GEMSTONE®

GEMFIRE®

ENTERPRISE

Developer's Guide

Version 6.5

September 2010

Send comments on this guide to docs@gemstone.com

GemStone Systems, Inc. ▾ 503•533•3000 ▾ www.gemstone.com

INTELLECTUAL PROPERTY OWNERSHIP

This documentation is furnished for informational use only and is subject to change without notice. GemStone Systems, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation.

This documentation, or any part of it, may not be reproduced, displayed, photocopied, transmitted, or otherwise copied in any form or by any means now known or later developed, such as electronic, optical, or mechanical means, without express written authorization from GemStone Systems, Inc.

Warning: This computer program and its documentation are protected by copyright law and international treaties. Any unauthorized copying or distribution of this program, its documentation, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted under the maximum extent possible under the law.

The software installed in accordance with this documentation is copyrighted and licensed by GemStone Systems, Inc. under separate license agreement. This software may only be used pursuant to the terms and conditions of such license agreement. Any other use may be a violation of law.

Use, duplication, or disclosure by the Government is subject to restrictions set forth in the Commercial Software - Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations (48 CFR 52.227-19) except that the government agency shall not have the right to disclose this software to support service contractors or their subcontractors without the prior written consent of GemStone Systems, Inc.

This software is provided by GemStone Systems, Inc. and contributors "as is" and any expressed or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall GemStone Systems, Inc. or any contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

COPYRIGHTS

This software product, its documentation, and its user interface Copyright © 1997-2010, GemStone Systems, Inc. All Rights Reserved by GemStone Systems Inc.

JavaGroups Copyright 1999-2004 Free Software Foundation, Inc.

The Java Software technologies are Copyright © 1994-2000 Sun Microsystems, Inc. All rights reserved

GNU Trove copyright 2001-2004 Eric D. Friedman. The PrimeFinder and HashFunctions classes in Trove are copyright 1999 CERN - European Organization for Nuclear Research. Copyright © 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

MX4J Copyright © 2001-2004 by the MX4J contributors. All rights reserved.

Antlr Copyright © 2005, Terence Parr. All rights reserved.

Commons Modeler Copyright © 2004 Commons Modeler. All rights reserved.

Trove Log4J Copyright © 1999 The Apache Software Foundation. All rights reserved. The Trove library is licensed under the Lesser GNU Public License, which is included with the distribution in a file called LICENSE.txt. PrimeFinder and HashFunctions classes in Trove © Copyright 1999 CERN - European Organization for Nuclear Research.

Copyright (C) 2006, Hitachi, Ltd. All Rights Reserved.

Copyright © 1994 Hewlett-Packard Company

Copyright © 1996,97 Silicon Graphics Computer Systems, Inc. Copyright © 1997 Moscow Center for SPARC Technology.

Copyright © 1998-2003 Daniel Veillard. All rights reserved.

Jgroups © 2001, 2002 www.jgroups.org

PATENTS

GemFire is protected by U.S. patent 6,360,219. Additional patents pending.

TRADEMARKS

GemStone, GemFire, GemFire Enterprise, and the GemStone logo are trademarks or registered trademarks of GemStone Systems, Inc. in the United States and other countries (trademark application pending for GemFire).

UNIX is a registered trademark of The Open Group in the U. S. and other countries.

Linux is a registered trademark of Linus Torvalds.

Red Hat and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries.

SUSE is a registered trademark of SUSE AG.

Sun, Sun Microsystems, Solaris, Forte, Java, Java Runtime Edition, JRE, and other Java-related marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Intel and Pentium are registered trademarks of Intel Corporation in the United States and other countries.

Microsoft, Windows, and Visual C++ are registered trademarks of Microsoft Corporation in the United States and other countries.

IBM, AIX, and developerWorks are registered trademarks of IBM Corporation.

W3C is a registered trademark of the World Wide Web Consortium.

