Vulcan Overview

Document Version 1.06 2/3/2005 8:48 AM

Table of Contents

Overview

Vulcan is a major redesign of the Firebird database management system. It undoes more than a decade of increasingly conditional code, starts integrating SQL into the core engine, and unifies the database access architecture.

Project Goals

Vulcan had four primary goals.

- *Portability*: Vulcan was developed simultaneously on four platforms: 32 bit Windows using the Microsoft compiler from Visual Studio 7, 32 bit Linux and 64 bit Linux for AMD64/Opteron using various versions of gcc, and 64 bit Solaris using the Sun Forte C++ compiler. Ports exist to 64-bit MVS UNIX, Itanium, and AIX. See the Portability section.
- *Single processor performance*: tests on Win32 demonstrate some performance improvement over Firebird 1.5. See Performance section.
- *SMP performance*: on a four processor SMP box, Vulcan benchmarks up to about 3.5 times the speed of a single processor system on cache intensive benchmarks. See SMP section.
- *Upward compatibility*: programs written for Firebird work unchanged on Vulcan when relinked. See Compatibility Section.

In the process of achieving these goals, Vulcan achieved a great deal more. Although Vulcan retains much of the code from Firebird, it makes radical changes to the code organization and architecture.

This paper outlines some of the major differences between Vulcan and Firebird 1.5.

New Features of Vulcan

Process Architecture

Vulcan resurfaces the original InterBase architecture and updates it to reflect modern hardware and software technology. For more information see the Process Architecture section.

Configuration Files

Vulcan configuration files are upward compatible from the Firebird configuration file, but have significant new capabilities. For more information see the Configuration Files section.

Configurable Security Managers

Different organizations require different levels of user authorization. Vulcan provides a path to the resolution of this requirement and introduces several improvements in the management of authentication information. For more information see the Configurable Security Managers section.

User Authentication

Vulcan configuration files can specify a per-database source for authentication information. Vulcan also adds SQL statements to create, update, and drop users. For more information see the User Authentication section.

SQL Integration

Vulcan moves DSQL from outside the engine to inside, making it more efficient, and eliminating duplicate metadata representation . For more information see the SQL Integration section.

Lock Manager

Signal based lock notification has been deprecated in favor of thread-based notification and the two variants of the lock manager have been recombined. See the Lock management section.

Call tracing

The Vulcan dispatch module (Y-valve) will print a session log the contents of which are controlled with configuration file parameters. The options are tracing calls, tracing results, or tracing SQL statements.

Vulcan Internals

Most of the changes to Vulcan are internal. Their effect on performance and reliability can be measured by client applications, but they don't change the API or other user interfaces. See the Vulcan Internals section for more information.

Garbage Collect Thread

Firebird 1.5 uses a separate thread for garbage collection when compiled as SuperServer. Firebird 2.0 uses cooperative garbage collection or a garbage collect thread, depending on the circumstances. Vulcan uses only cooperative garbage collection.

SMP

Firebird

The Firebird SuperServer is multi-threaded, but only one thread runs at a time, which was an appropriate architecture when a typical server system had only one processor. Each request operates in a thread. A built-in thread scheduler allows a thread to run for a fixed period of time, then stops it and starts another. This type of threading prevents a single, complex request from blocking all others. Firebird controls threads through a single, system-wide mutex manipulated by the THREAD_ENTER and THREAD_EXIT macros. Threads check the state of the mutex at convenient times and yield control as requested. In Firebird, a thread can never be interrupted while updating a critical data structure.

Vulcan

In Vulcan, threads run in parallel and are scheduled by the operating system thread scheduler. This algorithm works well on symmetric multi-processors, but requires a different mechanism for protecting critical data structures.

Thread Synchronization

Vulcan threads run concurrently and are synchronized, where necessary, by controlling access to specific data structures and objects. Wherever possible, data structures and objects are changed from one consistent state to another by a single, non-interruptible instruction. When an object must be held in a consistent state for reading or writing, the accessing thread must acquire a lock.

Objects typically provide their own locks. Most objects use instances of a SyncObject class contained in their class definitions. SyncObject provides shared and exclusive locks with exclusive lock requests managed in a “fair” (i.e.) fifo ordering. Shared locks allow multiple threads to read the values of an object instance. Exclusive locks allow one thread to change values in an object instance.

