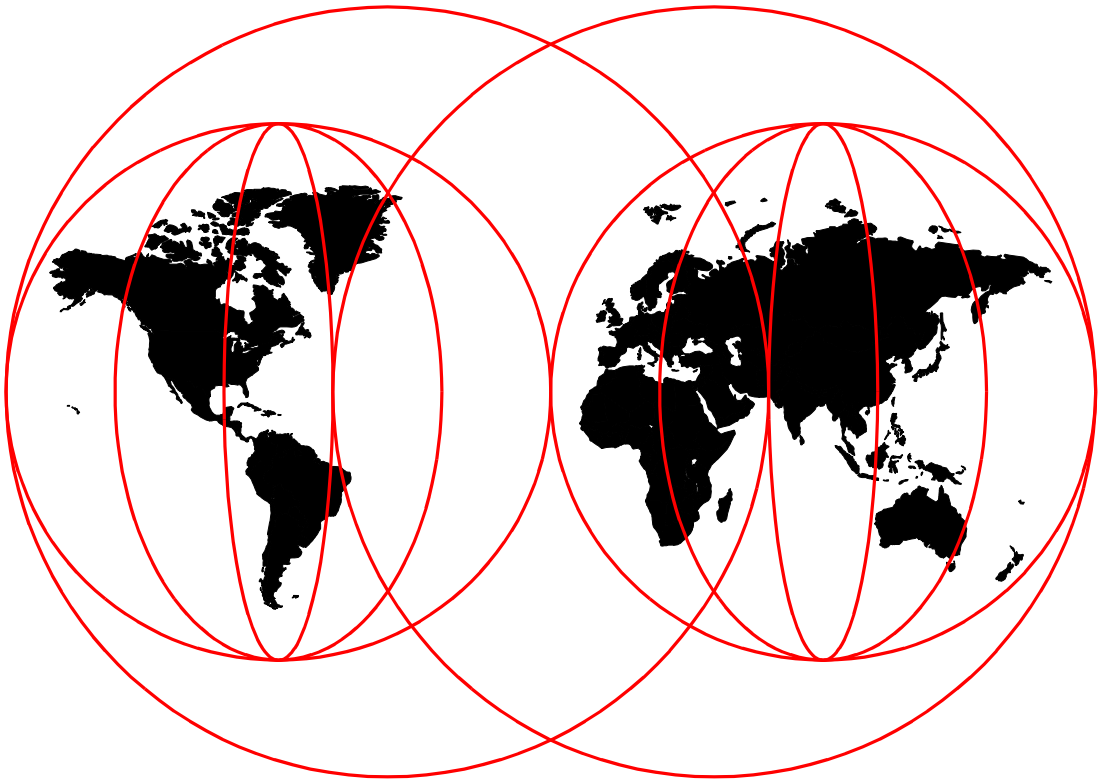
IBM SanFrancisco Performance Tips and Techniques

Gottfried Schimunek, Thomas Fanto, Maria Cristina Filorizzo, Armin Rauch, Roger Rolandsson, Mohit Sant, Jan Van der Sypt



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5368-00

International Technical Support Organization

SG24-5368-00

IBM SanFrancisco Performance Tips and Techniques

February 1999

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, “Special Notices” on page 203.

First Edition (February 1999)

This edition applies to IBM SanFrancisco, Version 1 Release 3 Modification 0 (V1M3M0).

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. JLU Building 107-2
3605 Highway 52N
Rochester, Minnesota 55901-7829

Contents

Figures

Tables

Preface

A SanFrancisco-based application can obtain good performance, but there are, and always will be, certain limits to this performance. To make a wise decision, it is important to have several pieces of information. First, what is the requested appropriate performance? Second, what is the performance that SanFrancisco can live up to for the type of application? If there is a match between these, take into consideration that the framework constantly improves its performance and a better performing version may be available by the time of deployment.

If the appropriate performance is reachable, and SanFrancisco is chosen for development of the application, the real performance work begins. You can achieve performance expectations by avoiding bad practices. A series of reasons can lay at the basis of a poor performing application. This redbook helps you to understand these reasons and offers solutions or alternatives when possible.

It is obvious that other characteristics and features of the framework are far more determining factors in choosing the IBM SanFrancisco framework as a base to build applications than its performance. Although performance should not be a major reason to vote against it, this redbook only considers performance issues.

The redbook is primarily intended for IBM SanFrancisco application developers, application architects, and performance consultants. It guides you through the different areas of performance and helps you to analyze a performance problem with the use of various tools, shows how application design patterns can influence performance, and describes the IBM SanFrancisco environment. It also provides suggestions for optimal performance, leads you through the technique of mapping the relational database to objects, and provides a comprehensive list of coding techniques for better performance of SanFrancisco applications.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Rochester Center.

Gottfried Schimunek is a certified I/T Architect at the IBM International Technical Support Organization, Rochester Center. He writes extensively and teaches IBM classes worldwide on all areas of application development and performance. Before joining the ITSO in 1997, Gottfried worked in the Technical Support Center in Germany as a Consultant for IBM Business Partners and Customers as well as for the IBM Sales Force.

Thomas Fanto is a Systems Engineer at Intentia Research & Development in Sweden. He has eight years of experience in the OOA, OOD, and OOP field. His areas of expertise include 14 years of ERP systems development, middleware architecture, and database systems. He has written extensively on IBM AS/400 Systems, Microsoft Windows NT, and Ericsson 2500.

Maria Cristina Filorizzo is an I/T Specialist at the Java Technology Center, IBM Semea Sud, in Italy. She has three years of experience in the OOA, OOD, and OOP field. She holds a degree in Computer Science from University of Bari, Italy. Her areas of expertise include intelligent software agents, application development based on San Francisco, and relational and OO database technology. She has written extensively on Java, publishing a series of articles for an Italian magazine. In 1996, she received the Outstanding Technical Achievement Award.

Armin Rauch is a Systems Engineer and architect at IDG Informationsverarbeitung und Dienstleistungen GmbH, Germany. He has four years of experience in the OOA, OOD, and OOA field. His areas of expertise include connection between OO and legacy systems and 15 years of application development.

Roger Rolandsson is a Systems Engineer at Intentia Research & Development in Sweden. He has two years of experience in the OOA, OOP, and OOD field. He holds a Bachelor of Science in Computer Engineering from the University of Linkoping, Sweden. His areas of expertise include application development, database, and object-oriented programming.

Mohit Sant is a Senior Software Engineer at IBM Global Services, India. He has two years of experience in the OOA, OOD, and OOP field. He holds a Bachelor of Engineering degree in Computer Science and Engineering from Sri Jayachamrajendra College of Engineering, Mysore. His areas of expertise include Java applications and JavaBeans development and networking.

Jan Van der Sypt is a SanFrancisco Support Engineer in Belgium. He has five years of experience in Object Technology. He holds a degree in Commercial Engineering from the Economical Highschool Saint Aloisius in Brussels, Belgium. His areas of expertise include Application Development with IBM Smalltalk and Java applications, which are based on the IBM SanFrancisco framework. He has been instructor at the Object Technology University, La Hulpe, Belgium.

Thanks to the following people for their invaluable contributions to this project:

Christopher Abbey
Randy Baxter
James Carey
Adam Dirstine
Tim Graser
Steve Halter
Jay Johnson
Jim Kidd
Steve Kiss
Wilson Lee
Kenton Lynne
Mike McKeehan
Stephanie Meizer
Steve Munroe
Mark Pasch
Phil Sanders
Jim Van Oosten
IBM Rochester

Achim Nogli
IBM Object Technology Consulting Practice Germany

Nathalie Vilaine
IBM Belgium

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “ITSO Redbook Evaluation” on page 217 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

  For Internet users `http://www.redbooks.ibm.com`
  For IBM Intranet users `http://w3.itso.ibm.com`

- Send us a note at the following address:

  `redbook@us.ibm.com`

Chapter 1. Introduction

Performance remains a hot topic. Everyone talks about it, wants the most of it, and sometimes confuses the mechanics behind it. This introductory chapter focuses on the question: What is meant by the best performance? It also serves as a guide to read this book depending who the reader is, and what his or her problem is. Two different situations may exist: the reader seeks information on performance to use it in developing, or the reader already has a performance problem and is looking for guidance on how to understand and solve the problem.

1.1 Performance in General

This section deals with performance in a general way and focus later on performance in a distributed IT environment. Additional information on performance can be found in several books in the bibliography. Chapter 2, “General Performance Issues” on page 7 deals with the different aspects of performance.

1.1.1 What Performance Is

Without looking for an exact, scientific definition in a reference, we all have a certain understanding of what good performance in information technology (IT) environments means:

- Fast
- Short response times, responsiveness

In general, performance is the ratio of a certain work, divided by the time it took to complete. In a computing environment, performance is usually measured by "how long this request needs to be serviced" or "how many requests this system can handle in a single second." All these phrases refer to the same ratio: work divided by time.

1.1.2 Why the Need for the Best Performance

It is a well-known fact that patience is not among the most widespread characteristics of human beings. Waiting for a machine to deliver an expected result is widely considered to be boring, even if only takes a short period of time. People using computers (end-users) get easily frustrated when they have to wait for computers to fulfill their requests.

Although this may not seem strong technical reason to invest resources to improve performance, it is among the most important reasons. Motivation, acceptance, and productivity largely depend on the infrastructure people use.

To be successful in a competitive market place, a company needs to be responsive to customers and improve troughput of the services and products they offer. Basically, companies need to be productive as much as possible. This high performance is only obtained if the people working in this company, and the infrastructure used, has the "best" performance.

When asked what performance people request, the usual response is "the best performance." This can only be achieved by investing an unreasonable amount of money. In an environment with limited financial resources, this is unrealistic. The

good thing about it that it may also be unnecessary. The main question becomes: What is appropriate performance?

1.1.3 What Appropriate Performance Is

The appropriate performance for an application, or a process, is precisely that performance is expected right from the beginning. For every situation, this may differ.

A fighter pilot will expect his jet to be reactive within milliseconds; where, an accounting program will attain the desired performance if an operation is finished within three seconds. A fighter pilot cannot live with a jet that has the performance of the accounting program, nor does it make any sense to have an accounting program finish its operation within a couple of milliseconds, as the operator will have to enter the necessary information anyway, which takes time.

In other words, the appropriate performance for a process is obtained if the user of that process can continue working without being held up by the process. If waiting times of a process are such that they are not interfering with the normal continuation of work in process, which may consist of manipulation by the operator or mere think time of the operator, the appropriate performance is obtained.

It is of no use to improve the performance of a business process that already has reached the appropriate level of performance.

Reaching the level of appropriate performance is not always easy, as an organization has limited financial resources. Performance can be improved by using faster machines, implementing better communication, or investing a lot of man hours in changing the code. All this takes money, which is limited. Compromises and wise decisions need to be made. The art is to spend the available financial resources in a way that the largest performance improvement is obtained.

To do so, a good understanding of the process is necessary, as the effort needs to be focussed on the bottleneck of the process. The performance of each process will always be determined by the point in the chain of actions that causes delays. This point is called the bottleneck. If this slowest part can be improved, making the delays disappear, the overall performance will improve. At the same time, another part in the chain will become the bottleneck. To improve the performance even more, this shift needs to be understood so that efforts can be focused on this new bottleneck.

1.1.4 Performance in a Distributed IT Environment

Performance analysis and problem determination in large, heterogeneous, and distributed environments is not easy. There are a number of benchmarks available for evaluating the performance of a CPU or how fast a terminal can display some graphics, and for many other purposes, but is it almost impossible to define methods to measure how good a distributed computing environment performs.

Performance tuning in a distributed environment concentrates mainly on:

- Achieving fast response times for remote requests.
- Circumventing bottlenecks that may slow down a whole infrastructure in any shared resources, such as servers.
- Providing scalability options for the expected growth of the overall system.
- Keeping the required investments in an acceptable range.

These involve thorough planning (design), methods for evaluating, and ways to detect and resolve performance-related problems. Every installation site needs to investigate which functions are the most important ones, what needs to be fast, as other people or services are waiting for the result. It must be obvious that a distributed application can not be faster than a local one, unless remote computing power is involved in a certain distributed transaction.

1.2 Where to Go from Here

Everybody involved in an IT development should be aware of performance issues. It depends on the task somebody is doing at a given time to what degree performance is an issue. The different actors in the development cycle make their contribution to performance at a different points in time.

The way this book can be used will greatly depend on the type of reader and in what stage of the development cycle of a SanFrancisco based application one is.

1.2.1 Where You Are in the Development

If you are unaware of the tips and techniques discussed in this book, the best place to be, in this case, is nowhere, meaning right at beginning of the development cycle of an application. If no work has been done so far, there is also no work that needs to be redone in order to improve performance. Performance tuning is an ongoing task, and the best place to start with it is directly from the beginning. The best way to proceed, in this case, is to read the book in a consecutive manner. Every part in this book will prove its usefulness.

If this book comes in right at the end of a development phase, where severe performance problems are suffered, several parts of this book will not be very useful if the application can not be reengineered. But still, many things are still possible: by focussing on the configuration of the different components in a SanFrancisco environment, performance can be improved. We recommend that you read Chapter 6, “Hardware and Software Configuration” on page 91, Chapter 7, “LSFN Configuration” on page 107, and Chapter 8, “Object Persistence, Databases, and Schema Mapping” on page 123.

1.2.2 What Your Job Is in the Development

This redbook has different audiences: decision makers, developers, architects, and performance people. Depending on who you are, the following advice is useful on how to proceed in reading the next chapters.

1.2.2.1 IBM SanFrancisco Application Developer

A developer takes the design object model as an input for his job. He implements the model, makes use of code generations, and plays an important role in the deployment of the application.

Developers should be aware of performance at code level. For them, it will be useful to focus on the chapters with coding tips and techniques on Java and SanFrancisco: Chapter 9, “Java Coding Tips” on page 147, and Chapter 10, “SanFrancisco Coding Tips” on page 163. These chapters will even be of interest to people involved in an ordinary development project with Java.

Code generators lower the burden on the developers, but sometimes it may be necessary to make changes to the generated code in order to obtain the best performing code. For more information, see Appendix B, “Modifying Generated Code” on page 201.

Since developers may be involved in the deployment of the application, it may also be useful to have an understanding of the performance issues that arise when configuring databases, configuring the Logical SanFrancisco Network and setting up the hardware and software. The following chapters deal with these matters: Chapter 6, “Hardware and Software Configuration” on page 91, Chapter 7, “LSFN Configuration” on page 107, and Chapter 8, “Object Persistence, Databases, and Schema Mapping” on page 123.

1.2.2.2 IBM SanFrancisco Application Architect

Architects will especially be involved in the first phases of a development cycle. They need to consider during analysis and design certain trade offs between flexibility/functionality and performance. It is therefore crucial to have a good understanding of certain constructs used in a SanFrancisco development. Some of these constructs will have an influence on the overall performance of an application built with SanFrancisco. Chapter 5, “Performance Aspects Using Design Patterns” on page 67 will definitely be indispensable.

The layout of the underlying relational database is important. Normalization of the databases tables, the choice of column types, and the choice of indexes will all be very important in performance. Sometimes an existing relational database is used. Also, in this case, good decisions can be made. Chapter 8, “Object Persistence, Databases, and Schema Mapping” on page 123 gives useful information.

1.2.2.3 IBM SanFrancisco Performance Consultant

Consultants dedicated to performance will be involved through all the different phases of a SanFrancisco development. Their primary task will be to monitor performance, give advise in decisions, and predict the impact on performance of certain decisions. It is clear that they will require the broadest range of experience possible in performance issues among their peers. The entire content of this redbook will be of interest, including the introduction, to measuring and mathematically approaching performance (Chapter 2, “General Performance Issues” on page 7).

1.3 Performance in SanFrancisco Based Applications: An Approach

This section presents you with a frame that can help you in deciding how to spend time and effort on performance issues throughout the different stages of a development cycle:

- Be clear about the application that needs to be built.
- Be convinced that SanFrancisco can offer the functionality and flexibility that is desired, now or in the future.
- Be clear about the appropriate performance that is needed for this application.
- Investigate whether the SanFrancisco based application will achieve these requirements, now or in the future. An analytical model might be the correct instrument to give the proper indications.
- Build an analysis object model focussing primary on the business requirements. Performance should not be an issue in this stage.
- Build a design object model while considering design decisions and their influence on performance. An analysis model can, in most cases, be translated in different designs. It is important to make wise decisions at this point, as some of them will constitute the core of the application, making them more difficult to change after implementation.
- In most cases, the design object model will form the input for a code generator. Do not start changing the generated code for performance at this time, as chances are great that it will not be the last time the code is generated, overwriting the modified code.
- During implementation of the business logic, remember the concept of Commands. They will be crucial for performance. It might be that some command object has been forgotten during design. Revisit the design model whenever necessary for updating. Also take into account the numerous coding tips on Java and SanFrancisco that are provided.
- Throughout the development, performance monitoring should be performed in order to detect, as soon as possible, shortcomings in design or implementation. This is best done by a dedicated person not directly involved in the coding. These tests of the code should be performed in as real conditions as possible. If this is not feasible, a simulation model should be developed. Performance testing should use relational databases and no posix stores.
- When the business logic is implemented, some modifications on the generated code can improve performance.
- The client code will mainly consist of GUI code that will execute commands against the servers. GUI code is definitely an area where a big performance hit can exist. Care should be taken to avoid too heavy and multiple reads to the database in order to populate lists and fields. Some techniques can be used to accommodate these requirements.
- While deploying the application, many parameters can be tuned. These configurations involve the database, the network, the hardware of the machines, and the SanFrancisco environment.

1.4 Understanding and Solving a Performance Problem

In the case of bad performance of a SanFrancisco based application, the first and most important questions are: What is the cause of the performance problem, and where does it originate? Without an answer to these questions, any changes to the configuration or the implementation of the application will probably not solve the problem. Chapter 3, “How to Find a Performance Problem” on page 23, helps answer these questions. When the problem is understood, advice and tips can be found in other chapters that helps solve the performance problem.

Chapter 2. General Performance Issues

This chapter deals with measuring performance and with the elements that influence performance. It is not meant to offer a thorough description of all aspects, but as an introduction, it covers the different topics. For readers already familiar with performance issues, this chapter may not provide new information; people that are novices in performance should read it. For a more mathematical foundation on performance measurements, other literature can be consulted.

2.1 Influencing Performance

This section covers the most important elements that influence performance. It is meant as an initial mindset on the different factors that play a major role in performance issues. These topics are dealt with in detail in relation to the IBM SanFrancisco framework in other chapters.

2.1.1 Architecture

The architecture in a distributed environment will have a significant influence on performance. Attention should be paid to minimizing the number of possible contention points when serializing information and while synchronizing processes.

A contention point is a position in the chain of actions that frequently causes waiting locks. This means that different processes would like to have an access to a particular element in the architecture. In a distributed, object-oriented environment, this element can be many things: a database, a harddrive, a queue, a particular object, and so on. It is crucial to understand these contention points in order to make them disappear. The solution to avoid contention can be to put a process on a different server, duplicate the element that creates the contention, or review the locking scheme.

While sending information over a communication link, the data needs to be streamed, transformed in a long series of sequential bytes. This serialized data needs to be transformed back by the receiver into the original data. Serializing does not refer to TCP/IP operations where the serialized data is packed to send over the link. Serializing refers to the operation that comes on top of TCP/IP. Serializing can be very expensive.

When two processes run on separate threads, they normally do not know about each other. In certain situations, they need to know. For example, a process requiring information that another process is calculating needs to wait until the second process comes up with an end result. If the first process needs to wait, it also needs to know when it can proceed. It is the second process that signals to the first one that it can proceed. When a process is put in a waiting state, it is called "suspended". When it resumes processing, it is "resumed". The whole mechanism of communication between processes is called "synchronization". Synchronization implies an additional workload. It can also lead to a degrading performance in the case of a bad architecture. It is possible that a slow thread performing a batch job in the background might be blocking the whole system.

Further information on this can be found in 10.1, “General Techniques” on page 163.

2.1.2 Software Path Length

In general, when talking about the code used, an important concept needs to be mentioned: path length. The path length of a task, or a business process, is the number of instructions that are executed in the CPU and the number of disk accesses that are processed in order to complete the business process.

It is obvious that the path length should be as short as possible. This will definitely depend on other matters that are influencing the performance, for example, caching.

2.1.3 Hardware

The faster that a machine used is, the faster the same task runs. The speed is certainly not only determined by the CPU, other elements of the infrastructure are important:

- The throughput of the internal bus
- The distance between components
- The speed of the hard disks used
- The number of disk arms of the hard disks
- The number of CPUs
- The thoughput of the network

Further information can be found in 6.1, “Hardware Recommendations” on page 91.

2.1.4 Synchronous versus Asynchronous Processing

Synchronous processing means that every step in a chain of actions is performed one after the other, where every following step is initiated only when the previous one has finished. This way of processing limits the throughput of the whole system to the throughput of the slowest element in the chain. The response time may be too long to obtain the appropriate performance.

In the example where a text editor sends a document to the printer one page at the time and waits for the printer to finish printing this page before sending the next one, synchronous processing is used.

In asynchronous processing, there is no handshaking between different processes. They just keep on delivering work to one another without waiting for acknowledgments at every turn. In order for a process to work independently, but still delivering input to the next process in the chain. There is a need to store end results from one process until the next process will come and pick it up as an input for itself. The intermediate stores are often referred to as buffers.

The different processes are aware of one another but only in the sense that they will wait when it is absolutely necessary. For example, if a buffer is full, no more end results can be put in, and the first process needs to stop until the second process clears the buffer during processing.

In the example where a text editor sends a document to the printer one page at the time but does not wait for the printer to finish printing every page before sending the next one, asynchronous processing is used. Therefore, the printer has a buffer where all received data is stored until the printer can print it out on paper. If too much data is received from the text editor, the printer will send a

signal that its buffer is full, which will cause the text editor to stop sending data. It waits for a period of time, after which, it resumes sending.

Asynchronous processing can increase the performance significantly in a distributed environment, as the different processes run on different threads. Also, on a single machine, asynchronous processing can be interesting to use. For example, the CPU can already process the display of the first part of the records while another process retrieves the following parts of the records.

2.1.5 Communication

Remote calls between a client and a server, or between two servers, are very expensive in terms of performance. They should be limited as much as possible.

A good rule of thumb should be: work with the data locally. This means it is a better idea to move the process to the machine where the data can be accessed directly without going over the network with the data.

In a heavily distributed environment, it is, therefore, essential to know where the data is stored that is used in a particular process.

Additional information can be found in 6.4, “Communication” on page 101.

2.1.6 Caching

Frequently and repetitively used information should not be retrieved from the harddisk to active memory over and over again every time it is needed. Instead, it should remain in active memory where it can be accessed directly without the overhead of an expensive IO operation or database access.

One must decide wisely what to cache and when. Not every piece of information can be cached, as this would consume too much of the available active memory, leaving too little for the actual processing.

Since a cache mechanism only has a limited amount of memory to its disposal, it is important that the correct information be in memory. The decision to put something in the cache and throw another piece of information out is based on an algorithm. The better job this algorithm does, the better the overall performance will be.

The performance gain obtained by a cache mechanism depends on:

- The time spent on retrieving the requested information if not found in the cache. This could be little, for example, when reading from a fast local harddrive, or a lot, when using a slow modem connection.
- The ratio of the number of times that requested information is actually in the cache (a cache hit) over the number of times it is not in the cache (a cache miss).

Using a cache mechanism that frequently encounters a cache miss will impair the performance even more than to not using a cache. In the case of a cache miss, the information needs to be retrieved anyhow. This makes the read in the cache an additional operation compared to directly retrieving the information without asking the cache first.

A difference needs to be made between a read cache and a write cache. A read cache only caches information that it has read. In the case of a change made to the information that is cached, this change is immediately written back to the underlying persistency. When using a write cache, a change made to the information is only written to the persistency when an explicit request is made or prior to removing the information from the cache.

An additional problem that may exist with the use of a cache is the updating of the cache whenever another process changes the cached information directly on the underlying persistency. An easy way to avoid this is forcing every access to the information to go through the caching mechanism. This is not always possible.

Additional information can be found in 10.1.1, “Caching” on page 165.

2.1.7 Prefetching Data

To limit the number of accesses to the hard disk, an algorithm can be used to predict what information will be asked next by the process that requested information. Instead of accessing the hard disk every time a piece of information is requested, a system can be used that already prefetches the data that the algorithm predicted that would be asked for later. This way, requested information can be returned from active memory, which is far more preferable.

This mechanism resembles caching, but the difference is that this is a proactive mechanism, where caching depends on repetitively accessing the same data.

2.1.8 Locking

Locking is the process of obtaining a right to access or to modify a component. This component can be a database, a file, or an object. Assume the component is an object. If there is only one process accessing or updating the object, there is no problem possible. The lock is granted. After processing, the lock is released. Whenever there is a second process that requests access to the same object, there may be a problem depending on the nature of the granted locks that already exist. If the first process gets a read lock on the object, other processes can get a read lock afterwards as well. A read lock grants access to read the object but does not allow any changes. A different type of lock is the write lock. This allows updating the object that the write lock was granted. A write lock can only exist when no other read or write locks exists. A process may request a write lock on an object to which another process was already granted a read write. In this case, is has to wait until this read lock was released before it is granted its write lock.

It should be clear that a bad locking scheme causes a lot of contention points that slow the system down. Two rules of thumb are: "release a lock as soon as it is no longer needed" and "never hold a write lock for a long time".

Other aspects need to be mentioned. A process does not always wait forever for a lock to be granted. A timeout can be used. This is the time that a process waits for a lock. If the time-out is too short, this may break the processing unnecessarily. For example, if a lock is requested on an object that is accessed over a slow communication link with a time-out of three seconds, may never result in obtaining a lock as it is very possible that the roundtrip time is more than the three seconds. Even if the lock could be granted, it is timed-out. A time-out period needs to be appropriate to the environment used. This period should not be too short to guarantee the possibility to obtain a lock. On the other hand, it should not

be too long so that the system remains responsive even if this means signaling failure of obtaining the lock. If the time-out period is indefinite, a situation of deadlock can occur.

A deadlock is the situation where two processes are waiting for a lock on an object that the other process has already locked. The lock is only released after the process gets the second lock for which it is waiting. Since both processes are in the same situation, there is no chance that they will ever get the second lock they want.

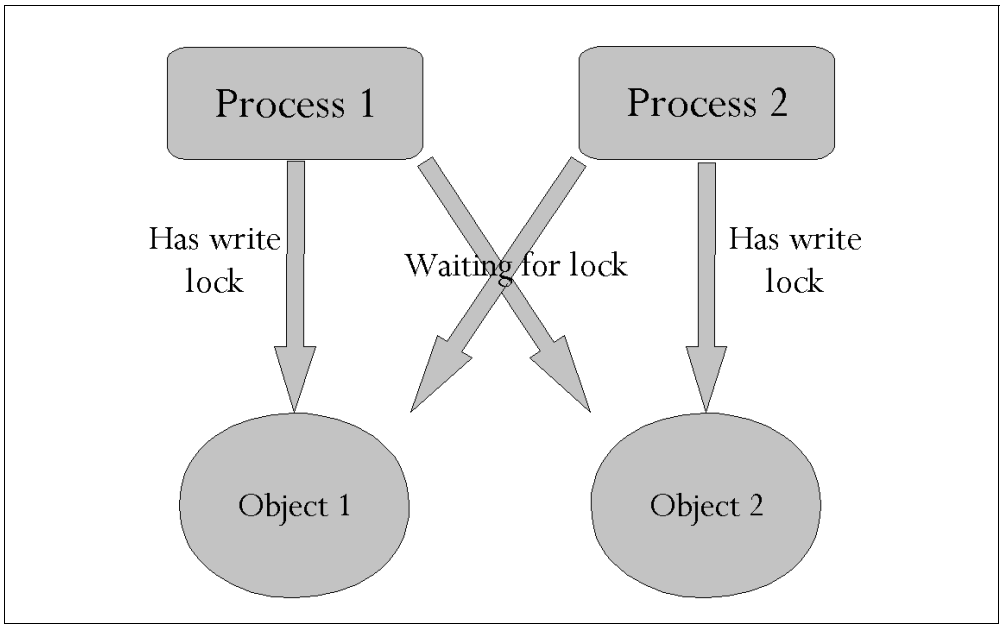


Figure 1. Dead Lock Situation

2.1.8.1 Optimistic versus Pessimistic Locking

When optimistic locking is used, a component is locked during read, a working copy of the component is created, and the lock is released again. Changes are made to the copy, and when committing these changes, the real component is locked for write, and the changes are made. The advantage is that the component becomes available for read access by other processes, hence limiting contention. The disadvantage is that the component also becomes available for write access. It is possible that another process has made a change to the real component between the first read access and the second write access. At that point, there is no longer any certainty on the state of the component. Different policies are possible: start over again, or investigate whether these changes are important to the proper modifications. It depends on the frequency that this situation occurs and on the effort to start over again. If it rarely occurs, and the modification is easy to do again, just start over again. In the other cases, recovery code is needed that tries to accommodate the requested modifications on the changed component. Sometimes this is not possible and forces the user to start over again.

When pessimistic locking is used, a component is locked for write for the entire duration of modifying the component. The lock is released after the changes are made. The advantage is that there is no risk that others have modified the component during the process. The component will always be in a coherent state.

There is no need for recovery code. The disadvantage is the increase in the possibility on contention. Other process may want to access the component for read but are locked out. Pessimistic locking is a good policy in situations where the transactions are rather short and where there would be a high risk on conflict, as encountered with optimistic locking.

There are some additional considerations to make when dealing with locking issues in the SanFrancisco environment. This information can be found in 10.2.2, “AccessMode and Locking” on page 179.

2.1.9 Object Oriented Issues

This section includes performance aspects specific to an object oriented environment such as the SanFrancisco environment.

2.1.9.1 Garbage Collection

Memory management in Java, and in some other OO languages, is done automatically. This means memory blocks are allocated when needed and freed up when no longer used. The allocation is done when objects are created. The deallocation is done periodically by a process called the Garbage Collector (GC). This process will check for objects that are no longer referenced. If so, it de-allocates the memory blocks occupied by that object. This process ideally runs in a low priority thread only running when the process has idle time. On a very busy machine, there is no idle time when the GC needs to run. In this case, the GC takes time away from the other business processes degrading the performance.

It is essential to limit the GC runs, definitely during heavy load periods. The best way to limit GC runs is to limit the objects created. Creating the same objects and abandoning them again over and over again will run the GC regularly. Reuse of objects is preferred.

Additional information can be found in 9.3, “Memory Management” on page 152.

2.1.9.2 Synchronization

Synchronization in Java is an expensive operation. It can exist on the level of a method or on the level of a code block. A synchronized method indicates that only one process at the time can invoke this method. Other processes have to wait. It is used as a kind of locking system to prevent objects getting in incoherent states. Invoking a synchronized method is a very expensive operation and should be used carefully. If there is no risk on incoherent state, it should not be used. An important aspect to know is that a lot of Java base classes use synchronization internally. The use of these classes can cause performance problems.

Additional information can be found in 9.4.7, “Synchronization” on page 159.

2.1.9.3 Exceptions

Exceptions in Java should be used only to signal real error conditions, not as a means to put back a return value. Only the abnormal conditions, the ones that happen rarely, should use exception throwing. Additional information can be found in 9.4.4, “Exceptions” on page 156.

2.1.9.4 Object Creation

If possible, objects should be reused rather than recreated. Depending on the size of the object, creation can be very expensive. Especially with the big objects, consider reuse. An additional benefit of reusing objects is the lower cost in garbage collection.

Additional information can be found in 9.3.2, “Reusing Objects” on page 152.

2.2 Measuring Performance

Discussion about performance does not mean anything if there is no way to use clearly defined measurements that enable comparing performance to a standard.

You should address performance only after having a fully tested and correct program. Complete the following steps:

1. Get agreement on the appropriate performance (see 1.1.3, “What Appropriate Performance Is” on page 2).
2. Measure the program’s performance under realistic conditions. If it meets the expectations, stop.
3. Determine the bottleneck part to focus on. Do not guess where it might be, but measure where it is.
4. Improve the performance of the part that was identified as the bottleneck.
5. Go to step two.

2.2.1 Common Measurements

Different measurements can be useful while profiling performance in a distributed environment. This section briefly describes several of the measurements.

2.2.1.1 Transactions Per Minute (TPM)

This is a measurement of raw troughput—how many transactions of a specific type can be executed in a single minute. Typically a software testing tool is used to generate the transactions against the server and measure the results. Depending on the server being measured and the type of transaction, Transactions per second (TPS) or transactions per hour (TPH) may be a more appropriate measurement.

After measuring the peak TPM, you should be able to estimate how many users your server can handle. For example, if tests show that a server could handle about 50 transactions per minute, and the average user issues two transactions per minute, the server should be able to handle approximately 25 concurrent users. Things get more complicated to calculate if a lot of contention exists as concurrent users need access, for example, to shared objects. In this case, the throughput is significantly less.

In most cases, TPM is a good for measurement figure for a SanFrancisco environment that is typically a business-oriented environment that has transactions that take, on average, a few seconds.

2.2.1.2 Response Time

This is a measurement of how long it takes from the time a transaction is issued until the server begins to issue results. Adequate response time varies greatly

depending on the type of transaction. It is best to survey the users to determine what response time is appropriate for a given transaction. Typically, the Maximum, the Average, and the 90th percentile measurements of response time are of interest.

2.2.1.3 Retrieval Time

This is a measurement of how long it takes from the time the server begins to issue results until all of the results are received. As with response time, adequate retrieval time varies based on the type of transaction, and typically the Maximum, Average, and 90th percentile measurements of retrieval time are of interest.

2.2.1.4 Network Time

Network time is a measurement of how long the data spends on the wire. It can be determined using a network sniffer or by executing simple transactions (which should have near 0 response time against the server both from the server and from a client workstation). The network time should be roughly the difference between both. Both response time and retrieval time are dependent, not only on the server, but also on the network.

2.2.1.5 Transaction Time

Transaction time is the sum of the response time and retrieval time—how long it takes from the time of initiating a transaction until the last result is provided.

2.2.1.6 Bandwidth

Bandwidth is the amount of data that can be transmitted over a channel per unit of time. Often, the percent Bandwidth is of more interest. These measurements apply to CPUs, networks, active memory, and so on. They are usually used to determine bottlenecks.

2.2.2 Capacity Planning

Capacity planning is the art of managing three different variables:

- The present and future applications
- The appropriate performance needed
- The configuration of the hardware and software used

It is clear that these three variables are interdependent. The applications that run on the available configuration result in a certain service level. The concept of service level is very much related to the appropriate performance level. The service level that is obtained needs to map to the requested performance level. To maintain the appropriate level of performance, forecasting methods are used to predict the service level with a given set of applications running on a given configuration.

There exist about three acceptable ways to predict the service level. Two of them are based on modeling. A model is an abstract construction that incorporates all the essentials from the real world. An ideal model simplifies the reality by taking into account the important elements while neglecting the less important factors. A model will represent the reality.

The two different modeling techniques used are:

- Analytical modeling
- Simulation

These two modeling techniques are discussed in the next sections. The third way to predict the service level is benchmarking. Benchmarking is discussed in a separate appendix.

In most situations, analytical modeling is the first step. It may even be a part of the decision making. Before any work is done, it is important to have a view on what kind of infrastructure is needed to obtain certain service level.

The next step is benchmarking. Before, and during development, benchmarks can give indications whether the application code is reached and not a performance level predefined by a set of standards.

In complex configurations with many interrelated influences on performance, it is beneficial to develop a simulation model. If correctly defined, this gives the best predictions.

A Word to the Wise

It is a myth to believe that a completely reliable model can ever be developed. The only thing that one may try to achieve is a model that minimizes it shortcomings in a way that they become unimportant to the end result. This end result always follows a distribution, even if the model gives exact figures. They can never be exact but always sensitive, in a certain degree, to a fault tolerance.

The best approach is always to validate the end results of as many methods you can use. The more they converge, the better the view of the system becomes.

2.2.3 Analytical Modeling

This method is generally accepted as a good way to predict the performance of applications because it:

- Deals with present and future application and systems
- Can predict service levels
- Is sufficiently adequate for most purposes while staying simple

An analytic model has two main components:

- The System, which is a simplified description of the major hardware items (processors, controllers, discs, tapes) on a configuration (real or potential).
- The Workload, which describes the number of times each device in the system is used by different types of transactions or jobs.

The combination of a System and a Workload is referred to as a Forecast. The modeling tool evaluates the Forecast and predicts what service levels will be provided by that combination of hardware and software. Hence, those three variables, the Applications, the Hardware, and the Service Levels, can be integrated into successful plans.

A baseline model is an analytical model built to represent the current state of the system and workload. The model is built from a standard system and workload monitoring sources appropriate to the hardware and operating system in use. As

you would expect, the model predicts service levels. In this case, the predicted service levels can be verified and tested by comparing them with the actual service levels measured by the system monitoring tools. This allows planners to verify the accuracy and applicability of the model before it is used to predict the future.

By modifying a verified baseline model, one can represent changes to the workload and to the hardware. The changed model predicts the levels of service that can be provided under these new circumstances.

The level of detail required for successful planning may well be different from that required to solve a performance problem. Some confusion can arise because both activities use largely the same data. While it is possible to use Analytic Models to resolve detailed application-specific performance problems, it is their long-term impact that is more important.

2.2.4 Simulation Modeling

Simulation is a modeling technique that is based on statistical mathematics. The main purpose of this section is to illustrate what simulation may encompass. A short description of the concept of a distribution of a variable is given, followed by describing the different steps that occur in developing a simulation model. An explanation on how simulation might be used in SanFrancisco finishes this section.

2.2.4.1 Distribution

The value of a variable, an element that influences the performance, can follow a distribution. This means that the value can never be exactly predicted, but in the mean time, it is certain that the value is always in a certain range. For example, the number of transactions a user launches against a server each hour can vary from 10 to 60, but every hour will be different. The value depends on coincidence, uncertainty, risk, and so on.

The most simple distribution is the uniform distribution. The value of a variable that follows this distribution has an equal chance on every possible outcome. For example, there is equal change for having 10 TPH, as for 46 TPH or 58 TPH. It should be obvious that many variables will not follow a uniform distribution. There exist many more. Discussing these distributions is out of the scope of this book.

2.2.4.2 Simulation

In a situation in which the elements that are important to a model follow a distribution, the model that abstracts this situation cannot be analytically composed. A simulation model is needed. In this kind of model, the input parameters follow a distribution, and the end result follows a distribution as well. This entire modeling technique is based on the probability theories. Figure 2 shows the general principals from a simulation model. The parameters that follow a distribution are fed into the model that calculates the distribution of the end result.

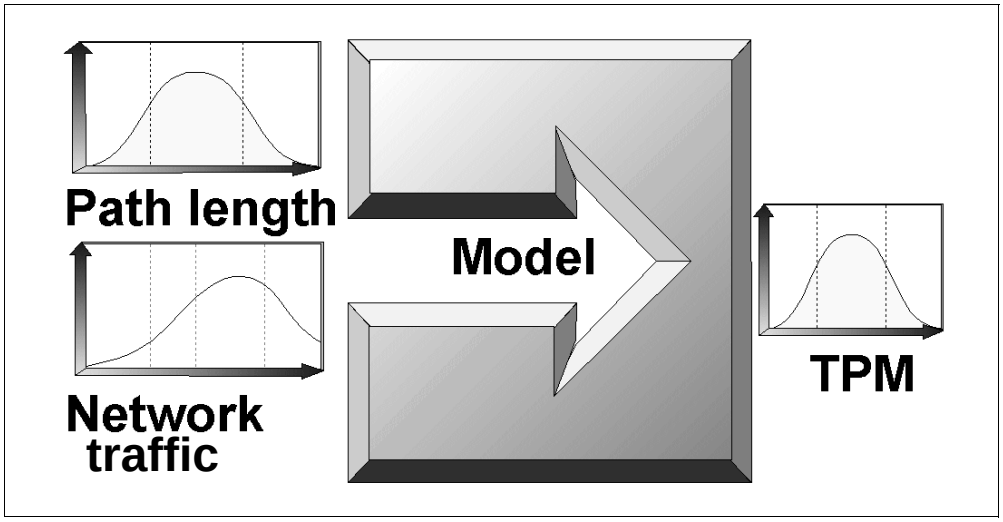


Figure 2. Simulation: Bird's-Eye View

2.2.4.3 Developing a Simulation Model

Simulation models are potentially very accurate, but they require a lot of skills and expertise to produce good results. If defined correctly, they can have a high forecasting value.

During the development of the simulation model, you must ensure that the model is correctly implemented and that it is representative of the real system. These two steps are called model verification and model validation. After the model development is complete, the next two issues faced are those of deciding how many of the initial observations should be discarded to ensure that the model has reached a steady state and how long to run the simulation. These issues are referred to as transient removal and stopping criterion.

2.2.4.4 Model Verification

The first step is needed to check whether the assumptions used in the model are reasonable, and the second step consists of checking whether these assumptions have been correctly implemented. Only a model that correctly implements the right assumptions gives useful results.

Different methods can be used to verify the model. It goes beyond the scope of this introduction to deal with them in detail, but some examples are:

- Top-down modular design
- Structured walkthrough
- Deterministic modeling
- Degeneracy tests
- Consistency Tests

2.2.4.5 Model Validation

Validation refers to ensuring that the assumptions used in developing the model are reasonable. That is, if they are correctly implemented, the model will produce results close to that observed in real systems. Model validation consists of validating the three key aspects of the model:

- Assumptions
- Input parameters and distribution
- Output values and conclusions

Each of these three aspects may be subjected to a validity test by comparing them with that obtained from the following three possible sources:

Expert Intuition
This is the most practical and commonly used way to validate a model. A brainstorming meeting of the people knowledgeable with the system and with the domain should lead to detection of counter intuitive output values that are produced by the simulation model. If these output values are not acceptable, it remains to be determined whether the results are caused by an implementation error or by wrong assumptions.

Real System Measurements
Comparisons with real systems are the most reliable and preferred way to validate a simulation model. Even one or two measurements add considerably to the validity of a simulation. In practice, however, it is rare that these measurements exist, since they may be too expensive to carry out, or the system does not yet exists.

Theoretical Results
In some cases, it is possible to analytically model the system under specifying assumptions. In such cases, the similarity of theoretical results and simulation results are used to validate the simulation model. This comparison should be used with care, as both may be invalid in the sense that they both may not represent the behavior of a real system.

2.2.4.6 Transient Removal
In most situations, only the steady-state performance, that is the performance after the system has reached a stable state, is of interest. The results of the initial part of the simulation should not be included in the final computations. This initial state is also called the transient state. The problem of identifying the end of the transient state is referred to as the transient removal. Therefore, the main difficulty is that it is almost impossible to be precise as to where the transient state ends. All methods for transient removal will therefore be heuristic, meaning based on a set of rules.

Certain systems never reach a stable situation. They need to be studied with more complex methods that make use of chaos theory.

2.2.4.7 Stopping Criterion
It is important that the length of a simulation is properly chosen. If too short, the results will be highly variable. If too long, computing resources and manpower may be unnecessarily wasted.

2.2.4.8 Simulation in SanFrancisco
Let us consider the following situation. A list of 10 items needs to be displayed on screen. Every row consists of one string that is retrieved from a distinct Entity maintained by a Controller. The entities reside in a container on a server running on a separate machine from the client machine. The container is linked with a DB2 database.

Then, ask this important question: What are the elements that influence performance of this operation? The performance will be measured in Transaction Time. The following elements can be taken into account:

- **Network load**: The higher the load, the slower the communication.
- **Garbage collector**: When the garbage collector kicks in, time is spent.
- **Database retrieval time**: Depending on the position of the disk arm, caching schemes of the database, and so on.
- **Paging**: When the operating system needs to page, time is spent.
- **Caching**: When Entity is in cache, no database access is needed.

Many other elements are important and should be included to have a useful model.

For every element, assumptions need to be made. This means that the possible input values are determined together with the probability of happening. For example, Garbage collection will happen occasionally but will have a big impact. The distribution of this element could be as in Figure 3. This is an example of an exponential distribution. It is unlikely that there will be a GC right after the previous one. As more time passes, the more likely it becomes that a GC will start. The impact of the Garbage Collection happening will be included in the model itself.

Somewhere in the model a formula will exist attributing a time penalty multiplied with the probability of the GC parameter and with the probability of the duration of the GC.

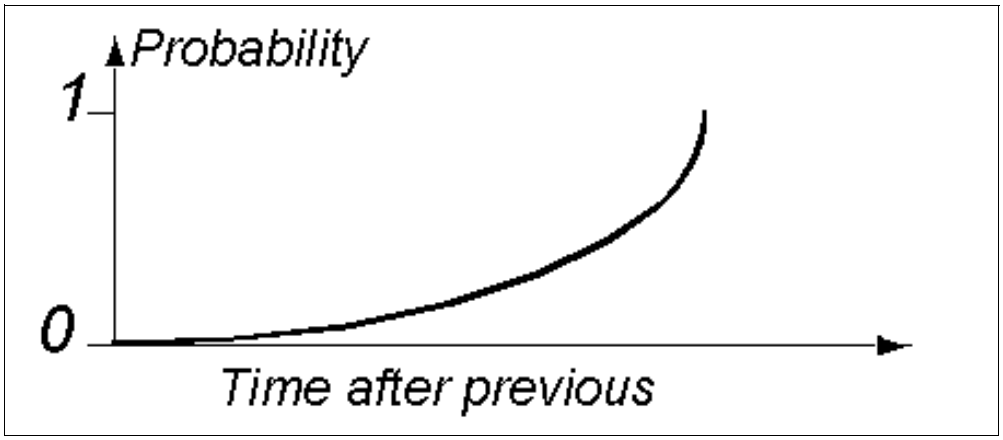


Figure 3. Distribution of the Garbage Collection

The end result of the simulation model will also follow a distribution. This distribution can be very similar to the one shown in Figure 4. This is an example of a gamma distribution. A precise answer is impossible to give. Answers are in the form of transaction time being less than 6 seconds with a probability of 90% or a transaction time of less than 5 seconds with a probability of 85%.

Needless to say, developing a simulation model is not easy.

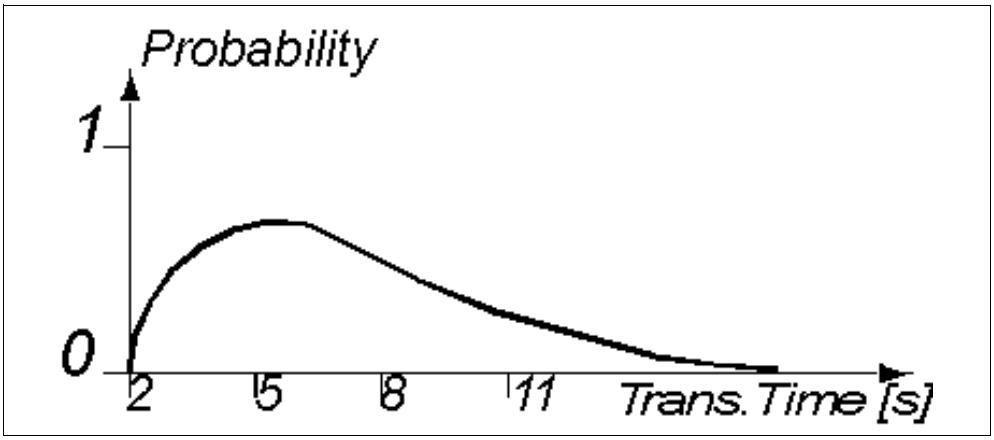


Figure 4. Distribution of the Transaction Time

2.2.5 Queuing

This section deals with different aspects of the queuing theory. It is typically a model that describes an asynchronous I/O process. At one end, information arrives, and at the other end, data is retrieved. In the middle, the information needs to wait before it is retrieved; it is queuing up. The mechanics of this process is described in a queuing model.

2.2.5.1 Introduction

As is often the case in computer systems, servers typically process many simultaneous jobs (for example, file requests), each of which contends for various shared resources: processor time, file access, and network bandwidth. Since only one job may use a resource at any time, all other jobs must wait in a queue for their turn at the resource. As jobs receive service at the resource, they are removed from the queue. All the time, new jobs arrive and join the queue. Queuing theory is a tool that helps to compute the size of those queues, and it views every service or resource as an abstract system consisting of a single queue feeding one or more servers. Associated with every queue is an arrival rate (A)—the average rate at which new jobs arrive at the queue. The average amount of time that it takes a server to process such jobs is the service time (Ts) of the server, and the average amount of time a job spends in the queue is the queuing time (Tq). The average response time (T) is simply Ts + Tq.

2.2.5.2 Stable Queues

If the arrival rate is less than the service rate (1/Ts), the queuing system is said to be stable. All jobs are eventually serviced, and the average queue size is bounded. On the other hand, if A > (1/Ts), the system is unstable. The queue will grow without bound. The product of the arrival rate and service time yields the utilization of the server (U = ATs). This is a dimensionless number between 0 and 1 for all stable systems. A utilization of 0 denotes an idle server, while a utilization of 1 denotes a server being used at maximum capacity. If the amount of time between job arrivals (1/A) is random and unpredictable, then the arrivals exhibit an exponential or "memoryless" distribution. This distribution is extremely important to queuing theory. A queue in which the inter-arrival times and the service times are exponentially distributed is known as an M/M/c queue, where the M's represent the Markov, or memoryless nature of the arrival and service rates, and the c denotes the number of servers attached to the queue. When

service history of a queuing system is irrelevant to its future behavior (only the current state of the system is important) that history can be ignored, greatly simplifying the mathematics.

The response time curve of an M/M/1 queue as a function of utilization is shown in Figure 5. At a utilization of 0, the response time is just the service time: no job has to wait in a queue. As utilization increases, the response time of the queue grows gradually. Only when the utilization approaches 1 does the response time climb sharply toward infinity.

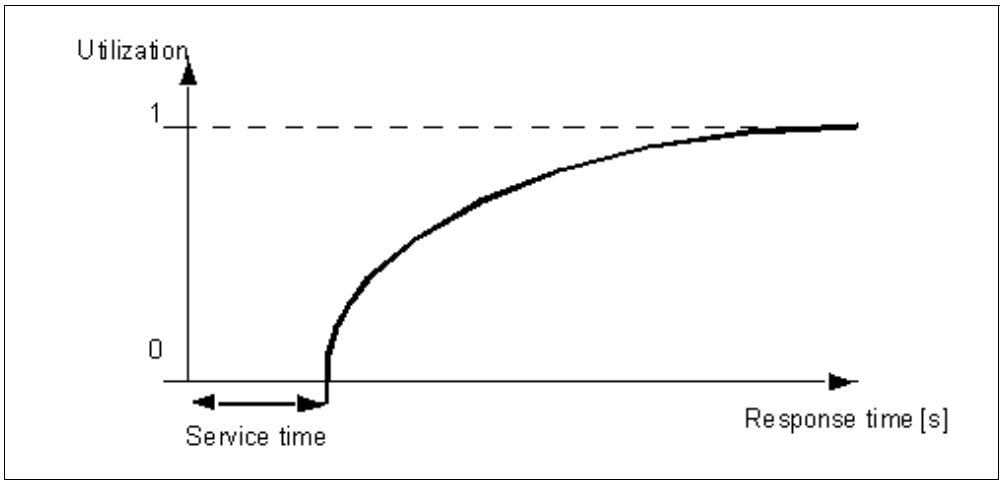


Figure 5. M/M/1 Queue - Response Time

2.2.5.3 Little's Law

Little's Law (N = AT) states that the average number of jobs waiting in the queue (N) is equal to the product of the average arrival rate and the average response time. Little's Law is surprisingly general and applies to all queuing systems that are both stable and conservative (no work is lost when switching between jobs). Little's Law is especially useful when applied to queuing networks. Typically, a single queue is insufficient for modeling a complex system, such as a Web server. In many such cases, a system can be modeled as a graph or network in which each queue represents one node. Such queuing networks are called open if new jobs arrive from outside the network and eventually depart from the network.

Chapter 3. How to Find a Performance Problem

This chapter offers hints on how to find and resolve the performance problems of your application. At first, you have to locate the source of the problem. After this, you find out why it is a problem, and look for ways to solve the problem. We provide a step by step approach to show you how to resolve it. By following this process, you can make sure that key parameters are checked.

3.1 Step-by-Step Approach

There are some things that you want to do when encountering performance problems. We provide you with step by step instruction to find and resolve your problem. First, you should concentrate on the environment your application runs in. Do not look for performance problems within your application unless you have verified the environment configuration. If you do, you may find yourself fighting problems that would not have occurred otherwise. The first four steps guide you through verifying your application environment:

1. Check the availability of resources:

   Your application needs several resources to function properly. It needs enough CPU power, a certain amount of memory, and enough disk space. There are some specific points that should be monitored over a longer time period:

   - The CPU utilization should be below 75%.
   - There should always be a considerable amount of free physical main memory left. For Microsoft Windows NT 16 MB, free memory is a good start.
   - There should be almost no paging activity. JVMs are greatly effected by paging due to garbage collection. It can happen that pages are loaded in the memory for garbage collection, which puts stress on all parts of the system.

   Refer to Chapter 6, “Hardware and Software Configuration” on page 91 for any hints on the actual configuration and a description of the appropriate tools. Additional information about tools for monitoring these activities can be found in Chapter 4, “Tools for Performance Analysis” on page 31.

   If you encounter a problem here, try to upgrade your machine. Do not forget to check all involved servers and clients. A problem on only one of several servers may be the reason for all your difficulties.

   Remember that, even if you monitor over a longer time period, there is still a good chance for some utilization peaks. Keep this in mind while planning the configuration of your machines. It is always useful to have some free resources left.

2. Check the configuration of your operating system and JVM:

   Some settings of your operating system or JVM inflict the performance of the application. Check if you have configured the key parameters properly. For example, a common mistake is to have the wrong settings for the heap size. These settings need careful adjustment. For example, normally you should have at least 25% of free heap space. For more information, refer to Chapter 6, “Hardware and Software Configuration” on page 91.

3. Check the configuration of your SanFrancisco installation:

   The installation and configuration of your Logical SanFrancisco Network (LSFN) has some influence on the application. Refer to Chapter 7, “LSFN Configuration” on page 107 for more information.

4. Check the configuration of your database:

   Every database has some settings that deal with different needs of applications. Some of these settings inflict the performance of your application. Refer to Chapter 8, “Object Persistence, Databases, and Schema Mapping” on page 123 for hints.

   After you have successfully completed the first four steps, your application environment should be in rather good shape. With this configuration, you can be sure that it is possible to have a performing application. If your application still does not show the expected performance, it is time now to look for bottlenecks or problems within your application.

5. Profile your application:

   For various reasons, an application normally spends a major amount of time in only a small part of the code for various reasons. Use a profiling tool to find out where the "hot spots" of your application are. A more detailed explanation on what to look for is found in 3.2, “Profiling the Application with JProbe Profiler” on page 25. There are some possible reasons to spend a considerable amount of time within one method:

   a. High workload:

      If this part of the code has to do much work or is executed multiple times, it is always useful to look for the best performing implementation. Refer to Chapter 9, “Java Coding Tips” on page 147 and Chapter 10, “SanFrancisco Coding Tips” on page 163 for some hints on performance optimized coding.

   b. Bad implementation:

      See above for hints on performance coding.

   c. Locking problems:

      Sometimes your applications waits a considerable amount of time for some resources. If these resources are locked by any other process, check for possible workarounds. See 10.2.2, “AccessMode and Locking” on page 179 for additional information on locking.

   To find out more about profiling an application, refer to 6.3.1, “First Steps” on page 98 and Chapter 4, “Tools for Performance Analysis” on page 31.

6. Check if you have coded correctly:

   If you find possible bottlenecks of your application, refer to Chapter 9, “Java Coding Tips” on page 147 and Chapter 10, “SanFrancisco Coding Tips” on page 163 to check if you already used the best performing implementation.

If steps 5 through 7 do not lead you to the application performance that you want to have, it is the time to think about the design of your application. This may lead you to re-implement a part of your application. Sometimes you have gone the wrong way in designing your application. There are many things that can go wrong, for example, object distribution or data traffic. For detailed information about these points, refer to Chapter 7, “LSFN Configuration” on page 107 and 5.2, “Command Pattern” on page 67.

7. Check the usage of design patterns:

   Some problems are best solved by the usage of certain design patterns that also have a performance impact. More about patterns and their impact on your application can be found in Chapter 5, “Performance Aspects Using Design Patterns” on page 67.

8. Check the overall design of your application:

   The overall design of your application is beyond the scope of this book. This also includes the object design and your application architecture. There are several things that can go wrong with an application. For these problem domains, you should consult books about Object Orientated Analysis and Design to find the possible bottlenecks of your application.

9. Re-evaluate your performance requirements

   After you have checked all previous points and still do not achieve a sufficient performance, check if your wishes can be reached with the available hardware and software. It may be the case that your application requires a completely different approach, which also is beyond the scope of this book.

3.2 Profiling the Application with JProbe Profiler

To find the actual bottlenecks of your application, there is no way around profiling. Do not try to guess where these bottlenecks are. Only measurement reveals the exact location of these problems. One way of measuring, the most common one, is profiling the application at run-time. Specific information about performance measuring tools, such as profiling tools, is found in Chapter 4, “Tools for Performance Analysis” on page 31. After profiling the application, there are several topics that are worth looking at. In this section, we use one of the profiling tools that are explained. This is the JProbe Profiler from KL Group.

This section describes these topics and give some examples how to find these problems.

3.2.1 Checking for Streaming

To check if you do a large amount of serialization in your application, order the profiling results by Cumulative Time. You can do this by clicking on the heading of the column. More information about the usage of the tool are in the already mentioned section.

To find the amount of streaming in your application, you look for *readObject* and *writeObject* method calls. An example for this is provided in Figure 6 on page 26. Having too much streaming indicates that you transfer too much data between client and server.

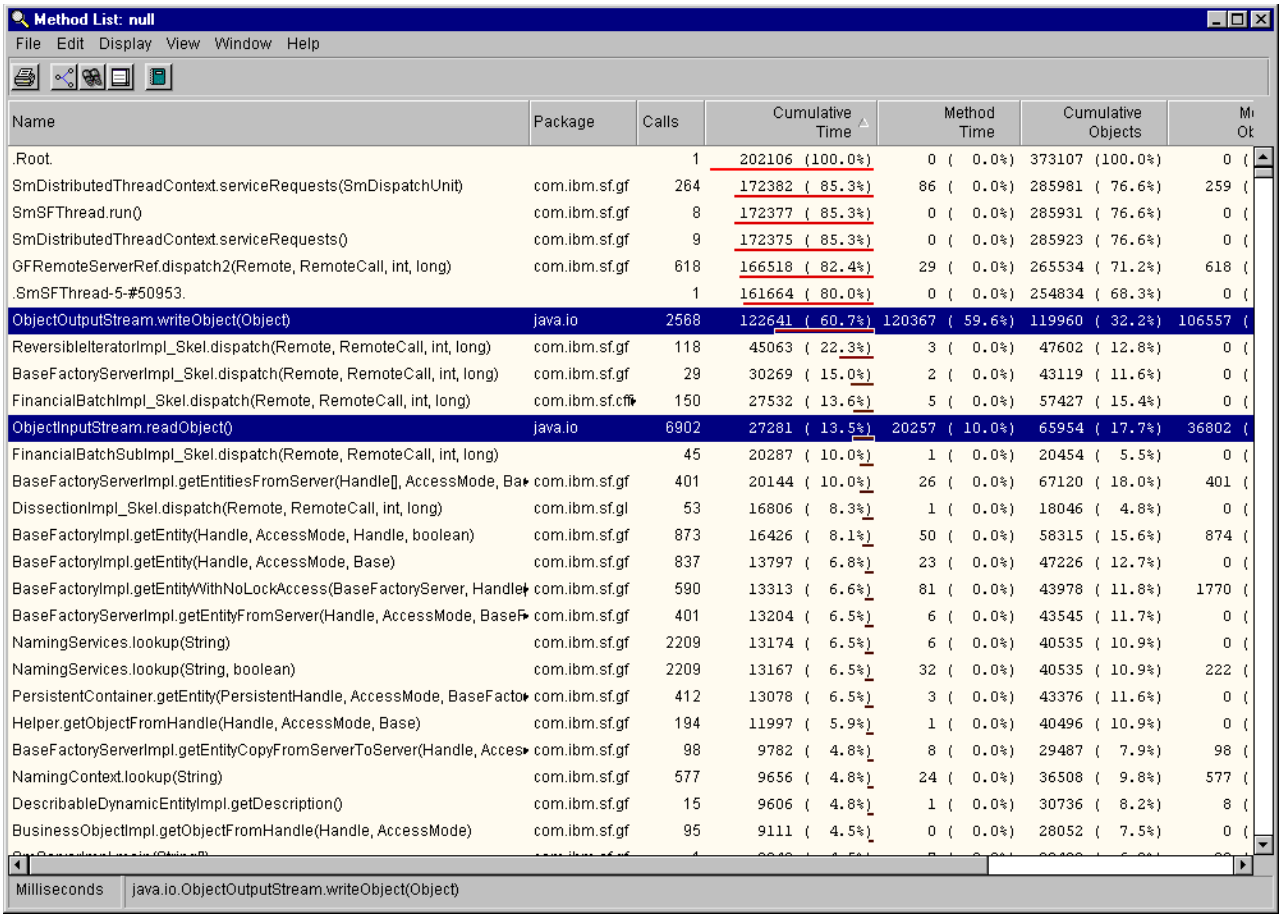


Figure 6. Serialization in the Profile

To reduce the amount of streaming, and thus the amount of transferred data, you should run more code where the BusinessObjects you are working with actually reside or minimize the data transfer by being more selective of what data is streamed. You can do this by using the command pattern that is described in 5.2, “Command Pattern” on page 67. This allows you to send a single command to a BusinessObjectServer and perform all the work there. If you need to display results of the work being performed, you should chose to have your command return the minimum amount of data possible. For example, instead of returning a BusinessObject or Entity, return a subset of its state as Strings and maybe a handle to the Entity. By doing so, you not only decrease the amount of transferred data, you also decrease the number of remote method calls.