Other company or product names mentioned herein may be trademarks or registered trademarks of their respective owners. Trademark specifications are subject to change without notice. All terms mentioned in this documentation that are known to be trademarks or service marks have been appropriately capitalized to the best of our knowledge; however, GemStone cannot attest to the accuracy of all trademark information. Use of a term in this documentation should not be regarded as affecting the validity of any trademark or service mark.

GemStone Systems, Inc.
1260 NW Waterhouse Avenue, Suite 200
Beaverton, OR 97006

Table of Contents

List of Figures ***13***
List of Tables ***17***
List of Examples ***19***

Preface ***25***
About This Guide 25
How This Guide Is Organized 25
Typographical Conventions 26
Other Useful Documents 27
Technical Support 29
Preserving Artifacts for Technical Support 29
Contacting Technical Support 30
24x7 Emergency Technical Support 30
Training and Consulting 30

Chapter 1. Introduction to GemFire Enterprise ***31***
1.1 Architectural Overview 36
1.2 Basic System Topologies 37
1.3 Services 39
1.4 Configuration Scenarios 41
Client/Server Caching 45
Partitioned Data Caching 46

Chapter 2. Programming with GemFire Enterprise ***49***
2.1 Running Your GemFire Applications 50
Running on Solaris and Linux 64-bit Machines 52
Running in Pure Java Mode 52
2.2 The Programming APIs 53
2.3 Configuring and Monitoring Your Applications 55
2.4 Defining Your Own Custom Classes 57

2.5 Configuring Database Connections Using JNDI 59
Configuring JNDI-Binding Properties 59
Configuration Examples 60
Encrypting Connection Passwords 64
2.6 Handling Exceptions and System Failures 65
2.7 Considerations for Your Host File Systems 66
2.8 About the Programming Examples 67

Chapter 3. GemFire Members and Member Caches 69

3.1 Distributed System Member Configuration 70
Distributed System Membership and System Topologies 70
Local and Remote Caching and Membership 71
3.2 Cache Management and Functionality 72
Caching APIs 72
3.3 Configuring the Cache 74
Cache XML Declaration File Requirements 77
3.4 Creating and Closing a Cache 78
Creating and Closing a Server or Peer Cache 79
Creating and Closing a Client Cache 80
Creating and Closing a Cache in a Secure System 81
Creating and Closing RegionService Instances in a Client with Multiple Secure Users 82
Changing gemfire.properties in the Cache Creation Call 84

Chapter 4. Data Regions 85

4.1 Region Overview 86
4.2 Region Types 87
Partitioned Regions 88
Replicated Regions 88
Distributed, Non-Replicated Regions 89
Local Regions 89
4.3 Data Hosts and Data Accessors 90
4.4 Region API 91
4.5 Creating and Managing Data Regions 92
Creating a Region 92
Invalidating a Region 94
Clearing a Region 95
Destroying a Region 95
Closing a Region 96
Determining a Region’s Status 96
Tracking Instances of a Distributed Region 96
4.6 Region Shortcuts and Custom Named Region Attributes 97
Storing and Retrieving Named Attributes 99
4.7 Region Attributes List 101
cache-listener 102
cache-loader 102
cache-writer 103
cloning-enabled 103

concurrency-level 103
data-policy 104
disk-store-name 106
disk-synchronous 106
enable-async-conflation 106
enable-gateway 107
enable-subscription-conflation 107
entry-idle-time 108
entry-time-to-live 108
eviction-attributes 108
hub-id 109
id 109
ignore-jta 110
index-update-type 110
initial-capacity 110
interest-policy 111
is-lock-grantor 111
key-constraint 111
load-factor 112
membership-attributes 112
multicast-enabled 113
partition-attributes 113
pool-name 115
refid 116
region-idle-time 116
region-time-to-live 116
scope 116
statistics-enabled 117
subscription-attributes 118
value-constraint 119
4.8 Creating Custom User Attributes for Regions and Entries 120
User Attributes Example 120
Limitations and Alternatives 121
4.9 Dynamic Region Management 122
The Dynamic Region Interfaces and Classes 124
Managing the Dynamic Region Factory 125
Creating and Destroying Dynamic Regions 127