“Fair” ordering means that requests for a lock on an object are serviced in order. If the object is locked in shared mode with no lock requests pending and another request is received for that lock in shared mode, the second lock is granted. As soon as a thread requests an exclusive lock on the object, additional requests for shared locks queue behind the request for the exclusive lock.

SyncObjects can be managed directly, but are more often managed by block local Sync objects that automatically release locks on exit or stack unwind.

Compatibility

Certain specific parts of Firebird 1.5 have not yet been completely implemented in Vulcan. They are: the WNET and XNET local server interfaces, Execute SQL Statement, and the Services API. In addition, some areas need additional work: Configuration File Structure needs some work to insure seamless upgrades. The Remote Architecture and the Use of XDR reflect pure early 1980’s technology and should be upgraded. Other areas offer opportunities that should be exploited to improve the product including Internal Engine SQL, and the Compiled Statement Cache. The Gateway Provider will provide a bridge from future InterBase versions to Firebird. Finally, the Event Manager appears to have suffered in the translation to Vulcan.

Process Architecture

Firebird

Firebird is distributed in three forms: SuperServer, Classic, and Embedded. Each represents conditional code and a conditional build.

SuperServer is a single process that accepts network connections and manages all database access. Coordination between requests is handled within the SuperServer. The server opens the database in exclusive mode and no other type of access is permitted to that database. SuperServer is multithreaded to avoid having a single request block others but does not support parallel thread execution.

Classic is a shared library containing database access, update, and lock management code. Each process reads and writes to the database independently. Access is coordinated through a shared memory lock table. A single process can make multiple connections to a database or connections to multiple databases. Requests to different connections can be interspersed, but are not multi-threaded. Classic provides parallel execution from different processes on SMP.

Embedded requires that all access to a database be through a single process that locks the database. Embedded provides the same level of multi-threading as SuperServer.

Vulcan

Vulcan provides fine granularity multi-threading with parallel thread execution in all configurations. It differs from Firebird in several ways, the most obvious being that Vulcan separates the functions of server and database access and controls database sharing through configuration file settings.

The Vulcan server is a process that accepts network connections and converts the requests conveyed in the network protocol to Firebird API calls. All database access goes through a thin primary shared library known as the Y-Valve to an open-ended set of data providers.

If the configuration file specifies that the database is not shared, network access to that database is equivalent to SuperServer. All connections share a single buffer pool. If the configuration file specifies database sharing, local access is similar to Classic, with any number of processes, each with a private buffer pool, synchronized by an external lock manager. If remote access is enabled on a database that allows sharing, Vulcan runs in hybrid mode. Remote clients access the database through a multi-client, multi-threaded server with a shared buffer pool that share the file with other local processes that maintain private buffer pools, all synchronized by the lock manager. With no server and database sharing disallowed, Vulcan runs in Embedded mode.

Y-valve

The primary library, firebird.dll or libfirebird.so, is called the “Y-valve” or “dispatch module”. It implements all user visible API entrypoints but does not perform any data management services itself. It loads and invokes other libraries called “providers” to do the actual work. All programs that access a Vulcan database do so through the Y-valve. There is no separate client library, classic library, or embedded library.

When the client asks for a database, the dispatch module finds engine, interface, gateway, or server that supports the requested database, whether local, remote, or even a different architecture.

In the configuration system, the Y-valve is a provider called `dispatch`.

Providers

A provider is a C++ object that implements the canonical provider interface class. The base interface class is extensible. A provider that does not support a particular function returns a proper OSRI error sequence.

Providers include actual database engines, remote interfaces to communicate with servers, gateways to another version of Firebird or InterBase, and any other data manipulation code which supports OSRI semantics.

Currently Firebird support ODS formats 9, 10, and 11 in a single code base. That support significantly complicates the code because it requires different execution paths depending on the result of tests for ODS version. Maintaining several parallel and intersecting execution paths is an engineering challenge at best. In the original InterBase, on-disk structure (ODS) changes were handled by including a provider that handled the old ODS and a provider that handled the new ODS.

The Vulcan architecture is similar, though the mechanism for choosing a provider is different. As the database engine evolves, we can expect to have a set of database providers, each handling a different generation of database files. A single application program can run against everything from an InterBase version 4 database to the most recent experimental version of Firebird.

Servers

A Vulcan database server is a program layered on the published database API. The current server handles TCP-IP connections following the Firebird 1.5 remote protocol. It connects to the Y-valve and then to providers for all data manipulation.