3.2.2 Remote Method Calls

A common performance leak is a large amount of remote method calls. A clear indication for this is a large number of calls to Stubs and Skeletons. These can be identified by their name, which always includes the extension *_Stub* or *_Skel*. An example for several Stubs is shown in Figure 7 on page 27. Remote method calls are necessary, but checking if all of them are needed is still a good idea and needs to be done. If possible, you should bundle multiple remote method calls in a command.

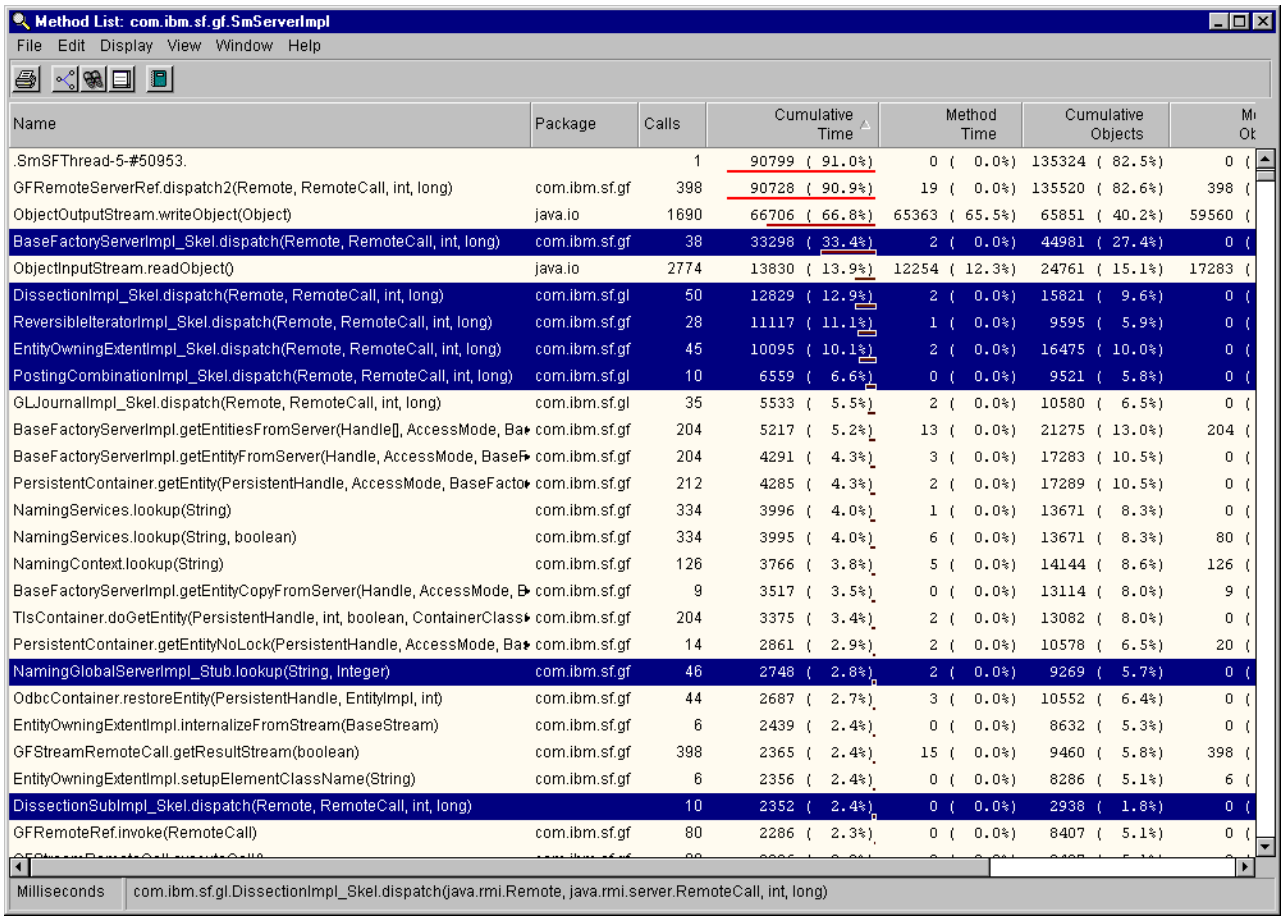


Figure 7. Skeletons and Stubs

To show the impact of a call to a skeleton, Figure 8 on page 28 provides a more detailed view of what is done implicitly by calling methods of a skeleton. You see that most of the time is actually spent in the *writeObject* method, and thus, with streaming. In this example, which is taken from the server side, you find the most time spent with writeObject. There is a simple matrix that tells you the meaning of these method calls, which is shown in Table 1.

Table 1. Methods for Streaming

	readObject	**writeObject**
Server	Parameters	Results
Client	Results	Parameters

By figuring out the time spent in these methods, you can check which amount of data is transferred. If the amount of time is high, for example 10,000 ms, you have to check your method calls for improvement. Remember, the actual scale of this window is mentioned in the bottom left corner of the windows. In Figure 8 on page 28, this is milliseconds.

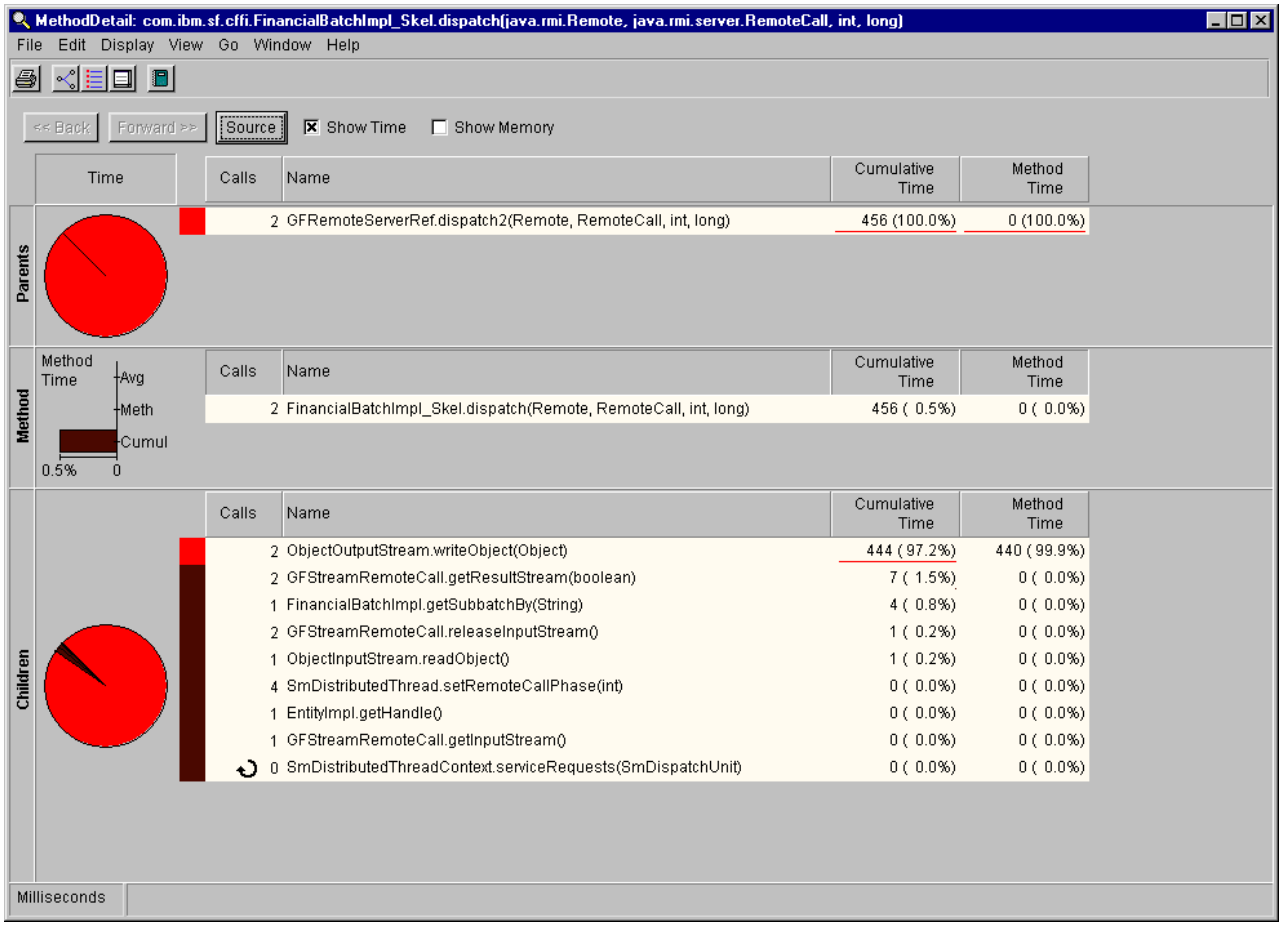


Figure 8. The Impact of Stubs and Skeletons

3.2.3 Creating Objects and Garbage Collection

Each creation of an Object takes time. Also, the destruction of the object by the garbage collector takes additional time. To find out if you create too many objects, instead of reusing them, look for *create* and *copy* methods of the BaseFactoryImpl class. Remember that only transient objects are garbage collected in a SanFrancisco environment. Figure 9 on page 29 shows some of the possible involved methods. Methods can be easily found if you change the sorting order of the screen by selecting the Name column.

The monitoring of how many objects are created and deleted is best done during run-time by looking at how many objects are in the memory at any given time. If the number of objects created is fluctuating a lot, then you should probably consider to reuse the objects you create instead of creating new ones and letting the garbage collector clear the unused ones. This monitoring can be done with either the JProbe Profiler or with OptimizeIt from Intuitive Systems, both of which are explained in Chapter 4, “Tools for Performance Analysis” on page 31. One easy way is to compare the number of objects at a point of time with the number of objects after garbage collection. Most profilers allow you to force a garbage collection. If you find a large difference between both states, you know that a lot of temporary or transient objects are created that live only for a limited time. For more information, refer to 9.3.2, “Reusing Objects” on page 152.

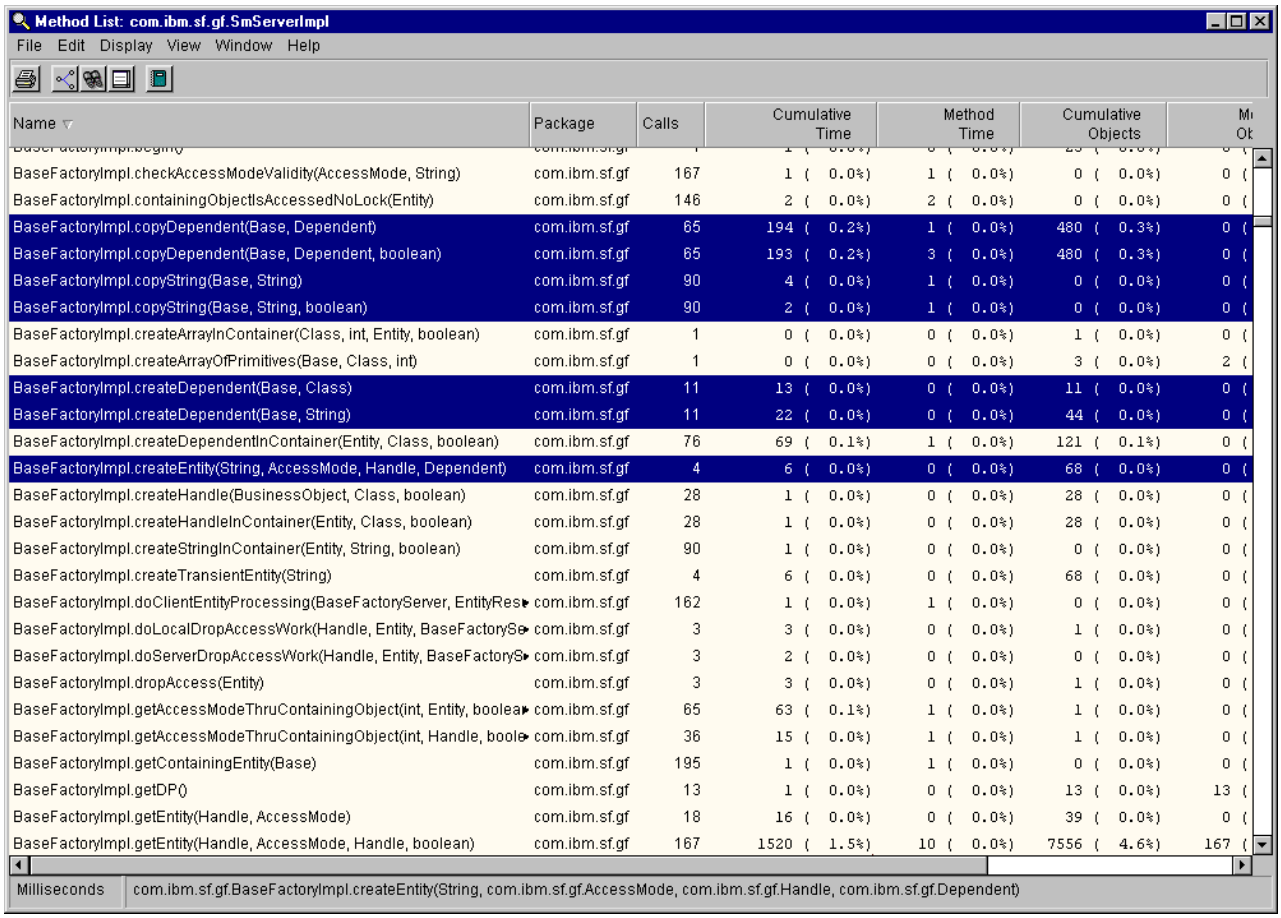


Figure 9. Copy and Create Objects

3.3 Using Strings and StringBuffers

Look for methods that handle Strings or StringBuffers. To start, look for *toString* methods and the *append* method of StringBuffer. Quite often Strings are overused, and too much time is spent in these classes. For additional information about this, refer to 9.4.1, “String Operations” on page 154. Figure 10 on page 30 provides an example about this.

If you find a large number of calls to these methods, check the following items:

- Reduce the number of Strings by replacing them with StringBuffer.
- Reduce the number of appends by trying to append larger parts.
- Check if all appearances of String or StringBuffer are really necessary.

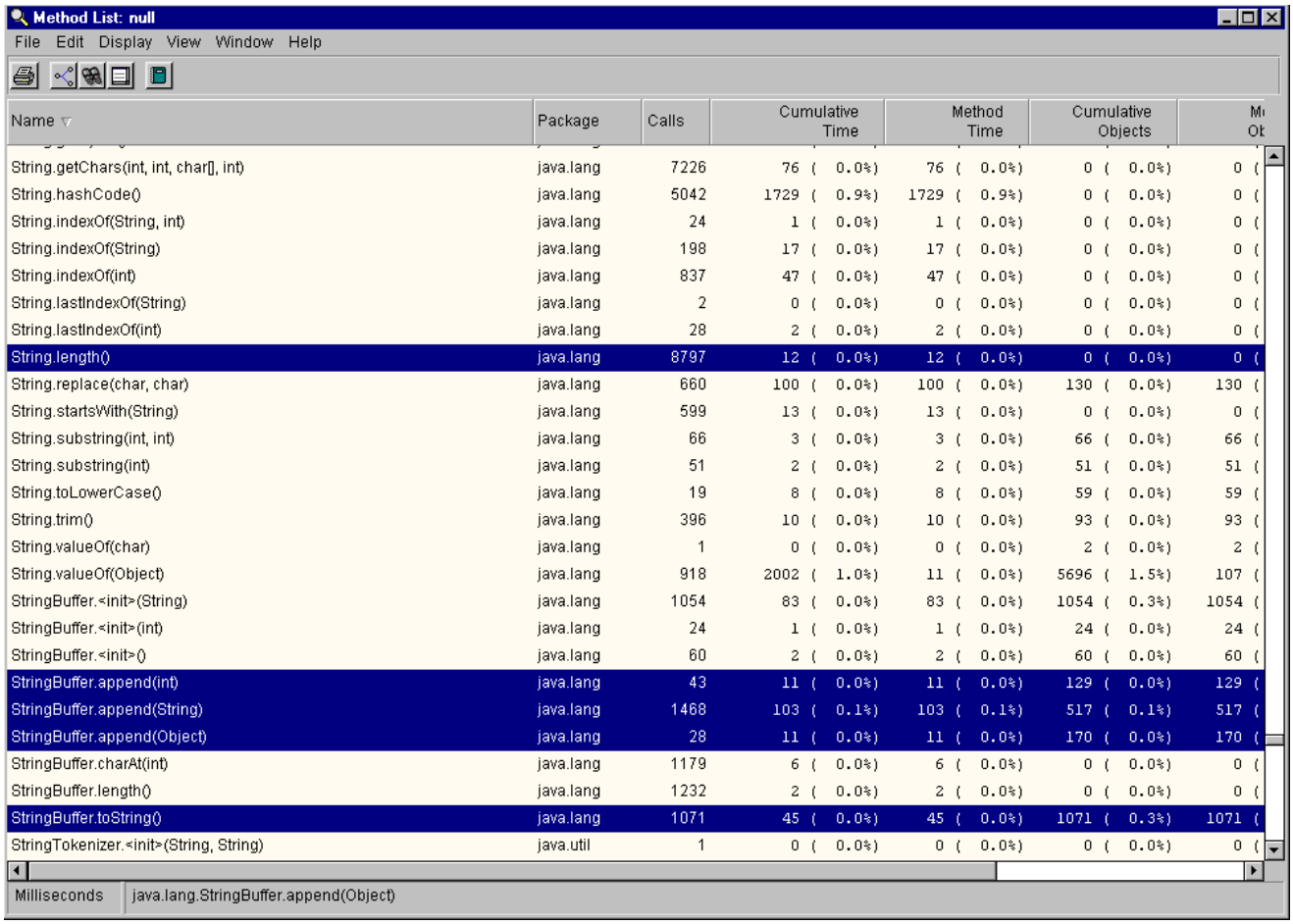


Figure 10. Using String and StringBuffer

Chapter 4. Tools for Performance Analysis

This chapter deals with aspects of how to find out how much time is spent in different parts of your application. An overview of this can be seen as a pie chart with different parts or slices as shown in Figure 11. There are, of course, different sizes on the different slices from one application to another, and perhaps also the slices differ. Slices of this pie are CPU-usage, communication time, database processing, and waiting time.

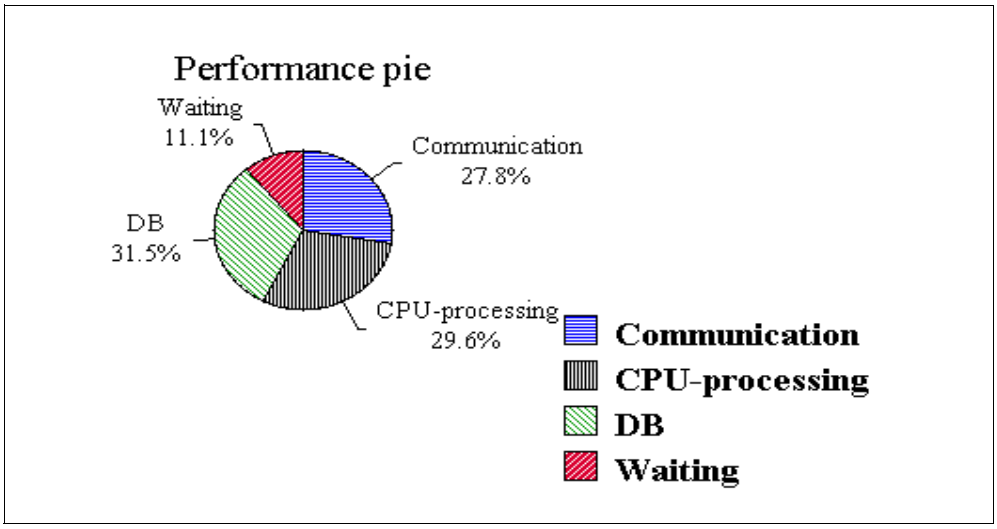


Figure 11. Performance Pie

When you have an actual performance problem, it is vital to find where you spend the time to find out where you can make the biggest gains. Most often, the biggest gains can, of course, be made in the most time consuming parts. Therefore, you start out in that part and split it up into smaller parts within the problem area. After this, you analyze them to find out which parts that are actually tunable.

To find the different slices of the pie and split them up into even smaller parts, you can use different kinds of tools that provide you with different amounts of information.

This chapter contains the following sections:

- Section 4.1, “Which Tool to Use”—We divide the pie into different problem areas, and then, give our recommendations on which tool, or tools, to use to find out how much time actually spent in each area.
- Section 4.2, “About Timing Methods”—We talk about different timing methods you can use when you are trying to find out how much time is spent and the pros and cons of using these methods.
- Section 4.3, “OptimizeIt”—We give you an overview on how to use the OptimizeIt tool from Intuitive Systems.
- Section 4.4, “JProbe”—We give you an overview on how to use the JProbe Profiler tool from the KL Group.

- Section 4.5, “Windows NT Performance Monitor”—We give you a short overview on how to use the Microsoft Windows NT Performance Monitor that comes with Windows NT.
- Section 4.6, “AS/400 Performance Tools”—We give you an overview on how to use the Performance Data Collector/Performance Monitor that is available on the IBM AS/400 System.
- Section 4.7, “The Container Cache Statistics Tool”—We give you an overview on how to use the Container Cache Statistics Tool that comes with SanFrancisco.
- Section 4.8, “Lock Analysis Tools”—We give you an overview on how to use two different Lock Analysis tools that come with SanFrancisco: the Lock Conflict Trace Analysis Tool and the Lock Contention Console Tool.

4.1 Which Tool to Use

This section gives you an overview on when to use various tools and what they can provide.

To get an overall picture of how much time is spent on the different slices, you should preferably use a more overall system performance analyzing tool. Of the tools that we discuss in this chapter, there is one that is for overall system analysis. This is the Microsoft Windows NT Performance Monitor. For example, when monitoring the processes, the NT Performance Monitor can provide you with information on how much CPU-time the database uses at different stages of your application. Other things that the NT Performance Monitor has information on are memory overcommitment, I/O bottlenecks, too large/too small heap sizes, and so on.

When digging deeper into the different slices, it is better to use tools that give you a more detailed picture of what is really happening inside the different processes.

The first two tools that we explain are OptimizeIt, from Intuitive System, and JProbe Profiler, from KL Group. Both of these tools are quite low-level performance analysis tools that should be used if you want to understand and solve performance issues in Java programs. These tools show the CPU-time spent by a certain program and even a certain method in that program. They also show how many instances of an object that are allocated at any given point of time.

These tools are excellent for showing you the most CPU-expensive points in your code. By looking at which methods are taking the most time, you can, if you have knowledge of what is done in the methods, distinguish between how much time is actual CPU-processing, I/O, and communication. For example, when time is spent in the method `SocketInputStream.read()`, you can assume that the time spent in this method actually was spent on communication.

These tools are also good for understanding how much memory your application is using at different points of execution and where it is spent. You can get indications of possible memory leaks and if your program is creating excessive objects. Temporary objects are usually rapid to allocate, but if too many temporary objects are allocated, the Java VM garbage collector will run more often which, in turn, means that, with most available Java VM, any Java program

pauses while the garbage collector is running. This means that the application will run slower, that is, the performance drops if the garbage collector runs more often than absolutely necessary. There is another element that may impact the performance of your application. That is, if your application uses a lot of memory, there will be less to use for the rest of the system, which may lead to the system starting to swap.

Since each container in SanFrancisco maintains a memory cache where it keeps recently accessed persistent objects so that when needed, it can retrieve them directly from memory rather than from disk. SanFrancisco provides a utility to display cache size and cache hit and miss rates. This is the Container Cache Statistics Tool. It can help you to optimize memory usage, and therefore, lower the number of disk accesses.

We then go through some AS/400 performance tools. These tools are both for overall system performance analysis and for more in depth performance analysis, such as which methods, or even lines of code, are performance hot spots. When it comes to waiting time, this can be caused by events, such as garbage collection, but also locking conflicts.

SanFrancisco provides two different lock analysis tools, the Lock Conflict Trace Analysis Tool and the Lock Contention Console. The *Lock Conflict Trace Analysis Tool* should be used during the application development phase to analyze trace data for lock conflicts. In SanFrancisco, different transactions can have read access to the same Entity at the same time, but only one transaction can have write access to an Entity. And while that write access is held, no other transaction can gain read or write access to the Entity. In this case, the other transactions must wait until the first one has released the lock.

The *Lock Contention Console* should be used at runtime to help isolate deadlock conditions that are due to a lock conflict. The tool captures a snap shot of lock data and provides options to display and analyze the data. The tool is primarily useful during a product test phase when running with multiple clients.

4.2 About Timing Methods

Depending on your particular project, you may want to measure time using different methods. Time could be measured in either elapsed time or CPU time. Most often, you may want to use the CPU time.

4.2.1 Elapsed Time

Elapsed (Wall Clock) time measures, in real-time, how much each method or line takes to complete. Elapsed time is often the most accurate measurement on your machine for this particular run but is affected by many different variables, such as CPU speed, other running applications, available memory, and so on. The data you get on your computer may not necessarily reflect how a program performs on other computers.

Note

Since elapsed time includes program pauses, sleeps, waiting for I/O, and so on, using this timing method may produce unexpected results. For example, if your program is running multiple threads, it can accumulate elapsed time simultaneously in all active threads. So, if 100 threads are running, the total time may be 100 times the expected time.

4.2.2 CPU Time

CPU time measures the number of CPU cycles spent executing code. Unlike elapsed time, CPU time does not take into account program pauses, waiting for I/O, and so on. Since it is not affected by as many variables as elapsed time, CPU time usually gives a better indication of how a program will run on other computers.

4.3 OptimizeIt

OptimizeIt, from Intuitive Systems, is a comprehensive Java profiler that allows developers to understand and solve performance issues in their Java programs. Much of the information in this section has been gathered from the documentation that comes with OptimizeIt, and this is available on the Intuitive Systems Web page.

It runs on both Microsoft Windows NT and Sun Solaris. With its advanced audit system, OptimizeIt delves into the Java virtual machine to provide detailed information about how a Java application, applet, or JavaBean uses memory and CPU resources.

With hot spot detectors and method call graphs, OptimizeIt's CPU and memory profilers make it easy to detect excessive object allocations or time-consuming algorithms. OptimizeIt is plug and play. There is no need to recompile your program with a custom compiler or to modify class files before the execution. simply run your program from OptimizeIt to start testing its performance. Because no code modifications are required, any Java code that your program uses is included in the profile.

OptimizeIt has two main components:

- **OptimizeIt user interface**—A window that displays profiles and controls for refining the profiles and viewing source code.
- **OptimizeIt audit system**—A real-time detective that reports the activity on the Java virtual machine back to the OptimizeIt user interface.

An evaluation copy of OptimizeIt can be downloaded from:
http://www.optimizeit.com

4.3.1 Testing a Java Program

To test a Java program, you need to launch your Java program with OptimizeIt's audit system. OptimizeIt's audit system runs in the tested Java virtual machine and reports profiling information to OptimizeIt.

OptimizeIt's program chooser allows you to launch the Java virtual machine and the audit system together for an applet or an application. The following steps show how to launch a simple applet:

1. From the File menu, select **Choose program...**
2. Click on the **Browse...** button and select

   ```
   c:\jdk1.1.6\demo\awt-1.1\lightweight\Gauge\example.html
   ```

3. Click on the **Start now** button to start the applet.

4.3.2 Testing a SanFrancisco Application

OptimizeIt can launch most applets or applications. However, some applications, such as large tools or Java servers, might require some special Java arguments. To test these applications, it is possible to run them from the command line and then attach from OptimizeIt. This feature allows OptimizeIt to run on a different machine than the tested Java program.

To test SanFrancisco, you have to launch it from the command line. To allow OptimizeIt to profile SanFrancisco, you need to launch it with the OptimizeIt audit utility.

To use the OptimizeIt audit utility, you need to add the following line in your classpath:

```
<install dir>\Intuitive Systems\OptimizeIt\lib\optit.jar
```

You also need the following line in your path:

```
<install dir>\Intuitive Systems\OptimizeIt\lib
```

OptimizeIt's audit utility is a set of Java classes and native code. The following command invokes the utility without any argument to print its options:

```
c:\> java intuitive.audit.Main
Options: [-port <portnumber>] [-dllpath <dir>] [-pause] [-dmp] [-nostdio]
ClassName arg1, arg2,...
```

The elements in this command are explained here:

- **-port**—Specify the port you want to use for the communication link between the audit utility and OptimizeIt application. If not filled in, the default port, 1470, will be used.
- **-dllpath**—Specify where OptimizeIt's dlls are if you don't want to change your path environment variable.
- **-pause**—Causes the launched program to pause immediately after launch.
- **-dmp**—Disables the memory profiler.
- **ClassName**—This is your application main class.

For instance, if you want to launch the Logical SanFrancisco Network server, LSFN server with OptimizeIt, perform these steps:

1. Start OptimizeIt.
2. Start the LSFN server from the command line with:

   ```
   c:\> java -ms6m -mx128m -DserverName=SFGSMProcess intuitive.audit.Main
   com.ibm.sf.gf.SmServerImpl
   ```

3. From the Program menu, in OptimizeIt, select **Attach**. The Attach panel appears (Figure 12).

If you want to launch SFBOProcess1 with OptimizeIt, perform these steps:

1. Ensure that the LSFN server is started.
2. Start OptimizeIt.
3. Start the SFBOProcess1 from the command line with:

```
c:\> java -ms24m -mx96m -noclassgc
-Dsfenv="%SF_BASE%\com\ibm\sf\etc\sfenv.ini" -DserverName=SFBOProcess1
-DGDPMsleepTime=60 intuitive.audit.Main
com.ibm.sf.gf.SmServerImpl -GSM_UID 1
```

4. From the Program menu, in OptimizeIt, select **Attach**. The Attach panel appears (Figure 12).

Note

Replace %SF_BASE% with the directory where you installed SanFrancisco, for example, d:\sf\sf130.

Hint

You may want to create batch files with the above Java commands.

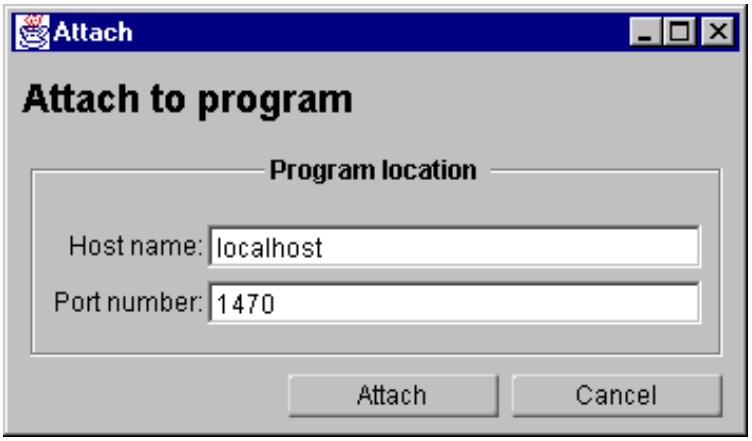


Figure 12. Attaching to a Running Java Program from within OptimizeIt

In the "Host name" input field, type the host name of the computer running your Java program. If it is in the same machine as OptimizeIt, leave it to "localhost". In the "Port number" input field, type the port number you want to use for the communication link between OptimizeIt and the audit utility. Change this value only if you used the -port option while launching your Java program.

4.3.3 Using the Memory Profiler

The memory profiler allows developers to track down object allocations. OptimizeIt displays. In real time, object allocations and can provide a graph showing which line of code is responsible for excessive allocations. You must, however, have access to the source code to view the actual lines in the code.

The first time you use OptimizeIt, it does not know where the source code is. Click on the **Browse...** button on the window bottom to select the correct source code. OptimizeIt also asks you whether or not you want it to remember the source code location. Click the **Yes** button.

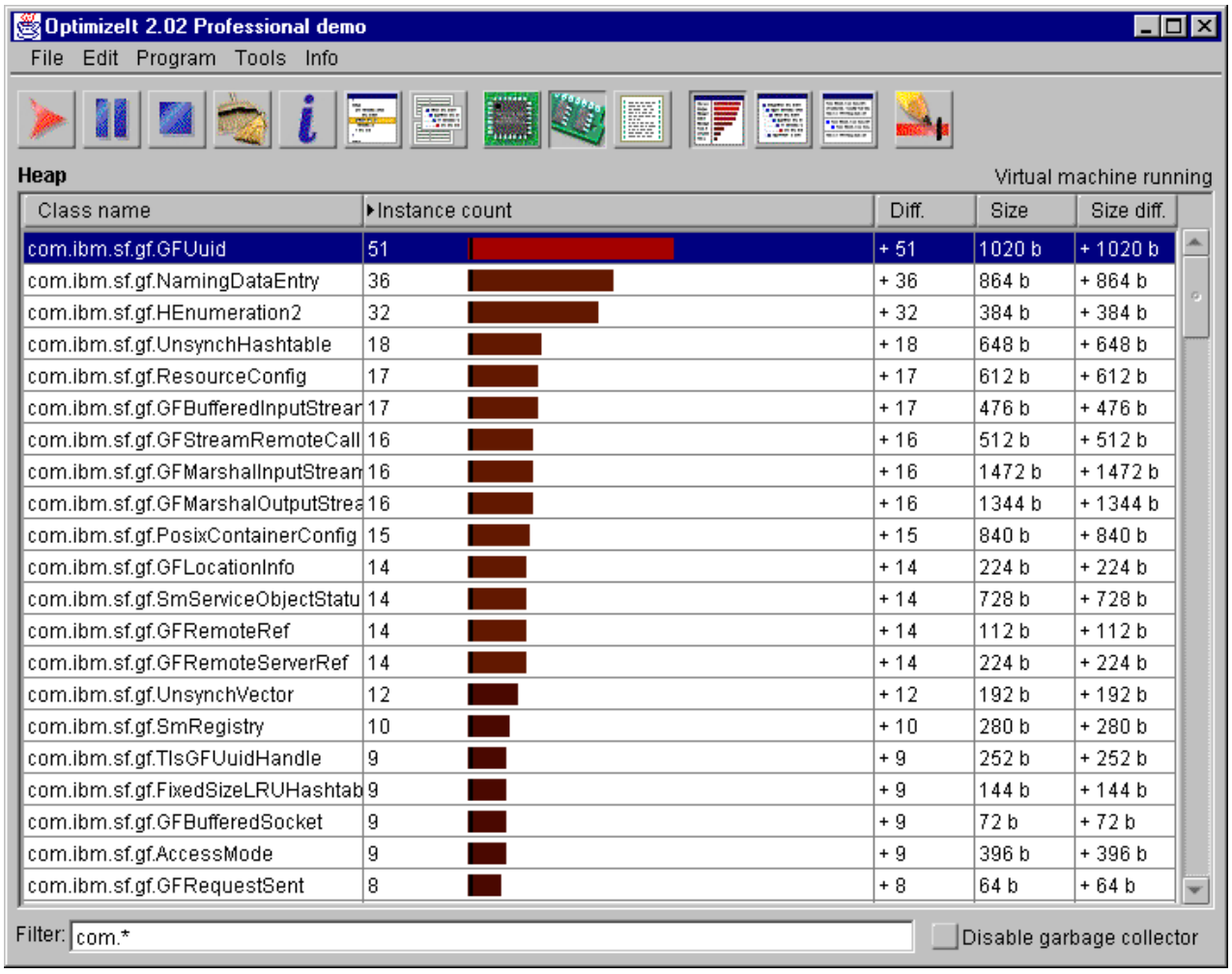


Figure 13. OptimizeIts Memory Profiler

First, you should click on the **instance count column header** to sort classes by number of allocated instances. Now, you always have the classes with the most instances on the top of the table.

The filter on the bottom can be used to display less classes. For example, if you want to test only SanFrancisco classes, type `com.*` in the **Filter field**, and all other classes will disappear. This is what we have done. Figure 13 shows the instances that are allocated after we have started the LSFN server, that is, when the LSFN server is up and running, ready to receive requests.

To measure how many instances are allocated when you run a certain program, such as entering a new customer to your system, you can first push the **broom icon** to run the garbage collector to get rid of nonreferenced objects. Push the **pencil icon** to set a mark at the number of objects you have allocated at this moment in time. You then run your application, and the difference in how many instances you have allocated will be shown in the "Diff" column in the table.

To find out where objects are allocated, you first identify a class with an excessive number of instances. Then, identify the code or the part of the program that is responsible for these allocations. Select the line displaying the class you want to focus on, and click the **Show Allocation Backtraces** button. OptimizeIt switches to Allocation Backtrace mode (Figure 14).

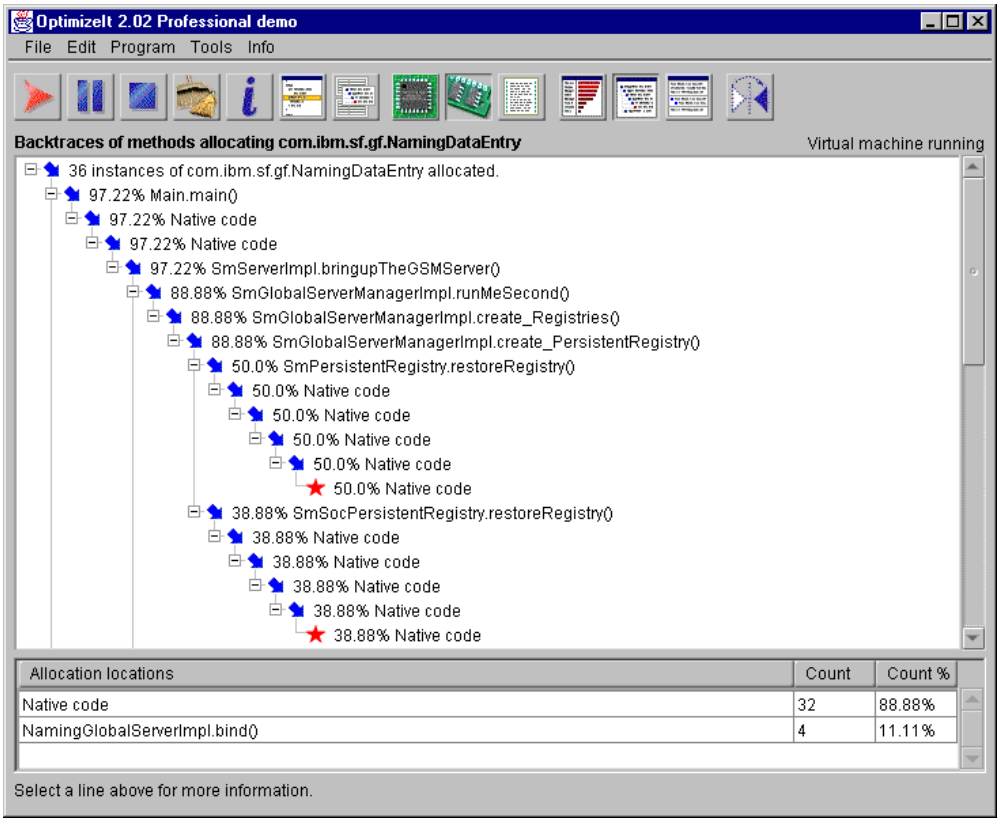


Figure 14. OptimizeIts Allocation Backtrace Mode

The top section in "Allocation Backtrace" mode traces calls from the first method where the allocations occur. By opening nodes in this view, you can see precisely where allocations originate. Any line with a star in front of it is a line that is responsible for one or more object allocations. The bottom section displays the names of methods responsible for object allocations.

By pressing the **Reverse Display** button in the toolbar, you can reverse the list to display backtraces from the place where the allocations take place to the allocated instances of the Java program (Figure 15 on page 39). This view can be useful when you need to focus on methods or lines of code responsible for object allocations rather than broad features of your program.

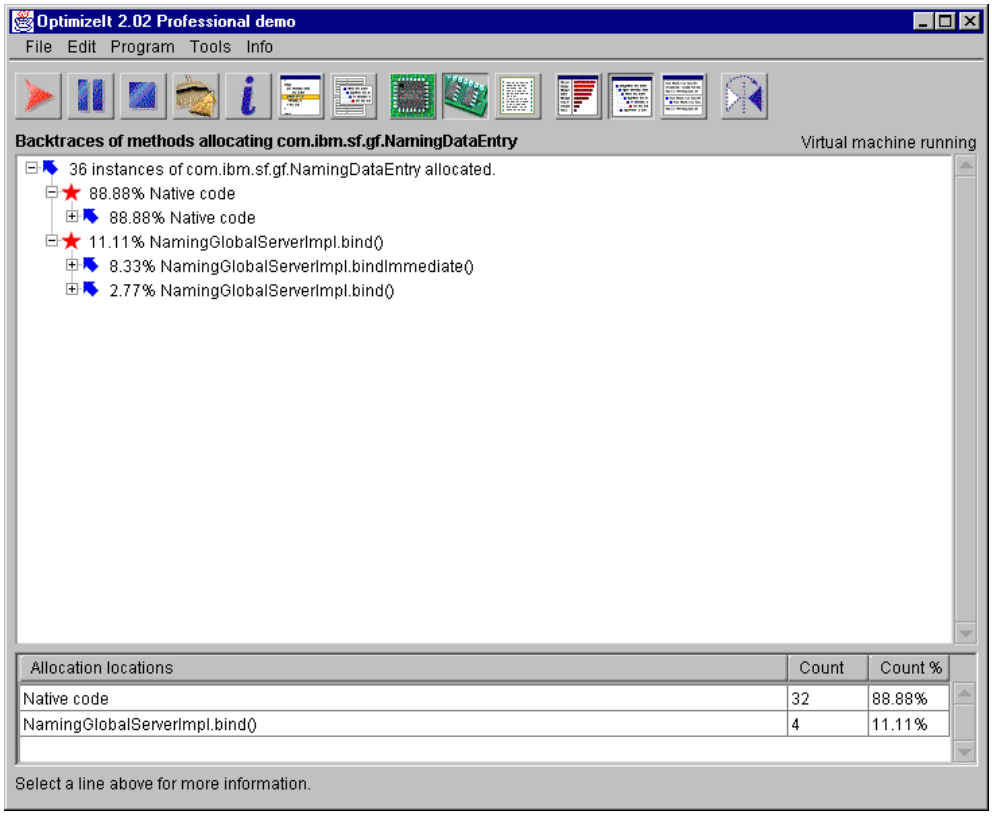


Figure 15. OptimizeIts Allocation Backtrace mode - Reverse Display

Using the memory profiler allows developers to minimize temporary object allocations. Temporary objects are usually rapid to allocate. However, they keep the garbage collector busy. With most available JVMs, any Java program may freeze for several hundred milliseconds when the garbage collector is busy. If too many temporary objects are allocated, the application can appear very slowly to the user because of these interruptions.

OptimizeIt's memory profiler also allows developers to discover how many objects are allocated and stay in the virtual machine. This can be very useful to verify that a document storage, for example, is really garbage collected when a document is closed.

4.3.4 Using the CPU Profiler

While the memory profiler allows developers to understand how to minimize object allocations, the CPU profiler allows developers to understand where the time goes. The CPU profiler can be seen as a recording device inside the Java virtual machine. Developers start the profiler, exercise whatever feature is slow in their application, then stop the CPU profiler. Precise data about where the time goes is then displayed.

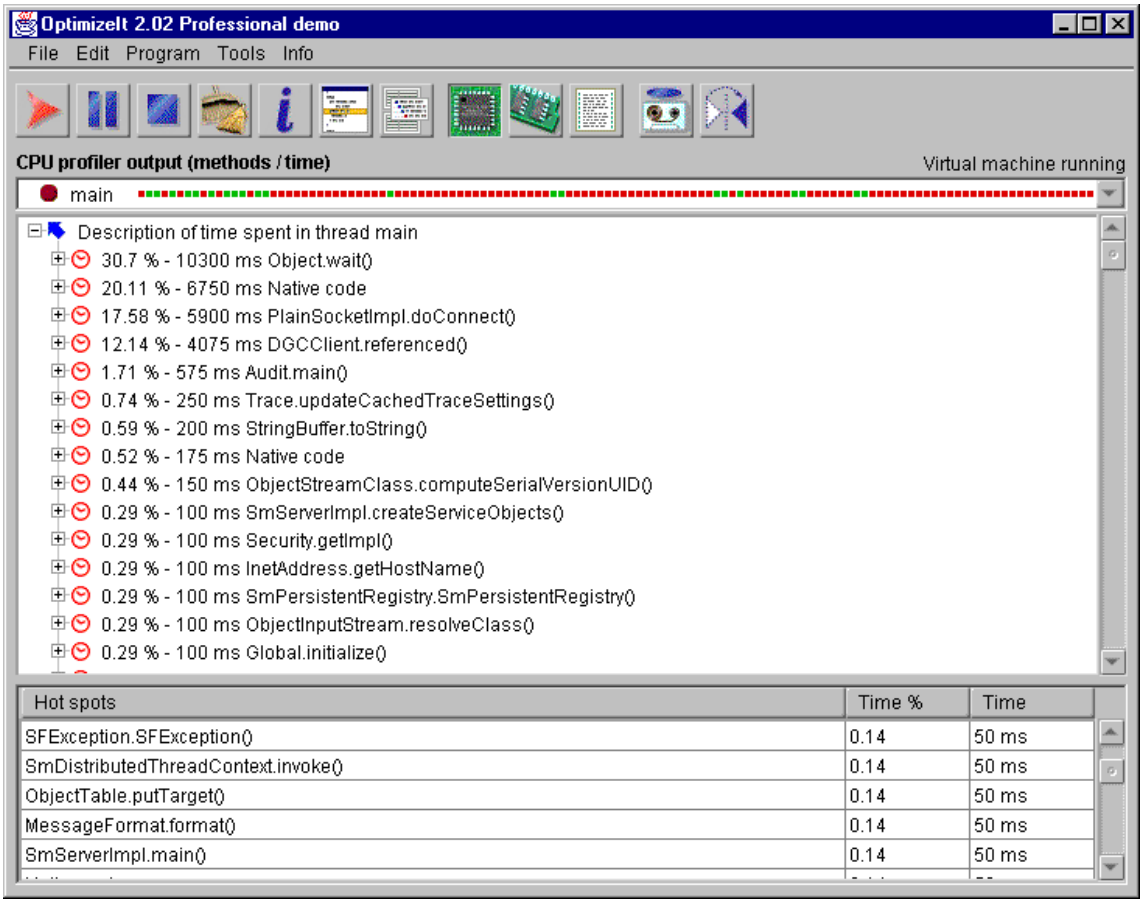


Figure 16. OptimizeIts CPU Profiler

When you want to run the CPU profiler, you must first click on the CPU profiler icon to switch to the CPU profiler. Then, click the start/stop icon, which looks like a cassette tape, to start recording the CPU. Next, run the application that you want to test for a minute or so and do whatever you want to measure (for instance, entering a new customer to your system from a client). When ready, press the **start/stop** icon again to stop recording the CPU. Now the CPU profiler displays some profiling information about the recorded session.

Figure 16 shows the results of where time was spent in the thread main when running the CPU profiler at the start up of the LSFN server. We did this by first starting the LSFN server from the command line, as in 4.3.2, “Testing a SanFrancisco Application” on page 35, but with the **-pause** option set. This started the application but paused it immediately. This gave us the possibility to attach to the application from within OptimizeIt and then starting the CPU profiler before we make the application resume. When the LSFN server was up and running, that is, when it was ready to respond to client requests, we stopped the CPU profiler in order to get the results that showed us where the CPU time was spent during start up.

You can, by clicking the **combo** box under the title "CPU profiler output," look at all the threads that where running during the session (Figure 17 on page 42), and from these threads, choose one thread or a thread group, that you want to examine more closely by clicking on it. Selecting a thread group shows how the

time was spent for all threads and thread groups belonging to the thread group. When you have chosen a thread, a description of the time spent in that thread are displayed in the CPU profiler.

In Figure 16 on page 40, the description of the time that was spent in thread `main` is displayed. Here we can see that 30.7% of the time spent in this thread was spent in the `Object.wait()` method. As you can see all of the methods displayed have a clock- like icon in front of them. This means that some time actually was consumed by this line. If, however, a method should have an arrow pointing downwards in front of it, this would mean that the method immediately calls a sub-method.

In the bottom of the CPU profiler, the hot spots of the displayed thread are shown. These hot spots are the methods in which most of the time is spent. If you double click on one of the methods in this part of the window, you get the actual code of that method displayed.

In the combo box with all the different threads, you can see that each thread has its execution time displayed as a dotted line. This line has different colored dots:

- Green dots mean that the thread was using the CPU.
- Red dots mean that the thread was waiting on a condition.
- Gray dots mean that the thread did not exist at the sampling time.

You can easily see which threads have been active and which have not. Since we only use monochrome pictures in this book, it may be difficult to see which color a dot has.

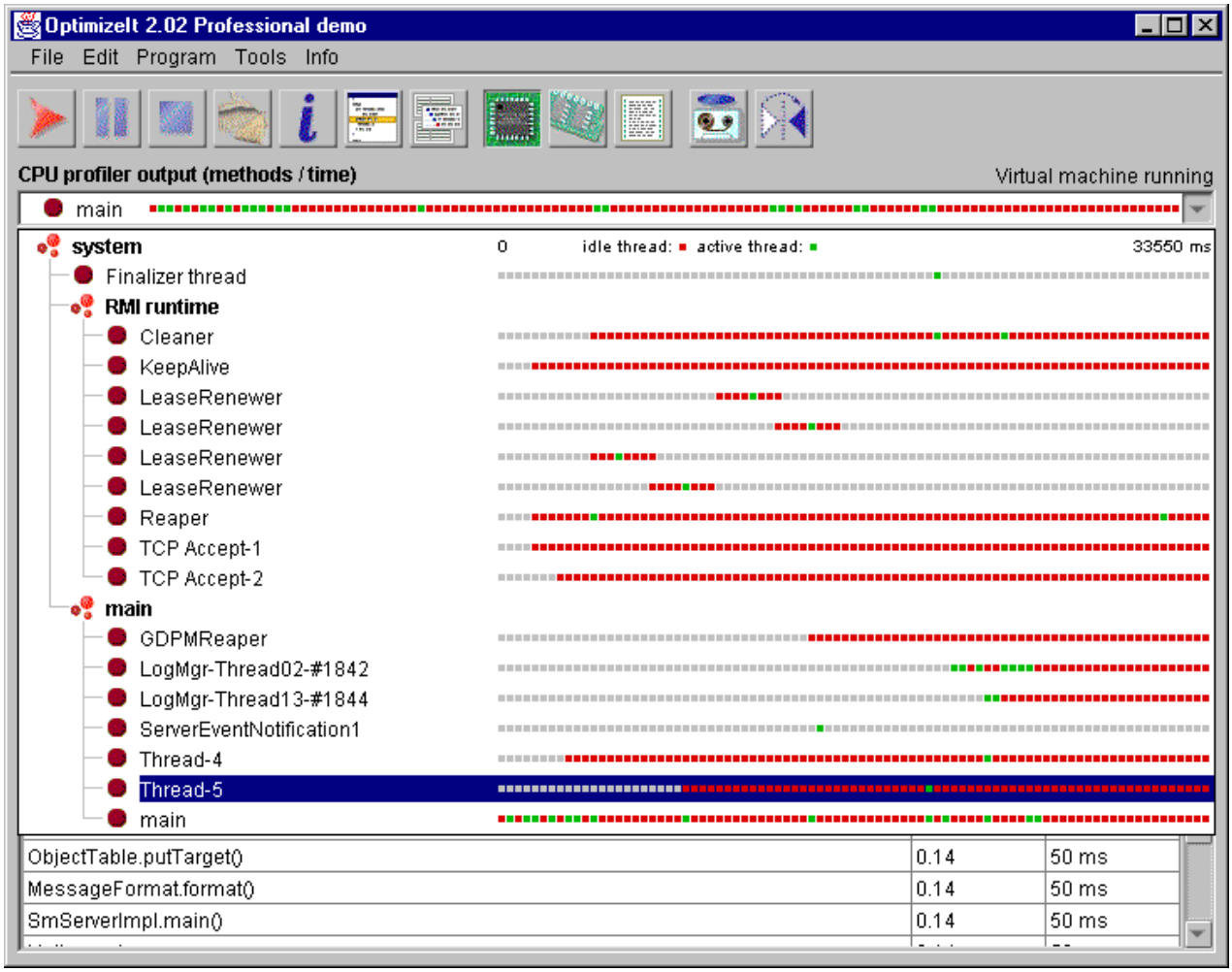


Figure 17. OptimizeIts Thread viewer

The CPU profiler records where the time is going. The time spent by the program waiting for a condition (I/O, monitor, and so on) is included in the data. This is useful to understand, for example, if a cache is necessary for a resource. In the source code, lines highlighted in yellow are the lines of code that were noticed by the profiler during the recording session.

4.3.5 Summary on OptimizeIt

OptimizeIt is a simple tool to use, and it gives you a great view over the entire application. OptimizeIt Version 2.01, however, has some features that may not be desirable.

One of the problems we found was the fact that OptimizeIt measures the CPU-time spent in elapsed time instead of CPU cycles. As discussed in 4.2, “About Timing Methods” on page 33, elapsed time includes program pauses, sleeps, waiting for I/O, and so on. Using this timing method may produce unexpected results. It would have been better if we, as users of OptimizeIt, could choose which timing method to use.

OptimizeIt also shows String objects, but not the objects underlying data held in a char[]. This gives the impression that there is no char[] allocation for each String

allocation that is made. Also, OptimizeIt does not have any good way to save and restore profiles.

Despite these problems, OptimizeIt is excellent for analyzing a Java application in the case of bottlenecks. It is a simple tool to use with you can quickly become familiar.

4.4 JProbe

JProbe Profiler, from KL Group, is a tool that helps Java programmers identify performance bottlenecks. Much of the information in this section was gathered from the documentation that comes with JProbe.

JProbe collects both timing and memory data. As your program executes, memory and object creation data is collected and displayed in the Memory Usage and Instance Summary windows. Timing information is collected either by line or by method and is displayed in the Call Graph using one of nine available metrics. Information can be viewed while the program is running, or it can be saved as a profile for later analysis and playback.

JProbe stores performance data in snapshots, which is a picture of the amount of memory and time your application is using at the time the snapshot is taken. To create a snapshot, run your program. At the time you want a snapshot, click the snapshot button. You can take multiple snapshots while running an application to compare different stages in the execution of a program. These are also useful to look at later or for comparison purposes.

JProbe uses five different windows to display data about an application. Each window displays data differently, and they can be used together to locate trouble areas in the application faster. The windows that can be used are:

- The *Memory Usage window* graphically displays how much memory your application has allocated at any given point of time. It also displays how much memory is available within the JVM.
- The *Instance Summary* and *Instance Detail windows* track real time instance creation. You can use the Instance Detail window to view information on instance creation of a particular class. The Instance Summary window can be used to force garbage collection.
- The *Call Graph window* displays the calling relationships between the methods. You can choose to show only the most expensive methods, calculated according to one of nine different metrics. You can easily locate the methods that are taking the most time or creating the most objects.
- The *Method Detail window* shows the percentage of memory and time allocated to a particular method's callers and descendants. This can be used to track down the program's most expensive methods.
- The *Source window* shows how much time and how many objects your code is using and does so on a line-by-line basis. Each method's most expensive line is highlighted to clarify where the problems might be located.

Further information about how to get started with JProbe Profiler and how each of the different windows are used is available in "Getting Started with JProbe Profiler," which can be downloaded from the same place as an evaluation copy of JProbe Profiler, at: http://www.klg.com

4.4.1 Testing a Java Program

When you decide to test an application or applet using JProbe, the first thing to do is to start JProbe. When running JProbe, you choose to run a program either by selecting **Run** at the first dialog that welcomes you to using JProbe or by selecting **Run** under the Program menu in the JProbe Profiler Console or by clicking the **Right Arrow** button in the JProbe Profiler Console.

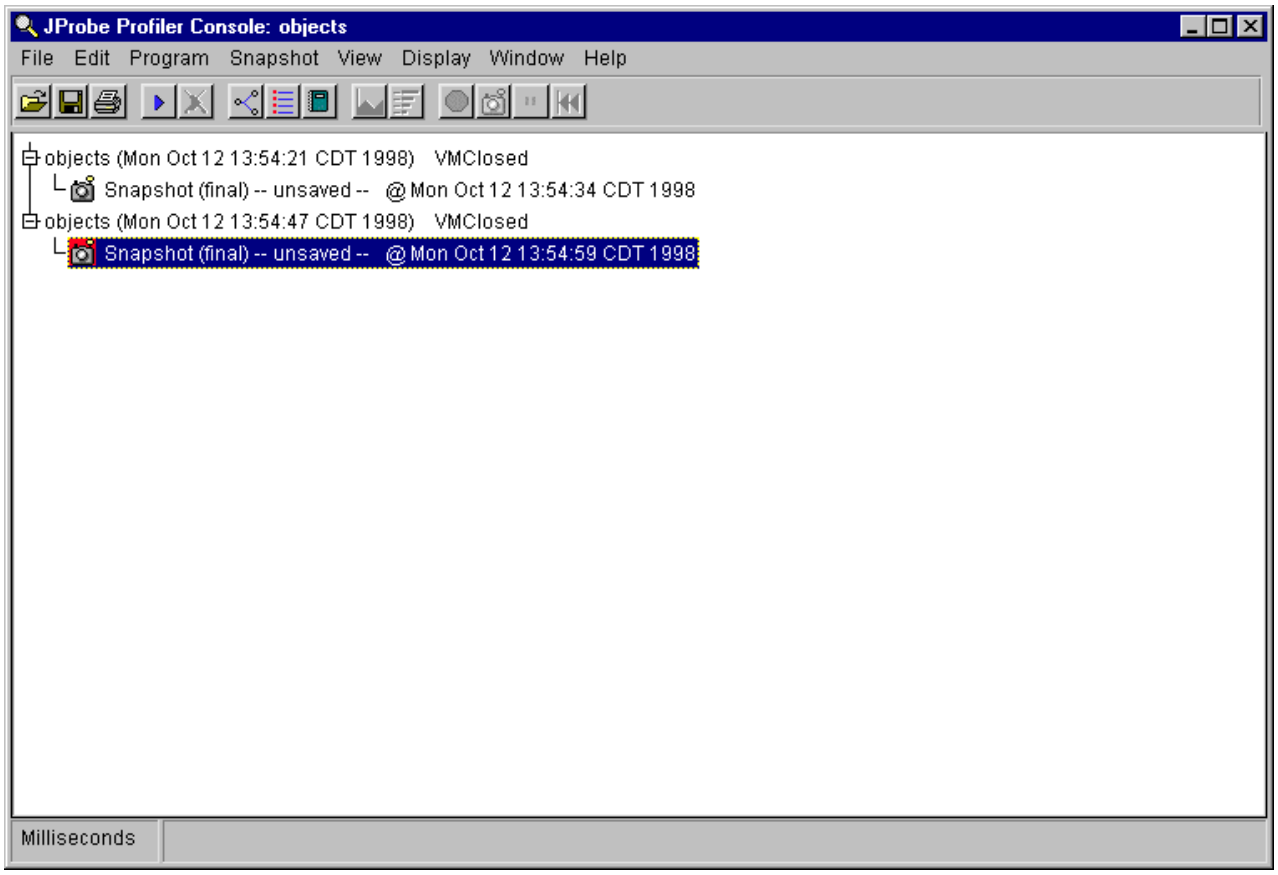


Figure 18. JProbe Profiler Console

When you have chosen to run an application, the Run settings dialog appears (Figure 19 on page 45). This dialog is fairly straight forward to fill with the necessary data in order for JProbe to be able to run the application. You have to specify whether it is an application or an applet you are about to run. The class file has to be supplied, if you choose to browse to the right file, then Working Directory and Source File Path will be automatically updated by JProbe. If the application takes any arguments, these must, of course, be filled in, and also if you want to run the application with any Java arguments, these must be provided (though not all Java arguments are currently supported).

