOM::writepro-file()	DBUtility.write-BEProfile

These behave with the same functionality of their utility counterparts.

Please refer to the C / C++ / Java Versant Manuals for a description of each of these APIs.

Passing of parameters to these functions is to be achieved through the use of the "`o_nvlist`" structure (class `NameValueList` in C++).

For more information on `o_nvlist`, refer to “Usage of o_nvlist” on page 568.

Functions to manipulate the variables of these types are listed in the table below.

C API	C++ API
o_addtonvlist()	NameValueList::add()
o_createnvlist()	NameValueList::create()
o_deletenvlist()	NameValueList::release()
o_firstnamefromnvlist()	NameValueList::get_first_name()
o_freevaluefromnvlist()	NameValueList::free()
o_getnameatcursorfromnvlist()	NameValueL- ist::get_current_name()
o_getnamecountfromnvlist()	NameValueList::count()
o_getvalueatcursorfromnvlist()	NameValueL- ist::get_current_value()
o_getvaluefromnvlist()	NameValueList::get_value()
o_nextnamefromnvlist()	NameValueList::get_next_name()
o_nvlistlength()	NameValueList::length()
o_removeatcursorfromnvlist()	NameValueList::remove_current()
o_removefromnvlist()	NameValueList::remove()
o_replacevalueatcursorinnvlist()	NameValueList::replace_current()
o_replacevaluebyindexinnvlist()	NameValueList::replace()
o_replacevaluebyvalueinnvlist()	NameValueList::replace()

USAGE OF O_NVLIST

```
typedef o_nvlisthdr* o_nvlist;
```

`o_nvlist` is a handle to a name value list. The structure `o_nvlisthdr` is kept opaque to the user. Only name and value pairs can be added to the list which is common when invoking command-line utilities.

Passing of parameters to various functions is achieved through the use of the "`o_nvlist`" structure (class `NameValueList` in C++).

Examples

The o_nvlist can be used in the following ways:

sch2db –D versantdb –y schema.sch

Equivalent of the utility `sch2db –D versantdb –y schema.sch' using C utility API is:

```
o_dbname dbName;
o_nvlist schdbList;
strcpy(dbName,"versantdb");
o_createnvlist(&schdbList);
o_addtonvlist(schdbList, "-y", NULL);
o_sch2db(dbName,"schema.sch",schdbList);
o_deletenvlist(schdbList);
```

Equivalent of the utility `sch2db –D versantdb –y schema.sch' using C++ utility API is:

```
NameValueList   schdbList;
o_dbname dbName;
strcpy(dbName,"versantdb");
schdbList.add("-y", NULL);
::dom->sch2db(dbName,"schema.sch",schdbList);
```

PASSWORD AUTHENTICATION FOR UTILITY APIS

For DBA utility APIs such as `o_makedb`, `o_createdb` etc, the user can specify DBA's password through "`-password`" option, if DBA is password protected.

Examples:

makedb - promptpasswd dbname

Equivalent of the utility `` `makedb –promptpasswd dbname` `` using C API is:

```
o_nvlist nvlist;
o_createnvlist(&nvlist);
o_addtonvlist(nvlist, “-password”, “mypassword”);
o_makedb(dbname, nvlist);
```

The following C utility APIs support database authentication using "-password" as above:

`o_addvol, o_comparedb, o_createdb, o_createreplica , o_dbuser, o_ftstool, o_makedb, o_makeprofile, o_removedb, o_removereplica, o_startdb, o_stopdb, o_vcopydb, o_vmovedb`

Equivalent of the utility `` `makedb –promptpasswd dbname` `` using C++ API is:

```
NameValueListnvlist;
nvlist.add("-password", "mypassword");
::dom = new PDOM();
::dom->makedb(dbname, nvlist);
```

The following C++ utility methods of class PDOM support database authentication using "-password" as above:

`createdb, addvol, comparedb, createreplica, dbinfo (PDOM::dbinfoapi), dbuser, ftstool, removedb, startdb, stopdb, vcopydb, vmovedb, removereplica and setdbid.`