Configuration Files

The Vulcan configuration file system is a superset of the Firebird 1.5 configuration file. The full syntax and semantics are documented in section on Configuration Files.

Firebird

Firebird has a single configuration file for the entire installation with no provision for separate control, either by user groups or by database. Originally, the configuration file controlled the size and organization of the lock table, a resource that is shared by all databases served by an installed version of Firebird. Versions 1.5 and 2.0 greatly expanded the number of parameters managed by the configuration file.

Vulcan

Vulcan balances need for database and application specific configuration values and the need for system wide management of configuration with cascading configuration files. The Vulcan configuration files control the mapping of databases to providers and security managers by defining those objects and their relationships.

Configuration file definitions

Vulcan configuration files define parameters and objects. Configuration parameters are fixed attributes like the lock table hash width, sort memory size, etc. Configuration objects include databases, providers, servers, security managers, and other as yet unknown object types. Configuration parameters that are declared globally within a configuration file become defaults. Configuration parameters that are declared within the definition of an object are local to the object.

A global default parameter overrides an intrinsic, built-in default parameter of the same name. A parameter defined within a provider object overrides the global default value for that parameter. A parameter defined within a database object overrides the same parameter defined in the provider. Parameters are scoped to support a balance between general policy and special case exceptions.

Database objects

Database objects are described by a file name that may include wild cards. The database object description has one of two formats. The normal format includes a provider.

1. a general database object that includes a TCP/IP address. This object describes all databases which have a name that contains a ":" character. The filename is left unchanged and the designated provider is remote. Thus a request to open a database with a file name of "caine:/usr/harrison/databases/foo.fdb" would be passed to the TCP/IP remote access provider.

```
<database *:*>
    filename    $0
    provider    remote
</database>
```

2. a specific database object. This object describes all databases with the name "msg.fdb" and redefines the name to point to the message database in the Vulcan install directory.

```
<database msg.fdb>
filename    $(root)/databases/msg.fdb
provider    engine8
</database>
```

3. a generic database object. This object describes all other databases and assigns them to the local provider.

```
<database *>
    filename    $0
    provider    engine8
</database>
```

A second style of database object simply changes the name of the database. This is a useful feature, but its behavior is unlike anything else in the configuration files.

4) A database object definition can replace the name of the database without giving it any other characteristics. For example, some of the Vulcan and Firebird sources are modules that must be preprocessed by gpre. Those files contain the name of a database to use in the preprocessing. Many of them reference yachts.lnk. The build configuration file replaces the name yachts.lnk with a full path to a database called metadata.fdb.

   When a database object definition replaces the name given on input, the configuration file manager reapplies all configuration files in order, using the new name rather than the name originally supplied.

```
 <database yachts.lnk>
     filename    caine:c:\harrison\vulcan\databases\metadata.fdb
 </database>
```

Provider objects

1. a provider object. This object is the local engine. Its attributes are the name and location of the lock file and the name and location of the shared libraries that can serve as that provider. In this case, the library can either be a 32-bit binary or a 64-bit binary. The Y-valve will attempt to open a database with the first library lists, then the second, and so on.

```
<provider engine8>
      LockFileName      $(root)/vulcan.lck
      library           $(root)/bin/engine8 $(root)/bin64/engine8
</provider>
```

   The Y-valve is a degenerate provider, which accepts one parameter, TraceFlags, indicating that certain operations should be written to its output stream.

```
<provider dispatch>
      TraceFlags  6
</provider>
```

Security Objects

2. a security manager. Security management is discussed in more detail <here>.

```
<SecurityManager SecurityDb>
      SecurityDatabase $(root)/security.fdb
      AuthAccount        user
      SecurityPlugin     SecurityDb
</SecurityManager>
```

3. a security plug in. Security management is discussed in more detail <here>.

```
<SecurityPlugin SecurityDb>
    library       $(root)/bin/securitydb $(root)/bin64/securitydb
</SecurityPlugin>
```

Cascading Configuration Files

The second major difference between Vulcan and Firebird configuration files is that Vulcan configuration files cascade. The Y-valve opens an initial configuration file that can provide all parameter and object definitions, like firebird.conf, or it can provide some definitions and include one or more additional configuration files. Those files can include other configuration files. Typically, the initial configuration file is the most specific and the included files are increasingly generic, ending with firebird.conf.

Objects can be defined at multiple levels. An earlier object definition overrides a later object definition. A system manager can choose to delegate control over various group databases to the groups themselves while retaining control over server configurations. An individual (or application program) can specify a personal or application configuration file that cascades to the group configuration file that cascades to the system wide client configuration files.