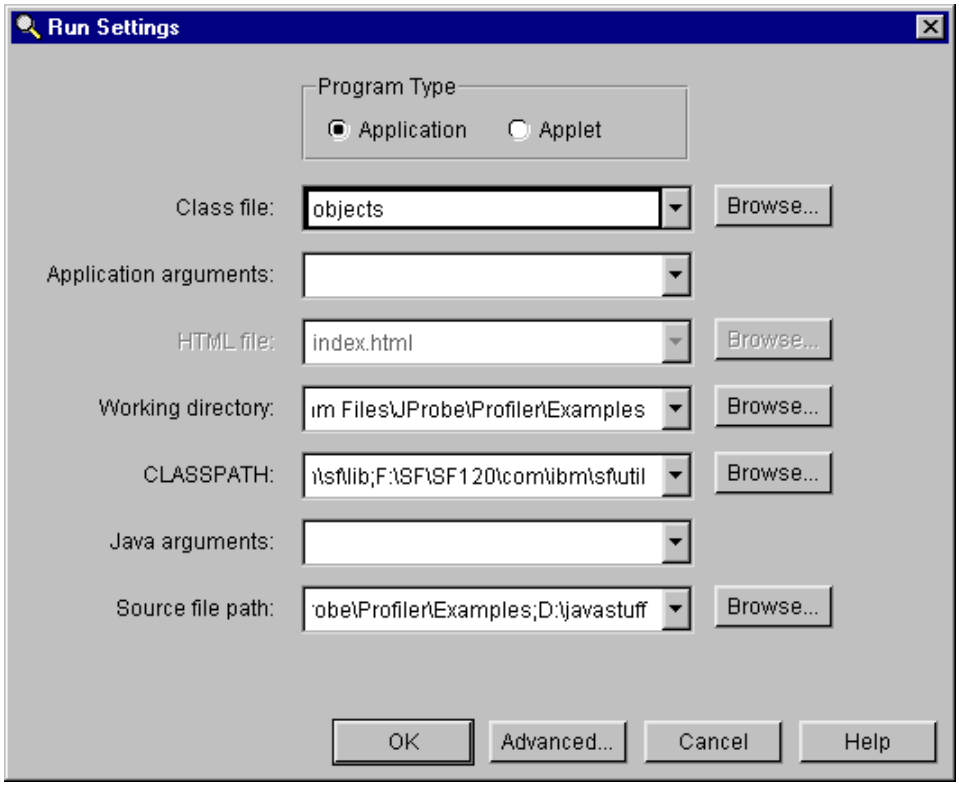


Figure 19. JProbe Run Settings

If you click the **Advanced...** button in the Run Settings dialog, the Advanced Run Settings dialog appears.

Use the Advanced Run Settings dialog to change JProbe's default run settings. The Measurement tab (Figure 20) specifies how JProbe times your Java application and the degree of granularity. If you change any of these settings, they are kept the next time you run an application.

If you choose **Line**, performance data is collected with a source line as the unit of granularity. If you choose **Method**, performance data is gathered with a source method as the unit of granularity. Using Line granularity provides more detail than Method, but requires line information to be in the class file, which means that your application must be compiled with the `-g` option.

The "How Time is Measured" section shows how JProbe is currently set to time your application, either by elapsed time, CPU time, or simulated time.

Note

Since CPU time data is collected by making a Windows NT-specific call, this method is not available on Windows 95 machines. When using CPU time, you can only collect method-granularity profile data.

When testing an application, we recommend that you use CPU time. As stated previously, it gives a better indication of how your application will run on other computers.

Note

JProbe adds an additional time measuring method to the two mentioned earlier. The new one is called Simulated time. Unlike elapsed time and CPU time, simulated time does not give timing information as such, but a relative comparison between methods in your program. Simulation is a method of measuring an application, where each instruction is assigned a specific time, and the times assigned to each line are added together instead of actually being timed.

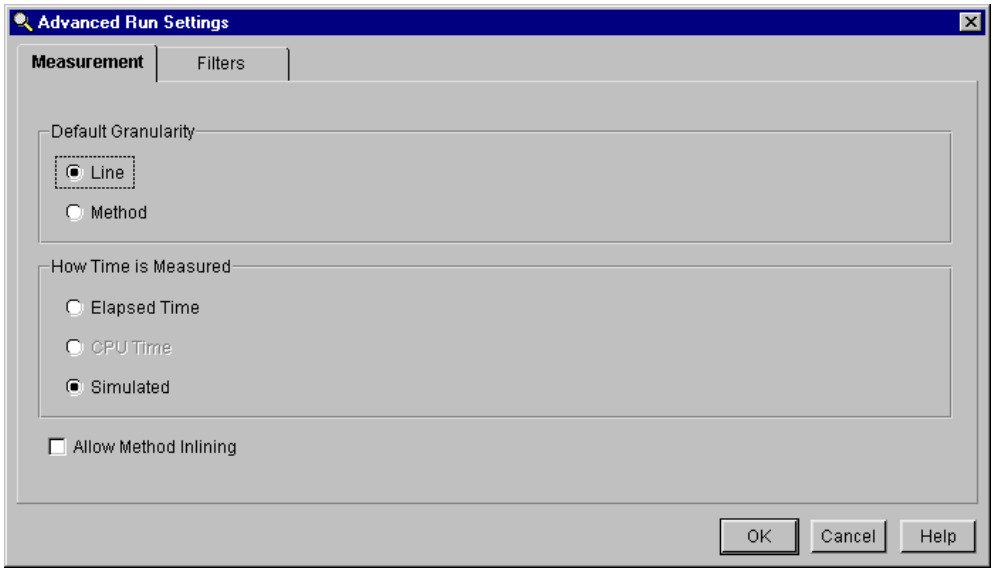


Figure 20. JProbe Advanced Run Settings - Measurement Tab

The Filters tab (Figure 21 on page 47) specifies any filters you want to apply to your program before running it with JProbe Profiler. By default, the times and object counts for `java.*` and `sun.*` are tracked but are not shown in the snapshot. Enabling the filter `*.run` includes all run methods in the snapshot, so you can track each thread in your application.

You can filter any package, class, or method. To add a new pattern that you want to filter, type it in the New Class field and click **Add**. Profiler applies filters to a program in the order they are entered. For example, filter all native Java methods except those responsible for handling I/O by hiding `java.*`, and then adding a filter to show `java.io`.

If you want only the SanFrancisco classes, you should add a filter to show `com.*` and then hide everything else. To remove a pattern from the filter list, select it by clicking the left-most edge of the **Filters** table and click **Remove**.

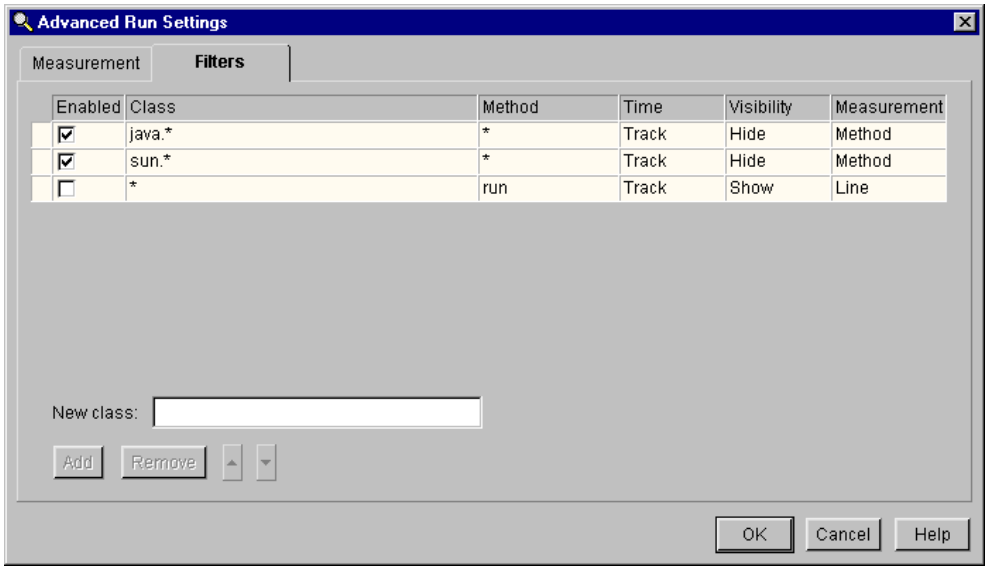


Figure 21. JProbe Advanced Run Settings - Filters Tab

4.4.2 Testing a SanFrancisco Application

JProbe, in itself, contains Jdk1.1.5, which it uses. To be able to run SanFrancisco through JProbe with JProbe Version 1.1, you have to replace the String class that is included in the classes.zip file that comes with Jdk1.1.5. The file classes.zip is located in: `<install dir>\JProbe\jdk115\lib`

You replace the String class that comes with Jdk1.1.5 with the one that comes with SanFrancisco, the last one is located in: `<install dir>\Sf\Sf130\java\lang`

Here is an explanation of how to replace the classes.zip file:

1. From within Explorer, copy the Java\lang\string.class file (located in the SanFrancisco directory) to another drive (for example, D:).
2. Using Winzip (or another unzip program), unzip classes.zip in the JProbe\jdk115\lib directory and delete String.class.
3. In the Winzip window, press the **Add** button, choose the String.class file you copied in step 1, ensure Compression is set to None, the Save extra folder information box is checked, and press the **Add** button. The String.class adds.
4. Close the Winzip window.

After replacing the String class, you run SanFrancisco by entering the correct values in the Run Settings dialog.

For example, if you want to run the LSFN server with JProbe, you fill in the values in the Run Settings dialog as shown in Figure 22 on page 48. The values in the different fields are:

- Class file: `com.ibm.sf.gf.SmServerImpl`
- Working directory: `D:\SF\SF130`
- Classpath: `%CLASSPATH%`
- Java arguments: `-ms6m -mx128m -DserverName=SFGSMProcess`
- Source file path: `D:\SF\SF130`

If you want to launch SFBOProcess1 with JProbe, fill in the following values in the Run Settings dialog:

- Class file: `com.ibm.sf.gf.SmServerImpl`
- Application arguments: `-GSM_UID 1`
- Working directory: `D:\SF\SF130`
- Class path: `%CLASSPATH%`
- Java arguments: `-ms24m -mx96m -Dsfenv="d:\sf\sf130\com\ibm\sf\etc\sfenv.ini" -DserverName=SFBOProcess1 -DGDPMsleepTime=60`
- Source file path: `D:\SF\SF130`

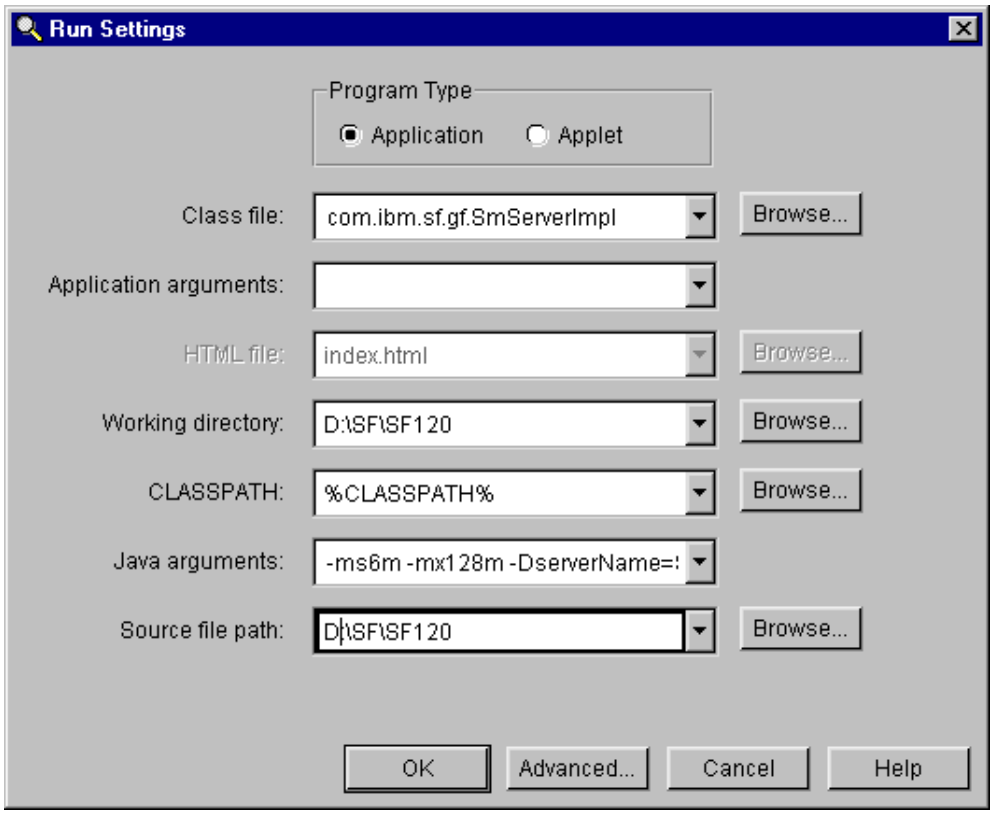


Figure 22. JProbe Run Settings - Running the LSFN Server

4.4.3 The Memory Usage Window

To view the Memory Usage window, select **Memory Usage** from the View menu in the JProbe Profiler Console.

The Memory Usage window (Figure 23 on page 49) allows developers to view the Java heap size and how much memory is currently allocated by the application. This does not include the memory required for JProbe or Java objects allocated during the initialization of the JVM itself. The status bar gives statistics on available and allocated memory.

Depending on how large the application is, you may want to change the refresh speed. By default, JProbe collects memory information once per second. Changing the refresh to collect data less frequently makes data collection less disruptive to the application.

By default, JProbe shows the application's entire memory graph. As more data is collected, the scale changes so that the entire memory usage graph fits in the dialog. You can choose to not change the time scale, and instead, just show the last few minutes of data. When one of these options is selected, any data past the selected time period scrolls off the graph.

You, as a user, can decide to start the garbage collector at any given time by clicking the **garbage can icon**. The garbage collector will then remove unused objects from the application. Removing unused objects should lower the amount of allocated memory.

A large memory gain after performing garbage collection may indicate that your program is creating excessive objects. If the number of objects for a given class continues to grow, this may indicate a memory leak.

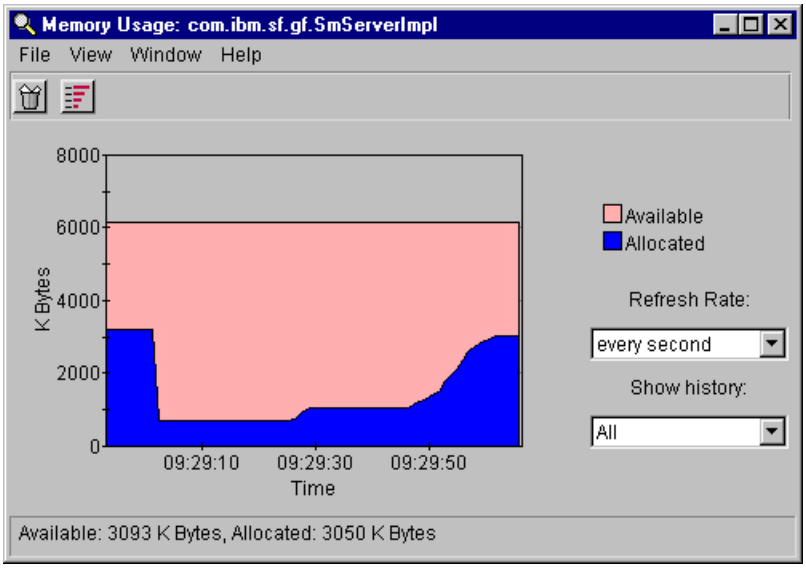


Figure 23. JProbe Memory Usage Window

If you choose not to view the entire graph and only view the last few minutes in the Memory Usage window, you can scroll the window by clicking and holding on the graph as you move the mouse in the direction you want to scroll.

4.4.4 The Instance Summary Window

To view the Instance Summary window, you should select **Instance Summary** from the View menu in the JProbe Profiler Console.

The Instance Summary window (Figure 24 on page 50) allows developers to view the number of live objects in the JVM while your program runs.

Since garbage collection is a time-intensive procedure, reducing unnecessary object creation in your program can increase performance. If the number of objects for a given class continues to grow, this may indicate a memory leak.

Use the color bars that underline the numbers in the Count and Memory columns to quickly locate the most active classes. The brighter the color and the longer the bar are, the more objects are created, or the more memory is used.

If you only want to show a subset of the data, use the Show Only field and type a specific class name to mask, click **Apply** to show only the classes that match the class name or pattern.

As well as when using the Memory Usage window, you can, at any given point of time, click the **garbage can icon** to perform a garbage collection. You can also set the refresh rate at the bottom of the window.

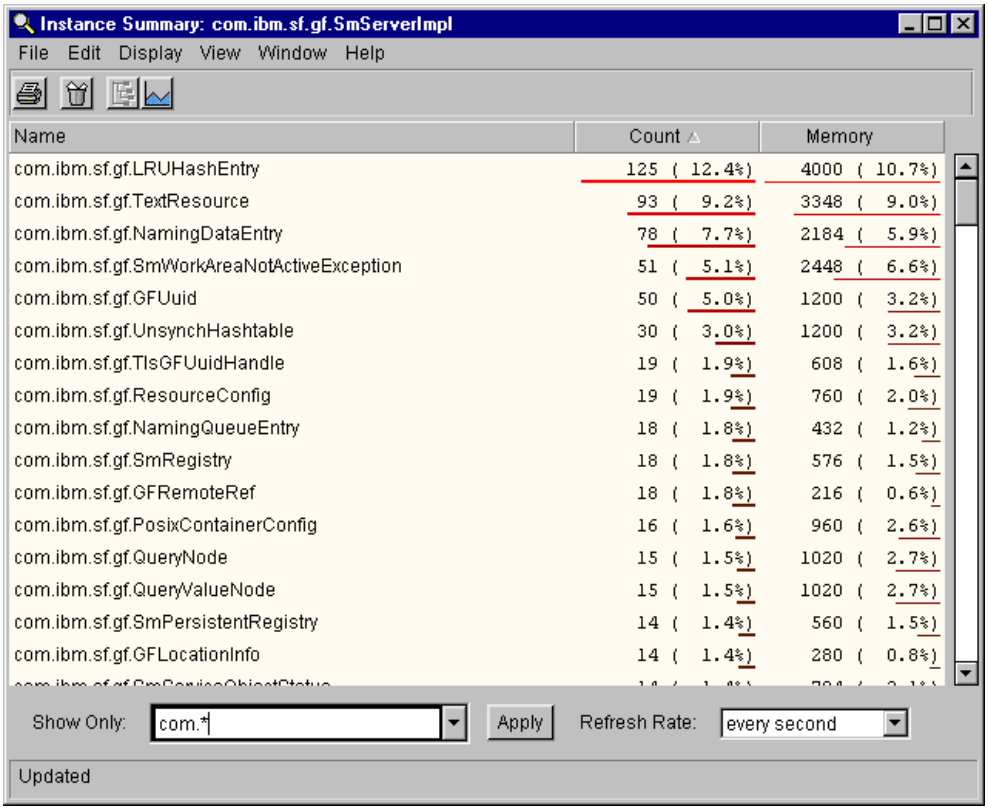


Figure 24. JProbe Instance Summary Window

4.4.5 The Call Graph Window

To view the Call Graph window, you should select **Call Graph** from the View menu in the JProbe Profiler Console. Use the Call Graph window, as shown in Figure 25 on page 51, to view a graph of all the methods in your program and their parents and children. The Call Graph window is divided into two separate panes.

The top pane shows the actual Call Graph. Select the number of top nodes you want to display and the metric you want to use to select the top nodes. Nodes are then colored according to the metric shown in the Color Methods By combo box.

For example, Method Time highlights those methods that spend the most time executing. The brighter the node color, the more expensive the method.

The bottom pane shows the method list and panner. The method list shows all the methods in your program (excluding those that were filtered at runtime). The methods currently displayed in the Call Graph are checked. Selecting a method

from the method list highlights the appropriate node in the Call Graph, and vice versa. Enabling a method's check box displays it in the Call Graph.

Use the Show Methods Top combo box to select how many methods you want to display in the Call Graph. You can show up to 50 methods in your application. You can manually add more nodes to the graph by using the right-click menu's Show/Hide submenu.

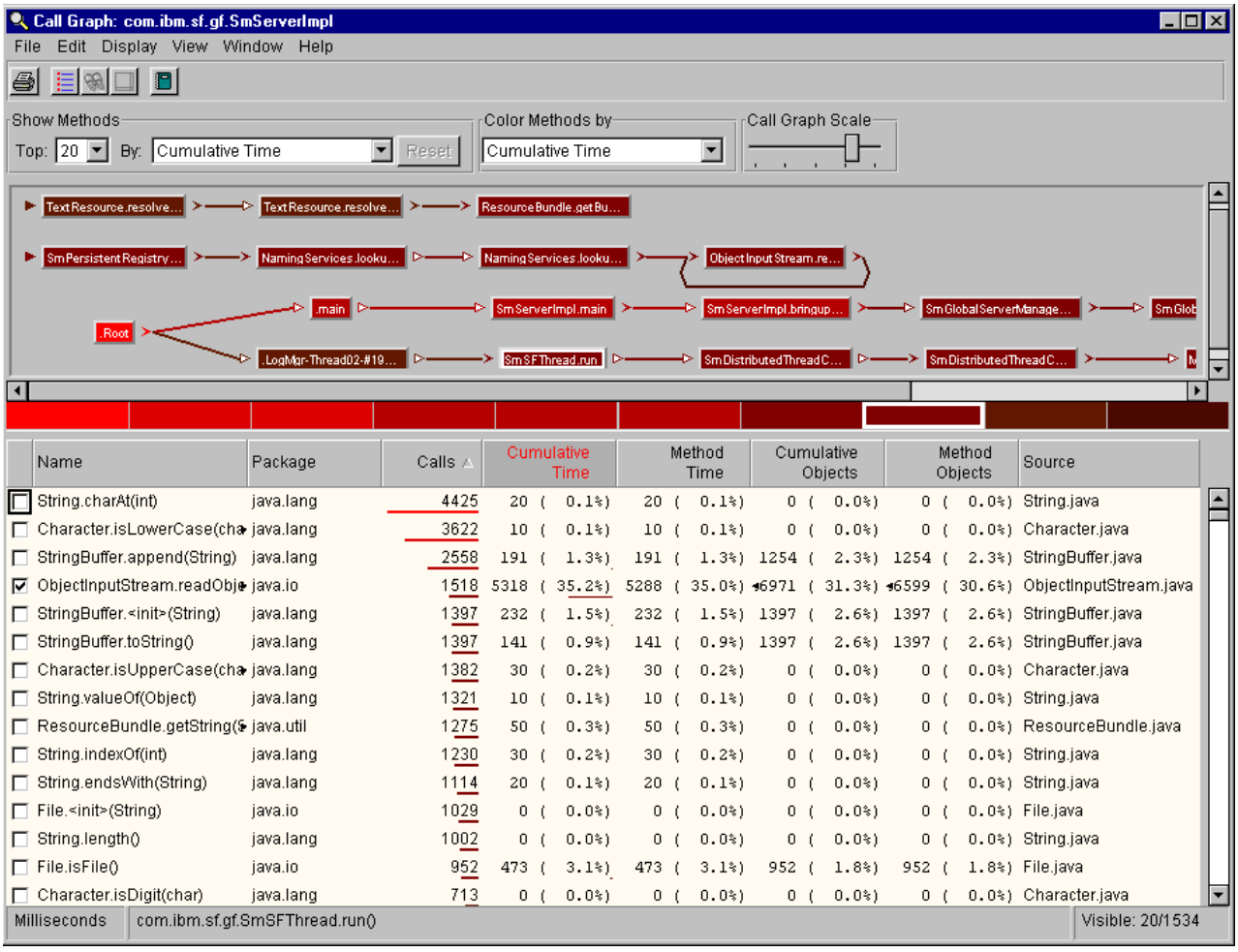


Figure 25. JProbe Call Graph Window

You can either exclude parts of the snapshot that you are not interested in, or focus on the parts in which you are interested by using the Subtree commands from the right-click menu.

Before running an application, use the Filters Tab of the Advanced Run Settings dialog to exclude profile data collection for unneeded packages. By filtering your profile, you can further target your search for only those methods you care about. For example, you may not want to include third party packages when looking for hot spots.

4.4.6 The Method List Window

To view the Method List window, select **Method List** from the View menu in the JProbe Profiler Console.

The Method List window, as shown in Figure 26, displays all non-filtered methods called in the application. Each of the columns represents a different metric and gives timing and object information for each method.

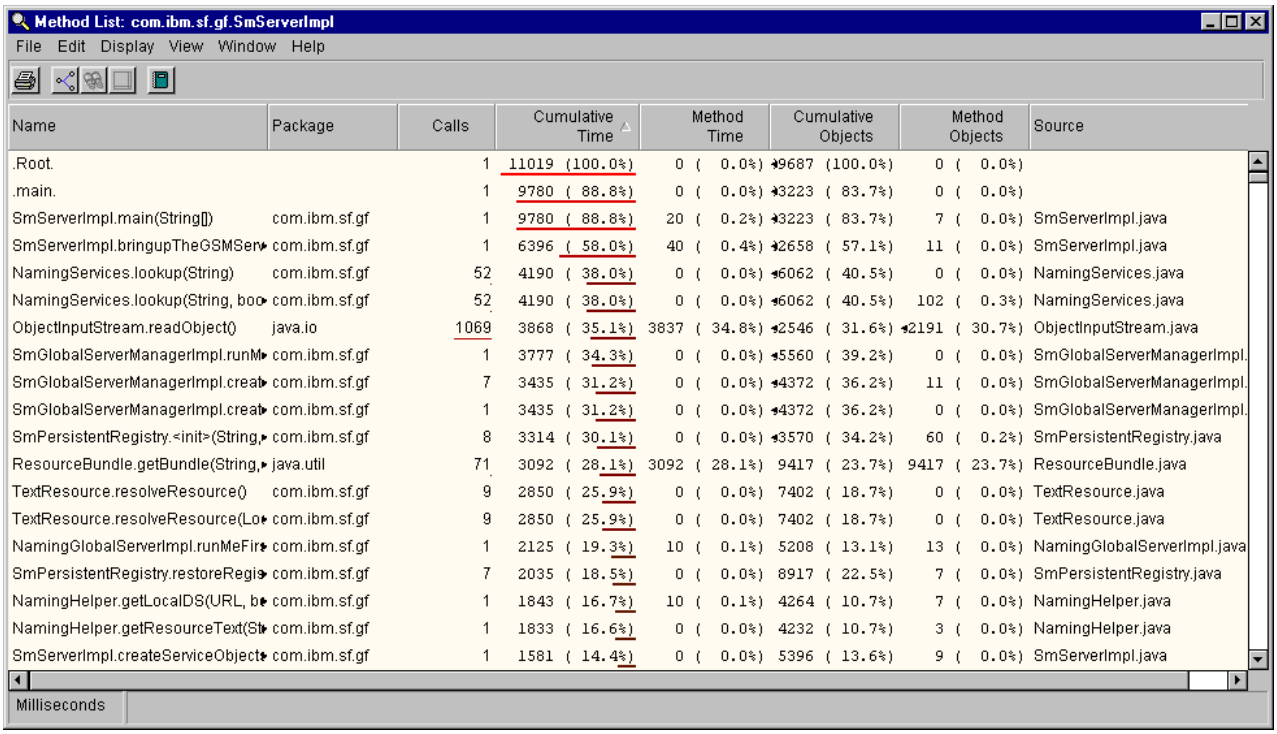


Figure 26. JProbe Method List Window

The Cumulative Time column shows the combination of self time and the combined time of all the method's descendants and exclusive recursive calls to its descendants.

The Method Time column shows the amount of time the method spent executing, excluding the time spent in its descendants.

The Cumulative Objects column shows the number of objects created during the method's execution, including the number of objects created in all its descendants.

The Method Objects column shows the number of objects created during the method's execution, excluding the number of objects created in its descendants.

4.4.7 The Method Detail Window

If you are viewing either the Method List window or the Call Graph window and want a more detailed description of a certain method, you only have to double click on the method of interest to have a more detailed display of the method in the Method Detail window.

The Method Detail window, shown in Figure 27, displays time and memory information about a specific method, including its parents and children.

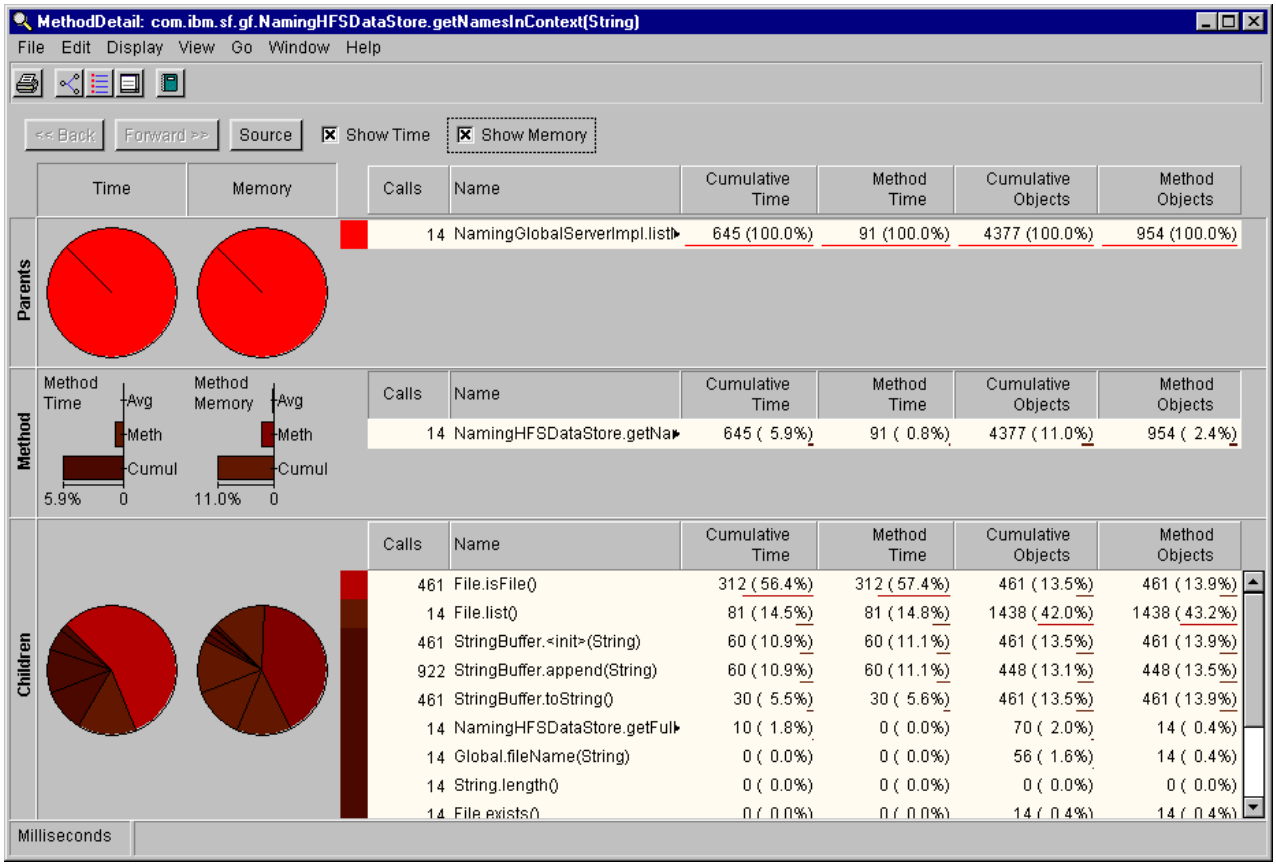


Figure 27. JProbe Method Detail Window

The middle section of the Method Detail window shows information on the selected method: the number of calls made to it from within the program, its Method and Cumulative times, and Method and Cumulative Object counts.

The Parents section lists the selected method's callers. The data shows how the method's time/memory is propagated to its parents.

The Children section shows how much of the method's time/memory for which each child is responsible. Double-click on a **parent** or **child** to view its Method Detail window.

Use the Show Time and the Show Memory check boxes to display time graphs and object creation graphs and data for the method's parents and children. The time graphs show how the method's cumulative time is propagated to its parents and children.

The object creation graph shows how the method's cumulative object count is propagated to its parents and children.

4.4.8 The Source Window

The Source window, shown in Figure 28 on page 54, is accessed from the Call Graph, Method List, Method Detail, and Instance Detail windows. When you have

chosen method, select the **View Source** command by right clicking on the chosen method. JProbe looks for the source code file from the path entered in the Run Settings dialog and displays it automatically. If it cannot find the file, JProbe prompts you to locate it manually.

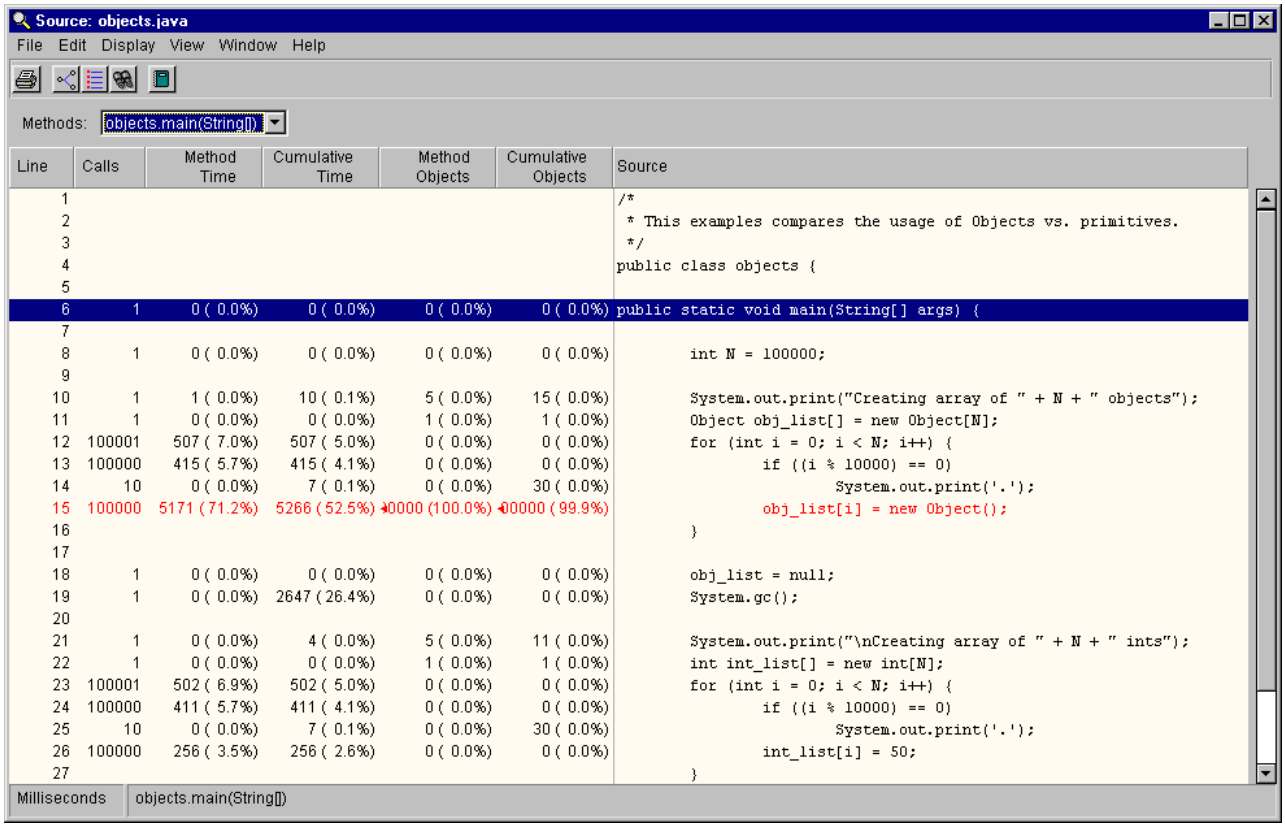


Figure 28. JProbe Source Window

The Source window is divided into two separate panes. The right side shows the source code. The left side shows performance data, including the line number, number of calls, and method and cumulative times.

The most expensive line in each method is highlighted in red to allow you to easily pinpoint problems in your code.

Time and object creation data is displayed for each line of the source and shows the amount of time spent/number of objects created for that line of code. The number in brackets is that line's percentage of the total method time/objects.

The example above is just a dummy example, but if you want to be able to view the source code of your SanFrancisco application, make sure that you have the source code available and that your application has line information, that is, that it has been compiled with the `-g` option.

4.4.9 Summary on JProbe

Since JProbe is a more complicated tool than OptimizeIt, it is also somewhat more difficult to use. It is, however, a more complete tool that includes features, such as the availability to choose timing method and the possibility of more careful examinations of single methods.

JProbe does, as compared to OptimizeIt, show char[] allocations in addition to String allocations. It has a better sorting of data, and it also provides the ability to save profiles for later use.

One disadvantage of JProbe is that it does not contain the actual Java VM that will be used when an application deploys. This makes it difficult to choose the JVM that you want to test your application on, which might be a big set back since there could have been new releases of the JVM, with new capabilities that you would want to use. Or you may even want to use a JVM that your own company has developed. OptimizeIt does not have this limitation, because with OptimizeIt, you can choose to use which ever JVM you want.

Besides the fact that you are bound to use the JVM that comes with JProbe, JProbe is a more complete tool than OptimizeIt. It does, however, require a little more effort because of its higher level of complexity.

4.5 Windows NT Performance Monitor

The Windows NT Performance Monitor is the premier system performance tool for Windows NT (3.52 and 4.0). It automatically comes with NT, can examine a wide range of system phenomena and is highly customizable. It can show you all kinds of statistics (over a dozen) that will help you understand what is going on in your system and is an excellent tool for overall system performance.

The NT Performance Monitor can be used to monitor any Windows NT system on the network, with appropriate permissions.

The NT Performance Monitor can be used out-of-the-box to:

- Observe current performance
- Log performance for later perusing
- Notify you when specified conditions occur
- Generate reports and graphs
- Output stats to standard spreadsheet files

You can start the NT Performance Monitor by following this path: **Start—>Programs—>Administrative Tools—>Performance Monitor**. Or, enter the command statement:

```
perfmon <settings file> [-c //computername]
```

You can choose to run the NT Performance Monitor in one of four views:

- Chart view is used to either view current activity or a saved log.
- Log view is used to save (log) current activity for later replay.
- Alert view is used for setting up alerts to watch for and record particular events and alerts you when they occur.
- Report view is used for publishing statistics.

On each computer, you can view the behavior of objects, such as processors, memory, cache, threads, and processes. Each of these objects has an associated set of counters that provide information about device usage, queue lengths, delays, and information used to measure throughput and internal congestion.

When monitoring a system, you are really monitoring the behavior of its objects. In the Windows NT operating system, an object is a standard mechanism for

identifying and using a system resource. Objects are created to represent individual processes, sections of shared memory, and physical devices. Performance Monitor groups the counters by object type. A unique set of counters exists for the processor, memory, cache, hard disk, processes, and other object types that produce statistical information. Certain object types, and their respective counters, are present on all systems.

Each object type can have several instances. For example, the Processor object type will have multiple instances if a system has multiple processors. The Physical Disk object type has two instances if a system has two disks. Some object types, such as Memory and Server, do not have instances. If an object type has multiple instances, each instance may be used with the same set of counters. The data is then tracked for each instance.

Two object types, Process and Thread, have a particularly close relationship. A Windows NT process is created when a program runs. A process may be either an application (such as SanFrancisco), a service (such as the DB2 DBMS), or a subsystem (such as the print spooler or POSIX). In addition to an executable program, every process consists of a set of virtual-memory addresses and at least one thread.

There is extensive Help text on using the Performance Monitor, what all the options mean and how to use them.

A detailed description of this tool can be found in the *Windows NT Resource Kit* (Microsoft Press) handbook.

4.6 AS/400 Performance Tools

There are several AS/400 performance tools on that provide everything from very basic performance data, for example, the amount of CPU time that a given test consumed, to very detailed performance data by method, for example, the amount of CPU time that was spent in a method.

The Performance Data Collector/Performance Explorer (PDC/PEX) is a comprehensive set of tools that provide the most data. When you are running this, you can use an option, called TPROF internally, that shows the hot spots by method, or even by statement.

A word of caution is that you have to be aware that TPROF uses a sampling technique for collecting data. Therefore, to get valid data that shows which statements are the hottest, the tests need to run for several seconds.

To use TPROF, perform the following steps:

1. Enable performance collection for the Java classes in the test by entering:

   ```
   CRTJVAPGM CLSF('classname') ENBPFRCOL(*ENTRYEXIT)
   ```

   For example, to enable collection for classes in /sf130/tpcc/gbof, enter:

   ```
   CRTJVAPGM CLSF(''/sf130/tpcc/gbof/*.class'') ENBPFRCOL(*ENTRYEXIT)
   ```

2. Create a PEX definition named TPROF using the following statement:

   ```
   ADDPEXDFN DFN(TPROF) TYPE(*TRACE) JOB(*ALL)
   MAXSTG(102400) INTERVAL(1) TRCTYPE(*SLTEVT) SLTEVT(*YES)
   MCKINST(*NONE) BASEVT(*PMCO)
   ```

3. Enter the following command statement:

   ```
   STRPEX SSNID(testname) DFN(TPROF)
   ```

4. Run the test.
5. Enter the following command statement:

   ```
   ENDPEX SSNID(testname)
   ```

6. Collect the TPROF data by entering:

   ```
   PRTPEXRPT MBR(testname) TYPE(*PROFILE) PROFILEOPT(*SAMPLECOUNT *PROCEDURE)
   ```

   The output will be in a spooled file.

For a detailed description of this tool refer to *AS/400 Performance Explorer — Tips and Techniques*, SG24-4781.

4.7 The Container Cache Statistics Tool

Each Container in SanFrancisco maintains a memory cache where it keeps recently accessed persistent objects so, when needed, it can retrieve them directly from memory rather than from disk. SanFrancisco provides a utility starting in SanFrancisco Version 1, Release 3 to display cache size and cache hit and miss rates. This information allows decisions for optimal entity cache settings.

4.7.1 How to Run

You run the Container Cache Statistics Tool from the command line by typing:

```
java com.ibm.sf.gf.ContainerStats
```

Enter the desired container, delay, duration, and interval when asked. Then, let it run to completion to get a summary report and do cleanup.

Remember that all counters that can be reset are reset on every retrieval, and for each container, there can be only one Container Cache Statistics Collector running at the time or unpredictable results will occur. Counters will, for example, be reset by one user while being read by another user.

4.7.2 Explanation of Output

`cpcty`	Maximum number of possible cache entries
`size`	Number of current cache entries
`szActv`	Number of current cache entries with active status
`szActvH`	High water number of cache entries with active status since last reset
`gets`	Total number of get requests
`gNStsfd`	Get not satisfied because entry not in cache

4.7.3 Example Output

```
Container Cache Manager Statistics
time    cpcty   size    szActv  szActvH gets    gNStsfd
11:11   8009    2760    0       0       0       0
11:11   8009    2760    0       1       4       0
11:11   8009    2761    0       2       10      3
11:11   8009    2761    0       0       0       0
11:11   8009    2775    59      59      2285    61
11:11   8009    2859    10      60      5583    114
```

```
11:11   8009    2896    47      59      5049    132
11:11   8009    2962    61      61      5701    167
11:11   8009    3003    47      119     6794    216
11:11   8009    3057    59      59      6500    177
11:11   8009    3121    41      60      5138    159
11:11   8009    3164    75      75      5969    213
11:11   8009    3198    59      119     6053    182
11:11   8009    3238    59      59      5618    136
11:11   8009    3258    0       59      2593    68

Container Statistics Collection Summary

        cpcty   size    szActv  szActvH gets    gNStsfd
Maximum 8009    5356    119     119     27862   836
Totals  -       -       -       - 355912  10561
```

4.7.4 What to Watch For

The main item to watch out for is if the "gets not satisfied because entry not in cache" is growing to very large numbers. If this occurs, you should probably consider to make the memory cache for that specific container larger.

You can, however, not set the memory cache for the different containers too large since this would decrease the amount of memory available for the rest of the system.

4.8 Lock Analysis Tools

IBM SanFrancisco provides tools to analyze locking problems and the performance optimizations for lock sequences.

Before using the locking tools provided by SanFrancisco, certain terminology must be explained.

Lock conflict
: A lock ordering or upgrade problem.

Serialization point
: A point at which two transactions can no longer run in parallel due to the fact that one or both transactions are getting exclusive access to a common entity.

Lock ordering
: A process to avoid lock ordering problems in which it is important to establish a lock hierarchy and access locks in order.

Lock Upgrade Deadlock
: When two transactions are trying to make a read to write lock upgrade on the same entity. The following example is a typical case of lock upgrade deadlock:

- Transaction T1 accesses entity E1 with a read lock.
- Transaction T2 accesses entity E1 with a read lock.
- Transaction T1 attempts to change E1, causing a read to write lock upgrade, T1 waits for T2 to release read lock.
- Transaction T2 attempts to change E1, causing a read to write lock upgrade, T2 waits for T1 to release read lock.

The example above results in a deadlock where transaction T1 is waiting for transaction T2 to finish, when at the same time, transaction T2 is waiting for transaction T1 to finish.

4.8.1 The Lock Conflict Trace Analysis Tool

In SanFrancisco, different transactions can have read access to the same Entity at the same time, but only one transaction can have write access to an Entity. And while that write access is held, no other transaction can gain read or write access to the Entity.

When deadlock occurs, a `LockUnavailableException` is thrown. This exception indicates that the access mode requested for an Entity was not granted within the specified time-out period (the default is 30 seconds). Deadlock is not the only condition that can result in this exception. If you are running an application with long-running transactions, you may need to increase the default time-out period to avoid this exception.

The Lock Conflict Trace Analysis Tool should be used during application development phase to analyze trace data for lock conflicts.

The tool highlights potential problems, such as lock upgrades, and can compare the lock ordering between two transactions. The tool can be used proactively in single client environments to prevent possible multiple client locking problems. Analysis of locking data is done statically after normal execution of an application.

The analysis process includes these steps:

1. Generate lock conflict trace data.
2. Start the trace analysis tool and load the trace data files.
3. Analyze transaction sets.
4. Display entity details.
5. Compare transactions.

4.8.1.1 Generate Lock Conflict Trace Data

To generate lock conflict trace data, you must perform these procedures (details follow):

1. Start trace on both the client and server processes.
2. Run the segment of the application that is of interest.
3. Stop trace on both the server and client processes

For more detailed information about activating lock conflict trace, look in the tool help and in the SanFrancisco documentation.

4.8.1.2 Start the Trace Analysis Tool and Load the Trace Data Files

To start the Trace Analysis tool and to load the trace data files, perform the following steps:

Note

SanFrancisco does not have to be running to use the trace analysis tool.

1. From the Command Prompt, start the tool by executing **TraceAnalysisTool**.

   Note: The tool uses Java Swing. If Swing is not in your classpath, you will receive an exception.
2. Select **File, Open** on the menu bar. The Open File dialog box appears.
3. Highlight the file to open and click the **Open** button.
4. Repeat steps 2 and 3 for each trace file.
5. Continue with the Analyze transaction sets procedure.

4.8.1.3 Analyze Transaction Sets

The Lock Conflict Trace Analysis Tool shows the Entity set associated with a transaction and highlights those Entities that have lock upgrades. Here is an example of the tool output:

```
Transaction 8 NoName Upgraded
Tran Id:-5487728891461103616.-2305842983292558782

      begin time: Thu May 28 10:49:26 CDT 1998

Entity Set {
            T8 E1r com.ibm.sf.cf.CompanyControllerRootImpl
            T8 E2r com.ibm.sf.gf.EntityOwningExtentImpl
            T8 E3r com.ibm.sf.cf.EnterpriseImpl
            T8 E62r com.ibm.sf.gf.EntityOwningExtentImpl
            T8 E63r com.ibm.sf.gl.AnalysisGroupImpl
            T8 E5r com.ibm.sf.gl.ChartOfAccountsRootImpl
            T8 E95r com.ibm.sf.gl.AnalysisCodeImpl

Upgraded >> T8 E96rw com.ibm.sf.gf.CommonMapImpl
            T8 E97r com.ibm.sf.gf.BTreeNodeImpl
            T8 E98r com.ibm.sf.gf.BTreeLeafImpl

      } commit time: Thu May 28 10:49:27 CDT 1998
```

The first two lines shows the integer number the tool assigned to the transaction ID.

> **Note**
>
> The Tran ID field is the unique ID assigned to this transaction.

The third line shows the date when the transaction was begun.

The entity set shows each entity accessed by transaction. The entity may be accessed more than once during the same transaction and may be accessed by other transactions.

```
T8 E1r com.ibm.sf.cf.CompanyControllerRootImpl
```

The "T8" indicates this is Transaction 8. The E1r indicates entity 1 was accessed in read mode. Possible lock modes include "r" for read, "w" for write, "rw" for read to write lock upgrade, "d" to indicate the lock was dropped, and "dw" to indicate a read lock was dropped, and a write lock was accessed. The "`com.ibm.sf.cf.CompanyControllerRootImpl`" is the class name associated with the entity.

4.8.1.4 Display Entity Details

Both lock upgrades, "rw", and drop and re-access, "dw", are flagged by special keywords to indicate they require additional analysis ("Upgraded >>" and "ReAccess >>").

In the example above, you can see that a lock upgrade is indicated like this:

```
Upgraded >> T8 E96rw com.ibm.sf.gf.CommonMapImpl
```

The trace data includes a stack dump of the code path associated with the lock upgrade so you can modify the code to correct the problem. As shown above, the Entity Detail for the CommonMapImpl was "Upgraded":

```
Tran Number : 8
Tran Id: -5487728891461103616.-2305842983292558782
Entity Number: 96
Entity Id: 218711701609080679.10995116277936
Entity Class: com.ibm.sf.gf.CommonMapImpl
Entity was first accessed in this tran at: Thu May 28 10:50:07 CDT 1998
Initial Lock: READ
Granted Lock: WRITE
Lock String: rw
Lock was Upgraded.

Stack Dump associated with Lock Upgrade.
java.lang.Exception:
at com.ibm.sf.gf.LockSet.traceLockObtained(LockSet.java:2584)
at com.ibm.sf.gf.LockSet.getWriteLock(LockSet.java:2011)
at com.ibm.sf.gf.LockSet.changeMode(LockSet.java:1486)
at com.ibm.sf.gf.LockSet.lock(LockSet.java:540)
at
com.ibm.sf.gf.PersistentContainer.getEntityNormal(PersistentContainer.java:806
)
at com.ibm.sf.gf.PersistentContainer.getEntity(PersistentContainer.java:473)
at
com.ibm.sf.gf.BaseFactoryServerImpl.getEntityFromServer(BaseFactoryServerImpl.
java:1193)
at
com.ibm.sf.gf.BaseFactoryServerImpl.getEntityRefFromServerToServer(BaseFactory
ServerImpl.java:961)
at
com.ibm.sf.gf.BaseFactoryServerImpl.getEntitiesFromServer(BaseFactoryServerImp
l.java:759)
at
com.ibm.sf.gf.BaseFactoryImpl.processServerHeldEntities(BaseFactoryImpl.java:1
605)
at com.ibm.sf.gf.BaseFactoryImpl.getEntity(BaseFactoryImpl.java:1507)
at com.ibm.sf.gf.BaseFactoryImpl.getEntity(BaseFactoryImpl.java:2255)
at com.ibm.sf.gf.Helper.getObjectFromHandle(Helper.java:423)
at
com.ibm.sf.gf.BusinessObjectImpl.getObjectFromHandle(BusinessObjectImpl.java:1
78)
at
com.ibm.sf.gl.ChartOfAccountsImpl.addOwnedPostingCombinationBy(ChartOfAccounts
Impl.java:2352)
at
com.ibm.sf.gl.ChartOfAccountsRootImpl_Skel.dispatch(ChartOfAccountsRootImpl_Sk
el.java:314)
at com.ibm.sf.gf.GFRemoteServerRef.dispatch2(GFRemoteServerRef.java:245)
at
```

```
com.ibm.sf.gf.SmDistributedThreadContext.serviceRequests(SmDistributedThreadCo
ntext.java:534)
at
com.ibm.sf.gf.SmDistributedThreadContext.serviceRequests(SmDistributedThreadCo
ntext.java:466)
at com.ibm.sf.gf.SmSFThread.run(SmSFThread.java:433)
```

Lock upgrades should be eliminated or masked. They can be eliminated by getting a write lock first or splitting the transaction into two transactions, one that primarily reads the Entities, and the other that changes them. Upgrades can be masked by getting a write lock on the Entity owning the entity that is upgraded. For the example, consider a case where an AnalysisCode Entity owns a Map, and the AnalysisCode is initially accessed with a write lock. Calling methods that cause the Map to have a lock upgrade would not be a problem because the write lock on the AnalysisCode serializes the changes to the map.

4.8.1.5 Compare transactions

The order in which Entities are accessed in concurrent transactions is important. If two transactions attempt to access the same entities in a different order, deadlock can occur.

Consider transaction T1 that accesses Entity E1 in read mode followed by E2 in write mode. T1 has an Entity set of { E1 (r), E2 (w) }.

Now consider transaction T2 that accesses Entity E2 in read mode followed by E1 in write mode. T2 has an Entity set of { E2 (r), E1 (w) }.

If T1 gets the read lock on E1 at the same time that T2 gets a read lock on E2, then neither transaction will be able to gain access to the next Entity in its set (because the other transaction has it locked). This results in deadlock.

To avoid this type of problem, establish an Entity hierarchy and lock Entities in that order.

The lock conflict trace analysis tool can compare two transactions and identify if a potential for deadlock exists. This is shown in the tool output here:

```
Comparing the lock ordering for the following transactions:
T3 NoName Thu May 28 10:47:57 CDT 1998
-5487728891461103616.-2305842991882493374
T6 NoName Thu May 28 10:48:41 CDT 1998
-5487598784016805888.-2305842991882493374

T3 T6 Class Name (associated with the 2nd transaction's entity numbers.)

E1r  == E1r com.ibm.sf.cf.CompanyControllerRootImpl
E2r  == E2r com.ibm.sf.gf.EntityOwningExtentImpl
E3r  == E3r com.ibm.sf.cf.EnterpriseImpl
E7r  == E7r com.ibm.sf.cffi.FinancialTransactionSourceControllerRootImpl
E8r  == E8r com.ibm.sf.gf.EntityOwningExtentImpl
E9r  == E9r com.ibm.sf.cffi.FinancialTransactionSourceImpl
E10r == E10r com.ibm.sf.cffi.TransactionTypeControllerRootImpl
E11r == E11r com.ibm.sf.gf.EntityOwningExtentImpl
E12r == E12r com.ibm.sf.cffi.TransactionTypeValueImpl
E13r == E13r com.ibm.sf.cffi.TransactionTypeValueImpl
E14r == E14r com.ibm.sf.cffi.TransactionTypeValueImpl
E15r == E15r com.ibm.sf.cffi.TransactionTypeValueImpl
E16r == E16r com.ibm.sf.cffi.TransactionTypeValueImpl
```

```
E17r == E17r com.ibm.sf.cffi.TransactionTypeValueImpl
E18r == E18r com.ibm.sf.cffi.TransactionTypeValueImpl
E19r == E19r com.ibm.sf.cffi.TransactionTypeValueImpl
E20r == E20r com.ibm.sf.gl.BudgetProfileControllerRootImpl
E21r == E21r com.ibm.sf.gf.EntityOwningExtentImpl
E22r == E22r com.ibm.sf.gl.BudgetProfileImpl
E24r == E24r com.ibm.sf.gf.LocaleControllerImpl
E25w SP E25w com.ibm.sf.gf.IdGeneratorImpl
E26r == E26r com.ibm.sf.gf.GlobalLocaleResourceControllerImpl
E27w == E27w com.ibm.sf.gf.CommonMapImpl
E28r == E28r com.ibm.sf.gf.BTreeLeafImpl
E29w == E29w com.ibm.sf.gf.DynamicMessageCatalogImpl
E30w == E30w com.ibm.sf.gf.CommonMapImpl
E31r == E31r com.ibm.sf.gf.BTreeNodeImpl
E32r == E32r com.ibm.sf.gf.BTreeNodeImpl
E33rw == E33rw com.ibm.sf.gf.BTreeLeafImpl
E34dw == E34dw com.ibm.sf.gf.EntityOwningExtentImpl
E35w != E36r com.ibm.sf.cffi.FinancialCalendarImpl
E36r != E37r com.ibm.sf.cf.FiscalCalendarImpl
E37r != E38r com.ibm.sf.gf.CommonMapImpl
E38r != E39r com.ibm.sf.gf.BTreeLeafImpl
E39r != E40r com.ibm.sf.cf.FiscalYearImpl
E40r != E41r com.ibm.sf.cf.FiscalYearImpl
E41r != E42r com.ibm.sf.cf.FiscalYearImpl
E42r != E43r com.ibm.sf.gf.CommonListImpl
E43r != E44r com.ibm.sf.gf.BTreeLeafImpl
E44r != E45r com.ibm.sf.cf.UndatedFiscalPeriodImpl
E45r != E46r com.ibm.sf.cf.DatedFiscalPeriodImpl
E46r != E47r com.ibm.sf.cf.DatedFiscalPeriodImpl
E47r != E48r com.ibm.sf.cf.DatedFiscalPeriodImpl
E48r != E50r com.ibm.sf.cf.DatedFiscalPeriodImpl
E50r != E51r com.ibm.sf.cf.DatedFiscalPeriodImpl
E51r != E54w com.ibm.sf.cf.NumberSeriesImpl
E54w != E55w com.ibm.sf.gl.GLJournalImpl
E55w != E56w com.ibm.sf.gl.GLJournalImpl
E56w != E57w com.ibm.sf.gl.GLJournalImpl
E57w != E58w com.ibm.sf.gl.GLJournalImpl
E58w != E35w com.ibm.sf.gl.GLJournalImpl

Summary of conditions:
Entities accessed in order: false (bad)
Lock upgrade before serialization point: false (good)
Serialization point exist: true (goodness depends on other conditions)

Interpretation:
Entities are accessed out of order, but it is not a problem because a
serialization point exists.

Recommendation:
Look at improving throughput by moving the serialization point later in the
transaction (but not after the out-of-order condition).
```

The tool shows the entities that the transactions have in common and the order in which they are first accessed for each transaction.

- "==" indicates the entities are accessed in order.
- "SP" indicates this is a serialization point.
- "!=" indicates the entities are accessed in a different order.
- "UP" indicates one or both entities have a lock upgrade condition.

The tool shows the summary of three conditions, interprets those conditions, and provides a recommendation. The conditions are:

- **Entities accessed in order**: true or false
  If the in-order condition is true, then the transactions can not deadlock due to an ordering problem.
- **Lock upgrade before a serialization point**: true or false
  If a lock upgrade condition exists prior to a serialization point, the point at which two transactions no longer execute in parallel, then a deadlock condition can occur.
- **Serialization point exist**: true or false
  The goodness of this depends on the other conditions. If the serialization point is prior to a lock upgrade or out of order condition, then it is preventing a deadlock condition. Having a serialization point limits how much the two transactions can run in parallel. This negatively effects throughput.

For out of order conditions, the tool recommends further analysis. If the entities being accessed out of order are only accessed in read mode, a deadlock will not occur. If write locks are involved, deadlock can occur.

A much more detailed description of how to use the Lock Conflict Trace Analysis tool is available in the SanFrancisco documentation at:
`<install dir>`
`\com\ibm\sf\doc\doc_de\base\ibmsf.sf.LockConflictTraceAnalysisTool.html`

The same information can be found just as easily by following these steps:

1. From the Start button, select **Programs—>IBM SanFrancisco—> Information**.
2. Use the search engine to locate the documentation on the Lock Conflict Trace Analysis Tool.

4.8.2 The Lock Contention Console

The Lock Contention Console should be used at runtime to help isolate deadlock conditions that are due to a lock conflict. The tool captures a snap shot of lock data and provides options to display and analyze the data. The tool is primarily useful during a product test phase when running with multiple clients.

4.8.2.1 Presettings

Before you run the Lock Contention Console tool, perform the following tasks:

Set a High Time-out Value

The time-out value determines the time a transaction will wait to acquire a lock on an entity. To effectively use the tool, you need to set a high time-out value. The default time-out value is 30 seconds. Once the time-out value is exceeded, the transaction receives a lock unavailable exception. In order to isolate a deadlock condition, you want to postpone the lock unavailable exception for a long time, typically 24 hours.

To set the time-out value to 24 hours, perform the following steps:

1. Ensure the logical SanFrancisco network is running.
2. Start the SanFrancisco Server Management Configuration console (SmConsole): From the Start button, select **Programs, IBM SanFrancisco**

(current version), Base Utilities, Server Management Configuration. Or from a command prompt, type `SmConsole` or `java com.ibm.sf.gf.SmConsole`.

3. In the SmConsole GUI: expand the network directory tree to see the process names.
4. Select the process you have a container configured in (usually **SFBOProcess1**, but can be different or in addition to). Information specific to the process appears to the right.

> **Note**
>
> You should change the time-out (steps 5-7 below) for each process that has a container assigned to it as defined in [sfdriver]\com\ibm\sf\etc\Global.name file.