NOTE:- An exception to the utility APIs is `o_setdbid`, for which the caller can specify DBA's name and password through `o_setuserlogin` API.

setdbid dbid dbname

Equivalent of the utility `setdbid dbid dbname` using a C API is:

Using `o_setdbid` with `o_setuserlogin`:

```
o_userInfo userinfo;
strcpy(userInfo.username, “username”);
strcpy(userInfo.password, “mypassword”);
o_setuserlogin(&userinfo);
o_setdbid(dbname, dbid);
```

Equivalent of the utility `setdbid dbid dbname` using a C++ API is:

```
dom = new PDOM();
o_userInfo userInfo;
strcpy((char *)userInfo.username, "username");
strcpy((char *)userInfo.password, "mypassword");
dom->setuserlogin(&userInfo);
dom->setdbid(dbname, dbid);
```

The utility APIs except `o_setdbid` will not use the information set by the `o_setuserlogin` API.

Regardless of whether the thread is in a session or not, the user must provide the utility API with the DBA password if DBA is password protected.

createdb

The next task would be to invoke "createdb" invoked from the command line as:

```
createdb versantdb
```

This could be invoked from a C program as:

```
o_createdb("versantdb", NULL);
```

Equivalent C++ program invocation is:

```
::dom->createdb("versantdb");
```

makedb

The utility "makedb" accepts an option "-owner" which also requires a value.

To invoke this utility from the command-line, the following command is to be specified:

```
makedb -owner vsntuser versantdb
```

To achieve the same through an application, the following piece of C code can be written:

```
o_nvlistmakedbList;
o_createnvlist(&makedbList);
o_addtonvlist(makedbList, "-owner", "vsntuser");
o_makedb("versantdb", makedbList);
o_deletenvlist(makedbList);
```

The same functionality in C++ would be as follows:

```
NameValueListmakedbList;
makedbList.add("-owner", "vsntuser");
::dom = new PDOM();
::dom->makedb("versantdb", makedbList);
```

o_writeprofile()

This API can be used to modify the contents of the back-end profile for the database. Normally, this is achieved by editing the back-end profile file "`profile.be`".

As an example, we want to add the following parameters to the back-end profile for the database "versantdb":

Parameter	Description
extent_size	8
heap_size	150
Logging	on
polling_optimize	off
commit_flush	on
Plogvol	4M physical.log 100
virtual_locale	"abc 123"
event_daemon	/opt/versant/nvlist/list "abc"

To achieve this through the application program, the following code snippet can be used:

```
o_nvlist   prflList;
o_createnvlist(&prflbList);
o_addtonvlist(prflList, "extent_size", "8");
o_addtonvlist(prflList, "heap_size", "150");
o_addtonvlist(prflList, "logging", "on");
o_addtonvlist(prflList, "polling_optimize", "off");
o_addtonvlist(prflList, "commit_flush", "on");
o_addtonvlist(prflList, "plogvol", "4M physical.log 100");
o_addtonvlist(prflList, "virtual_locale", "\"abc 123\"");
o_addtonvlist(prflList, "event_daemon", "/opt/versant/
                              nvlist/list \"abc\"");
o_writeprofile(O_BE_PROFILE, "versantdb", prflList);
```

vmovedb

```
vmovedb –threads 5 –C node employee person src_db dest_db
```

To implement the same through the C program:

```
o_nvlistvmoveList;
o_createnvlist(&vmoveList);
o_addtonvlist(vmoveList, "-threads", “5”);
o_addtonvlist(vmoveList, "-C", ‘node employee person’);
o_vmovedb(src_db, dest_db, &vmoveList);
o_deletenvlist(vmoveList);
```

The C++ program would have the following statements:

```
NameValueListvmoveList;
vmoveList.add("-threads", "5");
vmoveList.add("-C", ‘node employee person’);
::dom->vmovedb(src_db, dest_db, vmoveList);
```

sch2db

A schema file "`schema.sch`" is to be loaded into the database just created.