Configurable Security Managers

Firebird

Firebird supports a single hard-coded security model – a single, system wide, security database. Any user with access to one database has access to all databases on a system.

Vulcan

Vulcan introduces the architecture for loadable security managers. For the initial release, there is only one security manager, which emulates the Firebird security semantics. The architectural support for loadable security manager will permit installations to specify and enforce other security.

Security manager classes

The security manager is implemented with a C++ base class, like the provider interface. All security managers must support the interface, providing a framework for complex interactions between database engine and security manager while allowing future extensibility.

Like other configuration objects, security managers can be daisy chained. A single installation can support a mixture of security managers.

User Authentication

Firebird

The Firebird security management system depends on a database called security.fdb in the installation directory. All user authentication data for the installation is stored in that database.

Vulcan

The configuration files specify where Vulcan should look for user authentication data. That data can be in the installation-wide security.fdb, in another database, or in the target database itself. The configuration file can also specify that a database uses no authentication at all.

Vulcan also implements a user authentication extension to the published API designed for open-ended interaction with various loadable security managers. The user authentication facility also eliminates the public accessibility of account names and password hashes, significantly increasing database security.

Vulcan supports these SQL statements:

```
create user <username> password '<password>'
alter user <username> password '<password>'
drop user <username>
upgrade user <username> password '<password>'
```

SQL Integration

Firebird

Dynamic SQL (DSQL) in Firebird is implemented in the Y-valve rather than the engine. Each database attachment fetches and manages a private view of database metadata, incurring substantial overhead and delaying time to first record.

Vulcan

Vulcan moves SQL processing from the Y-valve into the database engine. This has several advantages.

Shared metadata cache

The SuperServer version of Firebird maintains a metadata cache within the engine and a metadata cache for each connection. Building a new metadata cache is a significant part of the cost of establishing a new connection. In a Vulcan shared server, all connections share a single copy of metadata.

Internal SQL execution

The Firebird engine does not use SQL directly. Internal database access is through GDML preprocessed by gpre using a special switch to generate internal rather than external calls. In Vulcan internal metadata management uses internal statements like this:

```
PStatement statement = connection->prepareStatement (
      "SELECT"
      "           vrel.RDB$CONTEXT_NAME,"
      "           vrel.RDB$RELATION_NAME"
      "     FROM RDB$VIEW_RELATIONS vrel"
      "     WHERE vrel.RDB$VIEW_NAME = ?");

statement->setString(1, viewName);
RSet resultSet = statement->executeQuery();

while (resultSet->next())
      {
      const char *contextName = resultSet->getString(1);
      const char *relationName = resultSet->getString(2);
            …

      }
```

Moving away from GDML and gpre is hard, but necessary.

Performance

To maintain performance while protecting shared resources, Vulcan reduces critical code paths. In particular, in Firebird 1.5 more than 500 routines, many on critical paths, start with a call to get_thread_data. Where thread data is required, Vulcan uses the normal parameter passing mechanism to provide it. This clean-up compensates for the interlocked instructions used to protect shared data structures.

Lock management

Firebird

Firebird has two lock managers. The SuperServer lock manager is part of the server, but still uses a defined, fixed size memory area to manage concurrent access to page buffers and other resources. The Classic lock manager is a shared memory area that all processes use to record their requests for access to shared database objects. Although the lock manager is discussed as if it were a separate entity, it is actually part of the SuperServer image and the Classic library. However, the Firebird lock manager carries with it years of history. Those years show themselves in the wide variety of ways that the lock manager attempts to establish communication between processes in Classic mode.

In particular, some systems require semaphores which may require a rebuild of the kernel. Others use signals, which may require a separate privileged process to transmit between clients from different process groups.

Vulcan

Vulcan has only one lock manager. The server can share a database with local clients. The lock manager uses a single mechanism for inter-process communication: threads. The need for a privileged resignalling process is gone.

Vulcan Internals

Object Lifetime Management

In Firebird all memory is allocated from memory pools and persists until the memory pool is deleted. Vulcan supports memory pools, but release-by-pool is deprecated. All new and revised code explicitly releases objects allocated. All objects track resources they allocate and release any remaining allocated resources in the class destructor.

Thread Data

The use of thread specific data other than thread management itself is deprecated. The "thread database" object has been retained, but is passed as a formal parameter to all methods and functions where it is required.