5. Change the time-out value from 30000 (30 sec) to 86400000 (24 hr).
6. Press the **OK** button.
7. Press the **Apply** button if it is enabled (the button is located towards the top of the window). The new time-out value is now set.

> **Note**
>
> The Apply button is only enabled when the SFBOProcess1 is running.

8. Continue with the Run the application procedure.

Run the Application

Run the application in a stressful way. The assumption is that you are trying to stress the application to verify it works. If the application appears to hang, it may be an indication that a lock conflict problem exists. Use the Lock Contention Console tool to gather and analyze the lock data.

4.8.2.2 Using the Lock Contention Console Tool

From a command prompt window, start the tool by executing:

```
java com.ibm.sf.gf.LockContentionConsole.
```

> **Note**
>
> If the tool does not start, the application hang condition is not likely the result of a lock conflict.

When started the Tool window appears, including the following frames:

- The Configuration Information frame: Not currently used
- The Actions frame: Use to collect data
  Note: Data can be collected multiple times.
- The Objects frame: Use to analyze lock data information stored in User dump

When you start the tool, gather the lock data. Then, display it for analysis.

How this is done is thoroughly explained in the SanFrancisco documentation at:

```
<install dir>\com\ibm\sf\doc\doc_de\base\ibmsf.sf.LockContentionTool.html
```

The same information can be found just as easily by doing following this path from the Start button: select **Programs—>IBM SanFrancisco—>Information** Use the search engine to locate the documentation on the Lock Contention Console Tool.

Chapter 5. Performance Aspects Using Design Patterns

It is said that a well designed model is the most important step towards a successful application, and that fine tuning the application comes afterwards. This still remains true while developing a SanFrancisco based application, but it does not hurt to be aware of certain decisions that can have an impact on performance.

This chapter deals with the impact design decisions will have on the performance of a SanFrancisco based application. Different design patterns and their influence on the overall performance are examined. Design patterns should be used wisely. Although they offer a higher level of abstraction, they imply a certain overhead that may lead to performance degradation.

For additional information on the design patterns in IBM SanFrancisco, the Extension Guide should be consulted.

5.1 What Design Patterns Are

A pattern captures the most common way in which problems are solved: when a problem is encountered for the first time, the solution has to be built up from scratch. The next time the same, or a similar problem, is encountered again, the experience obtained can be used to solve the problem more quickly. When a similar solution solves different problems, but although they are alike, the common elements of the problems, together with the solution used, forms a pattern.

Patterns help to provide a solution when a problem is encountered a second time. By writing it down, it provides help for someone else who encounters a similar problem. Patterns do not directly help when unsolved problems are encountered, but they do provide a means for thinking about and discussing the possible solutions at a higher level. Since patterns are usually put in general terms, a solution based on the pattern, in most cases, is more flexible.

5.2 Command Pattern

A brief description on this pattern is provided with some explanation why Commands are very useful from a performance point of view. A practical example on the use of patterns concludes this section.

5.2.1 Description

The command pattern encapsulates a set of actions or manipulations that need to be done as a complete unit. This approach implements the concept of a unit of work. All the instructions within a command are executed in one step.

The main advantage of using a Command object is that this Command object can be executed where the objects reside. This avoids many remote method calls. Figure 29 on page 68 illustrates this difference.

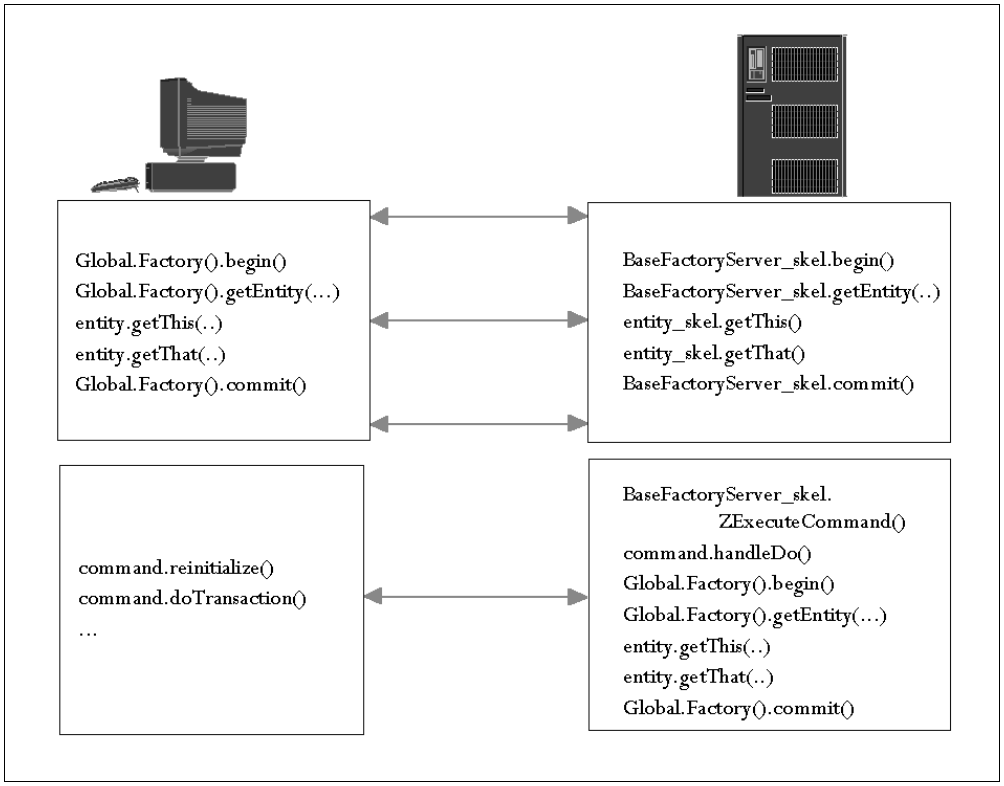


Figure 29. Use of Command Objects

Some additional advantages include:

- Monitoring the changes to business objects becomes easier, as these objects will only be changed through Command objects.
- The notions of process and task can be directly mapped to Command objects.
- Command objects are the owner of changes made to business objects. They can be implemented so that inverting the changes made is possible, although this undo function is not always needed.
- Additional security is available for commands that execute on the server through the concept of "server-only tasks".

5.2.2 Performance Impact

Since Command objects are atomic and "movable", it is possible to execute them at the location where the objects are that need to be accessed: on the client or on a server. This can result in a big performance improvements as a command is executed in the same memory space as the other objects that are accessed. Remote method calls can take many milliseconds to run (or even seconds if the communication link is very slow), as where local calls will only take microseconds.

Not only are many remote method calls avoided using Command objects correctly, but also the associated data transfer that accompanies accessing objects in a location other than their home is avoided. In the case of large objects, this can substantially increase performance.

Command objects can be executed on the server by setting the target correctly. In this case, the target must be a handle that refers to an Entity residing on the server. All additional state is by preference set by method calls inside the `doAll()` method.

There is a certain overhead in creating the Command objects. But because they are Dependent objects, they are typically lightweight and quickly created. So they do not imply a high level of overhead. Furthermore, a Command object can be reused. This is typically done through a reinitialize method. This method is specific to the Command object. The following code illustrates how an implemention can appear as shown in the following example:

```
public void reinitialize(Handle financialBatchControllerHdl,
   Handle[] ledgerItemHdls, boolean returnCmd) throws
   com.ibm.sf.gf.SFException {
      this.reset();
      this.setLocationHandle(financialBatchControllerHdl);
      this.setReturnCommand(returnCmd);
      this.setLedgerItemHdls(ledgerItemHdls);
}
```

This kind of method should be used in conjunction with the `reset()` method on the Command that places the Command object in the state in which it can again execute the `doAll()` method.

A consistent use of Command objects has a good impact on the overall performance because it can limit expensive remote method calls to a minimum. The application built on top of the SanFrancisco framework will benefit from this and so will other C/S applications, as it will lower the network traffic.

It must be mentioned that, in some cases, the use of Commands does not yield the best performance. If in doubt whether this approach is beneficial or not, start without the use of a Command object. Profile the operation as discussed in 3.2, “Profiling the Application with JProbe Profiler” on page 25, and reimplement using Commands if a performance problem is encountered.

5.2.3 Usages of Command Objects

Command objects can be useful in different operations on Entity objects. The following sections will describe in what context it can be beneficial to use Commands to increase performance.

Object Creation

The Command object is responsible for:

- Accepting the necessary parameters for a particular create.
- Allowing the user to select the necessary factory method.
- Executing on the server.

Object Deletion

The Command object takes in the Handle of the Entity that needs to be deleted and has one of these two actions:

- Calling the factory delete method if the object is loosely coupled.
- Calling the proper remove method on the owner if the object is tightly coupled.

Object Update

The Command object is responsible for updating the Entity on the server. It takes, as parameters, the Handle of that Entity and the values for the fields that need to be changed on that Entity.

Object Access

This is mainly used for viewing. The Command object retrieves the fields of an Entity that needs to be viewed. It takes the Handle of the Entity that holds the fields that contain the values to be displayed.

Collection Access

The Command object needs to be customized so that it can handle the different ways a collection can be accessed. The Command object returns a collection object that contains the useful information that can be accessed through the access methods on this collection. This collection is accessed locally on the client where the information is displayed. Command objects used for this purpose give a big performance boast. The following section "Retrieving a List of Information" illustrates the use of a Command in this context.

Example: Retrieving a List of Information

In many cases, retrieval of a set of data is needed to be displayed in a GUI. Usually this data will be contained in Entity objects, and, in most cases, only a small subset of the data of this object needs to be displayed. There are a couple ways that this data can be retrieved and displayed.

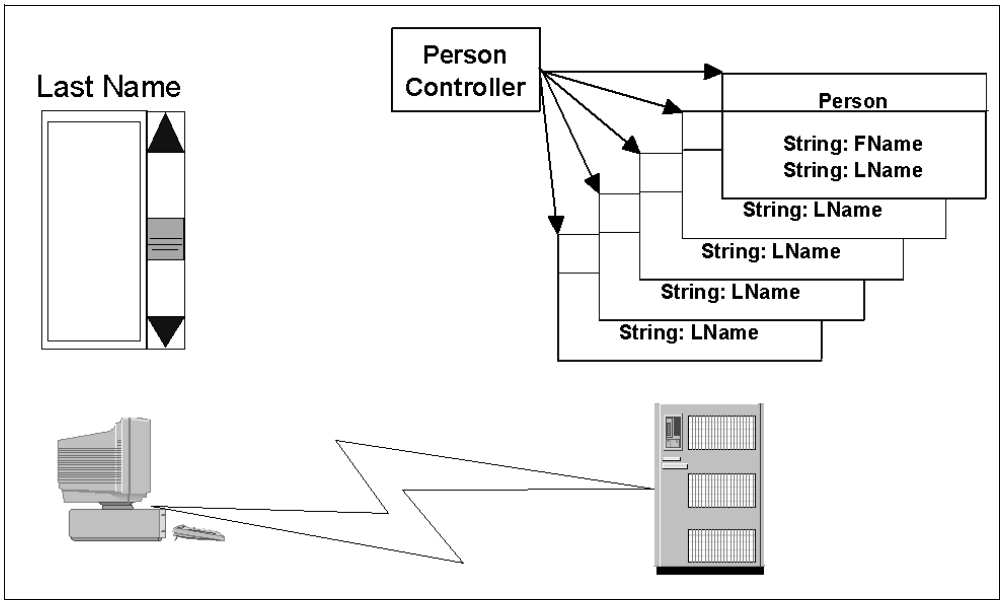


Figure 30. Retrieving a Subset of Data

The first way is to retrieve a collection of the Entity objects that contain the data. The collection will contain the handle for every object in the collection. This collection comes to the client, and the information is displayed on the screen. For every record displayed, the corresponding Entity object needs to be accessed separately on its server. This means that there are a large number of remote method calls to the server where the objects reside.

A far better approach is to use a Command object that executes on the server and gathers all the information to be displayed from the Entity objects. This is all done locally on the server machine. When all the information is gathered, the Command object is streamed to the client where the information can be accessed and displayed. Using this solution, not the entities containing the information, the information itself is returned. To enable the possibility that the corresponding Entity object from a record on the displayed list can be accessed, the Command object also needs to maintain the Handle objects. Typically, only a few records are selected requiring a remote access to the server using the Handle to retrieve the actual object.

This can be part of the client code:

```
// personCtrlHdl handle for the PersonController is referenced
// It is an object on the server
LastNameRetrievalCmd cmd = LastNameRetrievalCmdFactory.
         createLastNameRetrievalCmd(personCtrlHdl);
cmd.doTransaction();
Map names = cmd.getNames();
Iterator it = names.createIterator();
Vector v = new Vector();
String element;
while ((name = it.next()) != null) {
      v.addElement(new UIListItemInfo(new UITextResource(
         element), null, true, false));
}
UIListBox listBox = new UIListBox();
listBox.setListData(v);
```

This can be the implementation of the handleDo() method of the command: `LastNameRetrievalCmd`:

```
PersonController controller = getTarget();
Iterator personIt = controller.createPersonIterator();
Person person = null;
while ((person = personIt.next()) != null)
         ivNames.addElementBy(person.getLastName(),person.getHandle());
         }
```

The real object is retrieved from the Handle. An iterator is created for the controlled Person entity object. The iterator is used to walk through the collection of Persons; the name of every person is stored in a map with the last name as the element and the handle of the Person object as the key.

In case more information elements need to be retrieved, a map can not be used. A separate display object can be used instead. The class definition can appear as shown here:

```
public class PersonDisplayObject extends Object implements Externalizable {
   ...
   String ivLastName;
   String ivFirstName;
   Handle ivHandle;
}
```

5.3 Controller Pattern

A brief description on this pattern is provided together with some performance issues. An alternative to the use of Controller objects is proposed.

5.3.1 Description

Controllers provide two main functions. The first is controlling instances of objects of a class. The controller does things such as ensuring uniqueness of the instances. The controller also provides a central point for working with all of the instances, such as doing queries to find instances.

The second main function of a controller is providing a view of the set of instances of a class that are available to a particular company. Controllers isolate their user from the complexities of how the available instances are determined. Only during setup and maintenance, is it necessary to know what the particular controller being used and where instances actually reside.

A controller is an object that owns a collection of other objects. The other objects are of the same class (or at least the same superclass). A controller appears in a similar role as the master data table in a traditional system. All SanFrancisco controllers are implemented to hold one or several collections internally but present an external interface that hides the complexity of the implementation detail.

5.3.2 Performance Impact

Controllers offer a lot of functionality, but in the mean time, they also bear some performance risks. Especially the use of aggregating controllers (the second main function) can be very expensive to use. This functionality offers the possibility to ascend in the company hierarchy to inquire controllers at a higher level in this hierarchy in the case that a requested object was not found by the controller at a lower level in the hierarchy.

In the case of aggregating controllers, the links of the chain are not direct. Instead, an aggregating controller accesses its parent by first going to its associated Company (for which a direct link is held), accessing the parent Company through the organizational tree structure of Companies, and using the PropertyContainer interface of the parent Company to request the parent Controller.

In the case of a distributed Company hierarchy, this can be slow since it involves a lot of remote calls. Certainly, in many cases the higher controllers need to be accessed because the requested object is not found in the addressed controller.

5.3.3 Controllers without ExtentCollection

Controllers typically own objects that do not have a natural owner. Currency objects are an example. They are stored in a controller, which becomes a property on the Enterprise. In the cases where not a lot of objects will be controlled, it is overkill to use an EntityOwningExtent as the collection type to store the objects. Another type of collection might be more appropriate. It is easy to replace the implementation of the controller so that it uses another collection. What is lost is the query efficiency of normal controllers.

The steps needed to change the implementation of a normal EntityOwningExtent to an ExtentOwningMap are included in the following section on partitioning controlled Entities.

5.3.4 Partitioning Controlled Entities

When SanFrancisco is used in a highly distributed environment, could become necessary to have different objects of a class reside on different servers. The default controller pattern forces all instances that it controls to reside in the same container, which also means on the same server. In the case of an environment with several server nodes each representing, for example, a different company in the same hierarchy, all these companies need to reside on just one server. Since the other servers need frequent access to their company, this results in frequent remote calls. This is bad for performance.

There is a way that a controller is able to control Entities that reside in different containers. The following instructions take as an example the partioning of Company objects. The solution involves replacing the implementation of the CompanyControllerImpl class and replacing it with another implementation. These are the steps to follow:

1. Create an interface called XyzCompanyController that extends CompanyController. This class does not need to define any methods.
2. Create an implementation called XyzCompanyControllerImpl that extends DescribableDynamicEntityImpl (this is the important part since it bypasses the implementation provided in the com.ibm.sf.cf package) and implements XyzCompanyController. The simplest way to create this class is to copy the CompanyControllerImpl in a com.ibm.sf.cf package. After copying the class, change the inheritance as previously described, change the package, and locate all occurrences of the class EntityOwningExtent. The solution to the partitioning problem is to replace all occurrences of EntityOwningExtent with EntityOwningMap. Note, however, that there will be a couple of methods that EntityOwningExtents support that EntityOwningMaps do not. Simply change the method bodies that use these unsupported methods to empty method bodies.
3. Create an interface called XyzCompanyControllerRoot that extends XyzCompanyController. The simplest way to create this class is to copy the CompanyControllerRoot in com.ibm.sf.cf package. After copying the class, change the inheritance as previously described and change the package.
4. Create an implementation called XyzCompanyControllerRootImpl that extends XyzCompanyControllerImpl and implements XyzCompanyControllerRoot. The simplest way to create this class is to copy the CompanyControllerRootImpl in com.ibm.sf.cf package. After copying the class, change the inheritance as previously described and change the package.
5. Create a class called XyzCompanyControllerRootFactory. The simplest way to create this class is to copy the CompanyControllerRootFactory in com.ibm.sf.cf package. After copying the class, change the occurrences to CompanyControllerRoot to XyzCompanyControllerRoot and change the package.
6. Create a special factory for the CompanyFactory. Follow the directions described in the San Francisco documentation. Note that it is important that the location handle parameter is not ignored and is passed to the BaseFactory

when the Company is created. This is how the Company can be directed to the desired Container and Server.

7. Consider defining class replacement for the CompanyControllerRoot, specifying the XyzCompanyControllerRoot.
8. Whenever a Company is created, make sure that the location handle of the appropriate container is passed as a parameter. This ensures the proper placement of the Company object.

5.3.5 DController

Some objects that need to be controlled are not Entities, so they can not be held in an EntityOwningExtent. For these objects, which are mainly Dependents, a specialized controller is developed that uses a DMap as collection to hold these instances. A DController does not provide any query functionality, but it does provide keyed access. This keyed access is different from the one used by the normal controllers, as it is based on Specification Keys instead of the DMethodAccessKey.

Using DControllers could imply a performance risk because it uses a DMap to store its controlled Dependents. A DMap will function well if the number of held items remains reasonable. If too many objects are held in a DMap, performance will degrade rapidly. Typically, they should not contain more than 50 elements.

5.4 Property Container Pattern

A brief description on this pattern is provided together with some advice on the use of properties to get a good performance.

5.4.1 Description

The Property Container pattern is used to make it possible to dynamically add to, and remove, attributes from an Entity. These attributes are called properties and are accessed and maintained by an ID (String).

5.4.2 Performance Impact

The use of a property container should not be very critical from a performance point of view; although, care should be taken when accessing the same property over and over again. Instead of multiple access to the same property, the property should be cached after the first retrieval if possible. Compared to accessing an instance variable that will take some microseconds, accessing a property takes some milliseconds.

Some considerations are important while using the property container pattern. Properties are made persistent by streaming. There is currently no way that a property can be individually schema mapped to the database, as a property is, by definition, unknown to its container. As of version 130, Entities are therefore better performing properties than Dependents and Strings, as these are streamed, as a whole, where Entities are streamed using the Handle only.

The number of properties for a container should be limited as much as possible. The higher the number, the longer it takes to retrieve a property. This degradation is caused by the hashcode table implementation. It may very well be that, even for smaller numbers, the retrieval is slow. A solution to this problem is to change the

hash codes. The better they are distributed, the faster the retrieval process is. The easiest way to get to a better distribution is to use different and more diverse String identifiers.

To help reduce the number of properties, they can also be grouped into a Dependent or an Entity and added as a single property. This is an especially good idea if there are several properties commonly used together.

5.5 Policy Pattern

A brief description on this pattern is provided. The policy pattern is used in many different situations, some of which may be crucial for performance. A practical example on the use of this pattern is provided.

5.5.1 Description

The policy pattern, at its core, is the Strategy pattern from the Design Patterns book by Gamma, et al. It allows volatile business processes (or portions of the application) supported by algorithms to be easily customized. Policies can have different scopes, such as for the whole company or for a particular instance. The framework always includes at least one default implementation for each Policy.

5.5.2 Performance Impact

Default policies, the ones that the framework provides and uses if not replaced, do not always have the most appropriate implementation. In some cases, there are checks performed that are irrelevant to a particular situation. To avoid this, it is necessary to provide a policy that offers a more suited algorithm. Some ID generation policies can be implemented in a better performing way.

5.5.2.1 ID Generation Policy

The number generators that provide objects with unique numbers can sometimes be rather inefficient. Most of these implementations use a NumberSeries to generate the numbers. Often these are synchronized, limiting the throughput in case many objects try to access the ID generator at the same time.

To increase the throughput of a ID Generation Policy, the default implementation can be changed. For example, instead of using a generated ID based on a NumberSeries object, the handle of the Entity could be used. The handle, a 24 byte long unique ID, can be converted to a long instance. Of course, this trick does not always work. Sometimes restrictions are imposed on the sequence of the ID, as they need to follow a well-defined pattern.

In some uses of the Controller pattern, it is mandatory to have a unique key before the Entity object is created in the Controller collection. This is the case when the ID is the primary key in the database. It is not possible to use a ID generation based on the Handle in this case, as the Entity does not yet exist.

The following code illustrates how an ID generation policy can be implemented using the handle instead of a NumberSeries. The method `updateId(GLJournalDissectionController controller, GLJournal journal)` is implemented in the policy that will replace the default policy provided by the framework.

```
public void updateId(GLJournalDissectionController controller,
   GLJournal journal) throws com.ibm.sf.gf.SFException {
      //only retrieve and set new id, if not already set
      if (journal.getId() == null) {
         // get next unique value
         Handle handle = journal.getHandle();
         long id = handle.hashCode();
         String idAsString = FastConvert.longToString(id);
         journal.setId(idAsString);
      }
}
```

This method is called in the method `updateJournalId(GLJournal journal)` on the class `com.ibm.sf.gf.GLJournalDissectionControllerRootImpl`.

5.5.2.2 ID Generation in Batch Process

It may be possible to postpone the ID generation until the necessary resources become available. Consider putting the objects that request an unique ID in a queue where they wait to be processed further. During a low utilization period, a batch process can run that assign's unique IDs based on a NumberSeries.

It is even possible to combine this with the fast ID generation based on the Handle. A temporary ID can be generated from the Handle. Later on, these Entities are retrieved and run through the batch process that will assign them a unique ID based on a NumberSeries.

5.5.2.3 Validation policy

The validation policy is a special kind of policy. It is build into the implementation of an business object. Normally, for many methods that are present, an additional method is implemented that follows the naming convention `validateFor<methodName>()` where `<methodName>` is the name of the method that this validation method corresponds to. For example, a method `calculateMortgage(rate, amount, years)` will coexist with a method called `validateForCalculateMortgage(rate, amount, years)`. This validate method is invoked right in the beginning of the code of the other method. The purpose of the validate method is to check whether all the necessary prerequisites regarding the parameters passed are met.

The validation method returns an DResultCollection in case of a problem and null otherwise. It is more efficient to create this collection only in case a problem occurs and not upfront in the implementation of the validate method. Otherwise, the collection will never be used, as null will be returned in the case that no problems have occurred.

Especially, the `validateForInitialize(parameters)` can have a serious impact on the performance in a rather unexpected way. The initialize method is called by the create methods of the factory. The creation of the objects can be executed on a client process. If, for example, handle objects are passed as parameters, and the `validateforInitialize(parameters)` runs checks on these objects corresponding to the handles, a lot of remote calls can be the result, as the objects will probably not reside on the client process. A better approach is to use a Command object that encapsulates the creation calls and is executed on the server process that contains the objects referred to in the parameter list.

There is a mechanism put in place that will turn the validation on or off. This is discussed in 10.3.3, “Validation” on page 192.

5.6 Extensible Item Pattern

This pattern is heavy in use, and a good understanding of the performance implications is important before using this construct. Therefore, the overview of this pattern is explained. In some cases where not all the functionality of this pattern is needed, alternatives will yield better performance.

5.6.1 Description

The Extensible Item pattern allows the dynamic modification of an Entity's interface. This allows it to appear as if methods appear and disappear from an Entity. Such behavior is especially useful when having an Entity that changes responsibilities as it is processed. An area where this pattern is heavily used is Order Management. The core of each Order object is the Extensible Item pattern.

5.6.2 Concept

There are some important consideration to make when using this pattern. Since they can best be understood if the underlying mechanism is understood, some explanation is necessary.

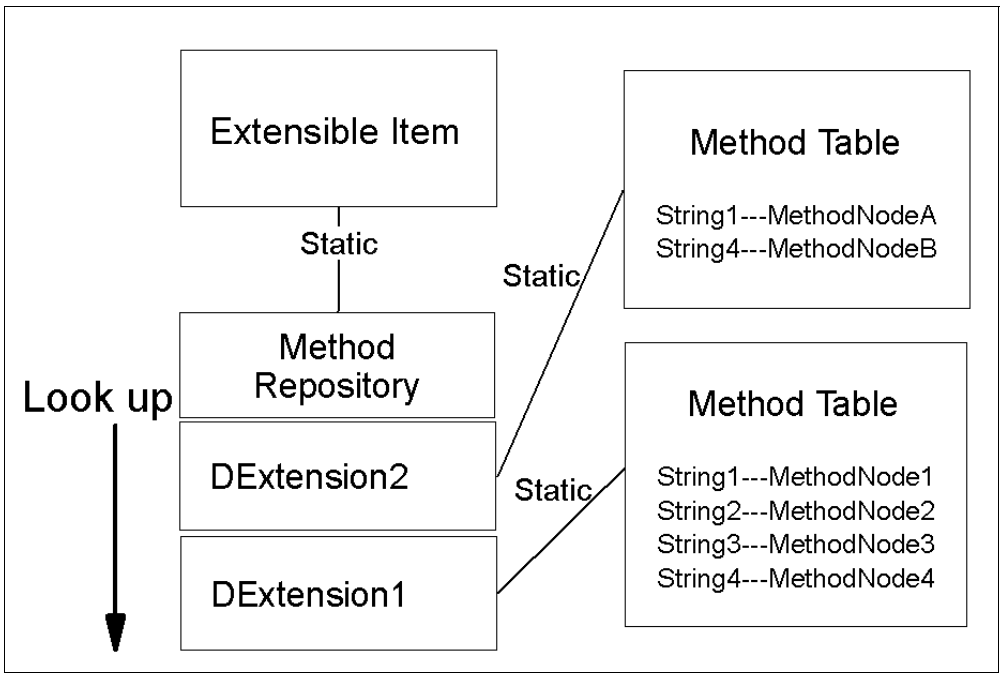


Figure 31. Structure of Extensible Item

Each DExtension maintains a Method table that is a map that holds the method nodes that the DExtension adds to the behavior of the Extensible Item. The keys in this map are the strings that will identify the method. These strings are used in the `invokeMethod()` method to execute the requested method. The method table of a DExtension is created the first time a method on a DExtension is called through the `invokeMethod()` interface on the Extensible Item; the next instances of the

DExtension already have the method table. This shortens the creation time of an extensible item that will maintain a DExtension.

The method execution follows the order that the DExtensions were added to the Extensible Item. The string ID that identifies the method that is requested is looked for first in the last added DExtension. This enables behavior being changed when other DExtensions are added to the Extensible Item. In Figure 31 on page 77, DExtension2 changes behavior by overriding String1 and String4.

5.6.3 Performance Impact

The largest trade-off when using this pattern is one of flexibility versus speed. It is a heavy construct that imposes quite some overhead. There have been internal changes (the static method repository) made to the implementation that makes the creation time shorter. This will definitely have a good influence on the creation time of objects that use this pattern. In the mean time, the path length for execution has increased slightly due to the additional level of indirection.

The fact that the method lookup goes through the stack from top to bottom can have a serious impact. In most cases, the first DExtension implements the most of the added behavior, so it is used the most frequent of all although it has the longest path length. The more DExtensions are pushed on the stack, the longer it is takes to execute a method on the first DExtension. There is no way to force a lookup to start other than from the top of the stack. If there is no need to override behavior, make sure that the most used DExtension is on the top of the stack. Also remove the DExtensions that are no longer useful.

The invocation of a method on an Extensible Item, on average, is about 20 times slower than a Direct Method call on an object. If repetitive method calls are needed on an extensible item, caching should be used, which avoid the repetitive slow invocation.

5.6.4 Alternative for the Extensible Item Pattern

If an object changes behavior throughout its life cycle, the Extensible Item is a powerful mechanism to accommodate these requirements. It is probably the best way for implementation. The Order object in the OrderManagement tower is definitely a good example. Without the Extensible Item pattern, it is not possible to offer all the required functionality.

There exist some situations where this pattern can be used, but are not indispensable. Consider the example of business roles. A person object can have the customer role and the supplier role at the same time. It is possible to design this problem while using the Extensible Item pattern.

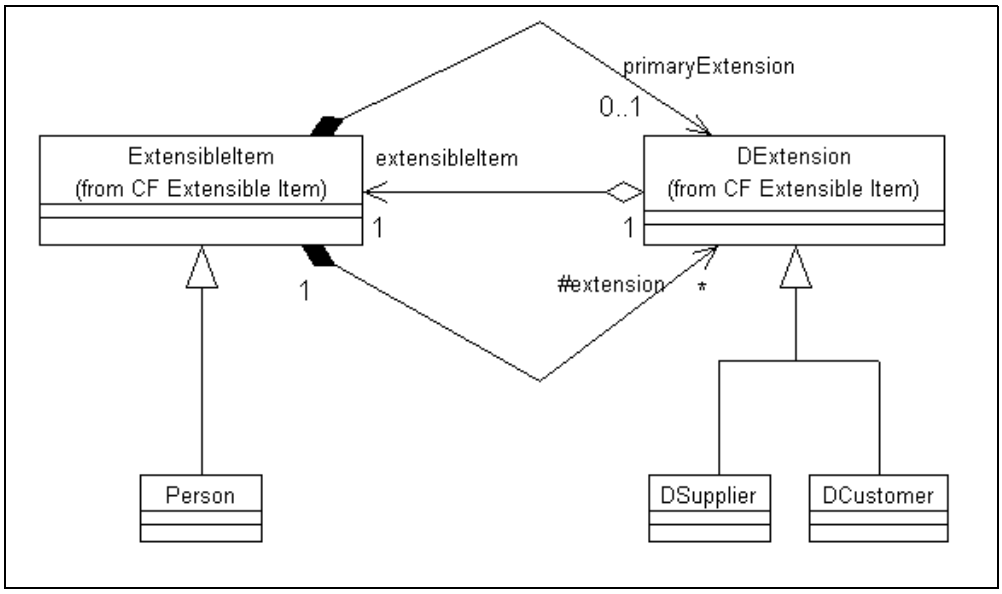


Figure 32. Solving the Problem with Extensible Item

There is another alternative that probably may perform better: using aggregation. Instead of letting the Person inherit from Extensible Item and the business roles from DExtension, Person can hold two instance variables that can contain the business role objects if needed.

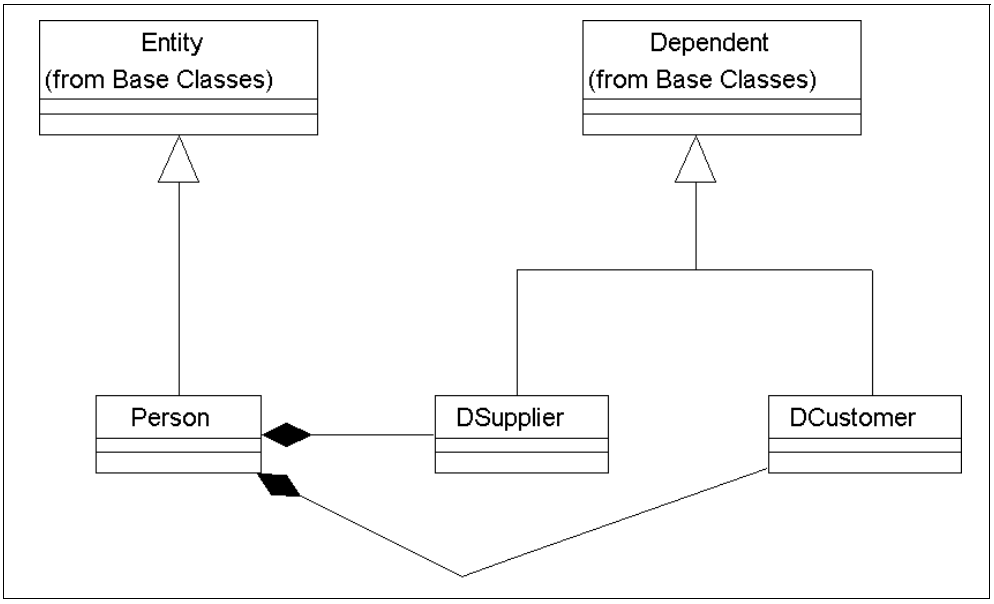


Figure 33. Solving the Problem with Aggregation

It should be clear that this design offers less flexibility but will definitely be more performing. The lesser flexibility shows up in the addressing of the business role. It is needed to have knowledge on the role that is played in a particular situation. It is not possible to add different roles other than the ones designed.

5.6.5 Replacing an Extensible Item Implementation

This section deals with how an implementation of an Extensible Item can be replaced by a more static approach as was described in 5.6.4, “Alternative for the Extensible Item Pattern” on page 78, without changing the client interface to the (alleged) Extensible Item. The recommended way of using the Extensible Item is through the Adapter approach, which eliminates the use of the `invokeMethod()` interface. This method should never be called directly by client code.

5.6.5.1 Adapters

A DExtension can hold all the functionality that is described by an Interface, but the Extensible Item that holds that DExtension does not implement that Interface directly. This means that an Extensible Item can not be cast to one of its DExtensions. In the example described earlier, a Person object can not be cast to a Supplier although it actually holds all this functionality. To solve this problem, an Adapter class was introduced. A DExtension can hold an Adapter that implements the proper Interface. When the Extensible Item is cast to one of its DExtensions, this Adapter is in fact returned that will act as the Extensible Item being this DExtension. The casting of an Extensible Item to one of its DExtensions is done through a method `castTo("DExtension_string")`.

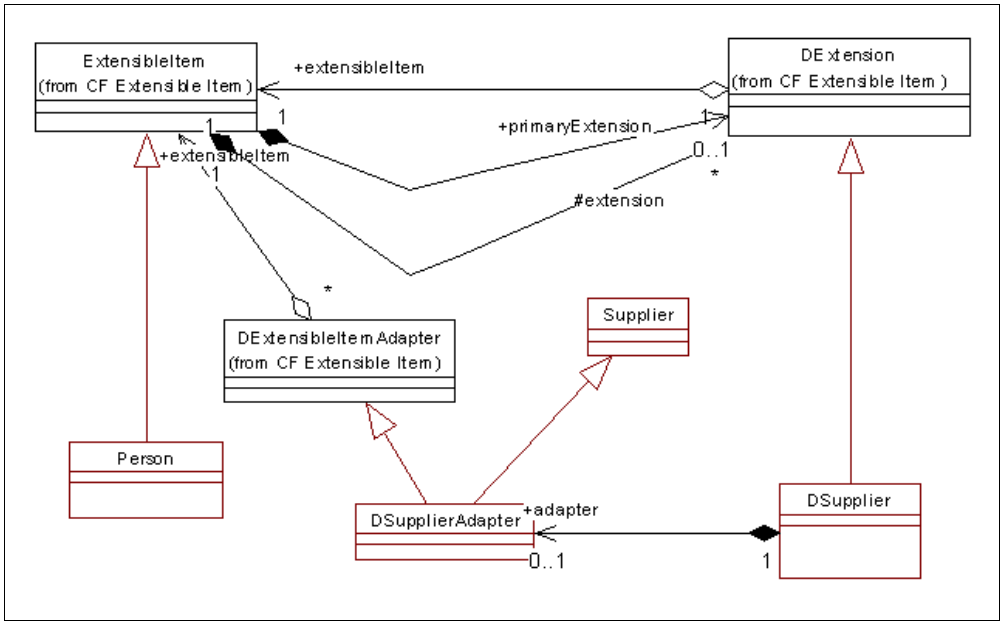


Figure 34. Adapters in the Extensible Item Pattern

An example can make it more clear. When in Figure 34, the method castTo("DSupplier") is invoked on the Person object, the object that is returned is a DExtensibleItemAdapter object that needs to be cast to Supplier. This DSupplierAdapter object implements the Supplier Interface that will define the functionality that the DSupplier extension also holds.

At this point, it is possible to call methods directly on the DSupplierAdapter as if it was the real Person object that was casted to Supplier. These calls do not use the invokeMethod() redirection mechanism directly.

5.6.5.2 Changing the Implementation

If the Adapter approach is used for an Extensible Item implementation, it is possible to change this implementation using a more static implementation.

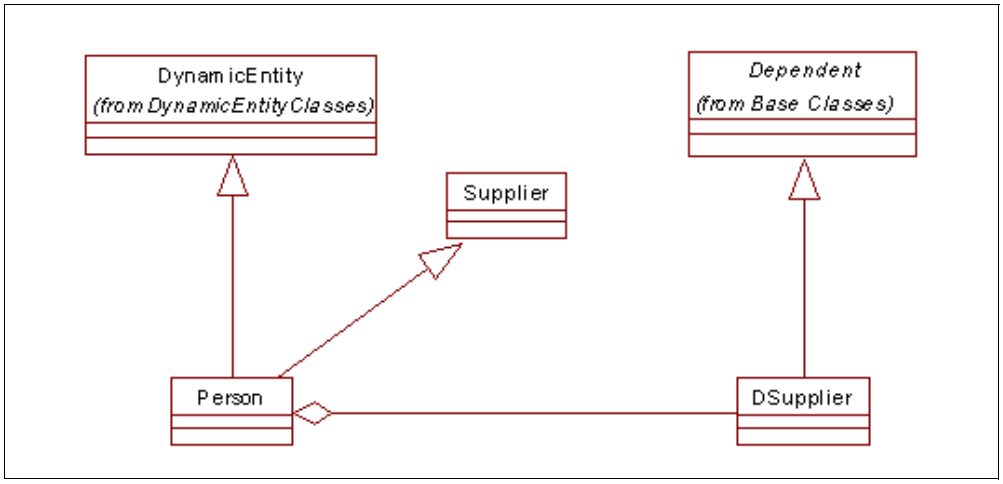


Figure 35. Static Implementation of the Adapter

Figure 35 illustrates the model that implements the same behavior proposed in Figure 34 but only without the Extensible Item pattern. The Person class needs to implement the Supplier Interface, but redirects the actual implementation to the contained DSupplier class. The Person class also needs to implement the `castTo(String)` method that will simply return this.

A more elaborate example that contains additional instruction can be found in the Warehouse Tower in A_Picking Extension.

This static implementation has a better performance than the implementation with an Extensible Item. It is advisable to be sure that the Extensible Item is creating a performance problem. Before the implementation is changed, run a profile on the code. It may very well be that the overall performance is only slightly increased by the static implementation, as the real bottleneck can be elsewhere.

5.7 Life Cycle Pattern

This section deals with the Life Cycle Pattern. A general overview is provided together with some advice on using this pattern from a performance point of view.

5.7.1 Description

Certain entities in the business domain, such as orders, traverse through a set of states during their existence. The traversal from one state to another is generally triggered through some external event, such as an order taker confirming an order. Quite often the states through which such a business entity may traverse and the events that cause these traversals vary from company to company and even within different types of the same business entity within the same company. Modeling of such business entities in a software application requires that the business object used to represent the business entity be flexible. It should allow for the various state transition paths that may be valid for different variations of

the actual business entity as well as to allow for the various events that may trigger these transitions.

The combination of the graph of acceptable state transitions of such a business entity, the events that trigger each state transversal, and the effects of each state transversal on the entity itself comprises the "lifecycle" of the business entity. Traditional approaches for representing business entities with complex state transitions (for example, the "State" pattern - see DesignPatterns, Gamma, et al.) require that the lifecycle of a business entity be integral to the business object that represents it. With this approach, changing a business object's lifecycle requires code changes to the class of the business object. This severely restricts the flexibility of the business object in adopting different lifecycle behavior. The approach used in the SanFrancisco business process components allows for flexibility in the definition and maintenance of a business object's lifecycle by placing the lifecycle behavior into a separate, dynamically-configurable object, thus allowing changes to the business object's lifecycle to be made independently of the business object itself. In the business process components, business entities with lifecycles are represented as LifeCycleManagedItem (LCMI) objects. Each LifeCycleManagedItem is associated with a LifeCycle (LC) object that encapsulates the lifecycle of the associated LifeCycleManagedItem objects. The ConditionChangeResult object encapsulates the necessary actions to be performed on the LCMI object when the corresponding event is signalled. These actions can include invoking methods and adding or removing Extensions.

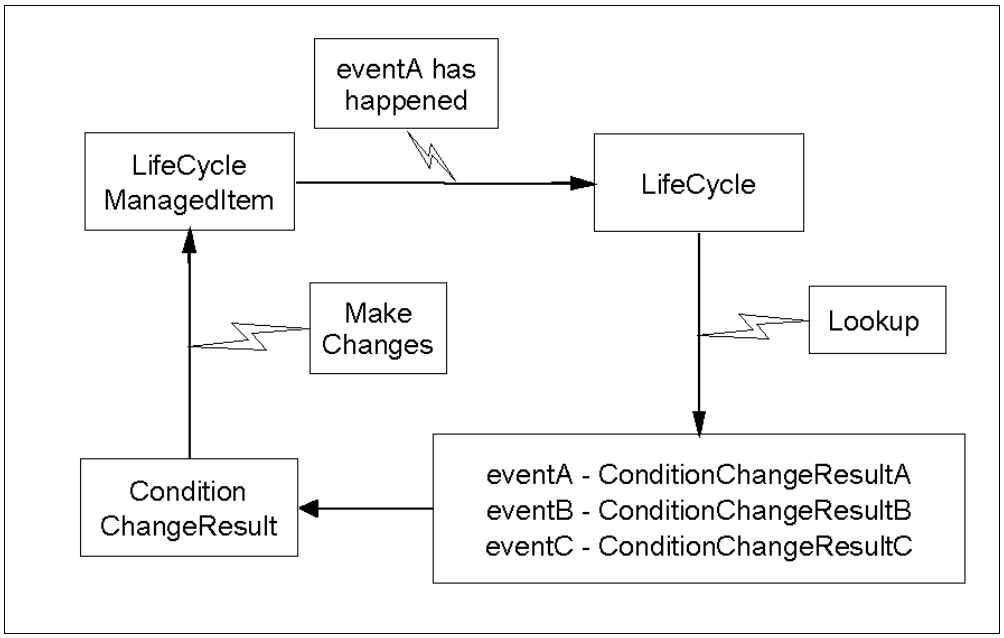


Figure 36. General View on the Life Cycle Pattern

5.7.2 Performance Impact

Although the lookup in the map to find out what changes needs to be performed on the LCMI object when a transition event is signalled to the LC object is very well performing, it is possible to avoid this lookup. The class that is used by default is DConditionSetKeyedLifeCycle. This class implements the lookup using a keyed query as described in Figure 36. It is possible to create an other subclass of the LifeCycle class that will have all the necessary transitions hardcoded using

conditional statements. This method is only valid while using the LifeCycle Pattern for developing additional LCMI objects. The LC, how it is used by the framework itself, should not be changed.

The transition IDs used (known as "conditionid's") should be selected wisely. Unnecessary lookup needs to be avoided. Also, the methods that signal transition events should be limited. Only the methods that actually can cause a valid transition should include the signalling.

In the case where it is not possible to avoid signalling of unnecessary events, the conditionid to condition map in the LC can be used as a filter to eliminate unnecessary LC state change lookups.

5.8 Cached Balances Pattern

The Cached Balances pattern is based on an underlying pattern: *Keys* and *Keyables*. Cached Balances are powerful but can impose a serious performance degradation. A possible alternative for using this pattern is proposed.

5.8.1 The Basis: Keys and Keyables

An application needs to work with data that can be identified by a set of otherwise unrelated items. This set of identifying items must be flexible so that items that are not always available or not of interest can be excluded, and new items can be added as additional identifying items. Within SanFrancisco, the set of unrelated items is called an Access Key, and the unrelated items within an Access Key are encapsulated in Access Keyables. In practice, the items are, in most cases, related and of the same type.

In addition to needing to be able to use Access Keys with particular unrelated items, applications also need to be able to specify groups of particular unrelated items. This is necessary to allow specification of what particular values can be used for each of the unrelated items. This specification is done using a Specification Key with associated Specification Keyables.

Using this pattern yields the best performance if the size of the keys used remains reasonable.

5.8.2 Description

The Cached Balances pattern provides the mechanism for aggregating (adding together) a set of things and keep this aggregation around so that they can be accessed quickly, or that aggregations that are a subset of it can be calculated quickly.

In order to cache aggregates, some means of identifying which set of criteria are of interest, and some means of capturing the aggregate value associated with each set of specific criteria is needed. This is done by using key/keyables. The Specification Key is used to define the criteria of interest. The Access Key is used to define a set of specific criteria. The Access Keys are mapped to the aggregated value for that specific criteria. The combination of a Specification Key, and the map from the Access Keys to their aggregated values, is called a Cached Balance Set.

5.8.3 Performance Impact

The Cached Balances pattern is a rich mechanism that offers a lot of functionality. Some considerations and advice can be useful while using this pattern.

5.8.3.1 Number of Maintained Cached Balances

There is a minimal and a maximum amount of cached balances that will define the useful area. Since there is a big overhead, even if there is just one Cached Balance to maintain, a minimum number of Cached Balances is needed to compensate the performance decrease. It is also needed to maintain them with the increase of functionality and improvement of response time when querying the cached balances.

On the other hand, there is also a limit on the maximum number of maintained Cached Balances that yields good performance. The time spent in updating all the different caches will simply become to long. The performance increase gained from easy querying is lost due to the performance decrease suffered from updating.

Some aggregates can be derived from others. For example, a cached balance set that aggregates over the color and the size of a product will also be able to calculate easily aggregates over only the color of a product. This process is called condensation. It is a way to limit the number of maintained Cached Balances, as some aggregates are already contained in others.

Figure 37 gives a representation of these situations. The minimal and maximal values for the number of maintained Cached Balances are impossible to determine empirically, as the nature of the balances and the frequency of updating and querying are important in this matter.

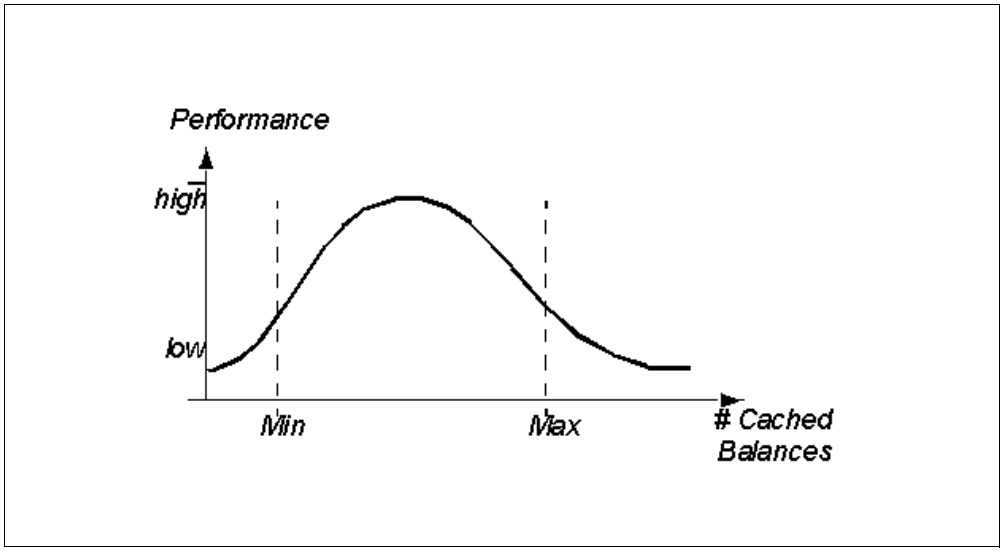


Figure 37. Performance with Cached Balances

5.8.3.2 Granularity of the Cached Balances

A Cached Balance should not be too specific. The more specific it gets, the more it starts reassembling the actual data. For example, if every product in your store has a different color-size-weight combination, it is of no use to have a cached balance that makes aggregates on every color-size-weight combination. This only

duplicates the information that is already present. What makes sense is to use aggregates, for example, on the different colors.

5.8.3.3 Asynchronous Use of the Cached Balances in GL

Cached Balances are used a lot in the General Ledger tower. As they may effect performance, a special mode of operation is introduced. It is possible to run the Cached Balances for the Dissections in an asynchronous mode. This means that, while posting journals, the dissections that compose these journals do not get aggregated in the Cached Balances right away. They are queued up until the Cached Balances are requested to synchronize.

The classes that play a major role in the way journals will be processed are:

- GLJournalDissectionControllerRootFactory
- GLJournalDissectionController
- GLCachedBalances
- DGLCachedBalancesBookingPolicy

In most cases, the GLJournalDissectionController is created before the GLCachedBalances. So, even while the create method on the class GLJournalDissectionControllerRootFactory takes optional parameters for the booking policy and mode of booking, these parameters are usually not used. These two variables will be set in the GLBalancesSetupCmd that calls the methods `setBookingPolicy(DGLJournalBookingPolicy)` and `setSynchronousBooking(boolean)`. When this second method passes `false` as a parameter, asynchronous mode is used. In this case, the booking of Journals is very fast, but the Cached Balances are not updated. Before the Cached Balances can be used, the method `synchronizeBalances()` on the GLBalances object needs to be called. This is a time consuming operation and should not be used in a interactive mode.

If the GLJournalDissectionController uses a synchronous booking policy, the `synchronizeBalances()` method should never be called. The balances will already be up to date, and it still can be a very time consuming operation.

5.8.4 Alternatives for the Cached Balance Pattern

If only a small number of aggregates need to be maintained, and if they are relatively easy to maintain, it is better to implement a proper caching system that maintains the required aggregates.

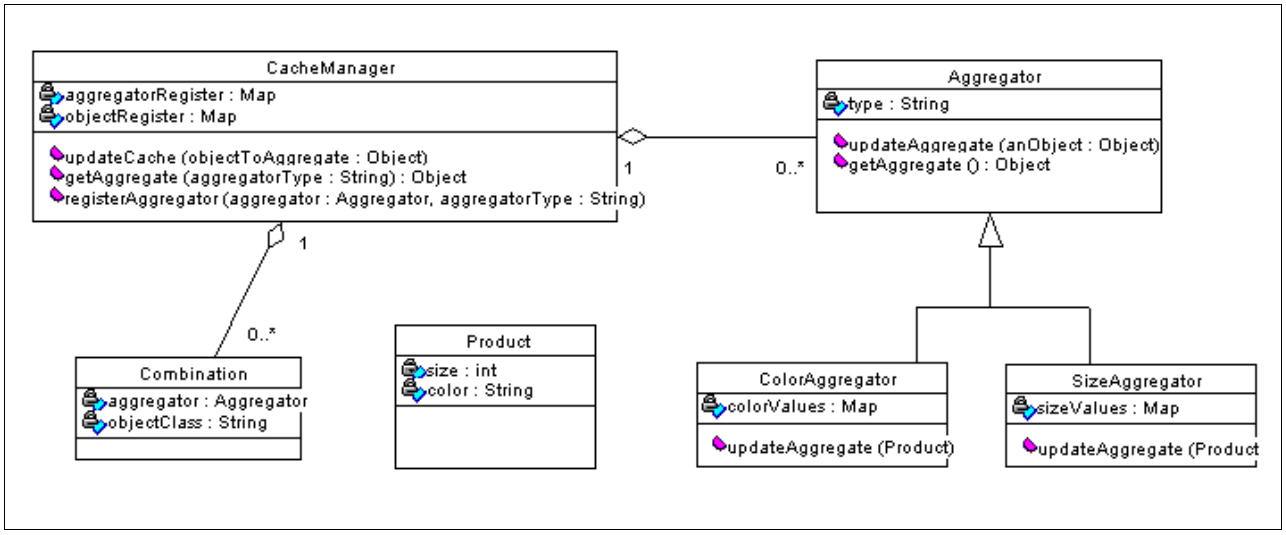


Figure 38. A Cache Manager Structure

Figure 38 proposes a caching mechanism that offers far less functionality than the Cached Balance Pattern. In certain cases, it may just be sufficient. The CacheManager class is the central entry point for the system. The Object Interaction Diagram in Figure 39 illustrates the flow during setup, update, and query.

During the setup phase, the different Aggregators are registered for the different Objects that play a role in the aggregation scheme. In the example, there are two Aggregators that calculate aggregates based on Product objects. The ColorAggregator maintains the amount of each product for every color. The SizeAggregator maintains the amount of each product for every size.

Whenever the CacheManager receives the call `updateCache(Object)`, it determines what aggregator objects play a role in this operation. Then it sends the method `updateAggregate(Object)` to every Aggregator instance that is registered for this type of Object. In the case of Object being a Product instance, the method `updateAggregate(Object)` is called on the SizeAggregator and the ColorAggregator.

Retrieving aggregate results from the CacheManager is done by sending the method `getAggregate(aString)`. The CacheManager looks up which Aggregator maintains the cache values for the identifier that was passed as a parameter and send the method `getAggregate(aString)` to this instance.

It is clear that the caching system proposed here is far from useful off the shelf. The main idea is to give an idea on how a proper caching mechanism could be implemented. It can even be far simpler. An Entity object could just hold on to cached items and keep counters internally as items are added or removed. The mechanisms proposed here are all fixed specification models and are not able to replace the Cached Balances used in the SanFrancisco framework itself where there are requirements for flexibility that are only offered by the Cached Balance pattern.

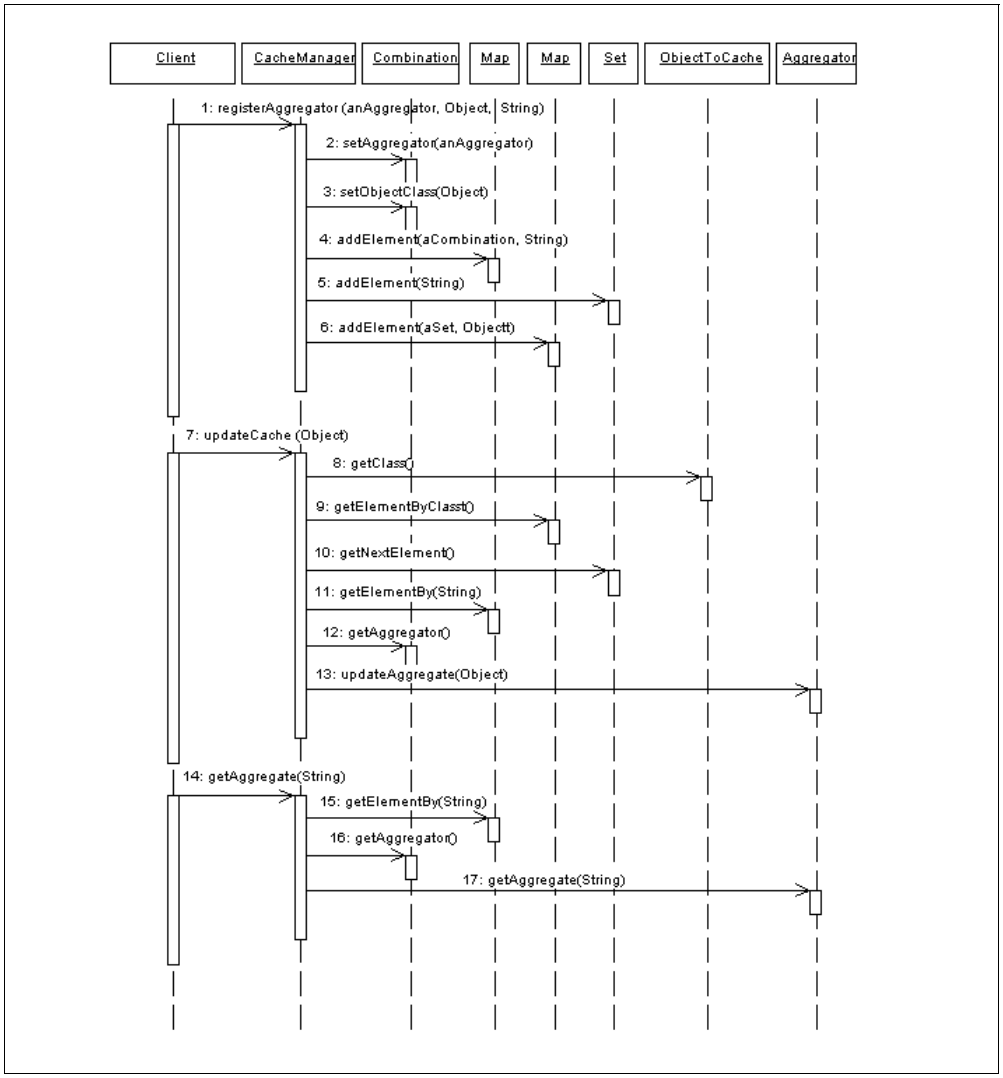


Figure 39. Object Interaction Diagram

5.9 Link Pattern

This section primarily deals with a SanFrancisco specific solution for the concept of a link object.

5.9.1 Link Object

A link object holds information of a link between two objects, where this information does not belong to either one of the objects but only to the relationship between them. In the case of a many-to-many relationship between these objects, a link object is needed. Figure 40 on page 88 illustrates a link object. Students can be enrolled for different classes; classes can have several students enrolled; every enrollment maintains information on its own.

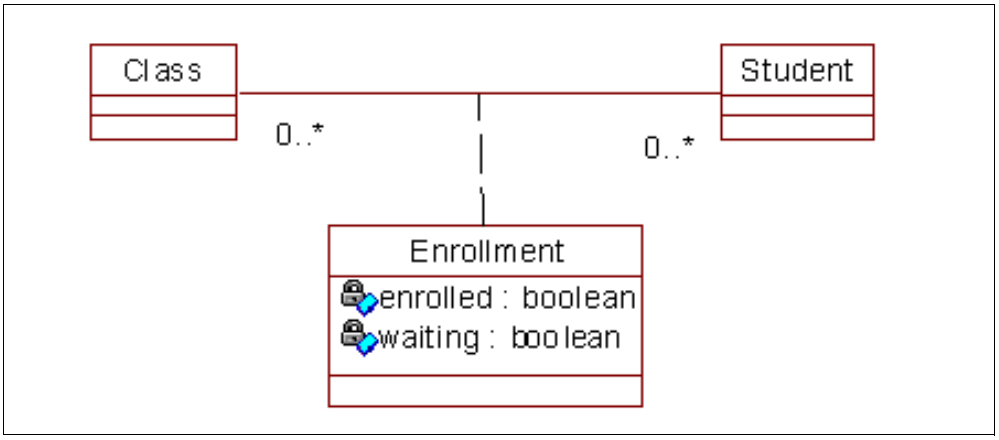


Figure 40. Example of a Link Object

5.9.2 Description

The link pattern that is discussed in this section is based on the concept of a link object but solves a SanFrancisco specific problem. Consider this situation. Product objects are maintained by a controller using an EntityOwningExtent that enables query pushdown on the ID of the instances. Warehouse objects are maintained by a second controller using also an EntityOwningExtent that enables query pushdown on the ID of the instances (as well as other attributes, of course). There exists between the product and the warehouse objects a many-to-many relationship. The following requirement exists: how to process a query that combines a where clause based on the product objects and on the warehouse objects. For example, retrieve the amount of all products that are in stock in warehouse A. The number of products that are in stock in a particular warehouse belongs to the relationship between the two objects. This is the concept of a link object. Figure 41 illustrates this situation.