The command line utility is invoked as: `sch2db –D versantdb –y schema.sch`

To implement the same through the C program:

```
o_dbname dbName;
o_nvlist schdbList;
strcpy(dbName,"versantdb");
o_createnvlist(&schdbList);
o_addtonvlist(schdbList, "-y", NULL);
o_sch2db(dbName,"schema.sch",schdbList);
o_deletenvlist(schdbList);
```

The C++ program would have the following statements:

```
NameValueList   schdbList;
o_dbname dbName;
strcpy(dbName,"versantdb");
schdbList.add("-y", NULL);
::dom->sch2db(dbName,"schema.sch",schdbList);
```

SS DAEMON ENHANCEMENTS

The Versant SS daemon is the Versant system services daemon that is invoked by "`inetd`" on Unix machines. This daemon handles all initial requests from the client application.

The "`inetd`" is configured to invoke this daemon during Versant ODBMS installation. Therefore root privilege is required to install Versant ODBMS completely. This proves to be a hurdle to embed Versant in an application.

In the Versant Object Database 6.0 new functionality has been added to the SS daemon. The daemon can now be executed from the command line (and hence does not require root privileges).

CHAPTER 22

Programming Notes

This Chapter lists general programming and usage notes.

This Chapter describes the following:

- Versant Name Rules

Versant Name Rules

Name length

Maximum name lengths and default values imposed by Versant are:

Name	Maximum number of characters
Site	223, should be null terminated
User	31
Database	31, should be null terminated
.h specification file	per your operating system and compiler
.cxx implementation	per your operating system and compiler
Class	16,268, should be null terminated, also limited by compiler
Attribute	16,268, should be null terminated, also limited by compiler
Method	16,268, should be null terminated, also limited by compiler
Session	31, should be null terminated, default is user name
Transaction	31, default is user name

Name characters

Versant does not allow spaces in names, including login and database names.

Name of remote database

Specify a remote database with `dbname@site` syntax.

Name of a class

Class names should be unique to their database.

When objects are migrated, class names cannot conflict with existing class names in the target database.

Name of an attribute

Attribute names should be unique to their class and superclasses.

Specifying a name with a variable

C/Versant and C++/Versant — When using a variable to supply a name, you must first allocate memory for the variable with `malloc()` or the equivalent for your operating system. Remember to add one character for the null terminator.

For example:

```
newAttrName = malloc(sizeof("ANewNameOfAnyLength")+1);
o_err o_renameattr(...
```

Names returned to a vstr

C/Versant and C++/Versant — Several methods and functions, such as `o_classnameof()`, return names to a vstr. When you are finished using the names returned in the vstr, remember to deallocate the vstr memory using a routine such as `o_deletevstr()`.

Name for null database

C++/Versant: In every C++/Versant method except `beginsession()`, whenever you see a database name argument, you can specify `NULL` to use the default database. Before a session starts, there is no default database, so you should specify a database when you begin a session with `beginession()`.

When you begin a session with the `PDOM beginsession()` method, the database specified becomes the session workspace and the default database. You can later change the default database by using the `PDOM::set_default_db()` method.

Changing the default database does not change which database is being used as a session workspace. It does change the database in which objects are created, since creating a new, persistent object with `O_NEW_PERSISTENT()` creates that object in the current default database. Logical object identifiers for new objects are generated from the session database, even if it is not the default database.

Name of nested collection

C++/Versant: C++/Versant allows you to nest Collection classes to any depth. For example, you can create a dictionary whose values are a list:

```
VEIDictionary<o_u4b,VIList<myObject> >
```

Name of attribute in query and index methods

C++/Versant: Methods which query or index on an attribute need a precise database attribute name even when referring to a private attribute. This precise name qualifies the attribute name with the class name, as in `Employee::age`. The query and index methods that need a precise name are `select(), createindex()`, and `deleteindex()`.

Precise attribute names are needed by these methods, because a derived class could conceivably inherit attributes with the same name from numerous superclasses along one or more inheritance paths. Even if there is no initial ambiguity, classes may later be created which do cause ambiguity.

For more information, please refer to the C Reference Manual and C++ Reference Manual.

Index