System Tables

In Firebird, compiled database access statements reference tables and fields by their identifiers (relation_id, field_id) rather than names. At one point the memory and processing necessary to maintain names was significant. Processor speed and memory cost have eliminated the benefit of that optimization, and Vulcan now uses names rather than identifiers. This change will allow "soft" system tables – system tables that accept and preserve user extensions.

Exception Handling

In most cases, Vulcan exceptions are thrown and caught as instances of the OSRIException class. The OSRIException carries the full context (information load) of the exception, eliminating the need to find a target status vector in order to throw an exception.

Class Encapsulation

Numerous internal mechanisms have been encapsulated as C++ classes including BLR generation, message formatting, message lookup, the mover, status vector, SQL parser, a string class, handle management, etc.

Namespace Management

Vulcan contains a great amount of code shared between the engine, the Y-valve, other providers, and external components. To promote code sharing while preserving architectural borders, shared Vulcan classes are qualified by soft, component-specific namespaces declared in the build configuration.

Coding Conventions

Vulcan code follows the class, member, and file name conventions laid out in the paper "Vulcan Rules" distributed in the Vulcan source tree.

Unfinished Work and Loose Ends

Remote Architecture

Parts of the interface for the various remote network modules have been encapsulated as a Port structure. The encapsulation, however, was driven more by pragmatics than coherent design. The encapsulation itself as well as the division of work between the base classes and individual networks modules could use a careful redesign. The interface is non-architectural, and can be changed at any point.

Use of XDR

The remote line protocol is defined in terms of XDR, the lower layer of Sun's remote procedure protocol. In practice, however, XDR adds nothing but cumbersome overhead. XDR should be replaced by simpler in-line code without impacting the line protocol or forward and backward compatibility.

WNET and XNET

The WNET and local transport mechanism XNET have not yet been converted to Vulcan.

Event Manager

The event manager has not been tested in Vulcan and can be assumed not to work.

Compiled Statement Cache

The Firebird 2.0 dynamic SQL handling has been implemented in the Vulcan engine as two classes, CStatement, a compiled statement, and DStatement, an instance of a compiled statement. This architecture can support a cache of compiled statements. To finish the implementation of a statement cache requires creating a statement cache manager to manage compiled statements and to validate security and access control to prospective compiled statements.

Internal Engine SQL

Vulcan supports internal engine SQL. Following implementation of the compiled statement cache, GDML should generally be replaced by SQL for metadata handling within the engine.

Execute SQL Statement

The Firebird 2.0 implementation of Execute Statement loops back through the API to provide call-level dynamic SQL. Vulcan's internal engine SQL provides a cleaner mechanism.

Configuration File Structure

Vulcan configuration files are designed to cascade, giving great flexibility, but have evolved in a rather ad hoc manner. Some careful thought should be given to the specific layout of configuration files to preserve site configuration information over Firebird upgrades.

The Gateway Provider

The Gateway Provider is demonstrable but incomplete. The Gateway is also compiled to support a single remote interface. It should probably be reworked to pick up information about the foreign system from the configuration file.

Services API

The services API is designed to be supported by a Services Provider. The Services Provider, however, does not yet exist.

The Commit Manager

Vulcan includes the skeleton of code to group commits. When a transaction commits, all pages it changed must be written to disk. Some of those pages are common to many transactions: the header page, the tip, generator pages, etc. Considerable saving can be achieved by grouping commits to those pages are written once for the whole group. The ability to group commits was not required to achieve the performance goals, but the structure to do so is in the code.

Portability

Firebird

Firebird V1.5 and V2.0 are sensitive to minor variations in C++ compilers, which is a serious liability given the somewhat cavalier attitude of compiler developers toward version-to-version compatibility.

Vulcan

Vulcan was developed on four compilers simultaneously: gcc 2.96, gcc 3.3.4, Solaris/Forte 5.5, and Microsoft VC7. To make that work, Vulcan eliminated the dependency on std.lib, minimized all other dependencies outside the core clib (c library), and reduced the complexity of template usage.

Temporary Installation Procedure

To be provided

The config utility

Vulcan includes a command line utility called config which reads the configuration files and reports on their contents.

Syntax and switches

Config accepts the following switches:

```
    -t            Trace configuration file opens
    -l            Print the contents of configuration files
    -i            Show installation directory
    -f filename   Config file name
    -h            Print this text
```

Config output

The most common use of config is to determine how a database will be accessed, using the –t option. Config first reports on the configuration files it opens in order, then reports any translation of the file name, then lists the provider and the shared library that implements the provider.