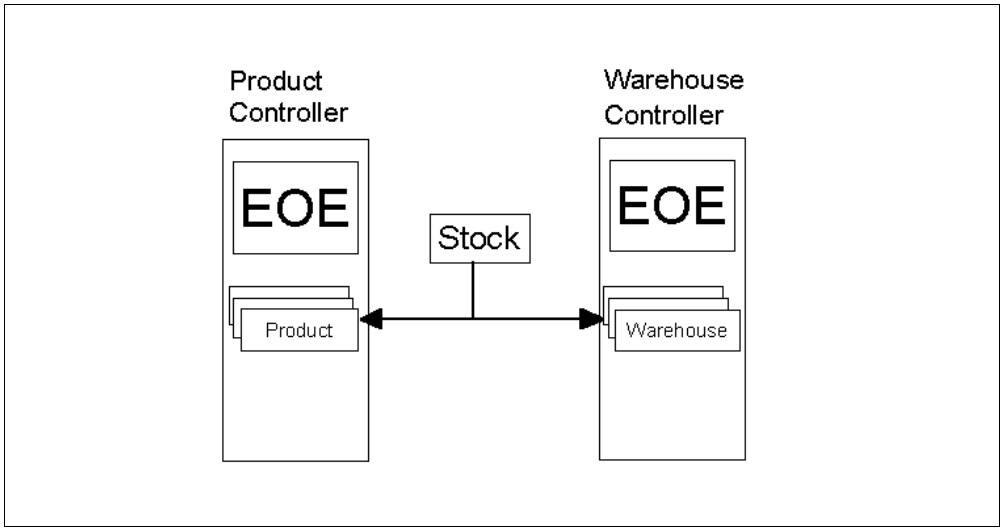


Figure 41. Link Objects in SanFrancisco

5.9.3 Performance Impact

To increase the performance of using a link object, it is important to increase the query performance. This is accomplished through the query pushdown mechanism that is only possible with an EntityOwningExtent. The solution becomes clear: the link objects are maintained by a controller that uses an EntityOwningExtent enabling query pushdown on its controlled elements. Figure 42 illustrates the concept of this Link Pattern in SanFrancisco. For implementation details, look in the package `com.ibm.sf.whs` for the classes ProductWarehouseLink and ProductWarehouseLinkController.

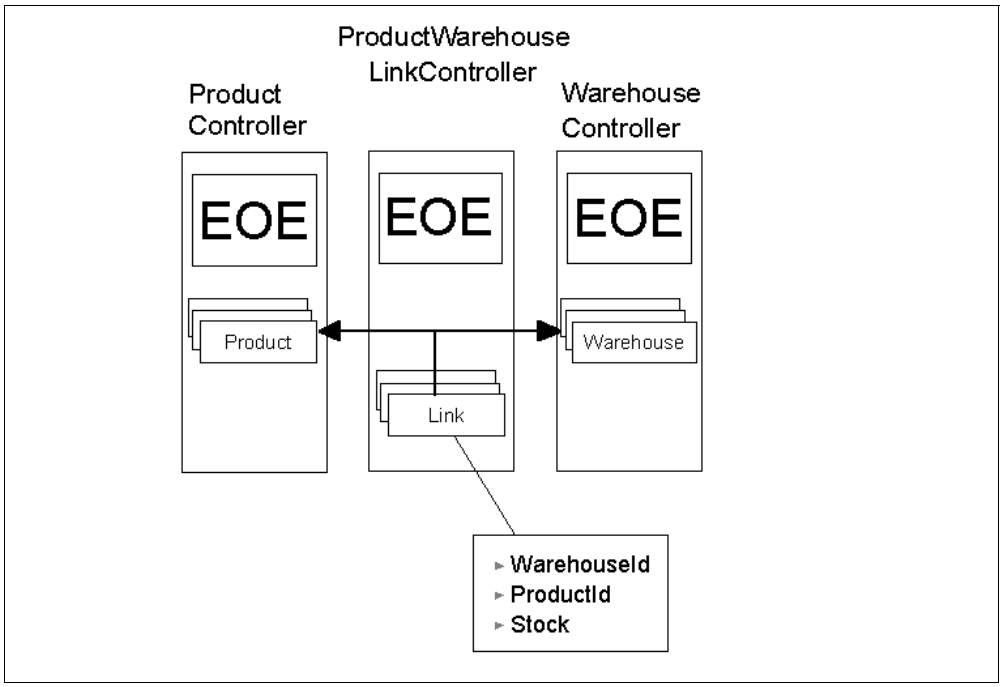


Figure 42. The Link Pattern

Chapter 6. Hardware and Software Configuration

This chapter provides information concerning the optimal configuration of your hardware and software for running IBM SanFrancisco (SF). Because this is a wide area, you may find additional ways to improve the performance in your own environment. This chapter represents the experiences that have been achieved during the development and test of the framework.

Additional information on the configuration of the LSFN (Logical SanFrancisco Network) and databases can be found in the following chapters:

- Chapter 7, “LSFN Configuration” on page 107
- Chapter 8, “Object Persistence, Databases, and Schema Mapping” on page 123

Note

- All information given in this chapter referring to Microsoft Windows 95 should also apply to Microsoft Windows 98 although we have no experiences on this platform yet.
- All information applies to the international English version of the operating system. For localized versions of your operating system, please consult your manual for the specific names used in your version.
- All information given, referring to Microsoft Windows NT, are tested on version 4.0. No tests have been made on version 3.51 or version 5.0. So far, we can not see any reason why the given information should not apply to these versions also. As mentioned before, we have no experiences on those.

This chapter is divided in five parts. The first part deals with the necessary hardware to successfully run a SF application. The second part gives some hints about operating systems; whereas the third part gives advice how to set your JVM correctly. The fourth part deals with communication related topics, such as networks and DNS. The final part of this chapter gives some hints how to run SF on a machine that does not fit the requirements that are stated in this chapter.

6.1 Hardware Recommendations

This section provides you with information concerning the minimum hardware requirements that we recommend for SF. You will notice that our recommendations are different from the minimum requirements, but we find these configurations more appropriate. If your current machine does not fit these requirements, the software may not run smoothly. Nevertheless, there is an additional section with information about running SF on a smaller machine. This should be a last resort solution and is definitely not recommended.

6.1.1 Memory

IBM SanFrancisco runs better when memory that is available. Also, keep in mind that most JVMs (Java Virtual Machine), or garbage collectors, do not tolerate paging. It will have a major performance impact to use paging for objects that are loaded to the memory just for garbage collection. To give you a general idea of

the memory requirements, the following table of the requirements on Microsoft Windows NT can guide you as a rule of thumb.

Table 2. Memory Requirements

System	Memory Required
OS kernel	~16MB
File cache	20+MB
DBMS and buffers	20++MB
SF Name Server Process	~6-16MB
SF BO Process	40++MB

In Table 2, the + sign means "more", whereas ++ means "much more". These numbers are depending on the size of the system and the application. Some of these features are present on every system, others are server related.

6.1.2 Client

A client is a system in a Logical SanFrancisco Network that provides access for the end user. It is playing the client role in a client/server environment. Because the larger part of the workload is concentrated on the server of this environment, including databases and most processing, the requirements for this machine are not very high. In fact, it can be a NC (Network Computer) with no permanent storage at all. Normally, this machine will present a GUI (Graphical User Interface) frontend to the user. For this reason, a good graphical equipment is necessary, such as:

- **x86/Microsoft Windows 95/NT**: 166MHz Pentium equivalent, 64MB RAM, 256 KB Level 2 Cache, 1024x768x256 color display
- **RS/6000/AIX**: Model 340, 80MB RAM, 1024x768x256 color display

Note

- If the application uses a GUI frontend to SF, we recommend an additional 16MB of RAM.
- Any AS/400 system can be used as a client if a GUI frontend is not used.
- If the application uses JavaBeans, the requirements may be higher (at least a 200 MHz Pentium equivalent with 80MB RAM).
- If the application is designed such that most object processing occurs on the client rather than the server, for example AccessMode is set to Local rather than Home, then memory and processor requirements may need to be increased.

6.1.3 Server

A server is a system in a Logical SanFrancisco Network that provides the computing power and data storage capabilities for multiple clients. To do so, the server will need sufficient DASD (Direct Access Storage Device) and processing resources to serve all demands. This need will increase with each additional client. The configuration that is suggested here should be considered as the minimum configuration for basic needs in small client/server environments.

However, since it is not required to run any GUI-based programs, its display capabilities can be less than the client system.

Capable of serving 1-10 client systems running SF applications with acceptable response times (1-10 sec) for typical medium weight transactions, a typical system configuration is made up of these elements:

- **x86/Microsoft Windows NT**: 300 MHz Pentium II equivalent, 512MB RAM, 512KB Level 2 Cache, 1GB free DASD (preferably on a SCSI array)
- **RS/6000/AIX**: Model 43P, 512MB RAM, 1GB free DASD (preferably on a SCSI array)
- **AS/400 system**: Model 170 feature 2160, 1GB RAM, 1 GB free DASD
- **HP-UX**: HP9000/879/K260 Series or higher, 1GB RAM, 1GB free DASD
- **Reliant UNIX (SINIX)**: RM Server with R10000 processor or equivalent, 1GB RAM, 1GB free DASD
- **SUN Solaris**: Ultra SU2-2000 or higher, 1GB RAM, 1GB free DASD

Note

- The size of DASD is heavily dependent on the application and the amount of data that is maintained.
- Depending on the application, for more than 10 clients, you will probably need to add memory on the order of 16-32MB per additional 10 clients.
- For x86-based systems, we recommend Windows NT Server 4.0 instead of Windows NT Client as the operating system.
- For data distribution reasons, multiple smaller disk drives are preferred over a single large one.
- Consider disk "striping", which means spreading data among different drives, either as a software, or preferably, as a hardware implementation.

6.1.4 Development

The development system is used by professionals to develop and test SF applications. The system should be large and fast enough to support sophisticated stand-alone SF applications (for testing, it does not need to support very large databases) and the use of appropriate tools such as IDEs, performance tools, browsers, and so on. For GUI applications, it will also need a sophisticated graphical display. Capable of developing SF applications and functionally testing and debugging them in a limited environment for unit tests:

- **x86/Microsoft Windows 95/NT**: 266 MHz Pentium II equivalent, 256MB RAM, 512 KB L2 cache, 1GB free DASD, 1024x768x256 color display
- **RS/6000/AIX**: Model 43P, 256MB Ram, 1GB free DASD, 1024x768x256 color display

Note

For x86-based systems, we recommend Microsoft Windows NT Client over Windows NT Server as the operating system.

6.1.5 Configuration

There is not much room for configuring the hardware. Assuming that you have the fastest settings for your BIOS and device drivers that give you a stable environment, the first thing is to take a look at the hard disk configuration. If you have more than one physical disk drive, spread applications, and databases, the operating system and swap files among the disk drives to average the disk utilization. This means that you have the operating system on one drive, the swap file on another, and so on. In this case, we are talking about physical drives and not multiple partitions on one drive. You can also use a RAID Level that handles the spreading automatically.

For heavy duty servers, you should invest in RAID SCSI controllers with a large hardware cache with at least 32MB. For smaller systems, a cache of 4MB will be sufficient. This controller will dramatically increase the I/O capabilities of your system. IBM SanFrancisco applications that process a lot of persistent objects make heavy demands on the I/O bandwidth of the server. If with a lot of clients, performance is slow, it is probably the I/O bandwidth that needs considerations.

To detect possible hardware problems, such as "insufficient hardware", you can consult the performance analysis tools that are shipped with your operating system or provided by third party vendors. Analysis tools are available for nearly every platform. Refer to the manual of your specific operating system for detailed information. Additional information is described in the following section.

6.2 Operating System

This section outlines some optimizations that are operating system dependent for getting the best performance possible from your IBM SanFrancisco application. In addition, this section includes all settings that we found to have an effect on the behavior of the software.

Note

For the specific versions of operating systems and other necessary software that are required by the framework, refer to your online documentation that is shipped with IBM SanFrancisco.

6.2.1 Microsoft Windows NT Server

Open the **Control Panel**, open **Network**, select the **Services** tab, select **Server** and click on **Properties**. You will see radio buttons for various options. Select the button titled "**Maximize Throughput for Network Applications**". This option prevents the file-serving cache from consuming too much of main memory.

Set the initial paging file size to at least twice the size of physical memory and the maximum to at least three times the physical memory. To do so, open the **Control Panel**, open **System**, select **Performance** and click on **Change of the Virtual Memory** section. Then select the drive (please refer to 6.1.5, “Configuration” on page 94), adjust the values, and click on **Set** and then **OK**.

For Windows NT, information about the actual utilization of your machine can be gathered using the TaskManager (press **Alt+Ctrl+Delete**, click on **Task Manager** and select the **Performance** tab). If you find insufficient memory (little or no

Physical Memory available) or processing power (CPU Usage > 70 percent), you may want to upgrade your system. Figure 43 shows an example how the TaskManager can appear.

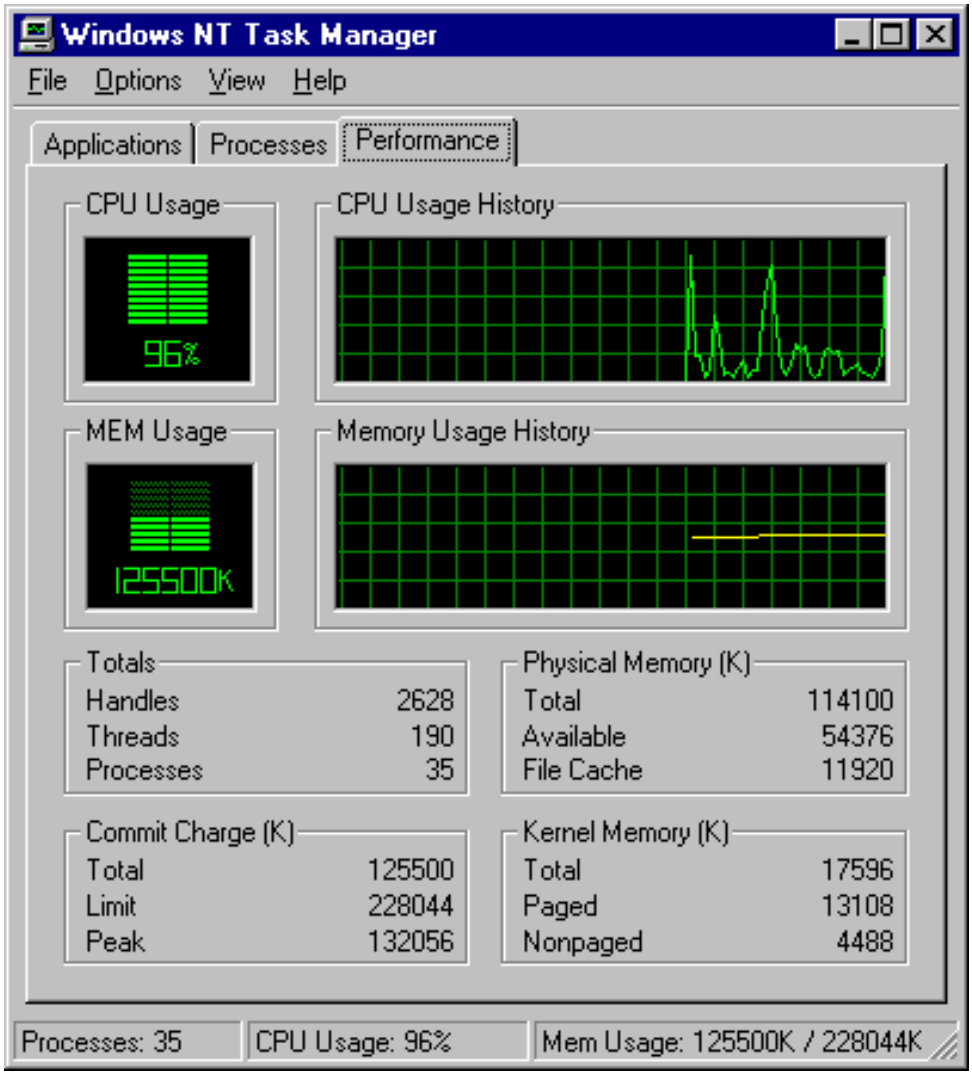


Figure 43. Microsoft Windows NT Task Manager

In this window, the CPU utilization is show as the CPU Usage bar in the upper left corner of the window. The number below the bar shows the actual amount. On the right side of this bar, you find the history of CPU Usage. If you only have some peaks above 75 percent, this is no problem. If the scale stays continuously above 75 percent, the system is low on CPU power. In the lower part of the window, you find the Physical Memory on the right side. If the Available memory is below 16MB on a Windows NT platform, you should upgrade the physical memory of your machine.

If you want to see more information about the system, select the **Processes** tab. This shows you the list of current processes and some values related to those. Figure 44 on page 96 shows an example for this view.

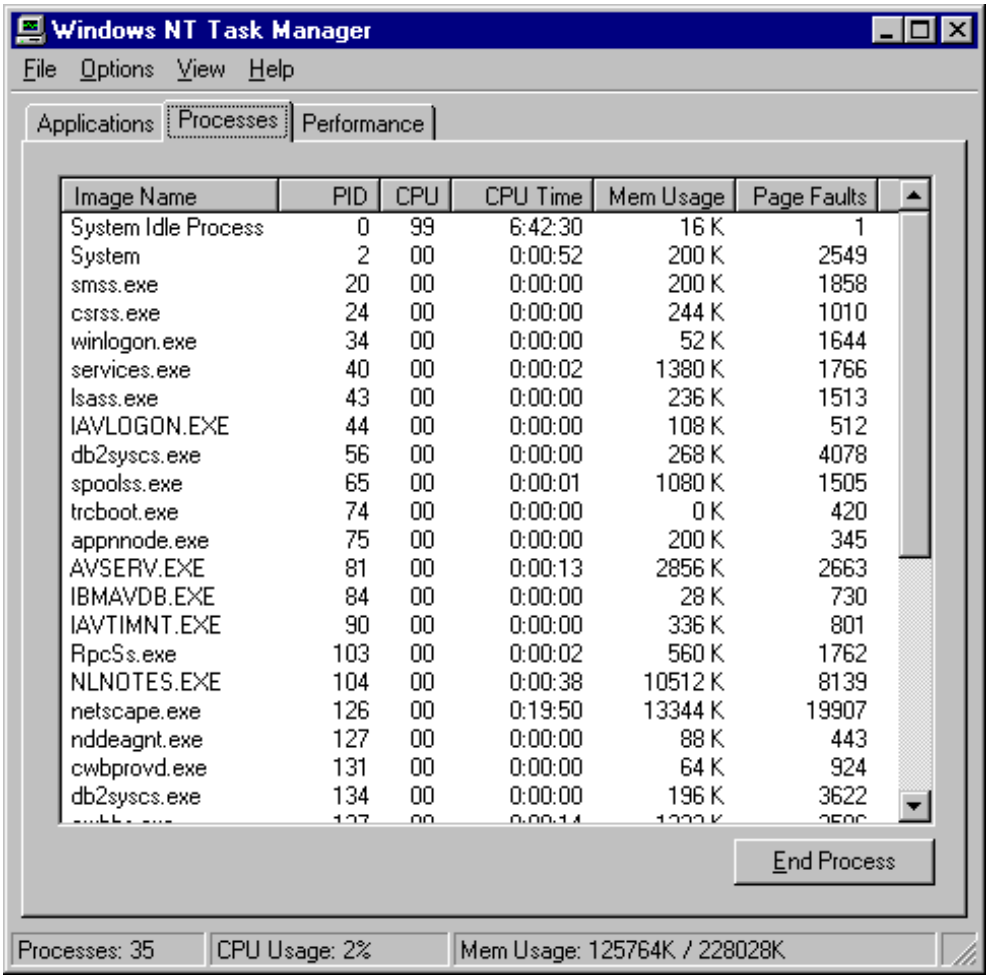


Figure 44. Processes View of the Task Manager

You find the amount of memory that is used by each process, the CPU time spent, and by looking at the Page Faults column, you can figure out the amount of paging. You may have to select additional columns to the default view. You can do so by selecting the **Select Columns** item of the View menu. The window that appears is shown in Figure 45. The default setting, for example, does not include the Page Fault column.

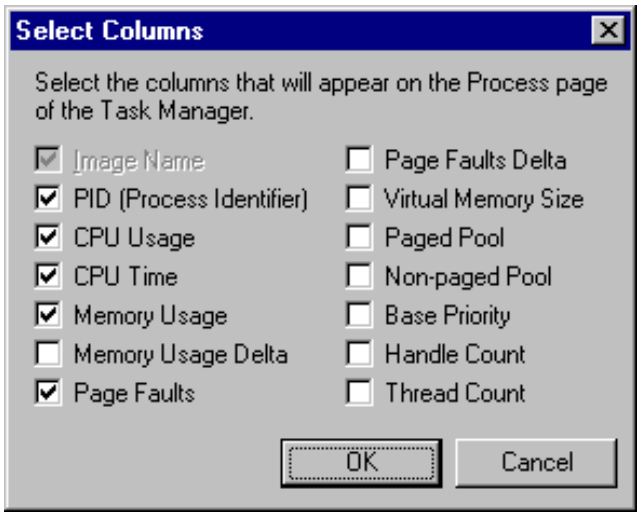


Figure 45. Select Columns View

Furthermore, Windows NT comes with a tool called Performance Monitor that can monitor various system parameters of your local or remote Windows NT systems. Further information on the Performance Monitor can be found in 4.5, “Windows NT Performance Monitor” on page 55.

6.2.2 Microsoft Windows 95

Set the minimum paging file size to at least twice the size of physical memory and the maximum to at least three times the physical memory. To do so, open the **Control Panel**, open **System**, select **Performance** and click on **Virtual Memory**. Then select "**Let me specify my own virtual memory settings**", select the drive (refer to 6.1.5, “Configuration” on page 94), set the values, and click on **OK**.

6.2.3 The AS/400 System

Similar to Windows NT, there is a performance measuring tool set on the AS/400 system. This set consists of the Performance Data Collector (PDC) and the Performance Explorer (PEX) and provides data on several performance issues. A more detailed description of this tool set can be found in 4.6, “AS/400 Performance Tools” on page 56.

6.3 JVM Configuration

This section provides information about the appropriate settings for your JVM to run SF applications. The main goal is to reduce unnecessary garbage collection.

When you run your Java program in a JVM, there are several things you can do to improve performance:

- You can change the amount of memory you allocate to the JVM that is controlled by the *-ms* and *-mx* flags. By default, it is measured in bytes. You can specify the amount in either kilobytes or megabytes by appending the letter "k" for kilobytes or the letter "m" for megabytes.
  - The *-mx* flag specifies the maximum amount of memory heap Java can use. For example: `java -mx16m` sets the maximum amount to 16MB.

– The *-ms* flag specifies how much initial memory should be allocated for the Java heap. For example: `java -ms16m` sets the initial amount to 16MB.

- You can disable class garbage collection through the *-noclassgc* option. This is recommended for all Business Object processes.

6.3.1 First Steps

When running IBM SanFrancisco applications, or any Java application in general, try to avoid all paging activity caused by the JVM processes. With the current garbage collector technology found in the base JVM, paging during garbage collection is extremely detrimental to performance. To limit this effect, follow this procedure to set reasonable JVM sizes:

1. Review present settings in the Global Server Manager (GSM), which are defined in the `StartLSFN.bat` file.

   By default, IBM SanFrancisco sets -ms to 6MB and -mx to 128MB. This value should be acceptable for most applications.

2. Analyze Business Object Processes

   Follow these steps to find appropriate -ms and -mx values for your system. For a description on how to obtain the necessary information below, refer to the system's manuals or contact your system administrator.

   a. Determine the main memory size of your system. For example, 128MB.

   b. Determine the operating system and file cache overhead for your system. For Windows NT, this is typically 36MB.

   c. Determine the database manager (DBMS) overhead for your system. For DB/2 on Windows NT, this is typically 12MB. For Posix, this should be ignored.

   d. Determine the size of the database buffers set for your database. For example, 10MB. For Posix, this should be ignored.

   e. Determine the size of the GSM process. For the Quick Sizing, this is 6MB.

   f. Determine the size of the client process or processes running on the server. This is typically at least 16MB for a Java Swing application. If you run the client processes on separate systems, this should be ignored.

   g. Add up the results of steps 2 through 6 and subtract from the value of step 1. Use this value for the -ms and -mx values of your Business Object process. If you have multiple processes, split the value amongst the processes. If this value is small (less than 48MB), see 6.5, "Running IBM SanFrancisco on Small Machines" on page 105. Because the Cache Threshold value has to fit into the heap size, based on the value, the Cache Threshold value may require adjusting to keep the cache within the bounds of the JVM heap. See Chapter 7, "LSFN Configuration" on page 107 for more information.

      Using the values in this example, 128MB - (36MB + 12MB + 10MB + 6MB + 16MB) = 48MB.

6.3.2 Fine Tuning

The values obtained in 6.3.1, "First Steps" on page 98, are sufficient to execute IBM SanFrancisco applications; however, for maximum performance, you should

set this to the actual memory size your application will require. To determine this, perform either of the following options:

- Run your application for an extended period of time, specifying the *-verbosegc* option when invoking the JVM. This can be done for the GSM, Business Object, and client processes. The JVM will display information when garbage collection is performed. If there is very little activity displayed after running the application for an extended period of time, the -ms value used for the process can probably be decreased. If there is an indication of the heap being expanded, the -ms value used for the process should be increased. However, if the activity that caused this expansion does not typically occur, the value can be left as is.

```
GC: managing allocation failure. need 7816 bytes, type=1, action=1>
GC: freed 10 objects, 2928 bytes in 4 ms, 6% free (3288/48328)>
GC: init&scan: 0 ms, scan handles: 2 ms, sweep: 0 ms,compact: 2 ms>
GC: managing allocation failure. need 7816 bytes, type=1, action=2>
GC: managing allocation failure. need 7816 bytes, type=1, action=3>
GC: managing allocation failure. need 7816 bytes, type=1, action=4>
GC: expanded object space by 24576 to 72904 bytes, 38% free>
```

Figure 46. Example for the Output of the -verbosegc Option

- Run your application for an extended period of time using a Java performance tool such as OptimizeIt (see 4.3, “OptimizeIt” on page 34) or JProbe (see 4.4, “JProbe” on page 43). The garbage collection activity, which is indicated as a call to Runtime.gc(), should not exceed 30-40 percent of the time. If the percentage of time exceed this, the -ms value used for the process should be increased. If the percentage of time spent in garbage collection is very small, the -ms value used for the process can probably be decreased.

If your find the free heap size falling below 25 percent, increase the *-mx* value to fit the needs of your application. To do so, it may be necessary to install additional RAM. If this is not possible, you can try to decrease the Cache Threshold value.

6.3.3 Timeout Setting

The JVM always try to free resources that are needed no longer. One type of these resources are connection. They are dropped after some time of being idle. To use the connection again, it has to be reconnected.

Certain SanFrancisco installations may require additional adjustments to the JVM settings due to communication latency and efficiency, especially Wide Area Network (WAN) installations. If you observe slow response times on the client after the client sits inactive for more than thirty seconds, and you also observe faster response times on the client with immediately successive activity, you might be suffering from excessive overhead in the re-establishment of the communication layers between the involved systems. This is a result of RMI's default behavior of closing idle RMI connections after an specified amount of time. In JDK 1.1.6, this default timeout value is fifteen seconds.

In this case, the RMI connection timeout interval, which is defined in milliseconds, can be adjusted when starting all client and Business Object processes in the

network. For the client processes, the following parameter can be added to the Java invocation:

```
-Dsun.rmi.transport.connectionTimeout=3600000
```

In this parameter, 3600000 is one hour. So, for example, the client invocation would look like:

```
java -Dsun.rmi.transport.connectionTimeout=3600000 com.xyz.MyApplication
```

To adjust the values for autostarted Business Object processes, edit the SFConfig.ini file in the root SanFrancisco directory. Modify (or add) a line that defines "theJVM" so that it is similar to:

```
theJVM=java -Dsun.rmi.transport.connectionTimeout=3600000
```

Manually started servers should be started with a command line argument similar to those for the client. It is necessary to restart the servers after setting the system property in SFconfig.ini. If this corrects the problem, the timeout value can be adjusted to a more suitable value. If this does not correct the problem, remove the parameter.

6.3.4 The AS/400 System

The AS/400 system supports the -ms and -mx options differently. The GCHMAX (Garbage Collection Heap MAXimum) parameter on the AS/400 system is analogous to the -mx option. On the AS/400 system, this value is by default set to *NOMAX. The GCHINL (Garbage Collection Heap INitiaL size) parameter on the AS/400 system is analogous to the -ms option.

6.3.4.1 Optimization Levels

For best performance, compile your code at optimization level 40. You can do this by using the CRTJVAPGM (Create Java Program) command on the AS/400 system. For example, if your class file is named "myfile.class", use the following command:

```
CRTJVAPGM CLSF(myfile.class) OPTIMIZE(40)
```

The increased optimization level has a strong effect on the performance of the application, especially on computing intensive applications.

Table 3. Impact of Optimization Level on AS/400 System

Optimization Level	Relative Performance
Interpreted	12.95
10	2.65
20	2.02
30	1.73
40	1.00

6.3.4.2 Heap Sizes

The AS/400 system platform has, different to most other platforms, the capability to perform a garbage collection asynchronously without halting the execution of your application. This reduces the impact of garbage collection on the overall performance, but there still is an impact.

Because the AS/400 system operating system does not allocate memory until really needed, you can leave the GCHMAX parameter with its default value of no maximum. However, if the heap size is too small, it may cause unnecessary synchronous garbage collection and impact your performance.

In the same way, the GCHINL has a considerable performance impact. It affects indirectly the frequency of garbage collection by initiating an asynchronous garbage collection each time the total allocation for new objects reaches this value. Because a larger value would increase the time for garbage collection, you should adjust this parameter to the particular needs of your application.

6.3.5 AIX

While executing some tests on the JDK 1.1.6 version on the AIX platform, the setting of *-noasyncgc,* which means no asynchronous garbage collection, helped to increase the performance. This may also be true for your application if you have enough memory or can control the garbage collection yourself by calling the *gc()* method. Normally, this setting forces a synchronous garbage collection, which should not occur, if possible.

6.4 Communication

The setup and installation of the communication protocol has a significant impact on the performance of IBM SanFrancisco applications. Knowing this, analyze and optimize your network before you run any such application.

Note

To reduce the impact of remote calls and the network at all, use the fastest available network topology. At the moment, this is 100 MBit Ethernet with PCI Adapters.

6.4.1 Network Drives

If your application has to be installed on a network drive, you will encounter a dramatic performance loss. This is due to the fact that every single class has to be loaded over the network into the client's JVM. Try avoiding this if you are concerned about performance. Otherwise, if you have no local drive, as with a network station, you should consider this fact during design. In this case, it would be a good idea to reduce the amount of classes to be loaded to a minimum and keep the workload on the server. In heterogeneous environments with both types of machines, you can use a dynamic balance between class loading and server utilization to have an optimal overall performance.

6.4.2 DNS Configuration

You can experience performance problems with IBM SanFrancisco that are related to a missing or malfunctioning Domain Name Server (DNS). Symptoms of possible problems are:

- Every remote method call takes anywhere from seconds to minutes to complete, making IBM SanFrancisco performance appear very poor.
- During installation, security priming fails when the `SFBOProcess1` process fails to start.

- When the Global Server Manager (GSM) process starts, the last line shows a short name, `localhost` or something else, but not the fully qualified name of the GSM system, which should be like `MySystemName.domain.com`.
- The GSM appears to start and run without problems. However, a client periodically fails with a `SmContextInvalidException, MSG_SM_210` message.
- With the evaluator CD, Microsoft Windows 95 systems fail when installed as servers with various symptoms. Windows 95 is not a supported server platform on non-evaluator installs.

The cause of the problem in all of the above cases is a missing or malfunctioning Domain Name Server.

6.4.2.1 IBM SanFrancisco with DNS

San Francisco uses RMI (Remote Method Invocation) for the communication between different systems. In order for RMI, and thus IBM SanFrancisco, to operate properly, the system RMI is running on must be able to resolve IP addresses. More specifically, RMI must be able to retrieve an IP address when it has a system name, and it must be able to retrieve a fully qualified system name when it has an IP address. This function is normally provided by a DNS. If IBM SanFrancisco is installed on a system where a DNS is not available on the network, additional configuration is required.

Most TCP/IP suites include a command, *nslookup*, which can be used to verify that a DNS is available and configured properly. If there are any questions about a DNS on the network, or if strange results occur when running IBM SanFrancisco, *nslookup* should be run. If *nslookup* fails, your network does not have a properly configured DNS, and additional configuration is required, or a properly working DNS must be established on the network.

Correctly Configured DNS Subsystem on a Single Host

This example is the output of a correct configured DNS subsystem on a single computer:

```
c:\>nslookup
Default Server: ns.acme.org
Address: 196.162.0.1

> exit
```

Correctly Configured DNS Server for a Network

This test can be executed from any machine on the network and should be run for each server in your LSFN with the following replacements made:

- `SFServer.DNS` is the fully qualified name for the server. For example, `"coyote.acme.org"`
- `SFServer` is the short name for the server without the domain information. For example, `"coyote"`
- `SFServer.IP.address` is the dotted-quad IP address for the server. For example, `"10.199.17.54"`

The important result from this test is that both `SFServer.DNS` and `SFServer.IP.address` return the same, correct, name and address. That the short name returns these correctly as well is a convenience. Deviations from correct

operation should be referred to your DNS administrator. The following example shows a possible output of the *nslookup:*

```
c:>\nslookup
Default Server: ns.acme.org
Address 10.199.17.12

> SFServer.DNS
Server: ns.acme.org
Address 10.199.17.12

Name: SFServer.DNS
Address: SFServer.IP.address

> SFServer.IP.address
Server: ns.acme.org
Address 10.199.17.12

Name: SFServer.DNS
Address: SFServer.IP.address

> SFServer
Server: ns.acme.org
Address 10.199.17.12

Name: SFServer.DNS
Address: SFServer.IP.address

>exit
```

Verifying DNS without a Full TCP/IP Suite

Some TCP/IP suites, for example, Microsoft Windows 95, do not include a *nslookup* program. On these systems, a similar verification can be done using the `ping` command. Issue the following commands from a DOS prompt:

```
c:> ping SFServer.DNS
c:\> ping -a SFServer.IP.address
```

Both of these commands should start by printing a line similar to:

```
Pinging SFServer.DNS [SFServer.IP.address] with 32 bytes of data:
```

Again, the important result from this test is that both `SFServer.DNS` and `SFServer.IP.address` return the same, correct, name, and address.

6.4.2.2 IBM SanFrancisco without DNS

It is possible to use IBM SanFrancisco without a DNS, but additional configuration is required.

On systems using TCP/IP without a DNS server, there is a Hosts file that must be updated with IP addresses and associated TCP/IP names. The syntax for the entries in this file is shown either in a sample file or in comments in the actual file. Updating this file allows the system to properly resolve those IP addresses. For IBM SanFrancisco to operate properly, every server and every client must have an entry in each system's Hosts file for every server and client used in the LSFN. When updating the Hosts file with system name and IP address pairs, make sure to use the system's fully qualified name, such as `MySystemName.domain.com`. The Hosts file is typically located at:

- `C:\WINNT\SYSTEM32\DRIVERS\ETC\HOSTS` for Windows NT
- `C:\WINDOWS\HOSTS` for Windows 95
- `/etc/hosts` for most Unix systems

Be aware that there is also a Hosts.sam file, which is an example file and is never used by Windows.

6.4.2.3 Tuning DNS

If you are running Windows 95 or Windows NT on a network that is heavily loaded, you might be able to reduce the network load and speed up your IBM SanFrancisco application at the same time by modifying your Hosts file. If TCP cannot find the IP address there, it makes a remote call to the network DNS server to find it. You can avoid this call to the DNS server by putting an entry for each IBM SanFrancisco server you access from your local system into the Hosts file.

Other platforms besides Windows have the same requirement that a DNS be available. However, these other platforms usually use a thoroughly designed network that almost always has a properly working DNS. Nevertheless, if a DNS is not available, consult your operating system's network configuration documentation on its equivalent to a Hosts file. Again, every server and client must have a Hosts file-like entry.

Note

The Hosts file has to be maintained manually. If the information in this file is not correct, even an installed DNS might fail. Be careful while using the Hosts file and judge between increased performance and increased maintenance work.

6.4.2.4 Using a Microsoft DNS Server on Windows NT Server

If you plan on using Microsoft DNS Server for Windows NT Server as your DNS, see the instructions below. There are cases where other applications function correctly, but IBM SanFrancisco does not because special steps are required to configure IP address to system name lookup (which is used by IBM SanFrancisco) in Microsoft DNS Server.

Use nslookup to verify if your DNS is already working with IBM SanFrancisco. If nslookup works, the DNS is setup correctly for use with IBM SanFrancisco. If it does not work, the following example will help you to properly configure Microsoft DNS Server. This is only one example of a process that was found to work. Your installation may require a different process.

Before beginning, make sure that DNS server is installed on the Server by opening **Control Panel**, selecting **Network**, and choosing the **Services** tab. If Microsoft DNS Server is not listed under Network Services, click the **Add** button and install Microsoft DNS Server, perform the following steps:

1. Under the start menu, choose **programs**, and then **administrative tools**, then **DNS Manager**.
2. Under the DNS menu, choose **New Server**.
3. Enter the new Server name and press **OK**.

4. Click on the newly created server, then go to the DNS menu, and select **New Zone**.
5. Choose the **Primary** radio button and press **next**.
6. Enter the name of the new Zone and the Zone file and press **Next** and then **Finish**.
7. Go to the Options menu and select **Preferences**. Make sure the Show Automatically Created Zones box is checked and press **OK**.
8. Now create a Primary Zone in the in-addr.arpa Domain. To do this, click on the server icon.
9. Choose **DNS** and **New Zone**, then click on **Primary** and then click **Next**.
10. In the space for the zone name, fill in the network (or use the subnetwork if it is a class B- or C-sized subnet) portion of the IP address in reverse order followed by in-addr.arpa. (For a network with an IP address of 192.168.102.0, the zone name would be 102.168.192.in-addr.arpa)
11. Enter the name of the zone file or use the default.
12. Click on the **Next** button, followed by **Finish**.
13. Now enter host names and IP addresses into the first zone and make sure that Create Associated PTR Record box is checked for each host.

The key for this process to make IBM SanFrancisco work involves steps 9-13. Without a Primary Zone in the in-addr.arpa domain, the DNS will resolve host names that allow programs like Client Access and Lotus Notes to function. It does not, however, provide reverse lookup for IBM SanFrancisco to run properly.

For further information about TCP/IP, refer to *Accessing the Internet,* SG24-2597.

6.5 Running IBM SanFrancisco on Small Machines

If you have a system that has less memory than we recommend for IBM SanFrancisco, there are currently some ways to reduce the amount of memory required. While it is recommended that you add more memory, there are situations, such as demonstrations, where it may be acceptable not to add memory. Nevertheless, in this discussion we will assume that the system has at least 64MB of RAM. If your system has more, it is possible to raise some of the suggested settings, as will be noted. All modifications should be made before priming the system.

Note

The hints given in this section are for the installation phase of SF. This means you have to do the changes before priming. If you want to change anything after priming, you have to use the Configuration Utility. For more information about this, refer to 7.2, “Configuring LSFN for Small Systems” on page 108.

6.5.1 IBM SanFrancisco Container Settings

The first step for making SF run is limiting the container cache settings.

> **Note**
>
> Current limitations in IBM SanFrancisco limit the container caches to a range of 100 to 1,000,000. These limitations may go away in a later release, allowing for a further reduction on some of the container caches sizes.

To limit the container cache, modify `com\ibm\sf\etc\Global.name` in the SF root directory. To change the cache sizes, find the appropriate lines and change them, as shown here:

```
[HOSTS]
     *=*,*,com.ibm.sf.gf.PosixContainer,*,100
[CONTAINERS]
     SFDefaultContainer=*_SFBOProcess1,com.ibm.sf.gf.PosixContainer,*,1000
     GFSecurityContainer=*_SFGSMProcess,com.ibm.sf.gf.PosixContainer,*,100
```

6.5.2 JVM Settings

Change the initial JVM heap size (-ms) settings. To do this, modify `com\ibm\sf\etc\Global.name` in the SF root directory as follows:

Set the Processes settings.

For larger memory systems, the `SFBOProcess1` initial JVM heap size should be set higher.

```
[PROCESSES]
*_SFGSMProcess=6000,*,30000,off,off,off,off,off,default
*_SFBOProcess1=12000,*,30000,off,off,off,off,off,default
*_SFBOProcess2=4000,*,30000,off,off,off,off,off,default
*_SFBOProcess3=4000,*,30000,off,off,off,off,off,default
*_SFBOProcess4=4000,*,30000,off,off,off,off,off,default
```

Chapter 7. LSFN Configuration

This chapter discusses various strategies that can be deployed to optimally configure a Logical SanFrancisco Network (LSFN).

This chapter has the following sections:

- Section 7.1, “Configuration Settings” on page 107—Describes the settings that can be done while setting up LSFN. These include settings that can be performed with the configuration tools.
- Section 7.2, “Configuring LSFN for Small Systems” on page 108—Describes the changes that can be done for small memory systems by specifying appropriate parameters for configuring containers and JVM.
- Section 7.3, “Exploring Topologies” on page 110—Provides some insight into the design consideration while partitioning with containers and also describes some commonly used topologies.

7.1 Configuration Settings

This section demonstrates the configuration settings that can be done with utilities that are a part of the SanFrancisco environment. You can make changes with your SanFrancisco configuration that can significantly improve performance.

7.1.1 Cache Threshold

Each container in SanFrancisco maintains a memory cache where it keeps recently accessed persistent objects so that, when needed, it can retrieve them directly from the memory rather than from the disk. The more objects the container can keep in the cache, the more likely it is that the particular object your application needs will be found in the cache. This can result in dramatic improvement of the application’s performance. The maximum number of objects a container keeps in its cache is a configuration parameter you can control. This can be done in the manner described below:

In the SanFrancisco Configuration Utility (java.com.ibm.sf.gf.CgRunTimeConfigurationView), perform the following steps:

1. Go to the Configure pulldown, then pick **Containers**, then either **Posix** or **Rdb,** depending upon the type of the container you want to configure. This will bring up a window listing the various existing containers of that type.
2. Double-click the name of the container in which you wish to set the cache threshold. This will bring up a window that contains a field for Max. cache size.
3. Set this to the maximum number of persistent objects that you want the container to cache in the memory.

Note

Keep in mind that the cache may consume virtual memory on your system but will make it more likely that your application will be able to find the persistent object it wants in the memory rather than having to retrieve it from the disk. It is best to set the configuration in such a way that the container cache fits in the heap allotted, and that the heap fits in the main memory.

Also, the logging service, and to an extent the security service, are low priority jobs and should be allotted fewer resources than the containers.

7.1.2 Garbage Collection

The garbage collector in JVM is automatically invoked. The user cannot control the activation of the garbage collection. The other major drawback would be if your JVM has a blocking garbage collector, unlike the AS/400 system, which has a non-blocking garbage collector. In blocking garbage collectors, all the threads in the JVM other than the garbage collector come to a halt, and nothing else can proceed if the garbage collection is going on. This could potentially cause a number of problems.

For example, if we run our setup on a single, large system, which contains a large amount of memory, it will run fine for some time. After the heap is filled, and the garbage collector is invoked, it can block the other threads in the JVM for an unreasonable amount of time since a large amount of garbage collection would need to be done. This implies the application will sit idle for this amount of time, and this may not be acceptable for certain applications. In such cases, you can consider running two smaller server processes, each with smaller amounts of memory. In this case, even if one of the server processes were involved in garbage collection, the other could carry on with added load.

It is important to set the correct heap size in the memory. If the heap size is smaller, the garbage collector would very likely be invoked more often, but it would run for fewer durations, and the blocking time would be reduced.

7.2 Configuring LSFN for Small Systems

If you have a system that has less memory than is recommended for SanFrancisco, there are currently some ways to reduce the amount of memory required. While it is recommended that you add more memory, there are situations, such as demonstrations, where less memory is acceptable. As a starting point, the remainder of this discussion assumes that the system has 64MB of RAM. If your system has more, it is possible to raise some of the settings, as will be noted.

7.2.1 Container Settings

Note

Current limitations in SanFrancisco (SF130) limits the container caches to a range of 100 to 1000,000. These limitations may go away in a later release, allowing for reduction in container cache sizes.

Before priming

If you have not primed your system as yet, it is possible to set the initial container cache settings. To do this, modify the `com\ibm\sf\etc\Global.name` as shown here:

1. Set the HOSTS container cache sizes:

```
[HOSTS]
*=*, *, com.ibm.sf.gf.PosixContainer, *, 100
```

2. Set the container cache sizes:

```
[CONTAINERS]
SFDefaultConatiner=*_SFBOProcess1, com.ibm.sf.gf.PosixConatiner, *, 1000
GFSecurityConatiner=*_SFGSMProcess, com.ibm.sf.gfPosixConatiner, *, 100
```

The same changes can be made for other containers you may define. For larger memory systems, the SFDefaultContainer cache size should be set higher using a 2MB per 1000 increment as a rule of thumb.

After priming

If you have already primed your system, changes to the container cache size must be either made through the Configuration tool under the Base Utilities menu or through invoking the DOS batch file `com\ibm\sf\bin\Configuration.bat`

To make changes with the Configuration tool, perform the following steps:

1. Start the **Configuration** tool.
2. From the menu bar, select **Configure**, **Container**, **Posix** (or **Rdb** if you are using database).
3. For each of the specified containers, double-click to edit the container attributes.
4. Change the **Max. Cache Size** to 1000 for the SFDefaultContainer and 100 for the others.

Note

For larger memory systems, the SFDefaultContainer cache size should be set higher using a 2MB per 1000 increment as rule of thumb.

5. Restart all the SanFrancisco processes to accept the modified settings.

7.2.2 JVM Settings

The JVM heap size settings that can be made are discussed in this section.

Before priming

If you have not primed your system as yet, it is possible to set the initial JVM heap size (-ms) settings. To do this, modify `com\ibm\sf\etc\Global.name`, modify the Processes settings. For larger memory systems, the SFBOProcess1 initial JVM heap size should be set higher.

```
[PROCESSES]
*_SFGSMProcess=6000,*,30000,off,off,off,off,off,default
*_SFBOProcess1=12000,*,30000,off,off,off,off,off,default
*_SFBOProcess2=4000,*,30000,off,off,off,off,off,default
*_SFBOProcess3=4000,*,30000,off,off,off,off,off,default
*_SFBOProcess4=4000,*,30000,off,off,off,off,off,default
```

After priming

If you prime your system, changes to the JVM heap size settings must be made either through the Server Management Configuration Tool under the Base Utilities menu or through invoking the DOS batch file `com\ibm\sf\bin\SMConsole.bat`

To make changes with the Server Management Configuration tool, complete these steps:

1. Start up the Server Management Configuration tool.
2. Click on the + sign for each system that is defined.
3. For each of the following processes, modify the Initial Heap Size (in Kilobytes) to the specified amount:

   ```
   SMSFProcess: 6000
   SFGSMProcess: 6000
   SFBOProcess1: 12000
   ```

   Note

   For large memory systems, the SFBOProcess1 initial JVM heap size should be set larger.

   ```
   SFBOProcess2: 4000
   SFBOProcess3: 4000
   SFBOProcess4: 4000
   ```

4. Restart all the SanFrancisco processes to accept the modified settings.

7.3 Exploring Topologies

This section explores various topologies for setting up an LSFN and trade-offs while making a choice for a particular configuration.

By topologies, we mean the physical layout of the Logical SanFrancisco Network in your organization. Before you start exploring the alternatives, you need to first understand the requirements of your application:

- What is the size of the application?
- What geographical boundaries does it span?
- What are the resources (hardware, system software, communication links) at your disposal?
- What are the restrictions on the availability and location of data?

To rephrase these with an LSFN perspective, consider the following points for a configuration of the LSFN:

- How are objects distributed in the system? That is, how many containers do you have and where are they located?
- What are the issues in communication? This would involve communication between objects and also communication for naming lookup and security information, and so on.
- Are you dealing with a single machine or with multiple machines? If you are dealing with multiple machines, then are these machines close together in a LAN, or are they geographically distant, connected by WAN links?

Before you proceed, a general overview of a typical LSFN will prove useful. Figure 47 on page 112 shows such a typical layout. Each of the three big blocks represent three separate nodes (machines) of the LSFN. There is one Global Server Manager (GSM), which has the global naming service. The other two nodes each have a Local Server Manager (LSM), which is for auto starting (and stopping) the processes on the respective nodes. Each of the nodes has been configured to run one or more server processes (BOProcess). Each node has containers, and the containers are assigned to their respective processes. Each server process maintains the following caches:

- Naming Cache—Stores the container/process mapping information of entities it has accessed recently
- Container Cache—Caches the entities it has retrieved from its containers

With this picture in mind, we will now try to explore the answers to some of the question we posed earlier in this section.

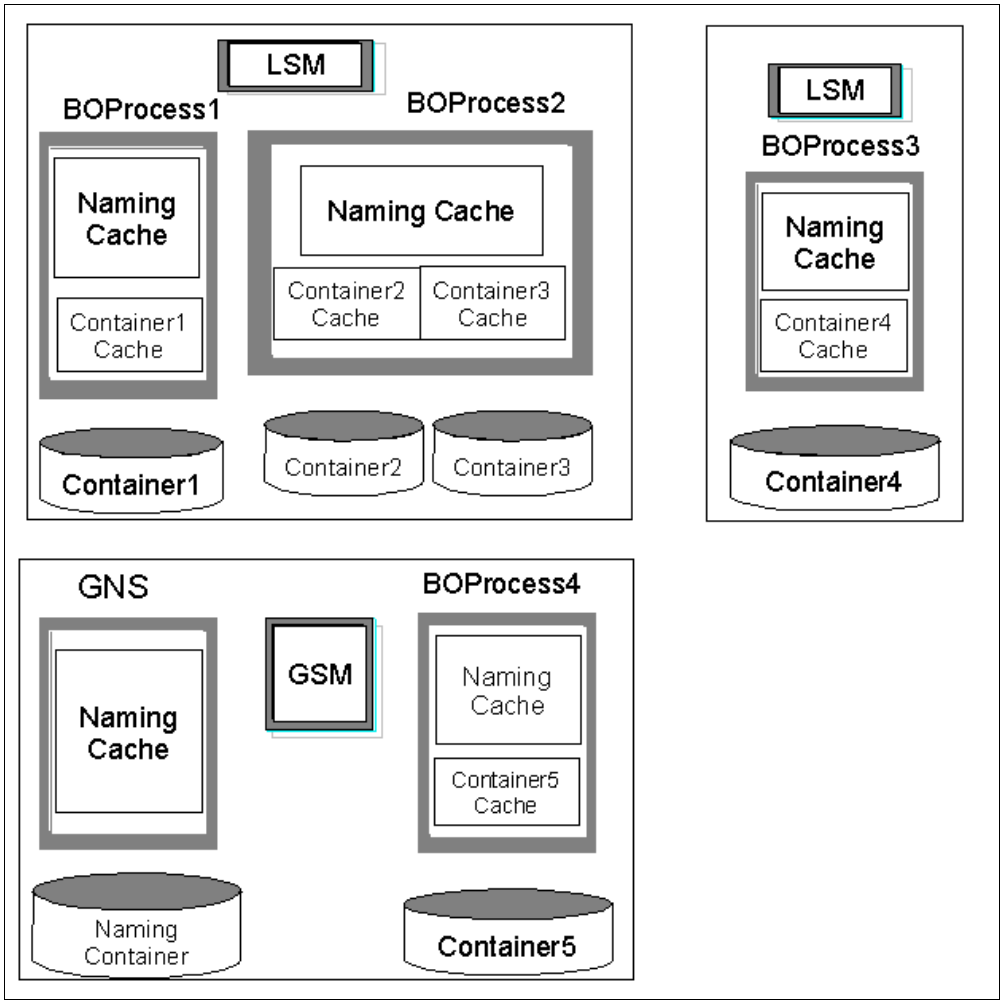


Figure 47. Distribution of Containers and Processes in an LSFN

7.3.1 Data Placement

Data placement refers to the distribution of containers and the issues concerning partitioning of data. There are a number of factors that need to be considered while deciding which data goes where. Some of these include the communication between server processes, the communication between objects, the scope of a given object, and communication with the name server. Accordingly, the containers have to be assigned to the server processes.

A point to keep in mind is that we cannot assume a given object as a black box and just work with its interface. The object may internally be requiring some other objects for a certain operation, and these objects may be located somewhere else. In the worst case, if these objects are called in a loop, then the performance of that operation is reduced drastically, and it may be very difficult to detect the source of the problem.

The partitioning of data can be done based on an acceptable algorithm for categorizing it. Which algorithm is acceptable largely depends on the application with which you are working. Presented below are some categorization techniques.

For information on partitioning of collection of entities, refer to 5.3.4, “Partitioning Controlled Entities” on page 73.

This shows the typical layout of an LSFN that encompasses multiple machines. In the TCP/IP sense, each of the three big blocks can be considered as three nodes. Each node has an LSM.

7.3.1.1 Categorization Based on Latency and Transfer Rate

This is mainly categorization based on the communication aspects. While distributing objects, we need to ensure that objects that work together are closely located. Communication between the objects could be considered at various levels of granularity:

- Communication between objects used by same server process and held in the same container, depending on the amount of caching, this can be of the order of a few micro-seconds.
- Communication between server processes that are on same machine - effectively, between objects used by different processes, given that the processes are in the same machine. This could be of the order of milliseconds.
- Communication between server processes on different machines - depending on how far the machines are located and the nature of the communication link, this could well be of the order of a few hundred milliseconds or even seconds.

To examine this in greater detail, we have to determine what operation we are going to perform and which objects are involved in a given process and to what extent they are involved. The question of the extent to which an object is involved is a good indication of its availability requirement and is determined by the role of the object. The roles of an object for an operation can be broadly designated as:

- **Working**—The object is of prime importance and needs to be in memory at all times. The operation is being performed on this object.
- **Active**—The object is involved in the operation but not at all times. It needs to be available only for a certain intermediate period, and then it can be released.
- **Reference**—The object is of relatively less importance and is seldom used. It would be required only in abnormal conditions if the normal course of execution changes.
- **Archival**—The object is of only historical importance and, for most of the time, not used at all.

The role that a particular object assumes depends totally on the operation we are performing. An object that is of referral nature for a particular operation may be the working object for another operation. Thus, the distribution of these objects would largely be determined by what operations we plan to carry out in a place and what the roles of the objects are in each of those operations.

To illustrate this, consider an example of a company with a number of warehouses with a master controlling warehouse. Each warehouse has a certain number of products, which may be present in more than one warehouse. Now let us consider the operation of replenishment of products in the warehouse. Replenishment process involves determining whether a product in a warehouse is below a certain pre-determined level (called the Re-Order Point), and if so, place an order for a certain quantity of that product (called the Economic Order Quantity) with the product supplier, which may be another warehouse or an

external supplier. Let us say the replenishment operation is carried out at a particular warehouse. The role of various objects involved here are as follows:

- **Working**—The product that is currently being examined for replenishment, the warehouse information, and the replenishment calculation policy
- **Active**—The other products that need to replenished in this warehouse and the replenishment source selection policy. The source selection policy determines which supplier to place the order with.
- **Reference**—The action to be taken if a product needs to be replenished but no supplier source for the product exists
- **Archival**—The past records of product delivery by the supplier of product.

Figure 48 illustrates this scenario.

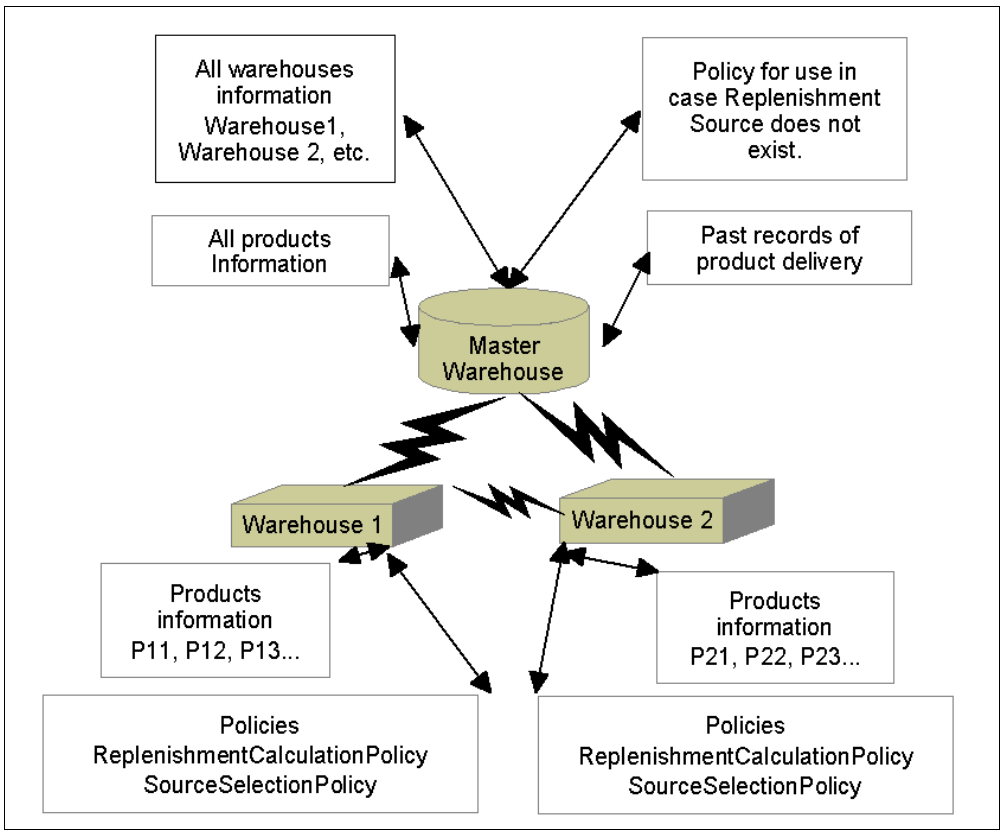


Figure 48. Distribution of Products and Policy Information among Warehouses

From this classification, the placement options become easier to determine. The working and active data should be close together; it needs to be in a single system and should be held in main memory most of the time. The reference information may be on a different system. For example, the policy that determines what needs to be done if no supplier exists for a product that has to be replenished will be common for the entire organization and, hence, can be maintained by the master warehouse. It can be requested by the local warehouse if the need arises. The archival information can also be maintained in the master or relevant warehouses and can be obtained by other warehouses if ever the need arises.

7.3.1.2 Categorization Based on Temporal Properties

Some data is inherently dynamic and some other, static. For example, consider a case where we have a central bank server that holds customer details, and a user needs to access this information. Now, if the user just accesses information, such as the name and address of the customers, then this kind of data would not be expected to change for a reasonable period of time. The user would not have to worry about synchronizing this information with the central server data, nor have to worry about getting the latest copy. But, on the other hand, suppose it is the customer's current account balance that is needed, and the user needs to debit a certain amount from the account. Suppose also that another user may enter some transaction that updates the customers balance between the first user getting a copy of it and using it. This case is clearly different. The difference is due to the different nature of data. In the first case, the data was static (historical, unchangeable), and in the second case, it was dynamic (continuously changing) and hence, it was vital that every user gets the latest copy of the object when making changes. There are different kinds of data and we can categorize them as follows:

- **Static** —This data is normally read-only. This kind of data can be located and duplicated at any number of places. There is no problem of concurrency or integrity of data here. Duplication of such data at places where it is required will also reduce the communication overhead that would have occurred if it were to be kept in a single place. Examples of such data include drawings of finished products, records of sale deeds for the past five years, and so on.
- **Pseudo-Static**—This is an intermediate between the static and dynamic but tending more to the static type. This type of data will remain mostly static, but provision has to be made for its alterations in case the need arises. This kind of data also does not face any concurrency problems.
- **Dynamic** —This is the interactive, shared, and changing data. This needs to be carefully handled by selecting the right kind of access mode that specifies the access location as to whether its going to be accessed LOCAL or HOME and what type of lock it is obtained with, and so on.

Let us examine more scenarios with regard to the placement of the dynamic kind of data. Let us suppose a manufacturer has five warehouses and has a server at each of them. Further, let us suppose that any user can issue stock from any warehouse as long as the user can "see" that stock. Also, suppose the manufacturer does not want to maintain a central server that provides stock details to all user of all warehouses. Finally, suppose that a user at one warehouse cannot access another warehouse's system. This kind of scenario can be implemented as follows:

- Each warehouse has a stock items database that contains all the items across all warehouses.
- The available stock is stored as elements of an array of five values (since we have five warehouses). Each element in the array is the available stock for that item from one warehouse.
- On a regular basis, a batch run is made in each warehouse. The batch run determines the amount of stock of an item in that warehouse and how much of this item should be "seen" by other warehouses. This information is then transmitted to the other warehouses, and their stock item databases are updated.

The drawback of this approach is that occasionally a user may tell a customer that a particular item is not available when in fact it may be supplied. The benefit is that the company will be able avoid a lot of communication overhead and also effectively partition the data. This is, of course, based on the intelligence of the algorithm in dividing up the data.

It is also very clear that such an approach may not be workable in some different situations. For example, suppose a bank branch were to tell you that you can draw only $20 from your account although you have $1000, because of the way they have designed their IT structure. The bank may soon not have the need for an IT structure at all!

7.3.1.3 Other Categorizations

The manner of categorization discussed above was slightly tricky, in that they did not have an inherent solution to them. The other, easier manners of categorization are mentioned below. Many a times, these may be the most preferred ways of partitioning data.

Categorization Based on Organizational Domains and Security

This type of categorization is based on the functional domains that an organization is made up of. These include the finance, the human resources, the marketing, the production, and so on. All data relevant to a department is categorized together. The other manner of categorization would be based on security, where you group data and define access rights for different categories of users.

Categorization Based on Location of Use

This type of categorization determines where the data is finally going to be put to work and places all such data at that location.

Categorization Based on Logical Domains

Data can also be very broadly classified as being local and global. But the classification is logical in the sense that neither the physical location nor the data are determining factors. For example, the information about family members may be considered as local to a family, but the members of the family may themselves be at different physical locations.

7.3.2 Communication Issues

Communication issues are very closely related to issues in data placement, which we explored in the previous section. In fact, the first option for categorization of data based on transfer rates and latency was a classification based on communication issues. Issues in communication are closely tied to the way the objects are distributed in the LSFN. However, there are some generic issues on network traffic generated within LSFN that should be kept in mind. This section discusses remote calls that are executed for reasons other than inter-object communication. The rule of the thumb is to have as few remote calls as possible.

7.3.2.1 Communication with GlobalNameService(GNS)

The flow chart in Figure 49 on page 117 shows the procedure that a server process follows when it needs to lookup an entity in response to a `getEntity()` call.

Each server process has a naming cache that holds information on containers it has accessed in the recent past (refer to Figure 47 on page 112) and the server processes they are configured to. When a server process needs to lookup an entity, it uses the entity handle (which is passed as a parameter in the getEntity() call) to check if its naming cache has information about this container. If it does not find the information, it contacts the Global Name Service and requests for this information. The GNS stores the container—BOProcess maps for each container. The requesting server process caches the newly found container—process map information in its local naming cache. It then determines if the server process of the container is itself or a remote one. If it is a remote process, then the call is forwarded to that process; otherwise, the entity is retrieved from the local container cache.

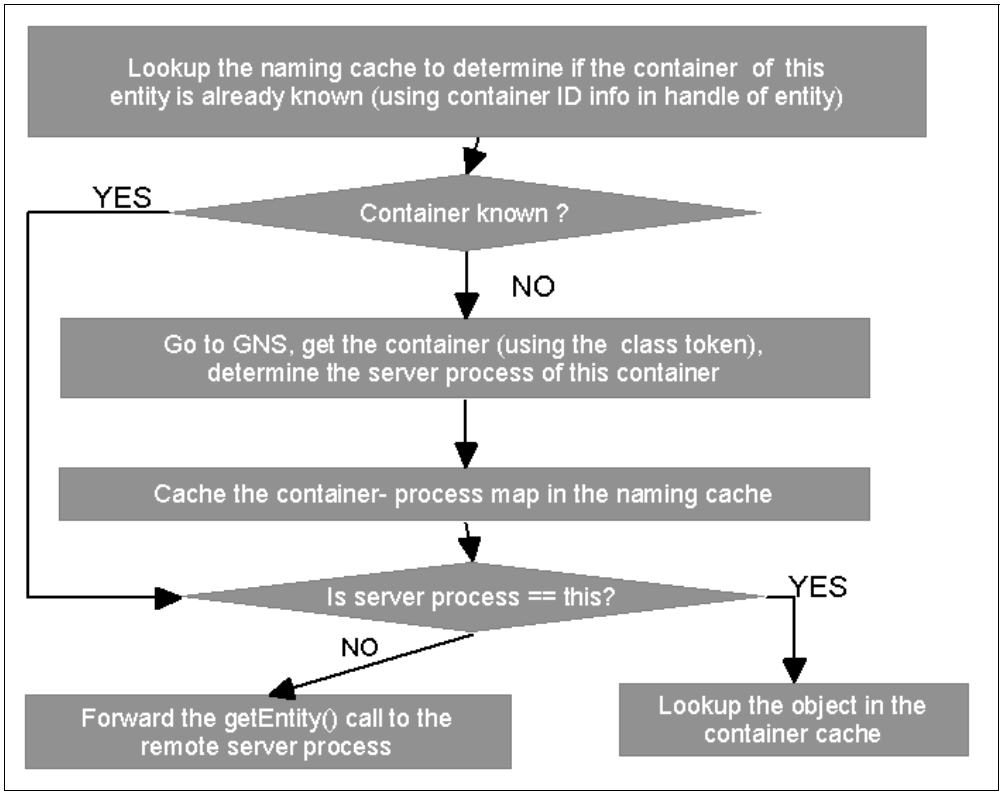


Figure 49. Entity Lookup by a Server Process - Initiated by a getEntity() Call

From this, we can infer that if the objects being accessed are from the same container or from a small set of containers. The GNS lookup is minimized since the container information is available in the naming cache of the server process. Therefore, the distribution of your objects has an impact on the number of the remote calls made for naming lookup.

Also avoiding the use of user aliases for specific object instances could reduce traffic for resolution of user aliases, which requires a contact with the GNS.

7.3.2.2 Communication for DPC, Transactions and Security

The discussion in this section applies to cases where the GSM and the server processes are on different machines. The Distributed Process Context, which was described briefly in the section on DPC initialization, requires a number of remote calls to be made between the server processes and GSM. This is

especially true in case a "write" operation needs to be done on the DPC, as opposed to a "read" operation that can be avoided by maintaining a local cache. Though operations with the DPC cannot be avoided, they should be kept at a minimum level.

Some operations that involve transaction management also require contacting the GSM. This is normally true in case the GSM was accessed to create or delete a user alias for an entity. If such cases, the transaction operations, such as commit(), rollback(), and so on would involve remote calls to the GSM. Once again, it is the use of user aliases and distribution of the objects that determines the number of remote calls that need to be made, even from the transactions perspective.

One of the first tasks the system administrator performs while configuring LSFN is to define the security policy. Among other things, this also requires them to make a decision regarding the use of digital signatures for the objects. Digital signing will be used to protect and verify all security objects that are passed through the network. Specifying that all security objects be digitally signed adds communication overhead to all client connections to servers. However, it protects against someone from building and executing a rogue client application that can fool a SanFrancisco client into thinking it is a legitimate one.

Therefore, the following rules should be observed as closely as possible:

- Minimize the number of remote calls.
- Keep the remote calls as closely packed together as possible. If a remote operation does not require to be performed immediately, it can be saved to a later time when all the remote operations can be performed together in a batch.
- Use local copies and caching wherever possible.

7.3.3 Some Commonly Used Topologies

In this section, we shall try to look at some often used topologies. For the sake of discussion, we shall consider our LSFN to be made up of nodes, where each node is a single machine in the TCP/IP sense. The issues we consider are the ones for a single node system, and then the issues when we have multiple nodes. Some of the issues for the single node system will also hold true in multi-node systems.

The following questions guide us to the answer:

- How many nodes does the system have, and what is the hardware/software configuration of each of these nodes?
- How are these nodes distributed?
- How many containers do you have; where do they reside, and how are the containers assigned to the server processes?
- Which server processes need to be setup and at which node?
- Which nodes are the LSMs, and where is the GSM located?
- What is the amount/kind of communication between the nodes, or effectively, the communication between the server processes?

These questions are answered for each of the above topologies.

Note

The discussion below assumes that your system is running only SF applications. If you are running any other large applications, please use your judgement before applying the hints given below.

7.3.3.1 Single Machine

This is a single node system. The server process that is mandatory is the GSM that has the naming service and the security. There will be at least one server process (let us call it SFBOProcess1) that provides the factory and transaction services. Optionally, you could have the conflict service and problem logging service too. There will be at least two containers. One container for use by the GNS, typically a POSIX container. The GNS has been found to perform best with a POSIX type. The other container will be the one containing your business objects. The container is configured with the SFBOProcess1. This container can be either POSIX or a relational database, for example DB2.

A part of this discussion related to configuration values for heaps and processes was dealt with in 7.2, “Configuring LSFN for Small Systems” on page 108.

Before we proceed further, we shall recap some facts, which you may already be well familiar with:

- The mapping between server processes to containers is one to many. Many containers can be associated with a server process; a container can be associated with a single server process.
- Though the mapping between containers to legacy database is many to one, this is not at all recommended from a performance point of view. The reason is that when your legacy databases are shared, you have to turn the caching off and access all objects with a pessimistic lock only. This can be a major performance hazard. As a rule of thumb, assign one container to one database.
- Each server process runs in its own JVM, which means it has its own heap and garbage collector.

Server Processes

The question is whether you would want multiple server processes. This may be true but only if you have multiple containers. You may want to assign one server process to one container. The table of memory requirements for a single system was given in Table 2 on page 92. Each additional process requires approximately 40MB more for an NT machine. If your server set is constrained by memory, then you can have a single process and assign all containers to it. If your server setup is not constrained by memory, you would be better off having multiple processes, as shown in Figure 50 on page 120.

The advantage of having multiple processes is that memory can be managed more efficiently. If a single process is assigned all the memory, it will work fine for quite a while until the garbage collector is invoked, but when the garbage collector starts, it blocks all operations, and since there will be a large amount of garbage to be collected, the blocking time could be unacceptable. With multiple server processes, the total heap is divided up among the server processes. Each

one invokes the garbage collector more often, and the garbage collector does not block for so long since the amount of garbage is less.

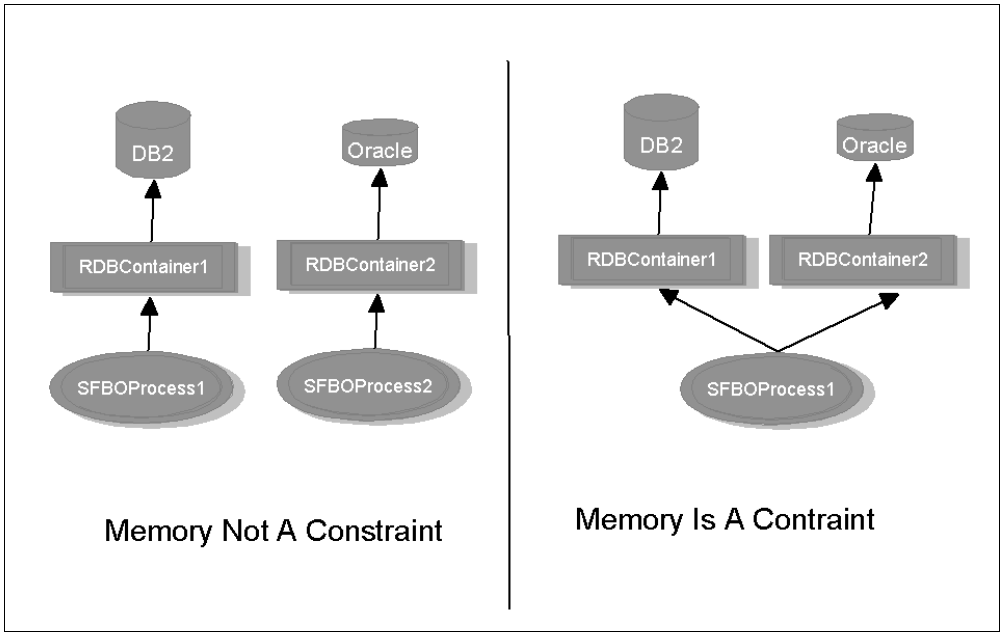


Figure 50. Assigning Server Processes to Containers

A minor disadvantage of working with multiple processes is that communication between objects in different processes will involve remote calls. The communication between objects in the same container, and also between objects in multiple containers that are assigned to the same server process, is faster than that between objects that are used in different server processes.

Containers

You will have multiple containers if you are going to partition your data. It is a good idea to partition your data, and this has been discussed in 7.3.1, “Data Placement” on page 112. On a single system, the partitioning may be for reasons of security, for the logical divisions in your business, and so on. The partitioning could also be done for reasons of efficient memory management by assigning a server process for each container, as discussed above.

The questions that arise here are the number of objects each container has and the cache size of each. It is a good idea to place frequently accessed objects in one container and to put the less frequently accessed ones in another. The ones with more frequently accessed elements would have to be allotted a larger cache as compared to the one with less frequently accessed objects. Also, the size of the containers will be decided by the average size of each entity in your system and the number of such entities in the container.

While assigning cache amounts to the containers, be careful not to cause too much paging in your system since that would reduce performance drastically. It is wise to assign cache only in the main memory and not to rely on virtual memory, as use of virtual memory results in more paging and hence, more I/O and consequently less performance.

7.3.3.2 Multiple Machines

In multiple machines, the issues are basically the same, except that we now have to consider communication delays too. There is a "master" system, which has the GSM and possibly one or more server processes. The other nodes may not have any server processes at all, they are pure clients, or may have one or more server processes (in which case they run a Local Server Machine or LSM for auto start, and so on). The client could be any of the nodes in the system.

Node distribution

The machines could be located close together in a LAN, or they may be in distant geographical locations. For machines close together, the issues are the same as that for a single machine with multiple processes. For machines distant from each other, the following pointers may be helpful:

- Place the GSM on the central and larger system.
- Place the objects (containers) near the place of use. It is a good idea to place them in the client location since that would reduce the network traffic for accessing objects.
- If the distant machine is a client, keep in mind that there will be a lot of communication for retrieving objects and for performing any operations on them. On a pure client (that is, with no server process), only the application code exists. Everything else runs on machines with server processes.
- Even if the client side has the containers, there will be some communication for GNS lookup, for DPC access, and for security information. Refer to 7.3.2, “Communication Issues” on page 116 for more details.

Note

The issue of determining the optimal configuration for a given topology is very specific to an application and is largely experimental. It is recommended that you set up a test environment that models your real life application or try different configurations in your existing application within the permissible limits. The empirical data thus collected, and its correct interpretation, combined with the advise given in this chapter, will help determine the optimal configuration settings for your application. The following references can provide some guidance in this direction:

- Chapter 3, “How to Find a Performance Problem” on page 23.
- Chapter 4, “Tools for Performance Analysis” on page 31.

Chapter 8. Object Persistence, Databases, and Schema Mapping

This chapter consists of two major parts. The first part, in a very generic way, describes the problem domain of how to map objects to relational databases. Every developer knows that object database mapping is a very important issue, and here we point out why it is so important. The importance of this is emphasized in the fact that object oriented systems have to solve a number of problems and issues that did not occur in the procedural or data driven approach to systems development. The approach taken here is to describe different scenarios. The most important issues discussed here are implemented in SanFrancisco. Some are not since they do not fit in a generic framework. Some functionality can be achieved in other parts of the framework, and some can be implemented by combining parts in the framework.

The second part describes the SanFrancisco way of solving some of these issues. It describes, very shortly, the default Posix store in IBM SanFrancisco and the RDB store in IBM SanFrancisco, including the DSM (default schema mapper) and the ESM (extended schema mapper).

The second part also describes database configuration for DB/2 on Microsoft Windows NT, DB2/400 on IBM AS/400 system, and Oracle on Microsoft Windows NT. Query Pushdown and EntityOwningExtent, which is a prerequisite for Query Pushdown to occur, are mentioned.

The purpose of this chapter is to advise the application engineer when to use which of them and why, and of course, which one that is preferred depending on the purpose of the kind of project the person is working with.

It will not deal with fine tuning the actual databases since this is an issue for the DBA (data base administrator) at specific customer installations and should also be abstracted away from the developer. It could also come into direct conflict with the legacy system in those cases where the SanFrancisco development works against legacy data.

8.1 Schema Mapping in General

This section describes the schema mapping issue in a very general way. Since schema mapping has to be done when object state should be persisted, special attention must be made to the different aspects on how to do it in Object Oriented Development Environments.

This is a compressed version of the article *Architecting Object Applications for High Performance with Relational Databases*, by Shailesh Agarwal, Ph.D. and Arthur M. Keller, Ph. D., Persistence Software Inc. found at Web site: www.persistence.com/products/wp_architect.html

8.1.1 Abstract

This section presents an approach for architecting object oriented applications for high performance with relational databases. This approach enables organizations to derive the benefits of object oriented technology while leveraging their investments in relational technology.

The key ideas of this approach are:

- Optimize business object mapping. Tune the mapping between business objects and relational tables to leverage relational technology.
- Perform object cache management. Use an application server object manager to share “high activity” objects among many clients, thereby minimizing database traffic.

8.1.2 Introduction

Object-oriented software development is rapidly becoming the leading approach for building flexible, scalable software systems in client/server environments. Additionally, over the past decade, relational technology has matured and has been widely adopted for managing corporate data. Relational databases have now become the standard data stores for online transaction processing (OLTP) applications. These two trends are motivating the need for building object-oriented applications that access relational databases. Developers building such object-relational applications face such difficult problems as:

- Mapping the objects from the application model to the relational schema in the database
- Managing objects in an application server to optimize performance
- Managing the locking and transactions to ensure data integrity
- Optimizing with consideration of the performance characteristics of relational databases

Each of these problems present several interesting issues worthy of discussion. However, this chapter focuses primarily on the performance issues for such object-relational applications and presents our experiences with architecting high performance object-relational applications.

8.1.3 Object-Relational Mediators

An object-relational application provides an object-oriented interface to relational data. In such applications, the application object model is mapped to a relational schema in the underlying database. Typically, such applications build or use a mediator for transforming object operations to relational database calls and vice-versa.

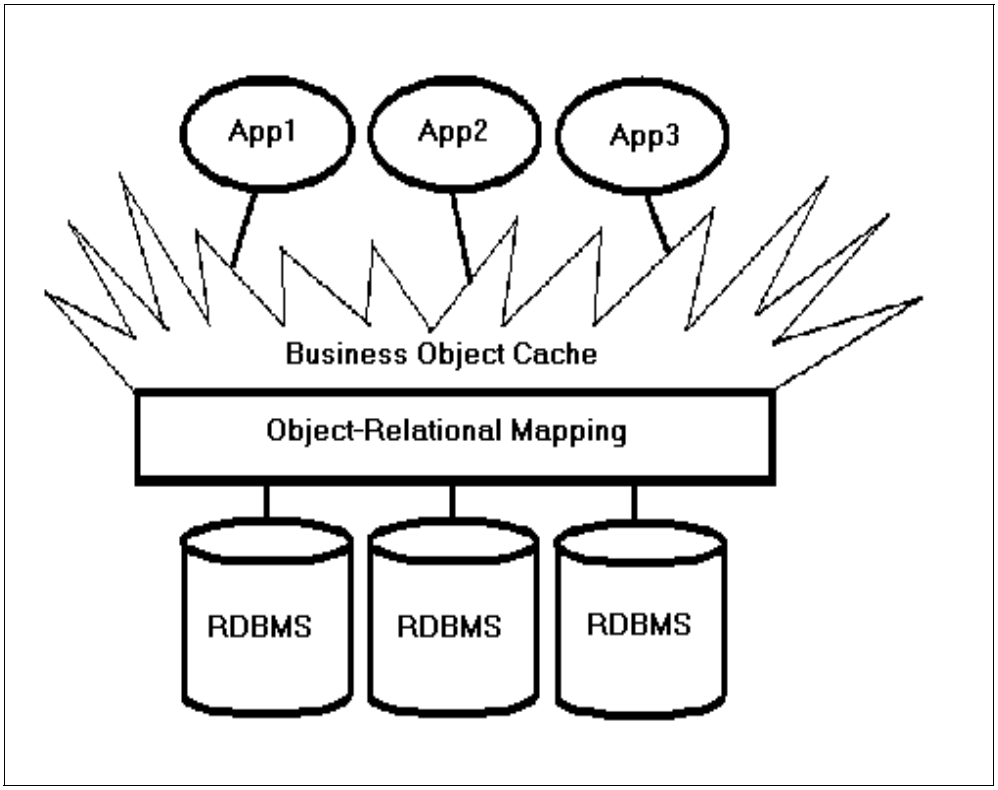


Figure 51. Object/Relational Mediator

As shown in Figure 51, the mediator maps business objects to relational tuples and also manages business objects in a shared cache on behalf of several applications.

The object management provided by the mediator can enhance performance through a technique we call “live object caching.” The mediator can also ensure data integrity with appropriate locking and transaction management. Tuples in a relational database are presented as instances of a class, and any updates on those instances are translated into updates of the corresponding tuples in the database. Additionally, the mediator could also provide a client-side object cache for enhancing performance. In a sense, such object-relational mediators enable an OODBMS (object oriented data base management system) interface to a legacy RDBMS (relational data base management system).

Based on over five years helping companies build large scale object-relational systems, Agarwal and Keller have found that there are a number of “ground rules” for achieving high performance. Developers who do not follow these rules risk building systems that do not scale well in operation.

In one memorable example, a team ported an application from an object database to a relational database. The port was completed without changing the object model at all. The resulting application ran properly, but where inheritance had assisted clustering in the object database, it imposed a high performance penalty in the relational database. Through Agarwal and Keller’s experiences with companies implementing large scale object systems, they have developed an approach to architecting object applications for high performance.

8.1.4 Achieving Performance

Agarwal and Keller's experiences has shown that object-relational mapping and high performance can go together, provided proper consideration is given to object mapping and object management issues. Developers can optimize the mapping between a set of interrelated business objects and a relational schema to leverage relational technology. Further developers can use a client-side object cache to enhance application performance. The basic approaches for optimizing object access to relational data are similar to the objectives for tuning the relational database itself. The two approaches are:

- **Optimize object-relational mapping**—Applications should be written to ensure an efficient mapping between application business objects and relational tables.
- **Optimize object-relational caching**—Applications should be written to take full advantage of object caching and navigation within an application server.

Architecting applications with these two approaches will enable them to achieve high performance. The following sections discuss these approaches in greater detail.

8.1.5 Optimize Object-Relational Mapping

The key objective of this approach is to pose only those queries and updates that can be processed efficiently by the server. There are potentially three ways to achieve this objective:

- Choose the appropriate object to relational mappings.
- Use query capabilities of relational databases.
- Take advantage of special performance features of relational databases.

Mapping efficiency between object classes and relational tables is the most critical factor for achieving high performance. "Pure" object models sometimes map poorly to relational structures, for example by requiring many joins to construct simple objects. Such a mapping can significantly deteriorate application performance. Hence, starting with an object model and attempting to map to a relational schema can sometimes lead to poor performance.

On the other hand, Entity-Relational (E-R) models have less semantic content than object models but can be converted naturally to object models, mapping entities to objects, and relationships to associations. E-R models can then be optimized through selective denormalization based on expected usage. The advantage of this approach is that such denormalization is well understood, while optimizing object models is not.

The best way to achieve this efficiency is to use a normalized relational schema (E-R model) as the basis for the corresponding object model. Developers will find that the simplest mapping between objects and normalized tables typically provides the best performance. Object-relational mapping requires the mapping of the following three major modeling concepts:

- Simple or aggregate classes or aggregation to relational rows
- Relationships to foreign keys
- Inheritance to relational rows or joins

8.1.6 Mapping Simple and Aggregate Classes

The most efficient mapping is most often the direct mapping of a class or aggregation of classes to a relational table. A more complex mapping can lead to performance penalties. Classes that represent a join between several tables can be expensive to construct and potentially impossible to update.

A simple class contains only simple attributes, such as integer and character, while an aggregate class can contain attributes that are themselves classes. The following figure shows mapping an aggregation of three classes to a single table.

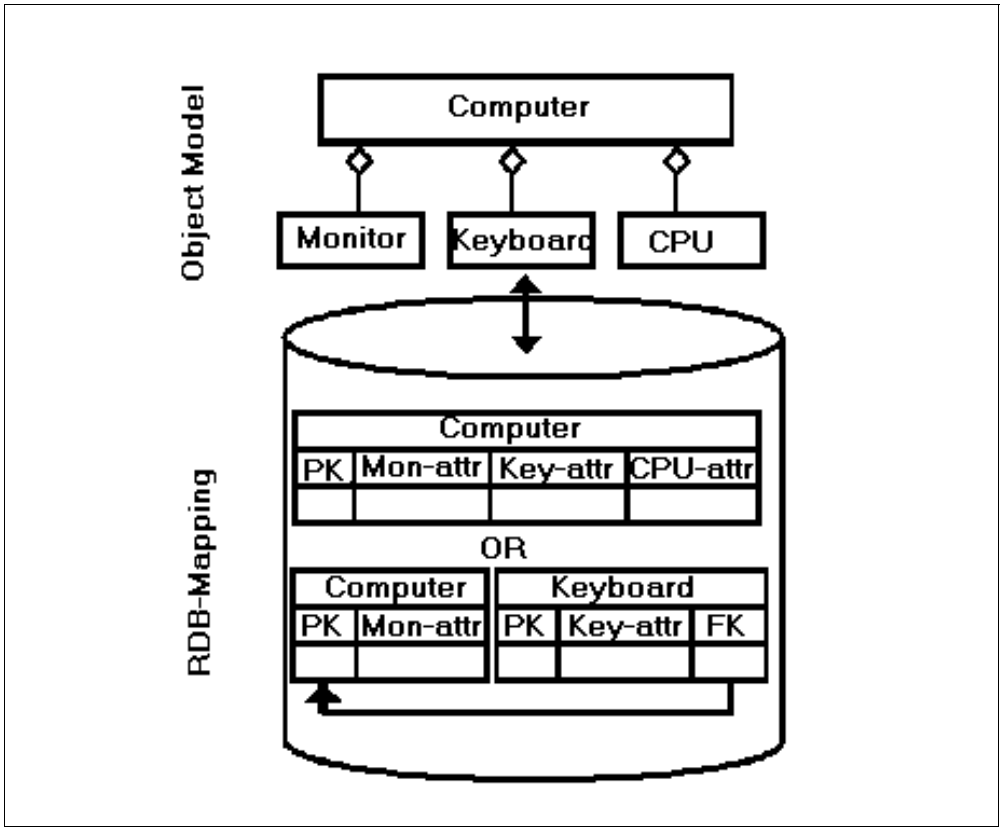


Figure 52. Aggregate Class Mapping

Two exceptions to this rule include creating projection objects to minimize network traffic and view objects for decision support.

- **Projection objects**—For tables with many columns, it may be efficient to map a projection of the table to a “projection class” and retrieve a full row only when it is needed. For example, a customer row may contain 50 columns, but only the name and phone number are needed most of the time. By creating a class that maps to just the needed columns, the developer can avoid having to pass unnecessary information across the network. Such a class must contain the primary key columns at a minimum.
- **View objects**—For decision support applications, it is often useful to create a view table that represents a database join and then map this view to an object class. This allows developers to take full advantage of relational algebra to hide the physical data model from the object application.

8.1.7 Mapping Relationships

Relationships between objects map to foreign keys between rows. How each mapping is implemented, however, has a significant impact on the performance and flexibility of the resulting application. Developers have several choices for mapping object relationships to relational tables.

- **Embedded foreign key**—This is the most common approach for one-one and one-many relationships. In this case, for a given class, the primary key of a related class is embedded in the class itself. This results in a performance characteristic better than the “distinct table”.

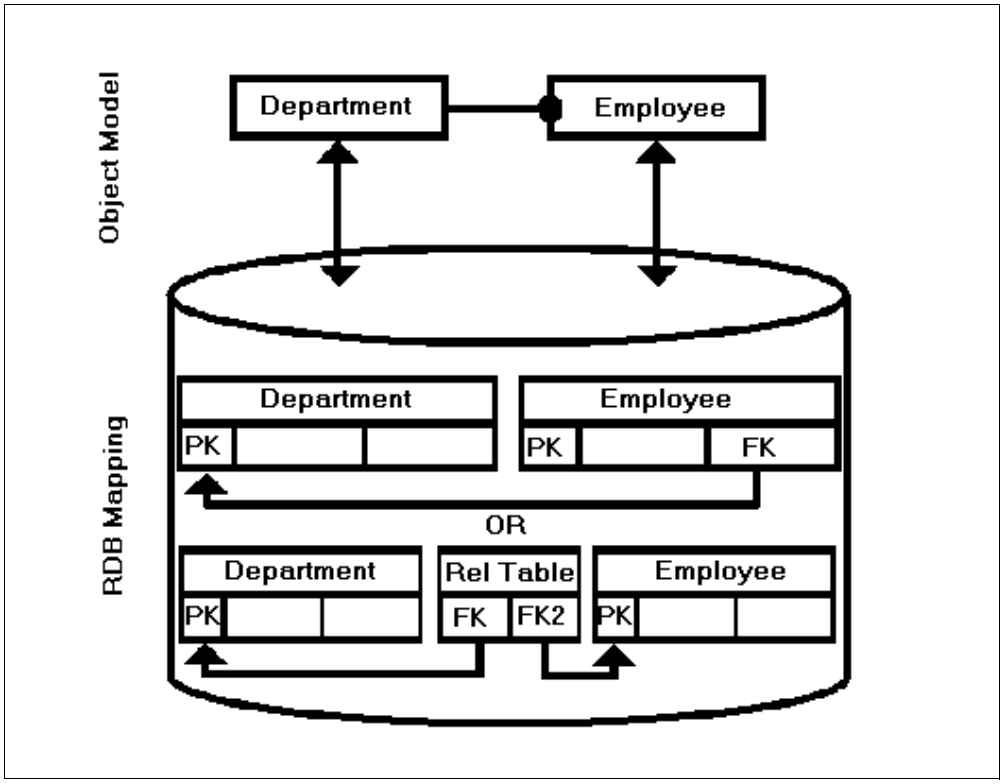


Figure 53. Relationship Mapping

- **Distinct table**—In this approach, the relationship is represented as a distinct table in the database. This approach provides the most modeling flexibility since it makes adding and removing relationships transparent to other tables. However, this approach can be expensive if the relationship is frequently traversed. In general, the embedded key approach is most efficient for one-to-one relationships and one-to-many relationships, while the distinct table approach is required for many-to-many relationships.

8.1.8 Mapping Inheritance

As with relationships, how object inheritance is mapped to the database can have a profound impact on performance. Here, particular attention to expected query paths plays an important role on the mapping choice. Each parent or child class can be either abstract or concrete. An abstract class is one that will never be instantiated on its own, and consequently, does not have any physical data associated with it (for example, no corresponding table). A concrete class is one that can exist on its own and typically will have its own associated table.

Developers have several choices about how to map inheritance into relational tables.

- **Horizontal Partitioning**—In this case, only leaf classes are mapped to tables and includes all of their inherited attributes. This approach may give improved performance since only one table needs to be accessed for instances of a given leaf class.

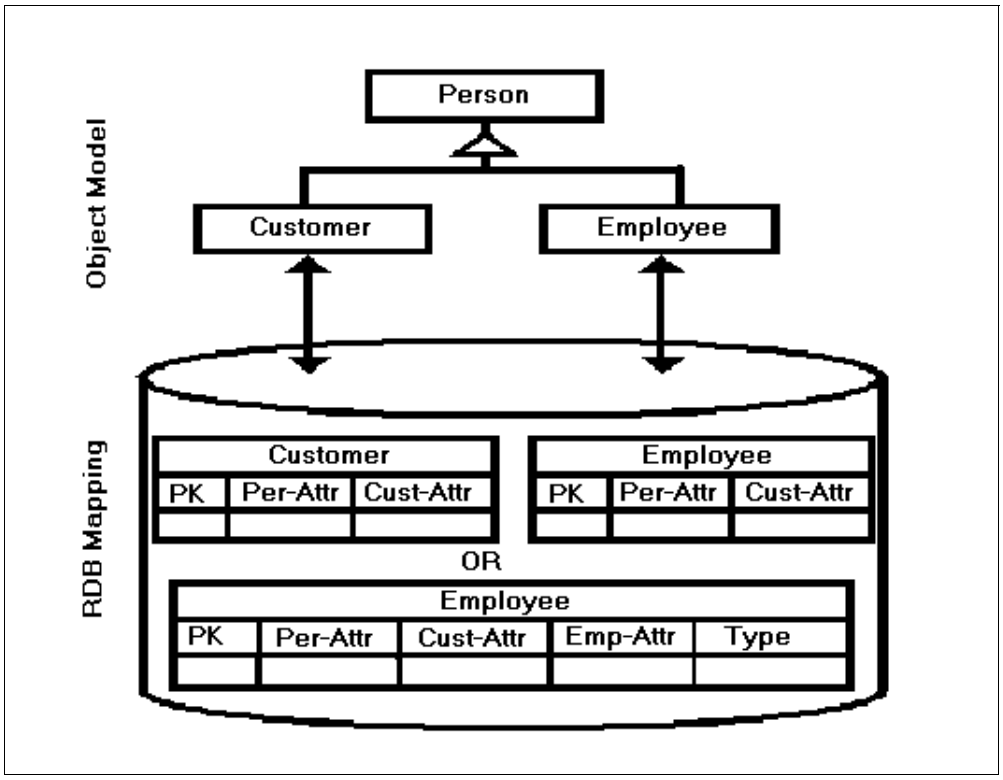


Figure 54. Inheritance Mapping

- **Typed Partitioning**—Another way to handle inheritance is to map all classes in an inheritance tree to a single table using a type column to distinguish between subclasses. This enables the retrieval of objects from multiple classes in a single query but violates normalization.
- **Vertical Partitioning**—In this type of mapping parent, classes in the object model class maps to a corresponding table. The classic object-relational mapping “mistake” is to create deep inheritance hierarchies that require multi-way joins to retrieve basic object information.

Queries made at the parent class level are implemented very differently depending on the inheritance mapping. If the parent class is concrete, a parent class query can be efficiently handled by querying the table corresponding to the parent class. If the parent class is abstract, the query must be run against each child table, a potentially time consuming operation. On the other hand, retrieving a single instance of a child class is much faster if all the parent classes are abstract, as all necessary columns are located in the child table. Child classes with concrete parent classes must perform expensive joins to retrieve instances.

8.1.9 Multi-Class Join Queries

Developers can gain significant performance benefits by using the sophisticated querying capabilities available with relational databases for ad-hoc access of objects. For the initial select of a set of objects, ad-hoc access through the relational join query mechanism using indexes is significantly faster than navigational access.

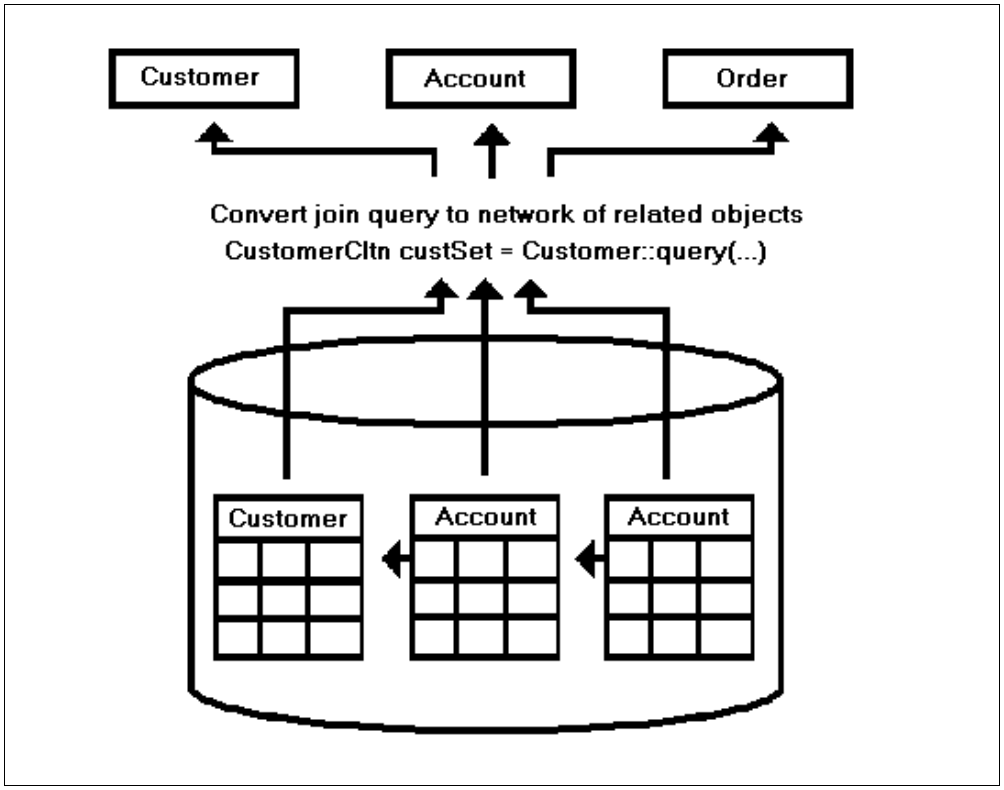


Figure 55. Join Queries

Once the data has been retrieved from the database, then in-memory navigational queries allow efficient use of the cached object information. One very important benefit of the object-relational mapping is that it allows developers to create hierarchies of related objects from “flat” tables, providing an elegant solution to the classic bill of materials problem for relational databases.

Object cache can solve “bill of materials” problems for hierarchical data.

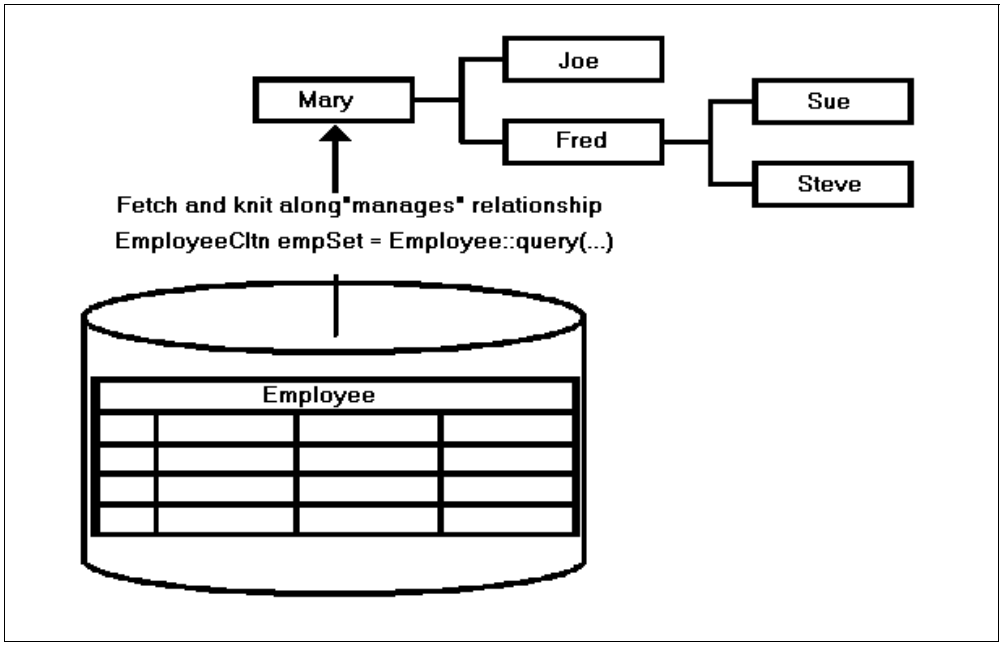


Figure 56. Object Knitting

To take advantage of this query support, appropriate indexes should be built on columns that are to be used for class and relationship queries. In general, indexes to be built include all primary and foreign keys in the database schema.

8.1.10 Live Object Cache

The basic model for object management is to cache data instances read from the database, register their primary key values, and respond to queries based on this cached data. As tuples are retrieved from the database, they are converted to objects and “knit” together according to the object model mapping to form a network of in-memory objects. High activity objects include reference data as well as core business data which is likely to be shared across transactions. High activity objects might include data such as network elements, flights, or portfolios that are being shared by many clients. In a live object cache, high volume transaction data, such as accounts, tickets or trades are kept in the database and only loaded into the cache on an as-needed basis.

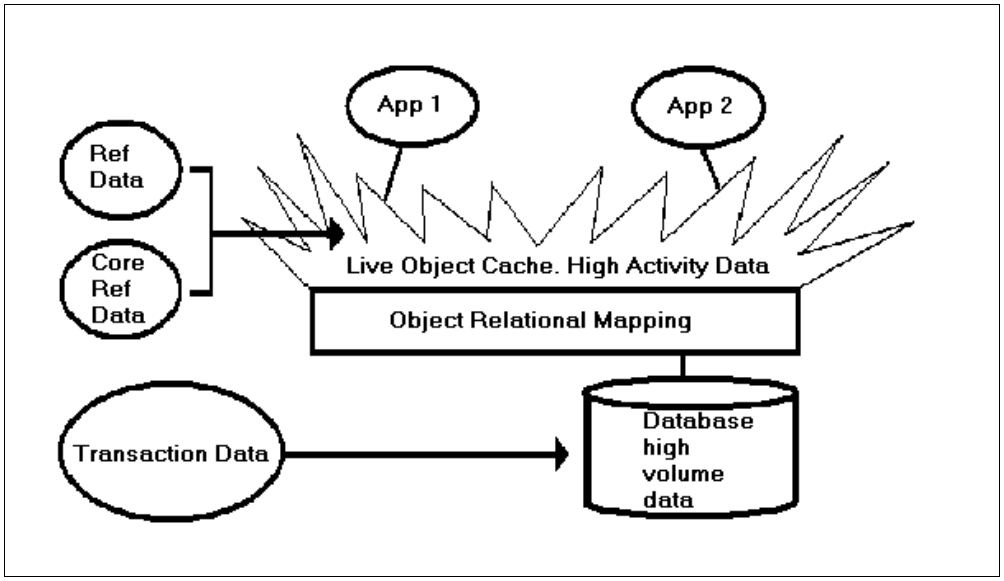


Figure 57. Live Object Cache

Cached objects can be shared between many users without paying the repeated overhead of querying the database. Further, navigating relationships between cached objects are orders of magnitude faster than querying the database. Cached objects may also be changed repeatedly without accessing the database, holding the database update until the transaction is committed. This prevents sending multiple updates for the same object to the database. In the case that the transaction is aborted, no changes need to be sent to the database at all. Object caching is also critical for ensuring object data integrity.

8.1.11 Optimizing Object Navigation

Using such an in-memory network, queries that follow foreign key pointers (navigation queries) can be performed fully in the shared cache once the basic object model information has been retrieved.

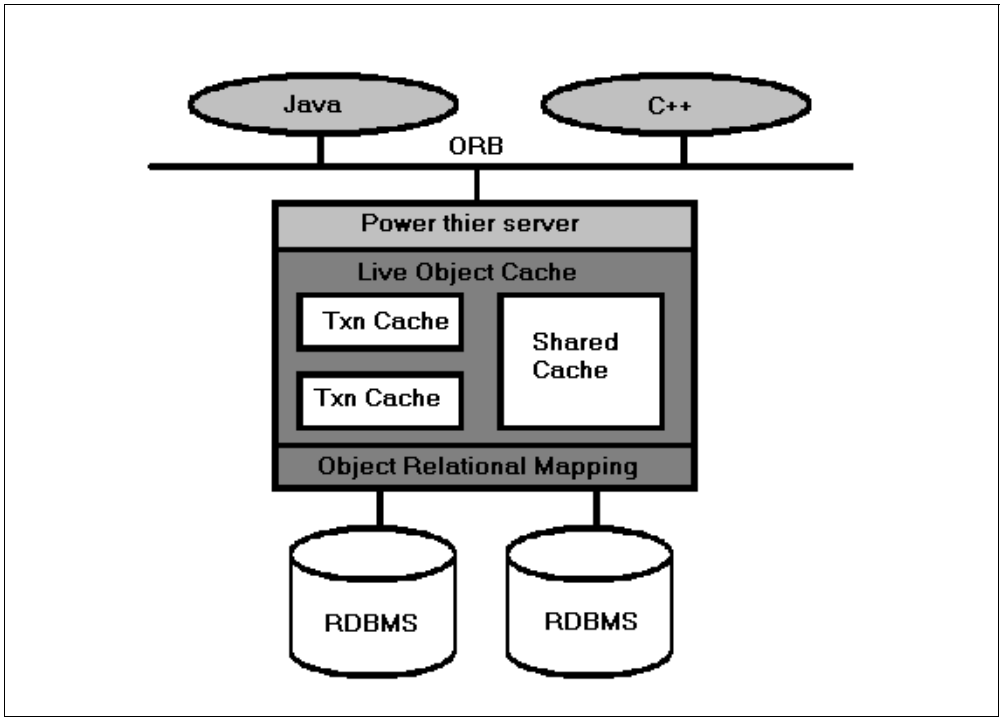


Figure 58. Transactional Object Cache

8.1.12 Transaction Isolation

Efficient management of a shared object cache can speed transaction throughput greatly. In a shared object cache, however, it is critical that uncommitted object changes from each client are isolated from other clients. Figure 59 on page 134 shows one way to achieve object isolation.

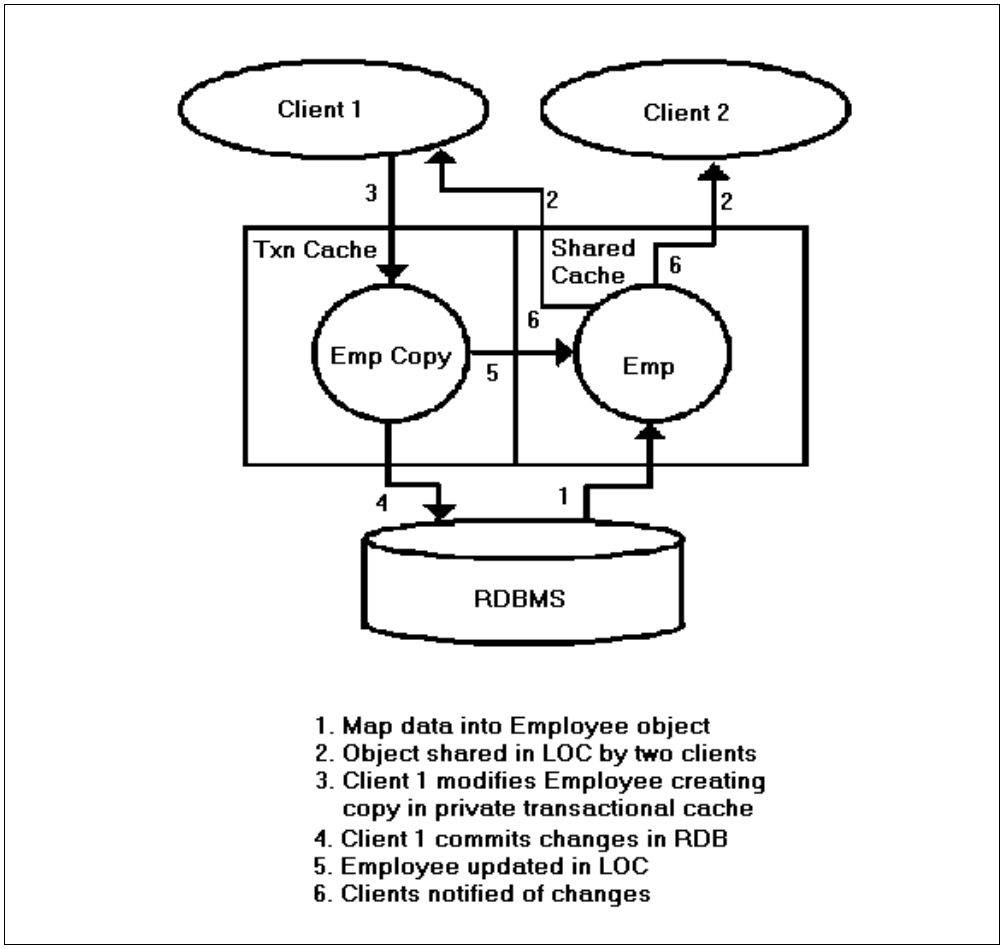


Figure 59. Transaction Isolation

To speed concurrence, an optimistic locking technique can be used. In an optimistic transaction model, no database locks are placed for cached data. Before sending the cache updates to the database, a check is performed to ensure that the data update is valid.

8.1.13 Conclusion

Developers can achieve high performance building object systems linked to relational databases, provided they follow a sound approach to object-relational mapping and object caching.

Here are some design guidelines for achieving high performance in object applications linked to relational data:

- **Classes**—Keep it simple. Map simple or aggregation classes to tables where possible.
- **Inheritance**—Balance retrieval speed against query efficiency. In general, horizontal inheritance is the best choice, as it speeds retrieval for child classes.
- **Object caching**—Optimize performance with a shared transactional cache. Caching “high activity” objects can speed access by three orders of magnitude.

- **Object navigation**—Knit retrieved objects into an in-memory object cache. Navigate objects within the shared cache to avoid less efficient database joins.

By following these guidelines and using the appropriate tools, developers can achieve the performance required for deploying large scale object systems.

8.2 The SanFrancisco Entity Cache

Each container in SanFrancisco maintains a memory cache where it keeps recently accessed persistent objects so that, when needed, SF can retrieve them directly from memory rather then reading them from disk. The more objects the container can keep in its cache, the more likely that the particular object your application needs will be in the cache, which can result in dramatic improvement of the application performance. To measure the need of size of the cache, it is advisible to run the Container Cache Statistics Tool described in 4.7, “The Container Cache Statistics Tool” on page 57. If it is possible, try to reach a hit rate of approximately 90%, so that there is always space left to use.

The maximum number of objects the container will keep in its cache is a parameter you can control in the SanFrancisco Configuration Utility by performing the following steps:

1. Start the SF Configuration Utility.
2. Go to the Configure pull down, pick **Containers** and choose **Posix** or `Rdb`, depending on the type of container that you want to configure.
3. Double-click the name of the container for which you want to alter the threshold. This brings up a window with a field for Max cache size.
4. Set **Max** cache size to the maximum number of persistent objects you want the container to cache.

Keep in mind that the cache consumes virtual memory on your system, so the performance gain can be a loss in other cases.

8.3 SanFrancisco Schema Mapping Cache

Each server process in SanFrancisco maintains a cache where it keeps SML (schema mapping language) files information. The information is in memory until the process terminates.

8.4 The Posix Store

The Posix store is the default way SanFrancisco stores state of Entities. The Posix store is very easy to manage and does not require any DBA assistance. It is very well suited for doing prototypical work since it does not require the developer to pay any special attention to it.

A container in Posix on Windows NT is physically a catalog (directory), and every Entity resides in a single file with a name that is a unique identifier that equals SanFrancisco internal representation for the object.

It is not recommended to use Posix store in a production environment. For a small number of objects, a Posix store can actually be faster than a relational database, but since every object resides in its own file, the access to persisted objects will be noticeably slower the more objects there are. This is also true for a relational data base, but there you have the entire power of RDBMS system to optimize the database activity for you.

8.5 The Rdb Store

The Rdb (Relational Database) store is used when you map the state of an Entity to a RDBMS (Relational Database Management System). Currently supported RDBMS are:

- DB2/400 on AS/400 Systems
- DB2 V5 (UDB) on Windows NT
- Oracle 8 on Windows NT, AIX, Solaris, HP/UX, Reliant
- DB2 on AIX

8.5.1 DSM (Default Schema Mapper)

The default schema mapper (DSM) provides automatic mapping from one class to one table. The DSM also maps the classes' attributes to default columns. A general set of rules help define and create the table at run time. Java reflection support obtains information about the Java class file at run time. An automatically generated object identity column becomes the primary key column for the new table. In addition, the DSM automatically generates a system defined partition key column for the new table. Partition keys are used to partition a table into multiple EntityOwningExtents (EOEs).

In simple cases when a class contains basic data types, the schema mapper maps each attribute to a column in the database. For example, Strings defaults to VARCHAR(128), and DDecimal to BINARY(a), and so on. Only certain dependant classes are streamed, not all. If the DSM would create more than one LONGVARBINARY column, it will instead stream the containing object. The DSM will only create one LONGVARBINARY column per table. Dependent classes, such as Address for example, that contain several attributes are just streamed out to a single column in the database. This makes query pushdowns impossible. Collections of Dependents are also streamed to a single column in the database and are impossible to do Pushdown queries against.

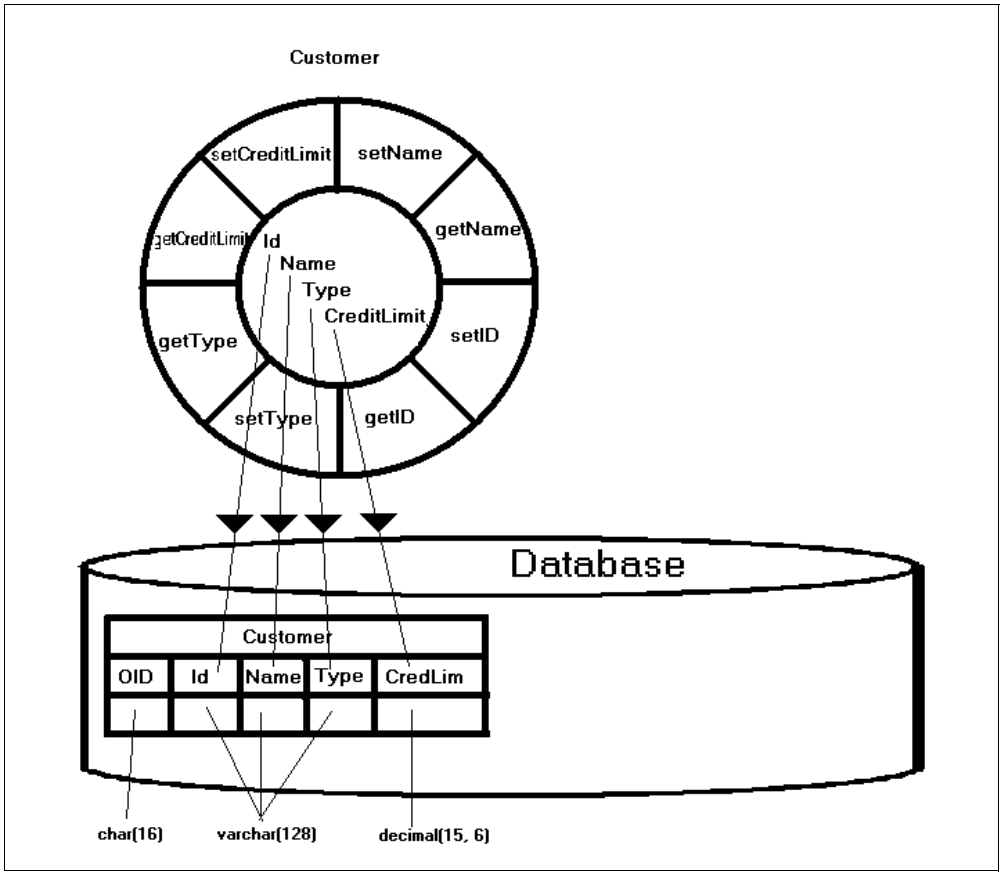


Figure 60. Simple Class Customer Mapped by Default Schema Mapper

For simplicity of this chart, partition key is omitted. By default, the DSM creates the tables with IBMSF as schema qualifier. The DBA (data base administrator) must create a user ID or a group profile with this name (IBMSF) and then grant the authorities to the users that should participate in database activity.

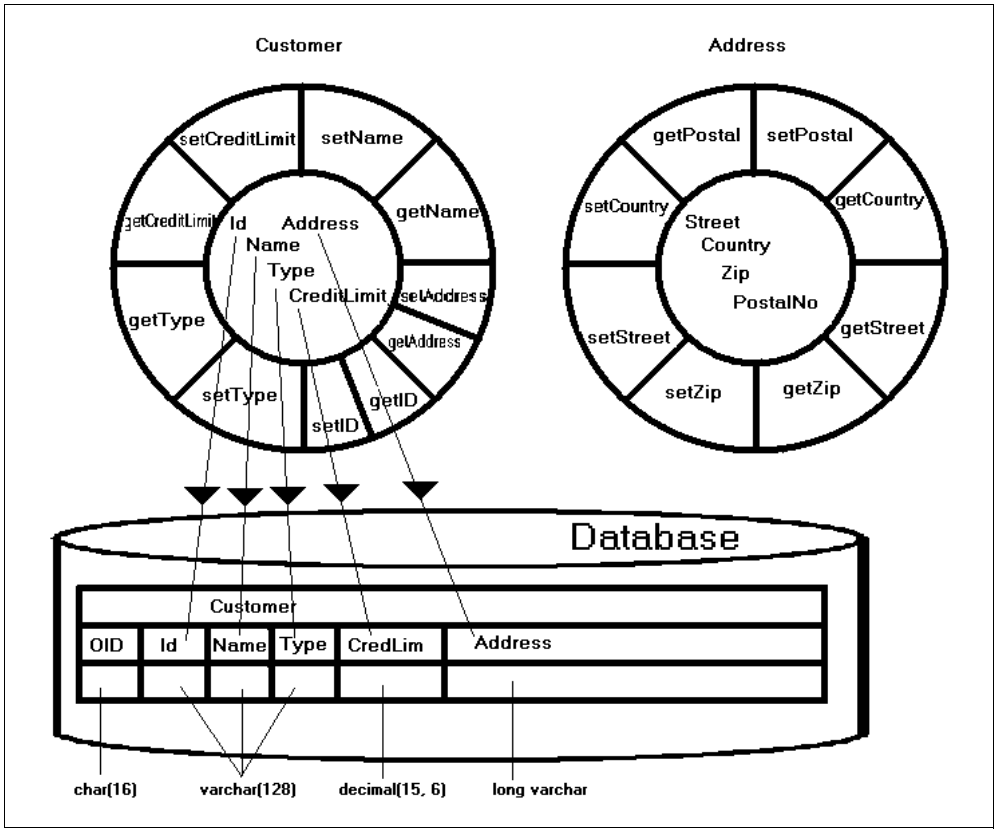


Figure 61. Class Customer with Dependent Class Address Mapped by DSM

For simplicity of this chart, the partition key is omitted.

8.5.2 The Extended Schema Mapper (ESM)

The extended schema mapper (ESM) enables you to map a class to either a new table or an existing table. The ESM initial mapping is similar to the DSM. In addition, you can change the attribute to use the column mapping to match a particular legacy table as long as the data base supports data conversion on that particular data type.

In simple cases when a class contains basic data types, the schema mapper maps each attribute to a column in the database. For example, Strings defaults to VARCHAR(128), and DDecimal to BINARY(a), and so on. Here a performance boost will be achieved in creating the table with CHAR and the actual length of the field(CHAR(35)). Fixed length data types generally take less overhead than variable length variables. An even bigger performance boost can be achieved by using UTF8 instead of the UNICODE default. It also saves space with US - Latin data.

In Extended schema mapper, it is also possible to map the Dependent classes that contain several attributes to columns in the database. This makes the attributes of the Dependent eligible for Query Pushdown. Collections can send handles map to other tables, which will help to get a more traditional database design. The optimizing capabilities of the database are used to achieve high performance.

The outcome from the schema mapper tool is a SML (Schema Mapping Language) file that contains the mapping between the class and the table. The SM (Schema Mapper) uses the SML file at run time to perform the mapping.

A class must be configured with a corresponding SML file in order for the ESM to map correctly. When using a class in multiple containers, the class can have a separate SML file for each container.

One restriction is that any classes or subclasses that are owned by an EntityOwningExtent **MUST** map to the same SML file. This restriction is based on the current implementation of EntityOwningExtent that requires that all elements contained in an EntityOwningExtent belong to the same table.

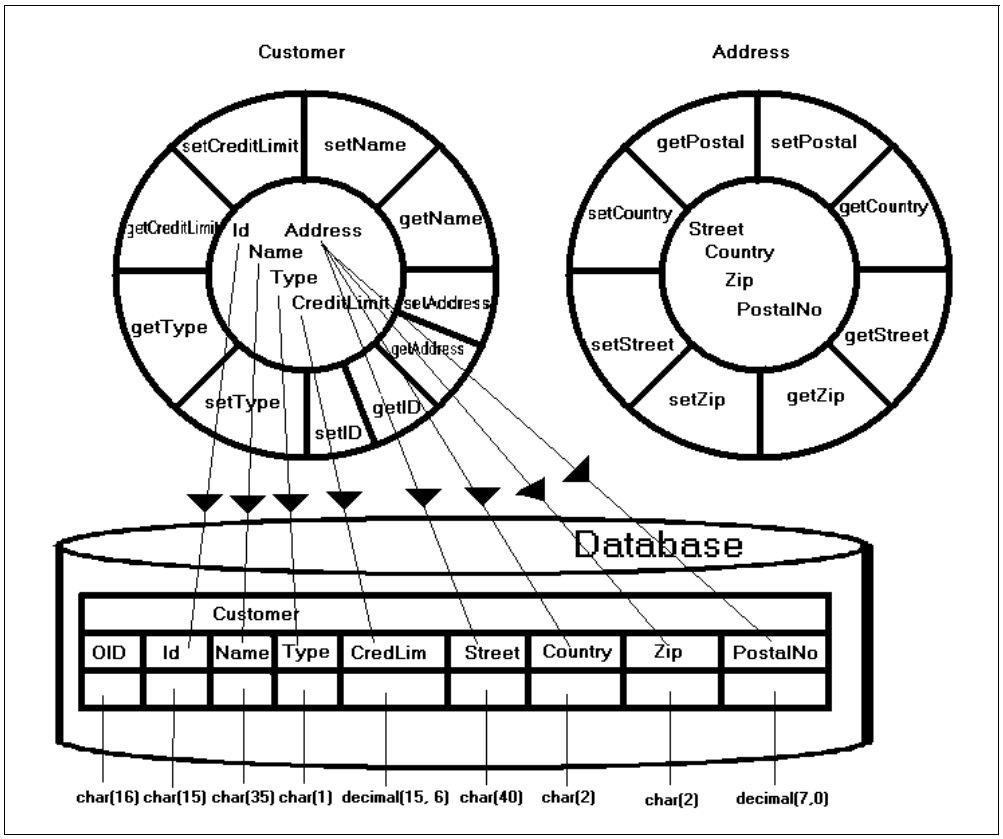


Figure 62. Class Customer with Dependent Class Address Mapped by ESM

Note that even the database data type column has been changed from the original defaults. For simplicity, partition key is omitted.