In this example, config reports that the installation configuration file for clients is the first one located, and that the file rogers.fdb be opened locally.

```
C:\harrison>config -t rogers.gdb
Opening \harrison\vulcan\install/client.conf
Opening \harrison\vulcan\install/databases.conf
Opening \harrison\vulcan\install/vulcan.conf
Looking up database name string "rogers.fdb"
  Matches "*", translates to "rogers.fdb"
    Provider engine8
      Library "C:\harrison\vulcan\install\bin\engine8": succeeded
```

This example comes from the vulcan build directory. There is a vulcan.conf file in the source directory that invokes a shared configuration file called build.conf. Build.conf renames yachts.lnk to be help.fdb – solving one of the long standing build problems.

```
C:\harrison\vulcan\src\qli>config -t yachts.lnk
Opening vulcan.conf
Opening ../build.conf
Opening \harrison\vulcan\install/client.conf
Opening \harrison\vulcan\install/databases.conf
Opening \harrison\vulcan\install/vulcan.conf
Looking up database name string "yachts.lnk"
  Matches "yachts.lnk", translates to
"../../install/help/help.fdb"
    Provider engine8
      Library "C:\harrison\vulcan\install\bin\engine8": succeeded
```

Configuration Files

Locating the initial configuration file

Vulcan locates a configuration file by trying the following steps in this order:

1. Translate the environmental variable VULCAN_CONF

2. Open `vulcan.conf` in the program's default directory
3. Open `~/.vulcan.conf –` vulcan.conf in the user's home directory on Unix and Linux systems.
4. Translate the environmental variable VULCAN and open client.conf there
5. Translate the environmental variable FIREBIRD and open client.conf there
6. Windows registry for Windows systems only.
    a. `SOFTWARE\\Firebird Project\\Firebird Server\\Instances`
    b. `SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App Paths\\Firebird`

Include statements in the initial configuration file lead to other configuration files.

Configuration file syntax – Parameters

The syntax for defining a parameter is the same as in Firebird

```
<parameter name>[=] <value>[ <value>…]
```

A newline terminates the parameter definition. Boolean parameters accept the values `yes`, `no`, `true`, and `false`.

For example

```
RootDirectory     /opt/vulcan
```

Or

```
RootDirectory = /opt/vulcan
```

Or

```
provider remote engine8 services
```

Or

```
DatabaseFileShared true
```

Vulcan accepts all the parameters supported by Firebird 1.5 and a number of new parameters. The file firebird.conf documents the firebird specific parameters. This section describes the Vulcan specific parameters.

`DatabaseFileShared – defaults to false.  This parameter is normally used within the definition of a database object.  If it is set to false, only one process can attach to the database at any one time. This is equivalent to embedded access or SuperServer access if the one process is the server.`

`LockFileName – defaults to null (0x0).  The parameter is provided so two or more copies of Vulcan can exist on a computer and ignore each other.  That configuration, while important and powerful, is also dangerous unless carefully designed.`

`SecurityDatabase – defaults to "none". The permissible values are "none", "self", and the file name for a security database. Typically an installation's vulcan.conf file will name a security database and the definitions of specific databases will override that definition with either "none" for databases that have no authentication, "self"`

```
for databases that contain their own authentication information, or an
application / installation specific shared authentication database.

Library - the name of a shared library that implements a particular
service.

TraceFlags - defaults to zero.  The flags are traceCalls 1,
traceResults 2, and traceSQL 3.  They can be combined.

SecurityManager - defaults to "SecurityDb". No other manager is
currently available.  This parameter is part of the security management
architecture to support security plugins.

CommitInterval - defaults to 200.  This parameter controls the
frequency of group commits under the (unfinished) commit manager.
```

Using firebird parameters

To emulate customized Firebird 1.5 behavior, you can add the line

```
      include $(root)firebird.conf
```

to the vulcan.conf file in the in installation directory. However, many parameters are more useful when applied to a specific database, and those parameters can be set in the object definitions in Vulcan configuration files. Others, like SortMemSize may be set differently for different processes and should be included in the vulcan.conf files for those processes.

Specific installations

If you are setting up multiple independent copies of Firebird and Vulcan, you should create a configuration file in the root directory of each installation that includes these parameters:

```
RootDirectory, LockFileName SecurityDatabase
```

Resource use

Some parameters should be set differently for different clients to distribute resources according to need.

```
SortMemBlockSize, SortMemUpperLimit, DefaultDbCachePages,
MaxUnflushedWrites, MaxUnflushedWriteTime,
```

Deprecated parameters

Due to changes in architecture, a number of Firebird parameters have no meaning in Vulcan.

```
RemoteFileOpenAbility - Vulcan does not support the ability to open
databases through NFS mounts.

CpuAffinityMask, TraceMemoryPools, LockSemCount, LockSignal,
LockAcquireSpins, SolarisStallValue, PrioritySwitchDelay,
DeadThreadsCollection, PriorityBoost,
```

TCP Server options

These parameters should be set in the server configuration file.
`GuardianOption, TcpRemoteBufferSize, TcpNoNagle, ConnectionTimeout, DummyPacketInterval, RemoteServiceName, RemoteServicePort, RemoteAuxPort, RemoteBindAddress,`

Non-tcp Server options

These parameters should be set in a configuration file specific to the affected server.

`IpcMapSize, RemotePipeName, IpcName, ProcessPriorityLevel, CreateInternalWindow,`

Compatibility with older versions

These parameters should be set at the process or user level
`OldParameterOrdering`

Lock and Event manager parameters

These parameters should be set at the installation level:

`LockMemSize,  LockGrantOrder, LockHashSlots, DeadlockTimeout,`
`EventMemSize,`

Access control parameters

These parameters should be set at the process or sever level:
`ExternalFileAccess, DatabaseAccess, UdfAccess,  TempDirectories`

Object definition syntax

Special tokens

$(install) – the actual installation directory.
$(root) – the translation of the parameter `RootDirectory` or the installation directory if that parameters is underfined.
$(this) – the current directory.

$0	- the current name
$1	- the translated value of the first wildcarded portion of the current name
$n	- the translated value of the nth wildcarded portion of the current name

Object definitions

The basic format of an object definition is:

```
<object     name>
parameter   value[ value…]
</object>
```

The known objects are server, provider, SecurityManager, SecurityPlugin, gateway, and database.

```
<provider engine8>
    LockFileName    $(root)/vulcan.lck
    library         $(root)/bin/engine8 $(root)/bin64/engine8
</provider>

 <SecurityManager SecurityDb>
    SecurityDatabase $(root)/security.fdb
    AuthAccount              user
    AuthPassword     lookup
    SecurityPlugin   SecurityDb
</SecurityManager>
```

Only one security plugin is supported currently.

```
<SecurityPlugin SecurityDb>
    library   $(root)/bin/securitydb $(root)/bin64/securitydb
</SecurityPlugin>
```

A database object definition can have one of two forms. It can map a database name, possibly wild-carded, to one or more providers and apply other parameters to the database like this:

```
<database *>
    filename    $0
    provider remote engine8 services
</database>
```

The other form of the database object definition changes the name given for the database without giving it any other characteristics. When the configuration file manager finds a database object which simply renames the database, it reapplies all configuration files in order, using the new name rather than the name originally supplied.

```
<database yachts.lnk>
    filename    caine:c:\harrison\metadata.fdb
</database>
```

Unless you include at least one provider in a database object, the only parameter you can set is the database name.

Using internal SQL

Syntax

Internal SQL is very much like JDBC. One starts by constructing a PStatement object by invoking the prepareStatement method of the Connection object on a SQL string that can include wild cards.

```
PStatement statement = connection->prepareStatement (
            "select"
            "   rfr.rdb$field_name,"
            "   rfr.rdb$field_position "
            "from "
            "   rdb$relation_fields rfr join"
            "   rdb$fields fld on"
            "      rfr.rdb$field_source = fld.rdb$field_name "
            "where"
            "    rdb$relation_name = ?");
```

The PStatement has a method for establishing values for the wildcards.

```
      statement->setString(1, rel_name);
```

Executing a query statement produces an RSet object.

```
      RSet resultSet = statement->executeQuery();
```

The RSet object iterates through the query results.

```
      while (resultSet->next())
            {
            int seq = resultSet->getInt(2);
            const char *fieldName = resultSet->getString(1);
            …
            }
```

Building Vulcan on Windows

Anyone who would like to build Vulcan at home and has access to Visual Studio 7 can do so, reasonably easily, following this procedure.