8.6 Legacy Data

As mentioned in the Extended Schema Map section, if you want to map Entities to legacy tables, this has to be done with the extended schema mapper.

8.7 When to Use What

The Default Schema Mapper provides the simplest way to persist the data to relational tables. However, it does not scale to perform advanced functions, such as subclassing, joins or specific data type mapping. It is nearly as easy to use in a prototypical stage as the Posix store.

Table 4. DSM and ESM Advantages and Disadvantages

Schema mapper	Advantage	Disadvantage
DSM	Does not require a previous knowledge of schema mapping by the user. Maps one class to one table. The schema mapper handles the mapping. The schema mapper creates the table automatically.	Applies to new tables only. Maps one class to one table. This means that it is not possible to share a table among classes. This results in a higher degree of normalization, which means that more joins may be used Individual object attributes may not map to columns. Instead, the entire object may be streamed into one column. Mapping may not be optimal. Query pushdowns are not available in all situations.
ESM	Supports legacy tables. Allows mapping of the class attributes to columns of the existing table. Allows you to change the data type to match the Java type. Allows usage of more functions like persistent array, null attributes, subclassing, joins, handles, etc. Performance benefits achieved in the sense that the administrator maps the class attributes to a more efficient data type and length. Mapping all attributes to columns allows query pushdown to be enabled.	Table must exist before the application runs. Must use the schema mapping tool to create the mapping. Must manage the Schema mapping language files. Requires more configuration.

8.8 Database Configuration

The following sections describe how to configure and set up different database systems to work properly with SanFrancisco.

8.8.1 Microsoft Windows NT DB2 5 - UDB

IBM SanFrancisco officially supports only version 5 of DB/2 on Intel and AIX. So far, we only have significant tuning experience with DB/2 on Intel/NT platforms. There are over three dozen performance-related parameters that can be used to configure DB/2. Of all these, the following are several that we have found that can have significant effects on performance with IBM SanFrancisco applications.

8.8.1.1 Pre allocating the Catalog, User, and Temporary Tablespace

When you create a database to hold SF business objects, you can specify which files will be used to hold the database tables and the size of those files.These are, in DB/2, called containers, not to be confused with object oriented container classes. Doing this creates the space for these ahead of time so that when SF application starts filling the tables with data, the database never has to pause to get file space or extend the current files. The following example shows an example DB/2 create database command that pre-allocates the database catalog at 20 MB in size (5000 x 4K pages), the user tables (containing the actual object state) at 1GB in size (4 x 65535 x 4K pages), and the temporary table-space at 16MB in size (4000 x 4K pages):

```
create database sfdb2 on 'e:'
catalog tablespace managed by database using
(file 'e:\db2data\sfcat00' 5000)
user tablespace managed by database using
(file 'g:\db2data1\sfusr00' 65536,
file 'g:\db2data1\sfusr01' 65536,
file 'g:\db2data1\sfusr02' 65536,
file 'g:\db2data1\sfusr03' 65536)
temporary tablespace managed by database using
(file e:\db2data1\sftmp0' 4096)
```

Note that table-space actually allocates space over four separate files (sfusr00 - sfusr03). This is to allow the database to use available empty space on one file while another is being extended.

8.8.1.2 Explicitly Setting the Number of DB/2 Buffer Pages

The other significant parameter we have found is the buffpage parameter that tells DB/2 how many 4K memory pages it should use for its data buffers. The key is to make this value as large as you can without overcalling memory to the point where you force the system to start trashing. The values we have found to work the best with our test applications are between 2000 and 20000 pages (roughly corresponding to a main memory size between 128MB and 1GB). This is for a system that is basically running SF applications and nothing else. Other applications that have memory requirements will effect this number. For example, to give 20MB to DB/2 for its database buffers, you can use the following DB/2 command:

```
update db cfg for sfdb2 using buffpage 5000
```

Because of a quirk in DB/2, explicitly setting the buffpage parameter, as previously suggested, only works if the database has been configured with the following buffer pool parameter:

```
alter bufferpool ibmdefaultdb size -1
```

8.8.1.3 Multi-Client Applications in DB2/NT

For using UDB in a multi-client application, set the parameter:

```
db2set DB2_RR_TO_RS=YES
```

This should be used to decrease lock contention (increase performance) but is only needed in a multi client environments. This is probably the normal setting for an average application.

8.8.2 IBM AS/400 DB2/400

If you plan to use DB2 for AS/400 systems as your data store, make sure it meets the requirements described in the following sections.

8.8.2.1 AS/400 Configuration Example

Create a SQL Collection on the AS/400 system. The SQL Collection is the schema of the table on the Schema Mapping Tool. In the AS/400 system, a Collection is a library where the database files reside. This can be done with the AS/400 command CRTLIB (create library) or STRSQL (to create an sql collection). Which one to use depends on if you are using the database natively or through SQL if you are using the default schema mapper

For the Default Schema Mapper (DSM), the Schema Mapper sets the default schema name to IBMSF. You can change the schema name by changing the qualified table name while you are configuring the Class Token. The schema name can be set at the container level.

You must have that Collection on the AS/400 system before running the SanFrancisco applications if you are using the extended schema mapper.

For the Extended Schema Mapper (ESM), the default schema name is IBMSF. The schema name can be set at the container level. You can change the schema name by changing the qualified table name on the Schema Mapping Tool. You must ensure that you create the Collection under the same schema name, for example, the table name is ibmsf.mytable. You can change to the table name mycoll.mytable.

Store the SML file on the AS/400 system (ESM only). Once you create the SML file using the Schema Mapping Tool, you need to store this SML file on your AS/400 system's IFS directory. In addition, you need to configure the SML root directory for each container. For each class, you must specify the SML file name from the root directory.

8.8.2.2 Restrictions on the Data Type to Use in the Tables

Currently, DB2 for AS/400 systems do not support the conversion from a CHAR column to a numerical column. You will receive the run time DataStoreException if you use this mapping.

DB2/400 does not support any column name that is one of the reserved words, such as LONG, DECIMAL, or BINARY.

8.8.3 Oracle on Microsoft Windows NT

SanFrancisco store Entities to Oracle using operating system authenticated connections to the Oracle database. This type of connection means that SanFrancisco will not pass a user or password to Oracle when obtaining connections. It is the system administrator's responsibility to start the SanFrancisco servers while logged onto Windows NT, using an authenticated Windows NT user and password. The existence of a operating authenticated Oracle user allows this Windows NT authenticated user and password to obtain connections to the Oracle database without supplying a user name or password.

The following text is a description of the SanFrancisco requirements for configuring and administering Oracle.

Create a Windows NT system user profile under which the SanFrancisco factory service process runs. You must sign onto Windows NT under this profile when starting the SanFrancisco servers. We recommend that you name this user IBMSF, but you may name it whatever you like. SanFrancisco persists Entities into tables that are schema qualified. In Oracle, an Oracle user of the same name owns every schema. Thus, for every schema that you use, you must create an Oracle user of the same name. When configuring Entities for a container, configuration can control the schema that you use. If using default or extended schema mapping, the default schema uses ibmsf. You can change this value for both default and extended schema mapped classes (see the documentation for configuring classes to containers and the Schema Mapping Tool). The space for the tables that SanFrancisco creates and uses is charged to this Oracle user. Thus, the Oracle user must be given sufficient quota size for the USER_DATA and TEMPORARY_DATA table spaces.

SanFrancisco depends on operating system authenticated users when connecting to the Oracle database. Oracle defines an operating system authenticated user by creating an Oracle user named <os_authent_pr><NT user> and configuring the user as authentication external. You can define the <os_authent_pr> value in the Oracle Instance Manager. By default, Oracle sets this value to OPS$. We recommend that you set this value to a null string (" "). For an NT user of IBMSF, and an os_authen_pr value of " ", an Oracle user named IBMSF (configured as authenticated external) allows the NT user IBMSF to connect to the Oracle database using operating system authenticated connections. All authority checking performs against this operating system authenticated Oracle user. Thus, this user needs privileges to any data that SanFrancisco accesses. Give the client authenticated Oracle user (IBMSF) appropriate privileges to access the database and perform operations on the tables configured to persist Entities. The CREATE SESSION system privilege is necessary to allow connections to the database. You must grant Insert, update, and select authority to any tables that persist Entities. When accessing tables that are in a schema different from the Oracle user making connections, you must grant special privileges (to access these schemas). Special privileges include: CREATE ANY TABLE, CREATE ANY INDEX, INSERT ANY TABLE, UPDATE ANY TABLE, and SELECT ANY TABLE.

8.8.4 Query Pushdown

Query pushdown is the process of converting a query over a set of IBM SanFrancisco Business Objects into a form that the underlying data store processes efficiently. When querying objects that are stored in a relational

database, it converts the query to an SQL SELECT statement. By converting to a query that the database understands, the restoring of every object in the table is prevented, and any indexes existing over the table can be used to further increase the performance of the query.

It is important to note that the distinction between object query and query pushdowns is transparent to the developer. A IBM SanFrancisco query operates on IBM SanFrancisco collections. There are no separate interfaces for performing a query pushdown. The only difference between the two approaches is performance. In most cases, query pushdown gives the same or better performance than object query. As collections become larger, the benefits of query pushdown increase dramatically. Therefore, it is to the developer's advantage that query pushdown be performed as often as possible.

8.8.4.1 Using Query Pushdown

An EntityOwningExtent is the only SanFrancisco collection that allows for query pushdown. The EntityOwningExtent class provides the ability to perform efficient queries against large-scale collections by tying the collection directly to the underlying data store.

The OOSQL (Object Oriented Structured Query Language) select statement is based upon methods that are invoked on objects. Since the database contains columns with data and no methods, there are limitations upon the types of methods you can use for the query. If these conditions are not met, a SanFrancisco query results in a Partial Pushdown or an object query. The following list shows the requirements on the query to be eligible for pushdown:

- Methods must not contain parameters.
- Methods must map directly to an attribute.
- Methods must return a supported data type.
- The attribute can not be streamed into a column with other attributes.

There are also requirements on the query statement itself, which include:

- It must not contain select compare or sort compare objects.
- It must not query over attributes that are stored in different tables. If Schema Mapping an object to multiple tables, only attributes stored in the primary table may be queried.

8.8.4.2 Method to Attribute Mapping

An object-oriented SQL query specifies methods on the objects being queried. To schema map an object to the database, the Schema Mapping Tool must map each attribute in the object to a column in the table. Since SanFrancisco query relies on method names, Query Pushdown must rely on an additional mapping from method name to attribute name. To do this, the Default Schema Mapper relies on a naming convention for the method to attribute mapping.

The naming convention is quite simple. If there is a get method, say getEmployee, then the DSM looks for an ivEmployee attribute. An iv replaces get. If this condition is met, then the method is eligible for Query Pushdown.

It is also possible for a navigation chain to push down to the database. A navigation chain is a list of methods that are performed to retrieve the desired element. For example, x.getBestEmployee()..getManager()..getName() would return the name of the manager of the best employee in the company. Each

method in the navigation chain must specify a method that maps to an attribute using the naming convention. Note that the last method in the chain must return one of the supported data types, but all methods previous to the last must return objects.

If the methods defined on the object do not follow the naming convention for the method to attribute name, then the Extended Schema Mapper can be used to manually specify the method to attribute mapping.

For more information on query pushdown, see the *Optimization with Query Pushdown* section in the *Persistent Object Planning* chapter of *Administering and Configuring IBM SanFrancisco* in the IBM SanFrancisco Base documentation.

8.8.5 EntityOwningExtent

The EntityOwningExtent collection is semantically similar to a Map, but it does not allow Entity elements to be added. Entity elements are always owned by the EntityOwningExtent collection and must be created "into" the EntityOwningExtent. That is, at creation time, the creation Location must be the EntityOwningExtent collection object. An EntityOwningExtent has semantics that allow it to be mapped to a table, or to a subset of a table, in a database. This allows many efficiencies in the implementation of EntityOwningExtent that are not possible with the other collections.

EntityOwningExtent can be configured so that the rows of an existing database table are implicitly elements of the EntityOwningExtent. With all other collection types, the only elements that are part of the collection are those objects that have been explicitly added to the collection. This makes EntityOwningExtent the only collection type that can be used to map to legacy data.

The EntityOwningExtent collections are intended to be the most scalable of the collections because they can be mapped directly to the database. The other Entity collections are designed to be more flexible but may not be as efficient for very large collections. For interface details, see the EntityOwningExtent Javadoc.

The other major difference between an EntityOwningExtent and a Map is that an EntityOwningExtent supports only method keys. That is, multiple keys are supported, but they must be defined from the methods supported by the elements' interfaces.

The concrete Map, on the other hand, supports a single key, which can be any arbitrary Base object or String.

SanFrancisco operations that might be pushed down to the database (like query() and getElementBy() on an EntityOwningExtent) always return the correct results. The results reflect any Entity creation, deletion, or update made in the current transaction without the programmer needing to make any explicit calls to syncEntity(). To accomplish this, the Foundation Layer may, at times, implicitly synchronize dirty Entities with the underlying datastore, but the programmer must not rely on when or whether that will occur. If you need to ensure that the actual database record is updated before commit() is called (for example, to trigger database operations like constraint or duplicate key checking), then you must use the syncEntity() method.

EntityOwningExtent that are accessed NO_LOCK have the following limitations:

- An EntityOwningExtent accessed NO_LOCK throws an exception when an attempt is made to insert an element or to remove an element.
- Query pushdown on an EntityOwningExtent accessed NO_LOCK does not see changes made to NO_LOCK copies of the elements in the EntityOwningExtent.

For more information see the *Programmers Guide*.

Chapter 9. Java Coding Tips

Like any other programming language, Java provides different ways to solve given problems. Some of these ways perform better than others. To increase the overall performance of your application, you should be aware of the implications certain code has. Some practices (like reevaluating the same expression in a loop) are just bad programming and not language dependent. Other practices (such as using those classes or constructs that give you the best performance while satisfying functional requirements) may depend on the language or the implementation of the language.

This chapter provides you with some basic information on Java performance. Because Java is a relatively young language, additional judgement is required. You may find your optimized code performing completely different with the next release of your JVM (Java Virtual Machine). It may be the best idea to take a look on the code that really inflicts problems. If you have no *serious* performance problems, you might be tempted to forget about performance tuning and just count on future versions of the Java implementation to do these optimizations for you—like the most good optimizing compilers for C and C++ do today.

Additional information and a variety of different hints about optimizing Java code can be found at: http://www.ibm.com/Java/Sanfrancisco/tips/ExtHome.html and http://www.cs.cmu.edu/~jch/java/

Also *Object-Oriented System Development* by Dennis de Champeaux, Douglas Lea, and Penelope Faure has a helpful chapter about performance tuning.

Note

All performance comparisons within this chapter are the result of our tests. The actual performance improvements for your own code may vary. You may not recognize any difference by simply running the source code. The differences are normally within the fluctuation of the provided measurement utilities. All becomes clear while using a profile that measures CPU time.

For all tests, we used a IBM Personal Computer 365, PentiumPro 200MHz, 112 MB RAM. To analyze the performance impacts of any given tip, we used JProbe for Microsoft Windows NT that includes a JDK that is based on version 1.1.5 and an instrumented JVM. Newer versions of the JDK, or different implementations of the JVM, may give different results.

9.1 The Idea Behind the Tips

Overall performance of an application depends on multiple factors:

- Hardware
- Hardware configuration
- Application architecture

And last, but not least, on the application code this is how a given architecture or design has been implemented.

It is clear that all designers and programmers must be aware of performance while writing applications. These tips and techniques should provide them with an easy way by following the rules that others have learned during prior experiences. We have divided all tips in three groups. The first is giving a general guidance that applies to most languages. The second part is resource (memory, in this case) related; whereas, the third one is more Java specific.

To give you an idea first of the overall performance implications of Java code, we have provided a table of the normalized costs of different Java operations.

Table 5. Java Operations Costs

Operation	Example	Normalized Time
Local assignment	i = n;	1.0
Instance assignment	this.i = n;	1.2
int increment	i++;	1.5
byte increment	b++;	2.0
short increment	s++;	2.0
float increment	f++;	2.0
double increment	d++;	2.0
Empty loop	while (true) n++;	2.0
Ternary expression	(x<0) ? -x :x	2.2
Math call	Math.abs(x);	2.5
Array assignment	a[0] = n;	2.7
long increment	l++;	3.5
Method call	funct();	5.9
throw and **catch** exception	try{throw e;} catch(e) {}	320
synchronized method call	synch Method()	570
New Object	new Object()	980
New array	new int[10]	3100

Note

Sometimes a performance optimized code results in a less readable code. In this case, you have to decide carefully which is more important, either a more readable code or a better performing one. Keep in mind that a more readable code is easier to maintain.

9.2 General Techniques

There are some coding techniques that are not specific to Java and can be applied to many other languages as well.

9.2.1 Loop and Counting

Common statistics reflect that you spend 90% of the time in only 10% of the code. The larger part of this code consists of loops. Because you spend much time inside loops, you would want to minimize the code that is inside the loop and reduce the number of iterations. There are different ways to decrease the evaluation time of a loop.

9.2.1.1 Moving Invariant Code

It is a good idea to move any code that is constant for the loop out of the loop itself. The following example shows the code that you may write quite often:

```
void doSomething()
{
   long test, result;
   result = 0;
   for(int i=0; i < 10000; i++)
      {
         test = 356 * 323;
         result = result + i*test;
      }
}
```

Obviously, the calculation of `test` always gives the same result. To reduce the execution of this code to the minimum, move this calculation outside the loop:

```
void doSomething()
{
   long test, result;
   result = 0;
   test = 356 * 323;
   for(int i=0; i < 10000; i++)
      {
         result = result + i*test;
      }
}
```

You can save 9999 calculations of the value of test with this little adjustment. Even if you do not have a variable in the first case, it is worth the effort to move constant calculations out of the loop.

9.2.1.2 Terminating Loops Early

You can spend much time in loops that already have the final result. The following example evaluates the complete array before it leaves the loop:

```
public boolean isReadOnly()
{
// return true only if all elements in the array are ReadOnly
   boolean readOnly = true;
   for (int i = 0; i < ivListenerIndex; i++)
   {
      readOnly = readOnly && ivListenerControlList[i].isReadOnly();
   }
```

```
   return readOnly;
}
```

The following improved example evaluates the code until it reaches the final result:

```
public boolean isReadOnly()
{
// return true only if all elements in the array are ReadOnly
   for (int i = ivListenerIndex; i > 0; i--) {
      if (ivListenerControlList[i].isReadOnly() == false)
      {
         return (false);
      }
   }
   return (true);
}
```

The time you gain with this improvement depends on the position of the first element that terminates the loop. If you have to check the loop multiple times, it may be a good idea to think about the order of the elements.

9.2.1.3 Using Local Variables

The fastest access mode to data is a local access. You can use this information to further improve the performance of loops by avoiding the access of global variables within the loop.

The following code uses direct access of the global variable:

```
int a1[] = new int [1000000];

public void test()
{
   int a2[] = a1; // included for comparison
   long start = System.currentTimeMillis();
   for (int i = 0; i < 1000000; i++)
   {
      a1[i]=55;
   }
}
```

By changing the access to the reference `a2`, you gain around 12% performance for the execution of this loop, as shown here:

```
int a1[] = new int [1000000];

public void test()
{
   int a2[] = a1;
   long start = System.currentTimeMillis();
   for (int i = 0; i < 1000000; i++)
   {
      a2[i]=55;
   }
}
```

The main reason for the time difference is revealed by looking at the generated byte code. The access to the global variable results in calls to the `getfield` operation that consumes much more time than a local access. You gain an

advantage by moving this operation out of the loop and hence using it only once. You can verify the byte code by decompiling the class file with the following command:

```
javap -c classname > filename
```

9.2.1.4 Avoid Unnecessary Method Calls

While implementing loops, always remember that method calls have a performance impact. It always takes time to execute any method call. For example, by writing:

```
String buffer = getBuffer();
for (int i=0; i<buffer.length(); i++)
{
   buffer[i]++;
}
```

instead of writing:

```
String buffer = getBuffer();
int len = buffer.length();
for (int i=0; i<len; i++)
{
   buffer[i]++;
}
```

you increase the number of method calls dramatically. This is safe to do, because in Java, Strings are immutable and can not change their size. For example, with a *buffer.length()* of 10 000, you will save 9 999 calls of the *length()* method.

In other cases, a method call can result in a remote call that might, depending on the connection, take several seconds of additional execution time. This dramatically increases the time saved that can be achieved by avoiding a call.

9.2.2 Using Buffered Data Streams

If you use unbuffered I/O streams, you may end up with single byte read or write operations. Note that the JDK (Java Development Kit) I/O classes use a lot of synchronization, so you might get better performance by using a single "bulk" call, such as *readFully()* and then interpreting the data yourself. Also notice that the Java "reader" and "writer" classes were designed for improved performance. For example, instead of using:

```
DataInputStream reader = new DataInputStream(new FileInputStream(in_file));
DataOutputStream writer = new DataOutputStream(new
   FileOutputStream(out_file));
```

use:

```
BufferedReader reader = new BufferedReader(new FileReader(in_file));
BufferedWriter writer = new BufferedWriter(new FileWriter(out_file));
```

9.2.3 Reduce Code Execution

While designing your application, you often have the choice between multiple objects and methods to get your job done. By being careful in your decisions, you can reduce the amount and weight of the executed instructions. Often, you want to do a simple task and call a method that does this for you. Unfortunately, the method belongs to an object with a completely different purpose. This object may

be too large, including too much functionality, which is more memory consuming. Even worse, it may delegate the job to other objects, which results in multiple object creations. With a little additional effort, you can find the object in which the task is delegated and use this object directly.

9.3 Memory Management

Although the price for memory is dropping, it is still a limited resource. You should consider the fact that Java is built for distributed environments and thin clients where size does matter.

9.3.1 Using Primitives

While using objects, you should always consider the fact that an object is a complex structure. On the other hand, a primitive is a lean structure. By knowing this, it is quite obvious that the performance of our application will increase if you use primitives instead of objects wherever possible.

To measure the difference, we ran the 100 000 iterations of the following two code examples:

Example 1: using `int`

```
int a2[] = new int[100000];
for(int j = 0; j < 100000; j++)
{
   a2[j]=10;
}
```

Example 2: using `Integer`

```
Integer a2[] = new Integer[100000];
for(int j = 0; j < 100000; j++)
{
   a2[j]=new Integer(10);
}
```

The difference of the runtime of both implementations is quite obvious, as shown in Table 6.

Table 6. Using int or Integer

int	Integer
10 ms	961 ms

With this difference in mind, you should carefully judge in which case a primitive can be used. It may be better to convert the data into primitives before an operation is made. This depends on the number of iterations your data runs through. The exact point after which a conversion is useful depends on the given problem. It may help to measure both implementations with test cases to select the better solution.

9.3.2 Reusing Objects

The creation of new objects takes a considerable amount of time and creates the need to get rid of these objects after they have been used. Especially for large objects, this fact has to be carefully considered. You may want to reuse these

objects by updating the fields of an existing object that is no longer used rather than generating new objects all the time. This also applies to lists of objects where you may provide an additional list of currently unused objects. There is an easy way of reusing objects that can be described in these steps:

1. Look on the free list to see if one is available to be reused.
2. If so, retrieve the reference to the reusable object. If the free list does not contain any reusable objects, create one as you normally would.
3. Take the reusable object off the free list.
4. Reinitialize the object (you may want to add a method that specifically does this).
5. Use the object as you normally would.

The following example shows how to follow the basic rules for reusing objects:

```
class Order
{
/* holds the recyclable orders */
private static Stack freelist = new Stack();

Order()
{
/* whatever it takes to create a new Order */
}

private Order reinit()
{
/* whatever it takes to initialize an old order into a new state */
}

public void recycle ()
{
   freelist.push(this);
}

public static Order getNew()
{
   if (freelist.empty())
   {
      return new Order();
   }
   else
   {
      return ((Order)freelist.pop()).reinit();
}
}
```

Instead of using *new Order()*, you use *Order.getNew()*,which generates a new object only if there are no preused left. Whenever an Order is no longer used, it is recycled and prepared for reuse by the *recycle()* instance method.

9.3.3 Reduce Object Size

Often there is more than one object that has the required capabilities. Similar to the already mentioned usage of primitives, you should carefully look at which object fits your needs best. For example, a list of objects can be stored in a

collection (in SF), a vector (refer to 9.4.6, “Vectors” on page 158), or an array. Perhaps you do not need the additional functionality of a collection. In this case, it is possible to choose an array that is a much leaner construct. This reduces the amount of memory required and increase your performance.

9.3.4 Free Resources

You may trust the garbage collector to get rid of all resources that are no longer needed. This is true, but do you know when the resources are collected? This is a major concern for external resources, such as files or database connections. If you no longer use those, free them so that the resources are available immediately, and you do not have to wait for the garbage collector to do this. Especially in long running methods, it may be worth to take a close look at your code to find out at which point these resources are no longer required.

9.4 Java-Specific Tips

Overall, there are some Java specific topics concerning the implementation of certain objects in Java.

9.4.1 String Operations

Because Strings are immutable in Java, you have to be careful about concatenation of Strings. Any concatenation of Strings results in the generation of new objects, which is a major performance cost. On the other hand, a StringBuffer is generated once and reused thereafter. Using of concatenations such as:

```
String test = "";
test += "Active ";
test += "Buffer";
```

performs worse than using a StringBuffer:

```
StringBuffer test = new StringBuffer();
test.append("Active ");
test.append("Buffer");
test.toString();
```

When executing it only once, you may not find any difference in the performance of both codes. To average the influence of one execution, we executed the examples 10,000 times. The significant difference clearly favors the usage of a StringBuffer if you have to concatenate more than two Strings.

Table 7. String versus StringBuffer

String	StringBuffer
4,181 ms	991 ms

Note

The actual execution times on your machine may vary.

You also may consider using a char array that will perform even better (refer to 9.3.1, “Using Primitives” on page 152). This solution, of course, decreases the readability of your code.

9.4.2 StringTokenizer

As in most other languages, there are different ways to divide a String into single tokens. The most comfortable way is the usage of the StringTokenizer. Unfortunately, the performance of the StringTokenizer is a disadvantage of this useful class.

The standard implementation using StringTokenizer (like suggested in the JavaDocs) is:

```
String result;
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens())
{
   result = st.nextToken();
}
```

This returns the expected results, but it does not perform very well. If you do not need the flexibility of the StringTokenizer, you may consider writing your own implementation for tokenizing Strings, such as:

```
int i1=0;
int i2;
String stringToParse = "this is a test", result;
i2 = stringToParse.indexOf(" ");
while (i2 >= 0)
{
   result = stringToParse.substring(i1, i2);
   i1 = i2+1;
   i2 = stringToParse.indexOf(" ", i1);
}
result = stringToParse.substring(i1);
```

This code is giving the same results as the first one, but it performs significantly better. During our tests, we ran both codes 100 000 times, which produced the results shown in Table 8.

Table 8. Compare StringTokenizer with Own Implementation

StringTokenizer	Own Implementation
2123 ms	1042 ms

Note

This implementation of the StringTokenizer is a good example where better performing code is less readable. Judge carefully when to increase performance and when to increase readability.

9.4.3 Function Inlining

A Java compiler can inline specific methods to reduce the number of method calls and increase the performance of an application. However, there are some restrictions for code being inlined by the Java compiler. The method has to be:

- Final
- Private
- Static

If your code often calls methods that have none of these modifiers, you should think about providing an additional method that fits these restrictions. This can increase the performance by elimination of unnecessary function calls.

A special form of code inlining is the usage of constants. Constants, which are variables that are declared static final, are important for performance because they allow a lot of optimization. For example, constant folding and inlining can be done if the compiler knows that the variable will never change its value. If you know that a variable is used as a constant, make sure to declare it as static final in order to reap the performance benefits.

Note

Remember that both the static and final modifier have to be set for inlining by the compiler.

Also, avoid calling new to initialize a constant String, for example:

```
static final String MY_STRING = new String("XYZ");
```

The previous example calls new and the String constructor when it is not needed, where the following code represents a better implementation:

```
static final String MY_STRING = "XYZ";
```

9.4.4 Exceptions

Exceptions are meant for situations that are truly unique. They require time and space to be instantiated. Performance-wise, it is expensive to catch an exception. Considering this, you should not design any method that has an exception as the normal result. Instead, this method should return a value with the necessary information. Otherwise, it is bad for performance to test every time for occasions that happen only if an error occurs. These cases should be covered by an exception. For example, testing for a negative return value to say you are out of memory would be unwise; that would be a situation calling for an exception case.

To enable the compiler to optimize your code, it is not a good idea to put a try/catch block around every method call. If possible, it is much better to put a larger block around several method calls instead of implementing several blocks for each method, as shown here:

```
public void some()
{
   try
   {
      some.method1();
   } catch (method1Exception e)
   {
// do something with the exception
   }
   try {
      some.method2();
   } catch (method2Exception e)
   {
// do something with the exception
   }
   try
```

```
   {
      some.method3();
   } catch (method3Exception e)
   {
// do something with the exception
   }
}
```

You should consider to include multiple method calls within one block, as shown in the following example:

```
public void some()
{
   try
   {
      some.method1();
      some.method2();
      some.method3();
   }
   catch (method1Exception e1)
   {
// do something with exception 1
   }
   catch (method1Exception e2)
   {
// do something with exception 2
   }
   catch (method1Exception e3)
   {
// do something with exception 3
   }
}
```

This improved implementation enables a JIT, or static compiler, to perform optimizations, for example, by rearranging code for increased performance. This optimization can only be done within a block. As described in the following different examples, sometimes using exceptions to terminate a loop is the preferred way to exit rather than constantly checking for a certain condition.

Original implementation with repeated call at the top of every loop to check if there are more elements:

```
while (iterator.more())
{
   some.method(iterator.next());
}
```

Improved implementation by terminating the loop through the exception path when an attempt to get the next element fails:

```
try
{
   while (forever)
   {
      some.method(iterator.next());
// Assume that some.method() can handle null
   }
} catch (CollectionException e)
{
   ...
```

```
}
```

Another alternative is:

```
long cardinality = coll.getCardinality();
for (long i=cardinality; i > 0; --i)
{
   iterator.next();
}
```

9.4.5 Hashtables

Java provides a very powerful hashtable function. However, it should be used with care if you have performance concerns:

Note

The hashtable you get with the JDK is synchronized so that it can safely be used by multiple asynchronous threads and give predictable results. However, if your hashtable is not accessed asynchronously, then the synchronization overhead is not necessary, and a non-synchronized hashtable will perform better from a strict code execution perspective.

- It is important to specify the correct size of hashtable for good performance. Note that the default constructor Hashtable you get from the JDK is set for a default capacity of 101 and a load factor of 0.75. If these values do not fit your needs, set appropriate values in the Hashtable constructor. We recommend a load factor of 0.75 with an initial capacity of about 1.25 times the average number of items you think the table will eventually need to hold. If you specify a smaller number than this, then you risk the performance cost of having to rehash the entire table to a larger size each time the load factor is exceeded.
- It is also important that the hashcode itself is appropriate for the data being put into the Hashtable. An efficient hashcode is one that:
  - Is easy to calculate
  - Spreads the data evenly over the domain of the key values

  The idea, of course, is to eliminate collisions as much as possible. A good hash function is tricky to design.

For further information on the hashtables and hashcodes in SF, refer to 10.1.9, “Hashcodes” on page 173.

9.4.6 Vectors

Java provides Vectors as data structures that are more powerful and flexible than arrays. However, they are also much slower, so if you need to use a vector, keep this in mind:

> If you are using a Vector to maintain a list of items that you will be inserting and deleting from, remember that it is always faster to add or delete from the end of the Vector rather than from the front.

For example, when using a vector to maintain a list of free elements, it is much faster to put the newly freed elements at the end of the vector using `addElement()` and get it from the end of the vector using `lastElement()` when you need one, as in the following code fragments:

```
Vector bufferList;
// add a buffer to the free list
public void addBufferToFreeList(Buffer freeBuf)
{
   bufferList.addElement(freebuf);
}
// take a buffer off the free list (exception if none available)
public Buffer getBufferFromFreeList() throws NoSuchElementException
{
   return (Buffer) bufferList.lastElement();
}
```

When you create a Vector without specifying an initial size, the Vector is created with an initial size to hold just 10 elements. Each time you have to add an element to a Vector that is already "full", Java automatically creates a new Vector of a larger size and copy all the old elements to the new Vector. While functionally this is fine, the extra work of creating another Vector and copying the elements adds up to significant performance overhead. This can be avoided if a right-sized Vector is created from the beginning. If you know how many elements the Vector will eventually have to hold, you should allocate the Vector to be of that size when creating it. You can also specify how much the vector capacity will grow once it fills up:

```
Vector myVector = new Vector(50,20);
```

This code creates a Vector with space to initially hold 50 elements. Its capacity will grow in increments of 20 elements.

9.4.7 Synchronization

Java was designed to be a multi-threaded language, and in keeping with that philosophy, the JDK has a lot of synchronization built into it. For additional information on synchronization, refer to 2.1.4, “Synchronous versus Asynchronous Processing” on page 8. Synchronization offers you a relatively robust multi-threading environment. However, due to the increase execution time of synchronized operations, it decreases performance to do this much synchronization, particularly if there is no real need for it. Therefore, synchronization has advantages and disadvantages. Not doing enough synchronization gives incorrect results and subtle timing window bugs that can be extremely difficult to pin down. Over-synchronizing results in bad performance. The JDK has chosen robustness over performance, so most of the classes and methods in the JDK are synchronized. For example, Hashtables, StringBuffers, and Streams are all extensively synchronized. If your program is not multi-threaded, or if its threads do not access the same objects asynchronously, then you can improve the performance of your program by writing unsynchronized versions of the methods that these objects use or find an unsynchronized version that already exists.

Keep in mind that you can synchronize on a block of code rather than an entire method. This particularly makes sense if the block of code is only on some of the logic paths through the modules. On the other hand, synchronizing on a method is slightly more efficient than a code block, so if the code block takes up most of the method anyway, you may just as well synchronize the whole method. In case you are in a highly threaded environment, this must be balanced by the need to synchronize across all methods versus just a single code block.

9.4.8 Casts and Instanceof Operation

Remember, it takes time to perform a cast or an instanceof operation. Different from C++, where it is resolved at compile time, a cast in Java requires runtime checking. This should be avoided if it is possible. Normally, there is no application without casts and instanceof, but this should be reduced to the least possible number. For example, instead of writing:

```
MyClass castedParm = (MyClass) parm;
if (someCondition)
{
   castedParm.someMethod();
}
else
{
   return;
}
```

which performs the cast even in the false case where it is never needed, you can implement:

```
if (someCondition)
{
   MyClass castedParm = (MyClass) parm;
   castedParm.someMethod();
}
else
{
   return;
}
```

This implementation avoids the unnecessary cast. In the same manner, an instanceof can easily be avoided in cases where a violation of an instanceof is really an exceptional case. In this situation, you may want to use an exception, but only if this case is really rare (refer to 9.4.4, “Exceptions” on page 156), always keeping in mind the additional cost of an exception. This can be done by writing:

```
try
{
   MyClass castedParm = (MyClass) parm;
   ...
} catch (ClassCastException e)
{
   ...
}
```

instead of using:

```
if (parm instanceof MyClass)
{
   MyClass castedParm = (MyClass) parm;
   ...
}
else
{
   ...
}
```

9.4.9 Using the API

Several Java classes provide methods for special needs that are faster than a normal operation. It is, for example, much faster to do an *arraycopy()* than looping over all elements to do the job. Sometimes the Java classes include too much functionality and can be replaced by a different implementation that better fits your needs. This can reduce the overall cost of the usage of these objects. Even if the object fits your needs, you may want to implement a single method that performs better than the original one. You can do so by overriding the original method.

9.4.10 Use JIT and Static Compilers

Normally, Java code is interpreted, which takes additional time that can be reduced. For an additional performance boost, make sure your application is running on a JIT (Just In Time) compiler that can increase the performance of your application by 30% up to 3000%, depending on your application. For a normal application, you can expect an overall increase of about 50%.

You should also consider the usage of a static compiler. These tools are now available from different companies and can increase the performance of your application even better than a JIT compiler, especially for client GUI applications.

Chapter 10. SanFrancisco Coding Tips

The purpose of this chapter is to suggest some coding tips that may be employed while using or extending the SanFrancisco framework. Some of the techniques described here are generic enough so that they can be incorporated in any SanFrancisco development work, while some others are specific to an architectural layer. Since the Foundation layer has the most active components, a judicious use of these can reap immediate performance benefits.

This chapter describes the following topics:

- Section 10.1, “General Techniques” on page 163—Outlines a number of coding techniques that can be fruitfully employed for any development work on SanFrancisco frameworks.
- Section 10.2, “Foundation Layer Coding Tips” on page 173—Details the performance trade-offs while using the various Foundation Layer objects.
- Section 10.3, “Common Business Objects Coding Tips” on page 188—Covers specific coding techniques that can produce better performing code while dealing with the CommonBusinessObjects in SanFrancisco.

10.1 General Techniques

Using the GBOB benchmark, it was inferred that a lot of time is being spent on transaction services and creation and retrieval of objects. Figure 63 on page 164 shows the time distribution for some commonly performed functions.

Note

The information presented in Figure 63 is just one example and is specific to one application, in this case the GBOB Benchmark. The time distribution may be totally different for your application. It is strongly urged that you create a similar profile for your application so as to obtain the time distribution that is more relevant to your application. For more information on how you can go about doing this, refer to 3.2, “Profiling the Application with JProbe Profiler” on page 25.

The following list shows some of the methods (on `BaseFactory`) that consume a lot of system time:

- `getEntity()/getObjectFromHandle()`
- `createString()/copyString()`
- `createDependent()/copyDependent()`
  - Cascading "`gets`" : `paymentTerms.getRemainingAmount().getAllocationIdentifier()`. Here, the call to `getRemainingAmount()` creates a new object, `DPaymentTermsDetail` for temporary use, so that `getAllocationIdentifier()` can be handled. This results in more garbage since the `DPaymentTermsDetail` object will be discarded after this use. Such a situation can be overcome by using the `get<name>ForRestrictedUse()` [`getRemainingAmountForRestrictedUse()`] method, which is provided typically for this kind of use. These methods are to be used when the returned object is not going to be modified.

– Stateless Objects (for example, Policies and so on)

– Immutable Objects

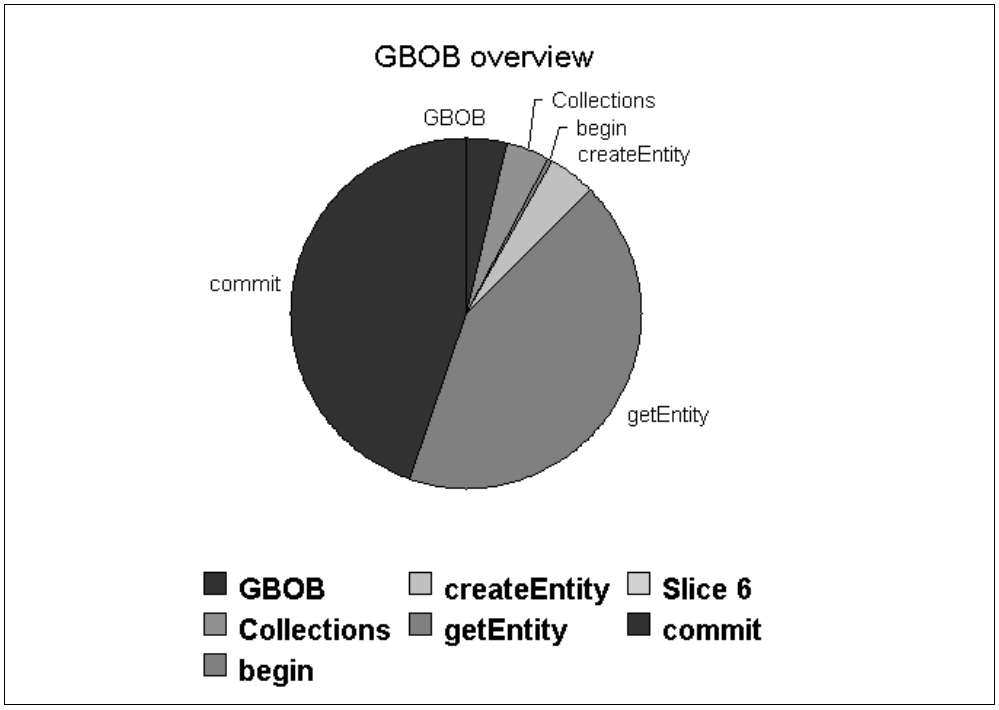


Figure 63. Where Time is Spent in a Test Application - The GBOB Benchmark

The reason for the excessive time consumption by these operations would become apparent once we become aware of the process involved in carrying out these operation. The flow charts in Figure 64 on page 165 show a macro level view of the process for some of the time consumption of the operations.

Each of the operations require an initial setup for setting local variables, and so on. The `getEntity()` operation checks if the requested object is currently in the container cache. If so, it checks if the object has been registered for the current transaction with a certain lock. If it does, a lock upgrade can be performed to lock it with the desired lock and then returned. If the entity has not been registered and locked, it needs to be locked with the requested lock and registered for the current transaction before it can be returned for use. In the worst case, if the entity is not in the cache at all, a new instance has to be created, its attribute values have to be obtained from the database, and the object has to be locked and registered before it can be returned back. In `createEntity()`, the stages are similar to the last scenario except that no cache check is performed since it is a new instance.

The `commit()` operation is more elaborate. After the initial setup, it goes into a loop checking each of the objects registered use in the current transaction. It determines whether an object has changed (that is, whether it is dirty), and if so, determines the type of operation that was performed on it. It then does the corresponding operation (write, insert, or delete), drops the lock on it, and starts the loop all over for the next object. After exiting the loop, the changes are committed to the database.

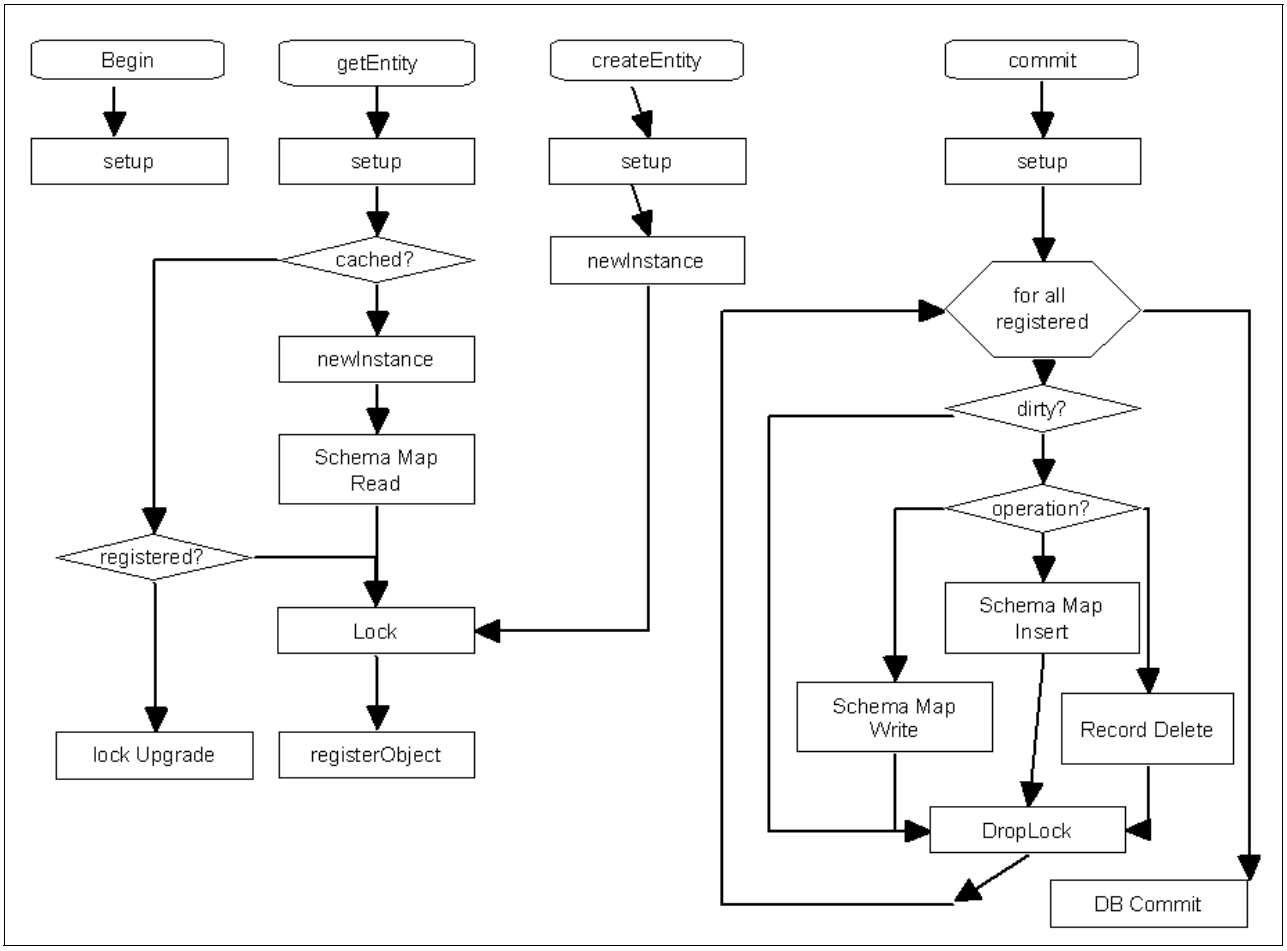


Figure 64. Flow Charts for Some Time Consuming Operations

Many times, due to faulty or careless programming, a large number of objects are created when it might not really be required. This results in lot of memory usage and also requires more garbage collection to be performed. This section describes many memory management techniques, such as caching, object selection, copy versus create, and so on. This section also explains some other common techniques that can be employed for performance benefit.

10.1.1 Caching

Method calls can be expensive if they are invoked repeatedly or cascaded, a practice that developers using OO languages very often resort to. A method call in Java is at least three times more expensive than referencing a local variable.

10.1.1.1 Intra-Method Caching

If the results of a method call are not going to change, and the results of the method call are going to be used more than once, the results of the initial call can be cached to a local variable for later usage. This can be called intra-method caching. Since SanFrancisco is a distributed environment, caching method return values are especially important because a method call may be a remote method call or internally invoke remote methods.

The `getEntity()` method is of particular importance to programmers in SanFrancisco. From a performance perspective, you have to know what you are going to do with an Entity after obtaining it. A `getEntity()` call is time consuming because it must go to the Business Object server process and get the object from the persistent store (at worst) or from its cache (at best) and return a local copy or a reference to its proxy (stub) depending on the access mode used. Now, if you are going to pass a local reference of the object, then simply pass the Entity reference obtained. However, if the Entity needs to be transported to a remote object or used in a Command that will be executed remotely, it would be prudent to pass the Handle itself. In such a case, you might not want to do a `getEntity()` at all since the creation of a proxy on your system is a waste if you are never going to call any methods on it. In such a case, just the Handle to the Entity should be passed in the command.

The following examples show common coding practices that need to take advantage of caching.

Example 1: Caching AccessModes

In the following piece of code, many AccessMode objects are created instead of creating them once and caching them in a local variable so that they can be used later:

```
Company company = CompanyContext.getActiveCompany(AccessMode.createNormal());
....
CurrencyController ctrl = getController(AccessMode.createPlusWrite());
....
Currency newCurrency =
CurrencyFactory.createCurrency(ctrl, AccessMode.createNormal(), ...);
```

The following improved version caches the AccessModes in a local variable and uses them:

```
AccessMode amNormal = AccessMode.createNormal();
AccessMode amPlusWrite = AccessMode.createPlusWrite();

Company company = CompanyContext.getActiveCompany(amNormal);
....
CurrencyController ctrl = getController(amPlusWrite);
....
Currency newCurrency =
CurrencyFactory.createCurrency(ctrl, amNormal, ...);
```

Example 2: Caching Global Factory Reference

The following example code has a loop where a reference to the BaseFactory is obtained each time a transaction is started and stopped and also when an Entity is obtained:

```
for (int i = 0; i < largeNumOfTransactions; i++)
{
   Global.factory().begin();
   ref = Global.factory().getEntity(handle, AccessMode.createPlusWrite());
   ....
   Global.factory().commit();
}
```

Instead, a local variable can be initialized to hold a reference to the BaseFactory. This can be used within the loop, as shown here:

```
BaseFactory factory = Global.factory();
AccessMode amPlusWrite = AccessMode.createPlusWrite();

for (int i = 0; i < largeNumOfTransactions; i++)
{
   factory.begin();
   ref = factory.getEntity(handle, amPlusWrite);
   ....
   factory.commit();
}
```

10.1.1.2 Inter-Method Caching

Another scenario where caching can be gainfully employed is the situation where the same data is accessed by a series of two or more method calls. Two areas worth considering are:

- Handle inflation
- Object navigation

Handle Inflation

Consider the following case:

```
someMethodABC(Handle handle)
{
   SomeEntity someEntity = factory.getEntity(handle, amPlusWrite);
   .....
   anotherMethodXYZ(handle);
   .....
   .....
}

anotherMethodXYZ(Handle handle)
{
   SomeEntity someEntity = factory.getEntity(handle, amPlusWrite);
   ...
   ...
}
```

There are two methods: `someMethodABC(Handle handle)`, which calls `anotherMethodXYZ(Handle handle)`. Both methods take the same parameter Handle and have to inflate it to obtain the Entity. In such a case, the second method, `anotherMethodXYZ(Handle handle)`, could be changed to `anotherMethodXYZ(Entity e)`, or it could be replaced altogether. This saves on the overhead of having to obtain the same object twice and provide better performance.

Object Navigation

On similar lines as the previous example, consider a case where there are two methods, `methodX()` and `methodY()`, which need to navigate several levels deep on an object to access data. This is typically by way of a series of `getXXX()` calls on a number of intermediate objects. In such a case, it is advisable to change the `methodX()` so that it stores the navigated results and then passes them onto the `methodY()`.

10.1.1.3 Process-Scoped Caching

Another opportunity for caching is with global data. When dealing with global data, the developer can cache the information scoped to a process in a static

variable or in the Distributed Process Context. See 10.1.7, “DPC Initialization” on page 172 for more information.

For more examples on method caching, refer to 9.2.1.4, “Avoid Unnecessary Method Calls” on page 151.

10.1.2 Object Selection

Often developers have a choice of different objects to accomplish a function. Selecting an object of the right weight is a critical issue for good performance. A good rule to follow is: Always use the simplest object possible, but no simpler. For example, if you need to store a number value, you could use the Java primitive `int`, a Java object `Integer`, or the Foundation Layer object `DInteger` (a dependent). Depending on what you are going to do with it, you would always be better of using the most primitive type that you can get by with, in this case, the Java primitive `int`.

An object that is heavier costs more both in time and space. It costs more to work with and requires more storage. Java primitive objects, especially `int`s, are very efficiently manipulated within the JVM (32-bits is a *natural* size to the JVM architecture). The disadvantage is that it is not an object, and hence, cannot be used like an object (for example, certain collections can only handle objects, not primitives). A distant second choice would be use of the Java `Integer` for the additional functionality that it offers as an object. Even in such cases, it is better to continue using a primitive `int`, and convert it to `Integer` only when you need that additional functionality. The SanFrancisco object, `DInteger`, is another choice. A Java `int` requires only 4 bytes, an `Integer` requires 20 bytes for the value and overhead, and SanFrancisco `DInteger` requires 24 bytes.

On similar lines, `boolean`, a java primitive, would be more efficient than using the `Boolean` object, which in turn, would be more efficient than the `DBoolean` SanFrancisco object.

An offshoot of the problem of object selection is when you have to make a choice between adding attributes to an object (by extending it) and adding properties to it. For example, let us say you have an Employee object, and you want to add the capability to store "title" information such as "Manager," "Director," "Architect," and so on. You could do this either by extending the Employee object and adding a String attribute "title," or you could add a Property "title" to the Employee object (if it is a DynamicEntity) using:

```
employee.addDirectlyContainedPropertyBy("Manager", "title").
```

Table 9 on page 169 can serve as a guide to make a choice between the two options.

Table 9. Choosing between Adding Properties to, or Extending, an Object

Extending and Adding Attributes	Adding Properties
To be used if it is going to be accessed often, or if the number of attributes is fewer in number.	To be used if large number of properties are going to be stored, or if these are going to be seldomly accessed.
To use this option in your application, you have to perform class replacement for sake of factory services, so that the extended object is used instead of the base object.	Adding a property internally results in creating and maintaining a `Hashtable`. Hence, it is an overhead to be used with fewer properties.
Query pushdown can be effected by appropriately mapping the newly added attribute to the corresponding columns in the databases. This results in improved performance.	The support for mapping properties of DynamicEntity to table columns will be available in a future release. It is not supported for release SF130.

For more details on object selection in general, refer to 9.3.1, “Using Primitives” on page 152. For information on properties, refer to 5.4, “Property Container Pattern” on page 74.

10.1.3 Object Streaming

When you need to pass an object parameter to a method that is executing remotely, the object needs to be streamed or flattened to be sent across the communication link to another process. There are several ways to stream an object, each with different performance characteristics. This section explores each of them in brief.

10.1.3.1 Declare as *java.io.Serializable*

You can simply define the class of the object to implement `java.io.Serializable` and be done with it. Java takes the responsibility for streaming all the attributes of the object except those that have been declared as `transient`. The streaming may not be very effective since the decision as to what is to be streamed is done at runtime. If streaming is done often or in a performance critical application, such a process may be not adequate.

10.1.3.2 Implement *java.io.Serializable*

To have more control over the serialization that Java does while streaming an object, you can implement the `readObject()` and `writeObject()` methods of the `Serializable` interface. These provide specific control over the streaming of the attributes of the object. However, it does not provide control over the parents of the object. Their serialization proceeds in the manner their classes have defined. For example, if one of the attributes of your object is a `Vector` that happens to be empty when you stream it, instead of streaming the empty `Vector`, just stream null. This can save the time necessary to flatten the `Vector` at the source as well as the time for fluffing the object at the target.

10.1.3.3 Implement *java.io.Externalizable*

If you want control over the streaming of an object, as well as that of its parents, then you object should implement the `java.io.Externalizable` interface. This means you need to implement the `readExternal()` and `writeExternal()` methods that provide control over streaming the object’s parents also. This can result in considerable savings in time, as the streaming process does not have to walk through the hierarchy of your object looking for `readObject()`/`writeObject()`

methods at each level. It also obviates the need to stream the parent's attributes if streaming your object does not need them.

10.1.3.4 GBOF Externalization

In SanFrancisco, streaming is used to transfer objects between client and server. The methods `internalizeFromStream()` and `externalizeToStream()` control which attributes of the objects are getting streamed. Every class (except the stateless classes) should implement these methods. The `com.ibm.sf.gf.Base` class implements this Externalizable interface. Its `readExtrenal()/writeExternal()` methods will call the SanFrancisco streaming methods. GBOF Externalization is about as fast as `java.io.Externalizable`..

10.1.4 Fast Conversions

Java allows conversions of any object to String object, and this is achieved by calling the `toString()` method on the object. For example, let us say you concatenate a number and a String, as in the following code:

```
for (int i = 0; i < someValue; i++)
{
   System.out.println("Loop counter value is " + i);
}
```

While this may not exactly be a performance hazard, if you really want to optimize on code, be aware that the JDK methods that convert the `int` to a `String` are currently inefficient. The SF Performance Team has written a class called `FastConvert` that converts the Java primitives into Strings a lot more efficiently than the JDK methods. The following code uses the `FastConvert` class from GBOF:

```
for (int i = 0; i< someValue; i++)
   System.out.println("Loop counter value is " + FastConvert.intToString(i));
}
```

Using `FastConvert` is eight to ten times faster than the built-in String methods.

The `FastConvert` class has a dozen other conversion utility methods for various types, such as decimal, Radix64, Hex, and so on. It is beneficial to utilize this class extensively in your code.

10.1.5 Tracing

A trace support is a powerful and flexible tool for debugging problems. You can have your own implementation, say `MyTrace`, for trace support. However, it can be a drag on performance if it is done incorrectly. The objective should be to minimize path length when tracing is not active. The following piece of code shows the recommended way of using a trace facility within methods:

```
if (tracingEnabled)
{
   MyTrace.println("Begin someMethod");
}
```

In this case, `tracingEnabled` is a static boolean variable that has been set to true or false depending on whether a trace should be taken at that point, as shown here:

```
static boolean tracingEnabled = MyTrace.isTraceLevelEnable(MyTrace.COMPONENT_X
| MyTrace.ANYTYPE)
```

`isTraceEnabled()` is an implementation you can provide to check if tracing is enabled for a component. When the class is first loaded, the `tracingEnabled` is set to true or false depending on whether tracing is currently enabled for `COMPONENT_X` (use you own `MyTrace` defined component name here). The only additional runtime overhead is the “if” test of the static boolean when tracing is not enabled for this component. To allow dynamic tracing to work, the `tracingEnabled` static variable should allow itself to be set by a static routine that can be called by the tracing support.

If you do not need the tracing to be turned on dynamically (that is, if you need tracing only for in-lab development purposes), you can condition the trace statement with a static final boolean that has been compiled to be either true or false:

```
if (finalTracingEnabled) {
   MyTrace.println("Begin someMethod");
}
```

where `finalTracingEnabled` is declared to be static final.

```
static final boolean finalTracingEnabled = true; // or false (or Debug.ON)
```

In this case, if `finalTracingEnabled` is false, the compiler eliminates the entire “if” statement as dead code. So, the compiled code looks as though the trace statement never even existed in the source code. Although this results in rather efficient coding, you lose the ability to dynamically set tracing ON or OFF without having to recompile your code. Note that you can use the static final boolean `Debug.ON` that has been defined for this purpose.

Finally, because of the need to keep class files reasonably sized, all unit test type traces should be of the `Debug.ON` variety (no code should be generated for them in the actual customer product).

10.1.6 Copy versus Create

This section is an extension of the 9.3.2, “Reusing Objects” on page 152. In most cases in SanFrancisco, it is faster to copy and modify a `Dependent` than to create a new one. When a transient `Dependent` is no longer needed, it can be recycled by reinitializing the object. A `Dependent` is copied when set as an attribute, hence, a transient `Dependent` used as an initialization parameter can be reused after reinitializing it. If a `Dependent` continues to be in use, it is generally faster to copy the `Dependent` to a new `Dependent`.

For example, the following code:

```
inStock = DDecimalFactory.createDDecimal(1, 1);
damagedStock = DDecimalFactory.createDDecimal(0, 1);
```

may be slower than:

```
inStock = DDecimalFactory.createDDecimal(1, 1);
damagedStock = ((DDecimal) Global.factory().copyDependent(null, inStock);
damagedStock.assign(0);
```

This may vary depending on the complexity of the logic to create an object and the amount of state that differs between the two instances.

10.1.7 DPC Initialization

A Distributed Process (DP) is based on the concept of a normal process, in that it is an anchor point for one or more threads that are actively working under it. The threads of a DP can be spread across one or more JVMs. The DP, like a normal process, has a context, or shared information space, that each thread in the distributed thread can access and modify. This context is called Distributed Process Context (DPC). Changes made to the DPC by one thread are available immediately to other threads in the distributed process.

A thread must be associated with a work area to access the DPC. Things that are scoped to the work area include transaction work, factory interface for NO_LOCK, and transient entities and security context.

During global initialization of an application, a distributed process and default work area is constructed and associated with client thread. When initialization is complete, the client thread has access to the DPC and can begin using the work area. The information stored here is typically timezone information, the locale currently in use, the precision values when working with `DDecimal`, the currently active company, and so on.

There are two ways to ensure that the DPC global data is initialized as soon as possible. You can:

- Define the initial default value in the `Global.name`.
- Set the value when it is not already set.

Currently, the DPC support allows primitives to be used in initial default values in the `Global.name`.

For handling situations where the DPC value may not have been set, the following logic should be used when attempting to retrieve the value:

```
DistributedProcess dp = Global.factory().getDP();
try
{
   scale = (Integer) dp.getSystemFieldBy("maxScale");
}
catch (SmContextFieldNotFoundException e)
{
   Integer scale = new Integer(5);
   dp.setSystemFieldBy(scale, "maxScale");
}
```

Additionally, the DPC allows for usage of any Java object (not just SanFrancisco objects). Here, using `Integer` instead of `DInteger` is a better choice.

10.1.8 Transient Entities

Generally transient objects are garbage collected when they are no longer referenced by any other object. On the other hand, transient entities are cached in Factory. They can be accessed by their handles.They are created as a result of:

- Query Result collections
- No-Lock copies
- Other usages, such as when an entity is explicitly created as transient by passing a null location handle.

These entities have to be explicitly deleted from the work area of the user, if not, they take up a lot of memory and thus cause performance degradation. They can be deleted by:

- `deleteEntity()`—Specify the handle.
- `deleteWorkArea()`— Results in deleting all the Entities of the work area, in addition to other information in DPC.
- End client process—Stops the client process and frees up the memory used by the Entities.

10.1.9 Hashcodes

Hashcodes are used by tables/collections for accessing elements. They are integer values returned by the `hashCode()` method, which all Java objects support.They are functionally similar to an index key used for looking up elements. Though hashcodes do not have to be unique, performance is best when no two elements in the same collection have the same hash code values.

While deriving `Dependent` subclasses, the `hashCode()` method must be overwritten if it does not inherit a suitable `hashCode()` method from some ancestor class. The value computed by `hashCode()` is normally computed from one or more instance variables of the object. A good `hashCode()` method should be fast and also returns values that tend to spread the instances of a class broadly across the range of integer values returned.

For more details on `hashCode()` method, refer to the *Programmers Guide* section on *Deriving Dependent Subclasses*

10.2 Foundation Layer Coding Tips

SanFrancisco is known to be a distributed environment. This means that objects can be made available anywhere in the network. Different options exist for accessing or working with a business object and choosing the right approach are crucial for application performance.

To make the right choice in accessing an object, you should know:

- How big the object is
- How many calls are going to be made to it
- What kind of data flows with those calls
- Communication link overhead
- Computing power of the server versus the client

Essentially, the choice you have lies within using a certain AccessMode and the use of Commands. These factors work together and have a cumulative effect on the performance. This section focusses on the use of AccessMode and Command objects and other Foundations objects that can have a profound bearing on the working of your application.

Figure 65 shows the distribution of time for some common operations in the Foundation Layer.

Note

The information presented in Figure 65 is just one example and is specific to one application, in this case the GBOB Benchmark. The time distribution may be totally different for your application. It is strongly urged that you create a similar profile for your application, so as to obtain the time distribution that is more relevant to your application. For more information on how you can go about doing this, refer to the 3.2, “Profiling the Application with JProbe Profiler” on page 25.

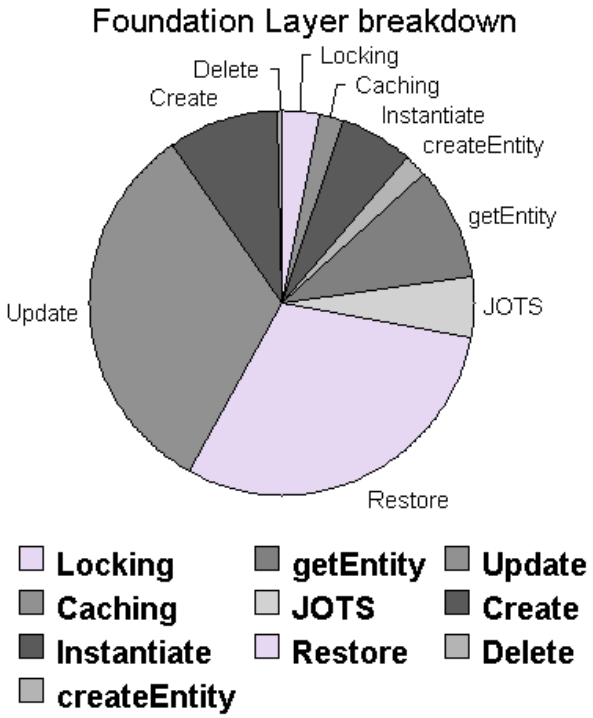


Figure 65. An Example of Time Distribution for Foundation Layer Operations

10.2.1 Commands

One of the most efficient ways to work with remote SanFrancisco objects is by well-planned use of the `Command` objects. A `Command` is a class of SanFrancisco that provides a way of wrapping up a number of method calls against an object. `Command`s are an effective way of shifting the execution of method calls to actual space of the target object on which they will be executed. Thus, the methods are executed in the same memory space and server process as that of the target.

10.2.1.1 Creating commands

Commands are Dependents and can be streamed and contained in another Entity or Dependent. This capability is used by business process components to

support logging of actions and undo/redo of commands. Commands are created by following the standard Factory pattern. The command factory, in turn, delegates the job to the BaseFactory.

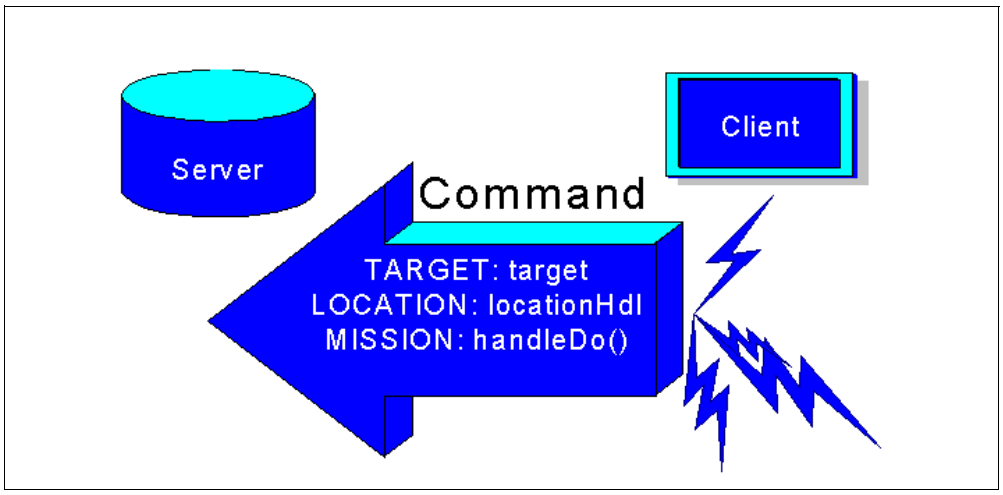


Figure 66. Command Loading - Specifying the Location of Execution

The Command factory has a number of static create() methods that allow you to specify different parameters while creating the command objects, depending on whether the target is a Dependent or Entity. These methods are reproduced here for reference:

```
static final <className> create<className>(Base target, <param1, ...>) throws
SFException;
//run command locally

For example:
public static final HouseAdjustRoomSizesCmd createHouseAdjustRoomSizesCmd(
House house, float percentage) throws SFException;
```

If the target inherits from Dependent, the following creation method is used:

```
static final <className> createContained<className> ( Base containingObject,
Handle locationHdl, boolean returnCommand, Base target, <param1, ...>) throws
SFException;

For example:
public static final HouseAdjustRoomSizesCmd createHouseAdjustRoomSizesCmd(
Base containingObject, Handle locationHdl,
boolean returnCommand, Handle houseHdl, float percentage) throws SFException;
```

If the target inherits from Entity, the following creation method is used:

```
static final <className> create<className> (Handle handleOfTarget, <param1,
...>) throws SFException;
//run in target server process, no return

For example:
public static final HouseAdjustRoomSizesCmd createHouseAdjustRommSizesCmd(
Handle houseHdl, float percentage) throws SFException;
```

```
static final <className> createContained<className>(Base containingObject,
Handle locationHdl, bolean returnCommand, Handle handleOfTarget, <param1,...>)
throws SFException;

For example:
public static final HouseAdjustRoomSizesCmd createHouseAdjustRoomSizesCmd(
Base containingObject, Handle locationHdl,
boolean returnCommand, Handle houseHdl, float percentage) throws SFException;
```

The factors that decide the actual place of execution of the command are determined by two parameters, which are passed to the factory `create()` method, namely the “target”, which can be either null, an Object (if derived from Dependent), or its Handle (if derived from Entity) and “locationHdl”, which can be either null or the handle of a persistent entity. The table in Figure 67 summarizes the location of execution for different input combination of these two parameters. For more details on other parameters, refer to the *Programmers Guide* for creation of Commands.

		Target		
		Object	ObjectHdl	NULL
locationHdl	Hdl	At the server containing the entity referred to by the locationHdl	At the server containing the entity referred to by the locationHdl	At the server containing the entity referred to by the locationHdl
	NULL	Locally, in the address space of the creator	In the server process containing the target, or if the targe is already accessed locally, then the command is also executed locally.	Invalid

Figure 67. Location of Execution for Different Values of Target and LocationHandle

If the target is specified by handle and no separate locationHdl is specified, target handle is passed to `BaseFactory` as locationHdl. This causes the command to be executed in server process of the target unless the target entity is already accessed locally, in which case, the command also is executed locally. This is the most efficient way of specifying the location of execution.

10.2.1.2 Commands and Transaction Scoping

Commands can be executed as independent transactions or as part of larger transactions. When they are used to define a transaction, they handle the transaction management by themselves with BaseFactory calls, such as `begin()` and `commit()`, and automatic recovery in case of failures with a `rollback()` call.

To execute a command atomically, use the `doTransaction()` call. It is not possible to undo/redo operations on the command if you use this method call. To execute a command as part of a larger transaction, use the `doAll()` call. This call allows undo/redo operations.

However, some framework provided commands that represent complex business tasks do not support undo/re-do. This can only be done by having the transaction

rolled back. Other commands that normally perform interactive tasks, such as data entry tasks, support the undo/redo operations. The `supportsUndo()` method call can be used to determine if a undo() is supported by a command or not.

If a lock is held on a business object that a command needs to modify, and the lock does not allow writing, the command can be programmed to attempt to upgrade the lock for modification. If the necessary lock type cannot be obtained due to a compatibility mode it has, an exception will be thrown.

10.2.1.3 Scenarios for Usage of Commands

It appears that top performance gains can be obtained from SF if most of the operations are performed using command objects. They can be used extensively for creating, deleting, and manipulating business objects and running complex business logic involving multiple objects.

`Commands` can be used specifically in the following scenarios:

- To retrieve a host of information from a remote object. All methods that need to be executed on the remote object are encapsulated in the `Command`.
- To execute a method call on a large number of remote objects. The `Command` can collect information from a large number of objects by executing a particular method call on each of them, collecting the results, and then sending it back to the caller.
- When the streaming costs of a business object are high.
- When the request involves access to other business objects.

Keep in mind, however, that using `Command` does not come free. It involves writing three new classes: the command interface, the factory for creating the command, and of course, the implementation. Creation of command also takes time. In cases where only one method call needs to be executed remotely on a single object, it would not be worthwhile to create a command. This is because of the additional costs involved in creating a command, streaming the command over to the target location, and returning the results back. In such cases, the standard method is recommended.

10.2.1.4 Generic Commands

Commands written should be as generic in nature as possible, so that commands used for a purpose can be reused, with some reinitialization, for use with other objects. For example, if we have to run a query on a collection, instead of hard-coding the command to be a particular type of collection, the collection to be queried can be passed as a initialization parameter. This command can then be reused with another collection for a similar kind of query by reinitializing the command with the new collection element. Use of generic commands reduces the the application code to write and causes the byte code to be loaded.

10.2.1.5 Reusing Commands

As with any other object, reusing `Commands` can also result in significant performance improvement. Thus, instead of creating a `Command` every time you wish to manipulate an object, you can usually reuse the `Command` (even those having state). The reuse is achieved by calling the `reset()` method on the command:

```
public final void reset() throws SFException, SFResetNotAllowedException
```

This method, in turn, calls the `handleReset()`, which changes the state of the command to NEW. This does not change the other initialization parameters, such as target, and so on, but allows another `doAll()` on the command. The other parameters could be changed by calling the appropriate public setter methods on the command class. This can be used specifically if you want to execute a series of commands that are almost identical.

If you have a command that does not have state, it can be reused as shown here:

```
MyCommand cmd;
...
if (cmd != null)
{
   cmd.reset();
}
else
{
   cmd = MyCommandFactory.createMyCommand(locationHandle);
}
cmd.doAll();
```

To reinitialize a command with a new set parameters, you should provide your own reinitialization routine. The following example shows how to re-use a `Command` that has state through a reinitialize method:

```
if (createDissectionCmd != null)
{
   createDissectionCmd.reinitialize(journalHdl, postingCombinationHdl.
   transactionValue);
}
else
{
   createDissectionCmd =
   CreateDissectionCommandFatory.createCreateDissectionCmd(journalHdl,
   (postingCombinationHdl, transactionValue);
}
createDissectionCmd.doAll();
```

Here is the implementation of the reinitialize method for the `CreateDissectionCmd` object:

```
CreateDissectionCmdImpl.java:
/*
-------------------------------------------------------------------------
/* Reinitializes this Command with the given values so that it can be executed
** again.
** @param Handle journalHdl (also the location handle of this Command)
**(Mandatory)
** @param Handle postingCombinationHdl (Mandatory)
** @param DTransactionValue (Mandatory)
** @return void
** @exception com.ibm.sf.gf.SFException Business Objects Framework Exception
**-------------------------------------------------------------------------
*/
public void reinitialize(Handle journalHdl, Handle postingCombinationHdl,
DTransactionValue transactionValue) throws SFException
{
   this.reset();
   ivJournalHdl = this.setHandleToHandle(ivJournalHdl, journalHdl);
```

```
        ivPostingCombinationHdl = this.setHandleToHandle(ivPostingCombinationHdl,
        postingCombinationHdl);

        ivTransactionValue = (DTransactionValue)
        Helper.setDependentToDependent(ivTransactionValue, transactionValue);

    }
```

10.2.2 AccessMode and Locking

AccessModes are used to control the way a shared, persistent Entity object is accessed. They are passed to the `BaseFactory`, and they determine lock type and access characteristics, such as the access location and lock wait time-out, for both Entity and any of its contained collection elements.

Figure 68 shows typical client server scenarios with different access modes. The AccessMode used by the client will determine whether the entities it accesses will be copied over to the client (that is, a proxy of the entity object is streamed over to the client), or they are accessed remotely on the server.

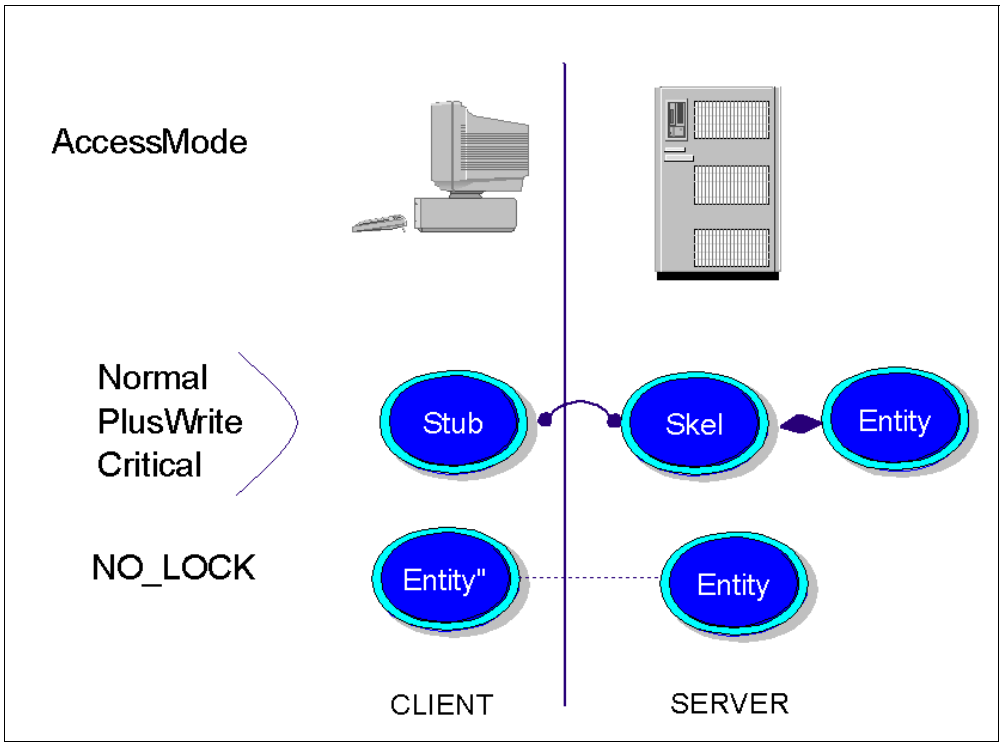


Figure 68. Object Distribution for Default SanFrancisco AccessModes

Programmers generally do not need to know the details of the various access mode possibilities. Special static create methods exist on the AccessMode class that return AccessMode objects suitable for most programming situations.

These methods for creating AccessMode objects are summarized here:

```
static final AccessMode createNomal() throws GFException,
SmWorkAreaNotActiveException;
//supports reading and upgrading to write, access location is HOME
```

```
//that is the server where the entity "lives".

static final AccessMode createPlusWrite() throws GFException,
SmWorkAreaNotActiveException;
//supports getting or upgrading to write access, access location is HOME.

static final AccessMode createCritical() throws GFException,
SmWorkAreaNotActiveException;
//supports exclusive access with high degree of isolation for short, critical
//tasks, access location is HOME.

static final AccessMode createNoLock() throws GFException,
SmWorkAreaNotActiveException;
//gets non-transactional, non-shareable access to data from the object
//access location is LOCAL, that is a copy is made on the client side.
```

The specific values returned from the preceding static create methods can be modified at run-time using the `set<name>()` methods. A modification affects all use within the callers DistributedProcessContext. These methods are used by first creating an access mode object using the static create method that allows specifying the values for every attribute of the object. Once an access mode object is created with the desired attributes, it is passed in the `set<name>()` method, which copies it for use in the corresponding subsequent static `create<name>()` calls.

10.2.2.1 Customized AccessMode objects

The primary purpose of creating a customized object is to use it on the static access mode methods to change the setting of the standard access modes (described above). Changing the standard access mode like this will normally result in the best performance because then all objects will be accessed consistently in that distributed process.

Attention

You should not directly use a custom access mode when accessing objects because you may run the risk of having significant performance or contention problems. This is because other code in the same transaction may use the standard access mode, resulting in a mixture of access modes, with some object being accessed differently from others. This mixture can cause bad performance due to excessive remote calls between the objects being accessed HOME and LOCAL. The alternative of trying to get all the code in the same transaction to use the same custom access mode object without changing the standard access mode is unreliable because of code that you didn't write being called. This includes IBM code (such as `AdhocQueryCmd`, which uses `createNormal()` access configured on the Entity that is the target of the command), or other third-party code that may be called without your knowledge because someone configured a third-party class to be substituted for one of your own or for one of the framework classes.

Creating a customized access mode object is done using the static create method on the AccessMode class. This method takes values for each of the access mode attributes as an argument as shown here:

```
static final create(int entityLock, // values defined in LockMode
```

```
int entityLocation, // values defined in AccessLocation
int entityCompatibility // values defined in CompatibilityMode
int waitTime) // values defined in Timeout
throws GFException;
```

The parameters that can have a bearing on performance are explained in the following sections.

10.2.2.2 Lock Modes

Concurrency control is achieved through the specification of the type of lock to get on the accessed Entity. Valid values for lock types are defined in the LockMode class with the following identifiers:

```
OPTIMISTIC
OPTIMISTIC_CLEAN
NO_LOCK
READ
WRITE
```

The details of each of these are available in the *Programmer's Guide*.

Approaches for Using of LockModes

A simplified way of looking at an application is to think of it as having three major stages:

1. **Read**—In this stage, data is collected from certain objects in order to present to the user. There are no modifications in user interactions.

2. **Work**—This is the stage with user interaction. The application needs to preserve requests of the user.

3. **Write**—Here, there are no user interactions, but the real objects are modified and committed.

Let us assume that each of these stages uses separate transactions for Read and Write stage.

Table 10 on page 182 shows what object access would be used in each stage of different approaches. The table also details the pros/cons for each approach and the performance trade-off while using the different approaches. The approaches are named for the dominant access mode used.

Table 10. Summary of Pessimistic, NO_LOCK, and Optimistic Approaches

	Pessimistic	**NO_LOCK**	**Optimistic**
Read	Pessimistic read access to collect information to display information to the user	Make NO_LOCK copies of relevant objects and use them to get information to display to user	Gets optimistic copies of relevant objects and uses them to get information to display to user
Work	Application uses local variables or UI object state to keep track of changes the user wants to be made	Use local variables or UI object state to keep track of changes user requests	Application makes modifications to the optimistic copies based on user interactions
Write	Pessimistic write access to make changes to real objects	Pessimistic write access to make changes to real object based on changes to NO-LOCK copies	Application can simply commit the transaction, which will succeed as long as no other transaction holds any pessimistic lock on the changed objects, and none of the objects have not been changed by other transactions
Pros/Cons	Must be careful about other transactions having changed your object during work stage. But this can be overcome in two ways. One is by maintaining a time stamp in the entity being changed, the other is by comparing all the attributes of the entity that was changed.	Allows use of an object view of the data during work stage. Current limitations with `refreshEntity()`, etc. that make it hard to program. Must be careful about other transaction having changed your object between the time you obtained the NO_LOCK copy and the time you got the write lock.	Allows object view of data during work stage. System will detect if objects have been changed by another transaction during commit time, and if so, will cause commit to fail. The burden of recovery from failure is on user. Current limitations with `refreshEntity()` make it hard to use.

Table 10. Summary of Pessimistic, NO_LOCK, and Optimistic Approaches

	Pessimistic	**NO_LOCK**	**Optimistic**
Performance	Best performing approach because only data to be displayed is copied. Makes it easy to use commands running on server for the read/write stage. Since pessimistic locks are not held during work stage, it reduces chances of a bottleneck caused due to restricting other transactions use of these objects.	Performance overhead of having to make NO_LOCK copies, which may be prohibitive if the objects are very large, such as collections other than `EntityOwningExtent`	Performance overhead of having to make Optimistic copies of objects, which can be very prohibitive for very large object, such as collections other than `EntityOwningExtent`

If the three stages Read, Work, and Write are carried out in a single transaction, then the scenarios are different. An object that is read and written in the same transaction would need to be accessed with a Write lock in the Read stage to avoid problems of lock upgrade during the work stage or during the Write stage. If you use a pessimistic lock, you also have to have a suitable scheme to avoid deadlocks. For more details on deadlock avoidance, refer to the *Programmers Guide* in the section on *Deadlock Avoidance*

10.2.2.3 NO_LOCK and Optimistic Access with EntityOwningExtents

Unlike other entities, NO_LOCK and Local access to EntityOwningExtents is not made by simply making a snapshot copy of the real object. The object provided when NO_LOCK or Local access is requested is a special object that delegates some of the operations performed on it to the real extent and the real underlying persistent data. Currently if an Extent is accessed Locally, it is implicitly overridden to HOME access. However, this does not affect the correct functioning of you application, performance may be significantly affected due to many remote method calls between objects that were accessed Local as requested and objects that got accessed HOME because the extent was overridden to HOME. It is better if your code doing the query and processing the results is running on the server rather than trying to use Local access mode from a client.

10.2.2.4 Access Location - HOME versus LOCAL

Accessing an object with AccessMode set to LOCAL causes the object to be copied from the server process owning its container into the process where it is being accessed. All method calls on that object are then local (fast). When you are finished updating the object, it is copied back to the server process that owns its container.

Accessing an object with access mode set to HOME creates a proxy or stub of the accessing system. The object itself remains on the server owning its container. All method calls on the object are then transferred by the stub to the server system where they will be executed directly on the object.

The LOCAL versus HOME question depends on whether the overhead of streaming the object back and forth (LOCAL) is more than that of having each

method call flow back and forth (HOME). The answer depends on a number of factors:

- The size of the object
- The number of calls made to it
- The kind of data that flows with those calls
- The amount of communication overhead
- The computing power of the server versus the client

In short, to answer the LOCAL or HOME question properly, you have to understand these different factors (some of which you may not even know) and how they inter-relate. The advise of our SF performance team is: “Access remote objects at HOME”. The team has found very few situations that LOCAL access wins. So, unless you are convinced that LOCAL access is better, use HOME.

One case where the LOCAL has been found to be the best is in connection with NO_LOCK mode. This is the case where you want to cache an Entity because you need to access it repeatedly for reading (across multiple transactions), and the state of that Entity is very unlikely to change while you work with it. Even if the Entity changed its state, the above still applies, as in the case:

- You were only interested in a subset of the Entity’s state
- The subset was not likely to change

Some of the Controllers introduced by CF and Towers are good examples for Entities to cache because:

- They go through an initial setup (where they are populated with the set of objects and from that point on typically do not change state any more).
- They are accessed multiple times (across transaction) to retrieve controlled objects.

For example, consider the `UnitOfMeasure` controller.

The set of `UnitOfMeasures` used by an application does not usually change at application run-time. It is determined during application setup. Thus, once the application setup is complete, the state of the `UnitOfMeasure` controller is unlikely to change. The following piece of code makes use of cached UnitOfMeasure Controller:

```
Company company = CompantContext.getActiveCompany(AccessMode.createNoLock());
cachedUnitOfMeasureController = (UnitOfMeasureController)
company.getPropertyBy("cf.UnitOfMeasureController");
BaseFactory factory = Global.factory();
factory.begin();
UnitOfMeasure unit_a = cachedUnitOfMeasureController.getUnitOfMeasureBy("A");
.....
factory.commit();
...
factory.begin();
UnitOfMeasure unit_b = cachedUnitOfMeasureController.getUnitOfMeasureBy("B");
...
factory.commit();
```

Follow these rules of thumb while using AccessModes:

- If you have many requests (methods calls) and streaming costs are low, use LOCAL access. The streaming costs are the costs for streaming the target of the method call plus the cost of streaming parameters and results.
- If you have very few requests, use HOME access.
- If you have many requests and high streaming costs, use Commands.

10.2.2.5 Compatibility Modes

A compatibility mode attribute in the access mode specifies whether, and how, the framework may modify the lock and location request to be compatible with an existing lock on the object or its container. The valid values for compatibility mode are defined in CompatibilityMode class with the following identifiers: NO_CHANGE, SELF and CONTAINER. The compatibility modes come into play when we try to access objects that are contained within another object. The containing object's AccessMode determines the type of lock that may be granted to the contained object. This is an important consideration since the appropriate lock may not be obtained for a contained object if its not compatible with that of its containing object. This could result in failures at commit time because a lock upgrade could not be obtained. For more details on Compatibility modes, refer to the *Programmers Guide* section on *Compatibility Modes* under *Advanced AccessMode Programming*.

10.2.3 Iterators

Iterators are a set of classes provided by SanFrancisco to traverse SanFrancisco collections. The following sections outline the points that you should keep in mind while using iterators.

10.2.3.1 Iterators and LockModes

Irrespective of the AccessMode specified for the collection, the elements of the collection are accessed using the default AccessMode createNormal(). If it is known in advance that the elements being accessed are going to be changed, then it would be a good idea to set the AccessMode to `createPlusWrite()` for accessing the elements. The `setAccessMode()` on the iterator can be used for this. The drawback of using the default `createNormal()` is that there is a contention risk when the changes are to be committed later.

10.2.3.2 Loops

If you are going to iterate over a SanFrancisco collection, you may attempt to utilize the `more()` or `hasMoreElements()` methods. For example:

```
Iterator iterator = collection.createIterator();
while(iterator.hasMoreElements())
{
   Object myElement = collection.next();
}
```

This loop is easy to understand, but the call to `more()` is unnecessary since SanFrancisco collections and iterators returns null on calls to the `next()` method when there are no more elements. You can use this feature to loop until `next()` returns a null. If you are accessing the collection remotely, remote method call overhead will also be added to the overhead of the `more()` method, resulting in a significant performance cost. An improved version of the loop eliminates the additional method call overheard. For example:

```
Iterator iterator = collection.createIterator();
Object myElement = iterator.next();
while (myElement != null)
{
   Object myElement = iterator.next();
}
```

10.2.3.3 Databases

When associated with persistent `EntityOwningExtents` in a relational database, iterators allocate resources when created. If your transactions are long running, you may want to consider forcing the iterator to drop its database resources when you are done with the iterator. This can be done by use of `releaseResources()` method on the iterator. This frees the resource for later use in this, or other, transactions.

If the user of the iterator neglects to call the `releaseResources()`, the iterator performs this method itself when the next transactional scope is started.

10.2.3.4 Reversible Iterators

Before the SF130 release, iterators supported both forward and backward iterations. The backward iteration proved to be a performance hurdle for usage with collections, especially `EntityOwningExtents`, because some databases do not support backward iteration. This required a lot of caching to be performed and used up memory. To overcome these problems, and also because most of the time only forward iteration is required, this functionality was removed from the Iterator and a new class, the ReversibleIterator, was introduced to allow for backward iteration. `Lists` and transient collections return reversible Iterators on `createIterator()` calls.

10.2.4 Collections

Choice of a particular collection can have considerable impact on performance. The table below shows the preferred choice of collection for a set of functionalities that are expected from it and for a given number of elements.

Table 11. Choice of Collections Determined by Size or Functionality Needed

Choice	Query	Pushdown	Iteration	Primitives	Legacy	Size***
DMap, List, Set*			#			10 to 50**
Map, List, Set	#		#			over 50
Extents	#	#	#		#	very large
Arrays				#		1-20

Note:
* Copy semantics
** Database column limits
*** Recommended size constraints

Special care has to be taken while returning collections as return values of methods. The following list outlines some pointers for such a use:

- When the returned value is used as a particular collection type, return the collection type.
  - For example, a collection should be used when the result is typically copied into a collection or will be queried over.
  - Avoid using a `DEntityOwning<collection>` since each copy owns the Entities and destroy them when its is destroyed.
  - If a large number of elements are returned, a `D<collection>` should not be used because a `D<collection>`, which contains dependents, copies everything when it is copied.
- When a constant fixed number of elements is to be returned, an array should be used.
- When a variable number of elements is to be returned, a vector should be used.

Note

Persistent SanFrancisco vectors are not available at this time, so if an array is not applicable, pick the right kind of collection: `Set`, `Map`, `List`, `DSet`, `DMap`, or `DList`, depending on your situation. Refer to the *Programmers Guide* for the suitability of a type of collection for a given situation.

For methods whose parameters and return values are collections:

- When the parameter is used as a particular collection type, pass in the collection type.
  - For example, a collection should be passed when it is copied into a collection or will be queried over.
  - Consider transferring ownership of non-Dependent collections.
  - Avoid using a `DEntityOwning<collection>` since each copy will own the Entities and will destroy them when it is destroyed.
  - If a large number of elements are returned, a `D<collection>` should not be used because a `D<collection>`, which contains dependents, will copy everything when its copied.
- When a constant fixed number of elements are passed in, then an array can be used.
- When a variable number of elements are passed in, then a vector can be used.

10.2.4.1 Arrays

Arrays need to be considered as a collection choice especially compared to Dependent collections (`DList`, `DSet`, and so on). Because arrays are lighter in weight, streaming and storage costs are much less.

It is important to know how an array is used and what function is needed. If only basic functions are needed (stepping through the collection one element at a time or especially indexing to specific elements), look at using an array. If keyed function or element uniqueness is required, `DMaps` or `DLists` would probably be required. For more assistance on selecting the correct collection, refer to *Collections* section in the *Programmers Guide*.

You can also use transient arrays to hold SF business objects (Dependents, Entities, and so on) that you need to hold temporarily (instead of creating a Collection for them). You can also use arrays as input parameters or return values for methods (again, rather than the relatively heavy weight collection objects). When working with a small number of objects, arrays are probably your best choice.

10.2.5 Miscellaneous

This section proposes the use of the Helper class and some recent enhancements made to SF, namely multiple entity retrieval and persistent arrays.

Helper

The Helper class can be used to copy dependents, comparing entities for equality, and so on. The helper methods should be called directly. For example:

```
Helper.setDependentToDependent();
```

The following example code compares two entities that are equal:

```
Helper.equals(thisHandle, thatHandle);
```

This way of comparing handles is preferred to comparing two entity objects directly because it saves the time for loading the entities in memory.

Multiple Entity Retrieval

The `getElementsBy..()` method on `BaseFactory` allows for accessing multiple entities by their Handle. This is more efficient as compared to retrieving each Entity separately since it reduces the number of remote method calls involved. This method also takes an AccessMode parameter that determines the type of access obtained for the elements of the collection.

Persistent Arrays

The use of SanFrancisco collection objects, such as Map, Set, and so on, may not really be necessary if the only need is to store a collection of elements. These SanFrancisco objects are heavier, and hence, take more memory and time. Instead, arrays could be used, and since their elements can be persisted, it makes sense to use them instead of the SanFrancisco objects.

DDecimal

The DDecimal factory class has been modified in the SF130 release to allow creation of DDecimal with a `long` primitive initialization value. This speeds up the initialization process.

10.3 Common Business Objects Coding Tips

This section discusses some of the issues that come up while using the Common Business Objects. Since SanFrancisco is organized on the concept of a company hierarchy, the section on CBOs details the structure of the hierarchy and the association of controllers and policies with the company objects. Also, it talks about validation and the use of keys and cached balances.

10.3.1 Company, Controllers, and Policies

A company in SanFrancisco serves more than the normal purpose of representing an organizational unit. It also acts as the placeholder, or parent of a

number of domain objects, that either do not have a parent object in the domain or that are generally used across the entire organization. These objects are tracked and managed by special objects called `Controllers`. The controllers, in turn, are held as properties by the company. Thus, we have two types of companies: context and non-context companies. To be eligible to be a context, a company should have all controllers and policies necessary for a given process contained either directly, or on, its ancestor companies. The exact controllers that are required depends on the application. A company that is currently associated with the context of a session is called the `activeCompany`. The user always retrieves any of these objects from the `activeCompany` by requesting the appropriate controller. For the user, it is as if the `activeCompany` holds all the controllers, whereas in reality, the controllers may be attached at different levels in the organizational structure. To make this clearer, lets assume we are dealing with the `PaymentMethod`(PM) objects, and our company hierarchy is as shown in Figure 69.

The `Enterprise` (just another Company, its significance being that it is the root in the Company hierarchy) is the root of the tree with a number of companies C1, C2, and so on under it. Let us say the `activeCompany` is directly under the company C1. The companies have the PM controllers attached to them. The enterprise has the root controller, which means it is the last in the chain of responsibility for searching for a given object. If its not found here, then the search of an item will fail. The company C1 has an aggregating controller attached to it, which means it can access objects in the PM root controller in addition to the object it contains. Aggregating controllers have access to all the objects that are above them in the hierarchy. Figure 69 illustrates this scenario:

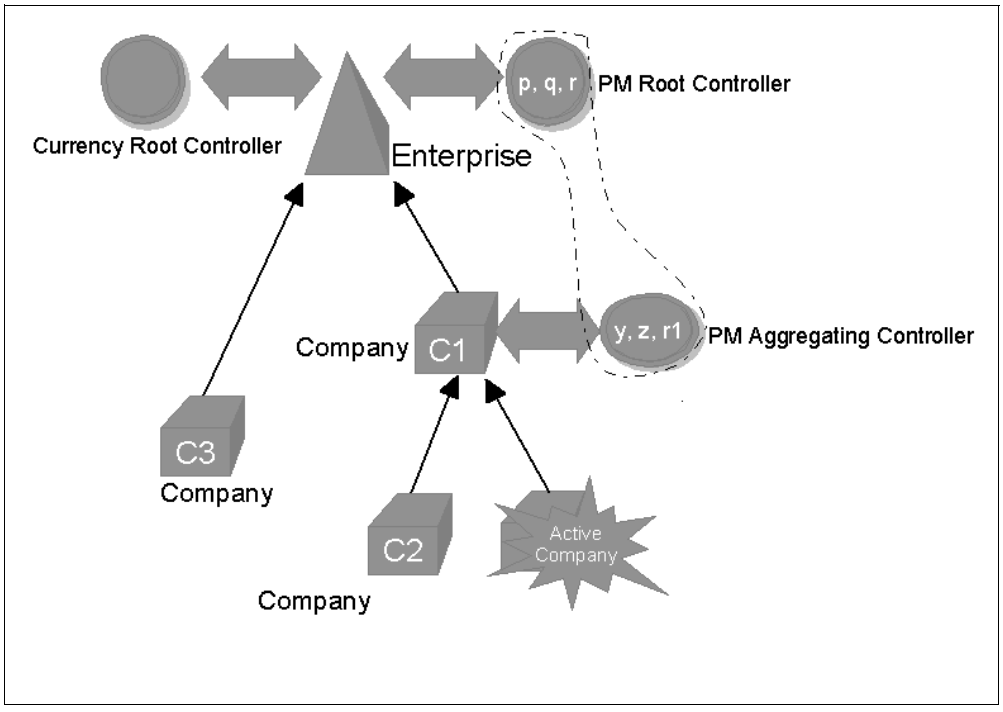


Figure 69. Company Hierarchy and its Association with Controllers

Now, let us say the user in the active company needs access to object "z." The company propagates the request to C1, which checks its PM aggregating

controller for the object. The object is found and returned. If the user now requests object "p," it is searched for in the PM aggregating controller of C1. At this point, the aggregating controller does not find it in its collection, but the search does not stop here. It passes the request on to the controller higher up in the hierarchy, in our case, the root controller. To do this, the controller internally requests its company, C1, for its parent company and passes on the request to the parent company's controller. Thus, the grouping shown in dotted lines is just a view that the user sees. The aggregating controllers pretend to be holding all the objects that are in the complete path higher up in the hierarchy. In reality, the logical scope of controllers increase as we descend in the Company hierarchy and decrease as we move higher up in the hierarchy.

The point to note here is that the distribution of the objects in controllers at various levels can have a considerable impact on the performance, especially for queries of the type described above. Also, the depth of the hierarchy, and which company is currently the active company for the user, will also determine the path length in the chain of responsibility for queries. If the company C1 and Enterprise are on physically different machines, it would involve remote calls over communication links.

The issues to be considered can be explained with the example of the `PaymentMethod` object used above. The `PaymentMethod` describes the means of payments accepted by a company, such as cheques, cash, and so on. Now, normally this would be common across the entire organization, and hence, this would be placed on the level of the enterprise (for example, objects "p," "q," and "r"). But, say the company is geographically dispersed and company C1 needs to introduce a new manner of payment "z." Also, the other means of payment are used. Since "z" is specific to C1, you place it in Company C1 and use a PM aggregating controller for accessing the other payments too. Also, if some type of payment, say "r," is not acceptable in its current form at company C1, it can be overridden and replaced by object "r1" in the company C1. Thus, if a user of C1 asks for object "r," they instead are returned the object "r1." Also, let us say there is a change in the object "p." This is an enterprise-level object, and thus, the change in it is automatically available to all the companies in the organization. For these, and many other organizational reasons, its desirable to have the objects distributed at various levels. All of this flexibility can lead to the performance problems described above. By a proper load balancing at all levels, these issues can be optimally sorted out. The answers to the following questions provide some direction in this regard:

- **Where is the root controller placed?** In the extreme case, every company could just hold a root controller and there would be no propagation in the chain of responsibility at all. This would, however, mean that the controllers contain all the objects in them. Also, there arises questions of keeping them all in sync since they are common for the whole organization. On the contrary, if the companies at a certain level, say C1 and C3, deal with totally different types of objects, and there are no commonalities in them, then each could hold a root controller.
- **Where is the `ActiveCompany` set?** Most of the time, this is decided by default as the current company returned by the `CompanyContext`. But the active company can be set at a different level in the hierarchy to reduce the path in the chain of responsibility. For example, let us say that C1 holds all the root controllers, and all companies below it in the hierarchy just pass the request up to it for all objects, then the active company could be set to C1, and this

would reduce the path lengths considerably. This would not allow for a company down in the hierarchy to override any functionality or use a new object.

The discussion for controllers applies largely to policies as well. The policy usage should be based on two criteria. The first consideration is whether the behavior being considered is flexible enough to warrant the use of a policy. If it is not likely that the behavior will change, or if that change could be accommodated by providing a different class (that is, another class that implements a common interface of the class being considered), it is probably unwise to introduce a policy due to performance and object size considerations. In this case, the behavior could be built into the object being designed. Users wishing to change the built in behavior could still do so by subclassing or introducing a sibling class and enabling that class through class replacement. However, if the behavior considered is flexible and likely to be customized, a policy should be introduced.

The second consideration is to determine the scope of the policy, whether it is application wide (applies for the entire logical SanFrancisco network, for example, a rounding policy for numeric values), or company specific (for example, leave encashment for employees). In the first case, where the policy is application wide, the appropriate policy subclass is enabled through class replacement using objects that create and contain an instance of the policy when they themselves are created. In the later case, where the policy is company specific, instances of the appropriate subclass of the policy are placed on companies in the company hierarchy as properties. Objects that use the policy do so by going to the active company and retrieving the policy when necessary. Because policies, like controllers, are held as company properties, their retrieval will follow a chain of responsibility upwards through the company hierarchy. For specific implementations, the chain of responsibility can be truncated at certain points by introducing terminal objects that return the required policy objects themselves instead of propagating the request. For more information on the chain of responsibility pattern, refer to the *Extension Guide's* section on *Working with chain-of-responsibility driven policies*.

Sometimes, the policy functionality itself needs to be changed. This would be true in cases where the default functionality provided by a policy may be too elaborate and not really required. For example, one implementation of the exchange rate policy, which converts from a given currency to another currency, does not do a simple table lookup for a direct conversion. It checks if there are any intermediate currencies through which the conversion can be done. If such a functionality is not really required, it could be replaced with a simple and more efficient direct conversion check. This would improve performance to an extent.

10.3.2 Euro Currency

Some changes have been made in the SF130 release for the incorporation of the Euro, though the discussion here applies in general. The exchange rate retrieval policy brings about some considerations while using fluctuating rates, such as trade offs between maintaining a lookup table for conversions between currencies and the coding algorithm for conversions. If you want to have the simplest algorithm for direct conversion table, you have to exhaustively list all the conversions you will encounter. On the other hand, if conversion can proceed through other currencies, then maintenance is reduced, but more efficient algorithm needs to be coded for conversion. For fixed rates, table lookup normally

proceeds in the triangular way, that is, by converting to an intermediate currency (the Euro) and then to the required currency.

Another factor is the information about the phases in which the changeover to Euro takes place. There are up to four phases, and each phase has the base currency, which is the primary currency being used and an optional report currency. This would have its impact to an extent on the creation of transaction values that is based on the period and the phase you are currently in. You may not want to add all the phases in the beginning, and this would reduce phase checks.

Another consideration that is not specific to the Euro is whether your exchange rate type is period based or date based. If your application is primarily period based, and you use a period based exchange rate type, then pass the period, instead of a date, which would require some processing to convert it to a period. The same applies if you are using date based exchange rates.

As a last note, if your application is used in a company where the Euro is not the base currency, then it would be a good idea not to use the `TransactionValueExtended` at all. This may improve performance.

10.3.3 Validation

Validation is an integral part of CBOF and the Towers. Since validation is intended to catch errors, theoretically, it should be optimized for normal operations, for example, it should be assumed that errors do not happen very often. One inclination is for a developer to create the `DResultsCollection` right at the beginning of a validation method. This practice, however, tends to be wasteful, since most of the time, `DResultsCollection` isn't needed (no errors) and is simply discarded. This increases the path length and the amount of garbage to collect. Developers should only create a `DResultsCollection` when it is needed.

Most of the time, the default policies that are employed perform an extensive number of checks. This may not really be required, and it is better to override the default policy and provide another one where the checks are minimized. Also, most of the time the GUI itself performs some of the validation checks. For example, let us say we are dealing with a policy that uses `BusinessPartner`s, which can be of three types. If the particular policy can be used with only `BusinessPartner` of the `Customer` type, then such a check would be included in the policy. This is not really necessary because the GUI allows the user to make selections only from a collection that displays `BusinessPartners` of the `Customer` type. At the same time, keep in mind that validations are an integral part of any process and skipping some checks might result in unacceptable behavior.

Additional performance tuning that can be done with the validation is with regard to the validation levels. The `ValidationContext` class can be set to work with three levels of validations defined in the `ValidationLevelEnum` class - `ALL`, `SEVERE_ERRORS_ONLY` and `NO_VALIDATION`. If set to `ALL`, then all errors and warnings validation logic are performed. If set to `SEVERE_ERRORS_ONLY`, only the severe errors validation logic is performed. If set to `NO_VALIDATION`, the validation logic is skipped all together. Depending on the criticality of the application, the validation could be set to either `NO_VALIDATION` or `SEVERE_ERRORS_ONLY` to reduce the amount of validation processing. Of course, for critical application, you have to use the `ALL` option.

The `ValidationContext` class also allows you to set the option for bundling the results (error messages). This is recommended since it saves the efforts for both creation and handling of the exceptions, which is rather expensive.

Appendix A. Internal SanFrancisco Tools - Schema Mapper Tool

The following sections described the major topics from the publication “Using Schema Mapper Tool” in the IBM SanFrancisco documentation found on the product CD-ROM at SanFrancisco Base Information, column Advanced. All references in the following sections point to this documentation. The developers that intend to use Extended Schema Mapping should print and read this documentation. This document is not included here since it may be subject to change.

A.1 Overview

A schema is the underlying structure of a database and the organization of its information. Using a relational database as the persistent store for IBM San Francisco Business Objects requires a mapping, or correlation, between an object model and a relational model, each of which has its own schema.

Schema mapping defines the relationships between an object schema and a relational schema. In other words, classes, objects, and attributes are mapped to tables, rows, and columns.

A.1.1 Schema Mapping Tool (SMT)

The SanFrancisco SMT plays an essential role in the integration of San Francisco Common Business Objects with new or legacy relational databases (RDB). Mapping Entity object attributes to table columns and mapping a unique object identity to an existing primary key are only two of the many schema mapping functions that help ensure RDB persistence.

SanFrancisco provides two different tools for schema mapping: the Default Schema Mapper (DSM) and the Extended Schema Mapper (ESM). The ESM requires a graphical user interface (GUI) to collect the necessary direct user input. Because the DSM is an automatic function that requires no direct input from a user, it does not use the SMT GUI.

For more information on the characteristics of the DSM and the ESM, see Integrating, storing, and managing persistent objects.

A.1.2 Schema Mapping Language (SML)

A SanFrancisco SML file contains the definitions of a particular schema map for a class, embedded objects, and subclasses. SML files extend the usefulness of a schema map through their reuse and modification.

A.1.3 Platforms and DBMS

The SML generated by the ESM is compatible with the following platforms with the specified DBMS:

- DB2 V5 on Windows NT 4.0
- DB2 on AIX
- Oracle8 on Windows NT
- DB2/400 on AS/400 systems

A.1.4 Schema Mapping Interface (SMT GUI)

The main interface of the SMT GUI is characterized by the four buttons (tabs) at the top of the interface and the large, active work area that fills the remainder of the frame. Clicking any of the four tabs displays the interface for the corresponding function of the SMT.

A.1.5 Schema Mapping an Object

Using the ESM to customize data mappings requires a full and elemental knowledge of the structure of the database.

A.1.6 Using the Extended Schema Mapper

The following issues are included in the work with ESM:

- Select a class
- Define user preferences
- Create schema map
- Map the object
- Map subclasses
- Assigning primary key
- Joining tables
- Mapping to multiple rows
- Save the schema map
- Configure the SML file
- Connect the data source
- Create required tables

A.1.7 Editing an Existing SML File

Alternatively, you can select an object that has already been mapped by specifying an existing SML file.

A.1.8 User Preferences

You can modify user preferences and save them to a preferences file. You can create as many preferences files as you want. When you are finished modifying the preferences files, you can choose one to use for your current schema mapping session.

A.1.9 Default and Override Tables

The name of the database table in which you will store the schema mapped object is the Default Table. You can override this designated table by specifying an Override Table.

A.1.10 Functions of the SMT

The schema mapping tool provides the following major functions: object mapping, defining a primary key, joining tables, mapping to multiple rows, and object sub classing. The tasks include working with following issues:

- Object mapping
- Map fields
- Primitive mapping
- Array mapping
- Stream object.

- Reference SML file
- Do not store
- Subclasses
- Handles
- Query methods
- Patterns
- Primary key
- Join
- Multiple rows

A.1.11 Other Considerations

The following SMT limitations apply to SanFrancisco Version 1, Release 2. These limitations will be addressed in future releases:

- Caution must be used when mapping more than one class to a single table. Deleting any object of a set of classes that are mapped to the same table deletes its corresponding row, effectively deleting all objects that are mapped to that row. A future enhancement will allow you to specify whether or not a deleted object will delete its corresponding row.
- No verification is performed against the database catalog or the original.class file. Incorrect table or column names that exceed the maximum length of a data type, and similar problems, result in a run time error.
- Query pushdown for Handles is supported only if the Handles are streamed. For specific details on data type mapping considerations and restrictions with respect to query pushdown, see Integrating, storing, and managing persistent objects.
- The Table Schema Assistant cannot "dynamically" create the table on a system different than the one in which the SMT is running. Instead, you must save the Create Table statement to a file, then use this file to create the tables on a different system.
- Please avoid using the "Window Destroy" (“X” button) as a Cancel function. Results may be unpredictable.
- Secondary tables may not use multiple row support. You must map large objects multiple rows in the primary table only.
- NT ODBC support is limited to single phase.

A.1.12 Table Schema Assistant

Create a new table directly from a SML file that was created by the Schema Mapping Tool. Starting the Table Schema Assistant causes it to parse the current SML file to generate both a CREATE TABLE statement and a DROP TABLE statement. Executing the CREATE TABLE statement from within, the tool creates a table that is needed to persistently store the class file that you schema mapped. Executing the DROP TABLE statement drops the table. It is also possible to use the ALTER TABLE statement in order to alter an existing table.

Note

You must have an active SML file for the Table Schema Assistant to work.

A.1.13 Start-up Options

Use the Schema Mapping Tool to map Java objects to relational tables.

A.1.14 Define New Schema Mapping

Designate the class name for the object that you want to schema map, and define initial user preferences, including data type mappings. For more information about the procedure, see Selecting a class to map.

A.1.15 Select SML File

Load an existing SML file to edit.

A.1.16 Select Preferences

Select initial user preferences to use in the current schema mapping or to save in a preferences file. For more information, see User preferences.

A.1.17 Mapping (User) Preferences

Change the listed mapping preferences and save the new preferences to a preferences file.

A.1.18 Data Type Mapping Preferences - Detail

Change the default schema mapping for a Java data type by defining a column definition for that type.

A.1.19 Object Mapping

Schema map an object by using one of the following types of mapping: Stream Object, Map Fields, Reference SML File, or Do Not Store. For more information, see the procedure Mapping the object or Object mapping in the Overview.

A.1.20 Field Mapping - Options

Define the mapping from a field to a database column. Options include the query methods that you want to use to associate a field and the database column definition and the pattern that you want to use for boolean, DBoolean, and DTime data types. For reference, the dialog displays the field name and data type of the field that you are mapping.

A.1.21 Select Query Methods

Map the get methods to use for query pushdown. A listed method that is specified in the “WHERE” clause of a query is eligible for query pushdown only if it is stored to a compatible column type. For reference, the dialog displays the name and data type of the field or object that you are mapping.

A.1.22 Substitution Pattern

Define the values that boolean and DBoolean Java data types are to use for the current mapping. Add, remove, edit, and order new and existing values for these data types. Ensure that the current values for the column definition support the pattern you want to specify. For more information, see Patterns.

A.1.23 Date Pattern

Examine or modify the date pattern that the DTime Java data type uses for the current mapping. Ensure that the current values for the column definition support the pattern that you want to specify.

A.1.24 Array Mapping - Options

The Schema Mapping Tool supports the mapping of a single-dimension array of Handles, Dependents, Strings, or primitive Java data types to a relational database table. Array elements must all be of the same type or class but can be a mixture of subclasses.

A.1.25 Defining Handles

Begin mapping a Handle. If you decide to stream the Handle, the process is simple and requires only that you define a column in which to stream the Handle data. Defining your own Handle types requires knowledge of the Handle, its contents, and the objects the Handle references. For more information, see the procedure Mapping the object or Handles in the Overview.

A.1.26 Mapping Handle Types

Begin mapping a Handle type. Options include the Handle Type, the type of mapping, and the definitions that identify the Handle contents and the objects that the Handle references. For more information, see the procedure Mapping the object or Handles in the Overview.

A.1.27 Defining Subclasses

Add, update, and delete subclass mappings. Mapping subclasses includes mapping the fields (or streaming them) and determining the class name or class ID. For more information, see the procedure Mapping the object or Subclasses in the Overview.

A.1.28 Subclass Mapping

Schema map a subclass by using one of two types of mapping: Stream Object or Map Fields. For more information, see Object mapping, Subclasses, and the procedure Mapping the object in the Overview.

A.1.29 Interface Mapping - Options

Select whether to use the associated Impl class or the interface class only. This dialog displays when you are mapping an interface, and an associated Impl class exists in your CLASSPATH.

A.1.30 Primary Key

Map the unique identity that is contained in the independent handle of this object to the primary key of a table. For more information, see the procedure Assigning a primary key or Primary key in the Overview.

A.1.31 Join

Build, modify, and inspect relationships between tables. All joins must chain back to the current default table for this SML file. For more information, see the procedure Joining tables or Join in the Overview.

A.1.32 Define Join Relationship

This display lists all the joins that you have defined in this schema map. Each join is made up of one or several column pairs. A column pair shows the relationship between two columns from two separate tables. A column pair uses the following format:

```
<primary table name>.<column name> = <secondary table name>.<column name>
```

Define the join (relationships) between a column or columns in a primary table and the corresponding columns in a secondary table. A list of column pairs in the Join On display indicates a Join. You can define several column pairs for the same primary and secondary tables.

A.1.33 Multiple Rows

Enable multiple row support to allow the streaming of Entity or Dependent objects in which the data stream is greater than the specified column length. This support focuses on mapping a large object into multiple rows of a column. Only those objects that are both streamed and mapped to a LONGVARBINARY column are eligible for mapping to multiple rows.

A.1.34 Exit Options

Confirm or abort the process of exiting the current schema mapping. Before exiting, select whether to save or discard any changes made since the last save.

Note

This dialog appears when you exit a schema mapping session, whether you have made any changes since the last save.

Appendix B. Modifying Generated Code

The section deals with the SF Code Generator for SF130 and with the possible enhancements that can be made manually in order to improve the performance. The code generator does a very good job, but at some points, improvements are still possible.

B.1 When to Make the Changes

It is clear that these kind of manual modifications to the generated code should only be done at the end of an implementation since changes cannot be permanently applied. Regenerating the code overwrites any manual modifications previously made.

B.2 Possible Changes

A series of points, where performance improvements, are possible are listed. All these tips and techniques are already discussed in the previous chapters.

B.2.1 Method getChildControllers() on Controller Objects

Controllers have a `getChildControllers(Company, DSet)` method. In this method's implementation, the "contains" check could be replaced with something like:

```
if ((result = childCompany.get...By (...)) != null)
```

This method returns null if the request can not be met, so the above expression gives the same reult as the "contains" check. As the result of the `get...By()` method is needed if the "contains" check passes, the result is already obtained. This is more efficient since the "contains" check essentially is eliminated.

B.2.2 Use of Iterators

There are a number of places in the method implementations where the code will use an iterator to traverse a collection. Rather than using:

```
while (iterator.hasMoreElements())  {...}
```

use:

```
while ((element = collection.getNextElement(iterator)) != null) {...}
```

since the `getNextElement(iterator)` method returns null if the next element is not found, and it is needed to do the `getNextElement(iterator)` anyway. This is more efficient since, essentially, the `iterator.hasMoreElements()` call is eliminated. Note that the specifics depend on the particular method implementation you are changing (for example, element may be named something else). Additional information can be found in 10.2.3, “Iterators” on page 185.

B.2.3 Caching of Global.factory

Already, many places will the generated code cache the `Global.factory()`. This is how it should be. In some areas, this is not yet completely done. The code needs to be changed manually in order to implement the caching of `Global.factory()`. Namely in create methods on the generated factories, some changes can be useful. Additional information can be found in 10.1.1, “Caching” on page 165.

B.2.4 Use of Local Variables

In many places in the generated code, a local variable is used for the unique purpose of storing the result of a method call just before returning the value of this local variable. This local variable can be eliminated instead of having code, such as:

```
{...
String name = person.getName();
return(name); }
```

the following code can be used:

```
{...
return(person.getName());}
```

B.2.5 Use of Helper Methods

The Helper class contains functionality that many high level base classes for business objects also implement. It is more expensive to use in business objects the inherited methods than call the Helper class methods. So instead of the code:

```
{...
Person person = setOwningHandleToObject(handle);
...}
```

use the following implementation:

```
{...
Person person = Helper.setOwningHandleToObject(handle);
...}
```

Additional information can be found in 10.2.5, “Miscellaneous” on page 188

B.2.6 Method addAllElements() on List Objects

If an business object implements a List object for example Person objects, a method `addAllPersons(Collection)` will be generated. In the implementation, an iterator loops over all elements calling the `addElement(Object)` on the List variable. It is better performing to change the code so that the List object itself will iterate over the collection. Use the `addElements(Collection)` on the List object for implementing the `addAllPersons(Collection)` method.

B.3 Other Changes

This list is definitely not exhaustive. Other modifications are still possible. All the recommandations from the coding tips described in Chapter 9, “Java Coding Tips” on page 147 and Chapter 10, “SanFrancisco Coding Tips” on page 163 should be respected in the generated code. Most recommendations are followed already, and future releases of the code generator will increase the compatibility with the recommandations.

Appendix C. Special Notices

This publication is intended to help IBM SanFrancisco Application Architects, Application Designers and Application Developers, as well as IBM SanFrancisco Performance Consultants, to analyze, optimize, and improve performance of their applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by the IBM SanFrancisco Business Process Components. See the PUBLICATIONS section of the IBM Programming Announcement for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBMs product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBMs intellectual property rights may be used instead of the IBM product, program, or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating

environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

APPN	AS/400
DB2	DRDA
Distributed Relational Database Architecture	IBM ®
OfficeVision/400	OS/400

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java, HotJava and Sun Solaris are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT Performance Monitor, Windows NT, and the Windows 95 logo are trademarks
or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Appendix D. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

D.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get ITSO Redbooks” on page 207.

- *AS/400 Performance Explorer — Tips and Techniques*, SG24-4781
- *Accessing the Internet*, SG24-2597

D.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

D.3 Other Publications

These publications are also relevant as further information sources:

- de Champeaux, Dennis; Le, Douglas; Faure, Penelope. 1993. *Object-Oriented System Development.* Addison Wesley (ISBN 0-201-56355-X)
- *Windows NT Resource Kit*, Microsoft Press (ISBN 1-55615-929-3)
- *Architecting Object Applications for High Performance with Relational Databases.* Persistence Software Inc. (www.persistence.com)
- Jain, Raj; Wiley, John. 1991. *The Art of Computer Systems Performance Analysis* (ISBN 0-471-503363-3)
- Sims, Oliver. 1994. *Business Objects.* Mc Graw-Hill (ISBN 0-07-707957-4)
- Shuey, Richard; Spooner, David; Frieder, Ophir. 1997. *The Architecture Of Distributed Computer Systems* (ISBN 0-201-55332-5)

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com/`.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER** – to order hardcopies in the United States
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** – send orders to:

	IBMMAIL	**Internet**
In United States	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** – send orders to:

IBM Publications Publications Customer Support P.O. Box 29570 Raleigh, NC 27626-0570 USA	IBM Publications 144-4th Avenue, S.W. Calgary, Alberta T2P 3N5 Canada	IBM Direct Services Sortemosevej 21 DK-3450 Allerød Denmark

- **Fax** – send orders to:

United States (toll free)	1-800-445-9269
Canada	1-800-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1) 408 256 5422 (Outside USA)** – ask for:

  Index # 4421 Abstracts of new redbooks
  Index # 4422 IBM redbooks
  Index # 4420 Redbooks for last six months

- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

Redpieces

For information so current it is still in the process of being written, look at "Redpieces" on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name | Last name

Company

Address

City | Postal code | Country

Telephone number | Telefax number | VAT number

☐ Invoice to customer number

☐ Credit card number

Credit card expiration date | Card issued to | Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

List of Abbreviations

APA	All Points Addressable
AS/400 System	Application System 400
BIOS	Basic Input Output System
BOB	Business Object Benchmark
BTPH	Business Transactions Per Hour
CPU	Central Processing Unit
CSM	Custom Schema Mapper
DASD	Direct Access Storage Device
DBA	Data Base Administrator
DBMS	Database Management System
DNS	Domain Name Server
DSM	Default Schema Mapper
E-R	Entity Relational
ESM	Extended Schema Mapper
GBOB	GBOF Business Object Benchmark
GBOF	General Business Object Framework (Foundation Layer of SF)
GSM	Global Server Manager
GUI	Graphical User Interface
I/O	Input/Output
IBM	International Business Machines Corporation
ISV	Independent Software Vendor
ITSO	International Technical Support Organization
JDK	Java Development Kit
JIT	Just In Time compiler
JVM	Java Virtual Machine
LSFN	Logical SanFrancisco Network
NC	Network Computer
OLTP	OnLine Transaction Processing
OODBMS	Object Oriented Database Management System
OS	Operating System
PEX	Performance Explorer
PDS	Performance Data Collector
RAID	Redundant Array of Independent Disk
RDB	Relational Database
RDBMS	Relational Database Management System
RMI	Remote Method Invocation
SCSI	Small Computer System Interface
SF	SanFrancisco
TBOB	Tower Business Object Benchmark
TPC	Transaction Processing Council
TPM	Transactions Per Minute
VTUNE	Visual Tuning Environment

Index

ITSO Redbook Evaluation

IBM SanFrancisco Performance Tips and Techniques
SG24-5368-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
_ **Customer** _ **Business Partner** _ **Solution Developer** _ **IBM employee**
_ **None of the above**

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction _____

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

_____

_____

_____

_____

What other redbooks would you like to see published?

_____

_____

_____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_____

_____

_____

_____

_____

SG24-5368-00

Printed in the U.S.A.

IBM SanFrancisco Performance Tips and Techniques

SG24-5368-00