Check out the sources from sourceforge using the module name vulcan and any CVS client. The account is anonymous with no password. The CVSROOT is `/cvsroot/firebird.` If you are using a version of CVS that does not check out empty directories, you'll need to create a Databases directory under src.

Read the file README.VulcanWin32.txt in the top-level directory. It instructs you to

1) define VULCAN as <vulcan root>\install

2) put %VULCAN%\bin on your path before the paths to Firebird or InterBase

3) run the command file boot_copy.bat

4) copy or rename the file vulcan\builds\VisualStudio7\Vulcan\Vulcan.snl.template to vulcan\builds\VisualStudio7\Vulcan\Vulcan.sln

Then start Visual studio pointing it at the solution in vulcan\builds\VisualStudio7\Vulcan.

5) in the Visual Studio/Tools menu, pick Options/Projects, then Visual C++ files. If the list does not include <source root>/install/bin and <bison install directory>/bin, add them.

6) click on build solution and it should churn happily for a while and eventually report that it built 22 projects with no failures.

The build automatically creates all the necessary databases, message files, dynamic header files, etc.

Local connections will work immediately. To allow remote and loopback connections, start the server %VULCAN%\bin\inetserver.exe with the switches -a -d -t. The configuration files are deeply involved with the connection between a utility and a database. The best way to guess what the config files are doing to you is to run the config utility with the -t option.

The config utility traces back through the sequence of configuration files, then reports the translation of the database name, the name of the provider, and whether the provider was found.

```
C:\Harrison>config -t help.fdb
Opening c:\cygwin\home\vulcan\install/client.conf
Opening c:\cygwin\home\vulcan\install/databases.conf
Opening c:\cygwin\home\vulcan\install/vulcan.conf
Looking up database name string "help.fdb"
 Matches "help.fdb", translates to
"c:\cygwin\home\vulcan\install\help\help.fdb"
   Provider engine8
     Library "c:\cygwin\home\vulcan\install\bin\engine8": succeeded


C:\Harrison>isql help.fdb
Database:  help.fdb
SQL> show version;
ISQL Version: WI-V2.0.0.4027 Vulcan 1.0 Development
Firebird/x86/Windows NT (access method), version "WI-V2.0.0.4027 Vulcan
1.0 Development"
on disk structure version 11.0
SQL>
```

If you have problems building vulcan, please contact me, aharrison@ibphoenix.com

Thanks,

Ann

Building Vulcan on Posix system

1) Check out the sources from sourceforge using your favorite CVS into a directory somewhere. For this example, call the top of the directory tree `/home/vulcan`
2) Set and export (depending on the style of your shell) the environmental variable `VULCAN` to be the directory called `install` under the vulcan directory. In this example it would be `/home/vulcan/install`.
3) Set and export the environmental variables `ISC_USER` and `ISC_PASSWORD`. SYSDBA and masterke are good – change the password later.
4) If you are building a 32-bit version of Vulcan, add `$VULCAN/bin` to your `PATH` and `LD_LIBRARY_PATH` variables.
5) If you are build a 64 bit version of Vulcan, add `$VULCAN/bin64` to your `PATH` and `LD_LIBRARY_PATH` variables.
6) `cd` to the `src` directory under the build tree. In this example `cd /home/vulcan/src`
7) Add execute permission to the files `set_platform`, `autogen.sh`, `build` and `boot_build`.
8) Invoke the autogen script `./autogen.sh`
9) Run the `set_platform` script with one of these arguments: `{ linux32 | linux64 | solaris64 }`. For example `./set_platform linux32`
10) Run the build script file with no arguments. `./boot_build`

If you have problems building vulcan, please contact me, aharrison@ibphoenix.com

Thanks,

Ann

Building Vulcan on 64-bit Linux

Here is the full command sequence from a SuSE 9.2 AMD64 build, which already had the environmental variables defined:

```
cvs -z3
-d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/firebird
   co vulcan
cd vulcan
./autogen.sh
cd src
./set_platform linux64
boot_build
rehash
```

If you have problems building vulcan, please contact me, aharrison@ibphoenix.com

Thanks,

Ann

Document history

V1.0 first draft
V1.01 removed installation instructions
V1.02 corrected errors including classic on windows
V1.03 added Linux instructions and improved configuration section
V1.04 corrected and restated various problems, corrected linux instructions added 64 bit build example
V1.05 incorporated Helen's comments
V1.06 removed details of Firebird use of the security database that changed between 1.5 and version 2.0 and added the need for user and password environmental variables in the Posix builds.