Chapter 5. Data Operations 129

5.1 The Entry Interfaces and Classes 130
5.2 Adding an Entry to the Cache 131
Creating Entries in XML 132
Creating Entries Through the API 134
5.3 Accessing an Entry 135
Batching get Operations 137
Accessing Only the Local Cache 138
Getting Different Object Types From the Cache 139
5.4 Updating an Entry 140

Safe Entry Modification 140
Behavior of the put Operation in Partitioned Regions 142
Batching put Operations 142
5.5 Invalidating and Destroying an Entry 143
Local and Remote Operations 143
Destroy and Invalidate in Partitioned Regions 144
Determining an Entry's Status 144

Chapter 6. Configuration and Management by Region Type 145

6.1 Partitioned Regions 146
Rebalancing Partitioned Regions 150
Partitioned Regions with Redundancy 152
Custom Partitioning and Data Colocation 157
6.2 Replicated Regions and Other Distributed Regions 161
How Distributed Regions Work 161
How Replicated Regions Work 162
The Difference Between Distributed and Replicated 163
Creating a Replicated Region 163
Creating a Distributed Region 164
6.3 Local Regions 168

Chapter 7. General Region Management Options 169

7.1 Using Disk for Backup and Overflow 170
Programming Example for Persistence and Overflow 173
Performance Considerations 175
Entry Retrieval in Overflow and Persistent Regions 175
7.2 Keeping Your Data Current With Expiration 176
Partitioned Region Note for Idle Time Expiration 179
Interaction Between Expiration Settings and netSearch 180
Configuring Expiration 181
7.3 Controlling Memory Use With Eviction and Overflow 182
Partitioned Region Note for Eviction 182
Configuring Eviction 184
7.4 Controlling Heap Use With the Resource Manager 186
How Background Eviction is Performed 187
Configuring the Resource Manager 188
Configuring GemFire for Heap LRU Management 188
Setting Your VM's GC Tuning Parameters 189
Monitoring and Tuning Heap LRU Configurations 189
Resource Manager Configuration Example 192
7.5 Controlling Data Flow With Application Plug-ins 193
Plug-ins in a Partitioned Region 193
Plug-ins in a Distributed Region 195
Loading Entries Through a Data Loader 196

Chapter 8. Client/Server Architecture and Configuration Basics *201*

8.1 Client/Server Caching Features 202
8.2 Standard Client/Server Deployment 203
Simple Configuration for Server Discovery 204
Configuration With Server Grouping 205
8.3 The Client/Server API 206
8.4 Configuration Properties and Examples 207
Client Cache Configuration 209
Client Pool Configuration 210
Examples 213
8.5 Client/Server Data Flow 216
Server-Initiated Data Flow 219
Data Flow Between Clients 223
8.6 Example Scenario: Very Small Client Caches 224

Chapter 9. Client/Server Advanced Topics *227*

9.1 Client/Server Connections 228
Creating and Maintaining Connections 230
9.2 Balancing Server Use 232
Pool Connection Conditioning 232
Grouping Your Servers 233
9.3 Client Operations to Server 234
How Client Threads use Pool Connections 234
When Operations Don’t Return Quickly 234
9.4 Single-Hop Partitioned Region Data Access 236
9.5 Subscribing to Server Events 238
Discarding Duplicate Events 239
Highly Available Subscription Delivery 241
Durable Subscription Queues 246
Failover with Subscription Queues 254
Limiting Subscription Queue Memory Use 255
9.6 Controlling Thread and Connection Use 259
Controlling the Number of Server Connections to Peers 260
Controlling the Number of Server Threads Used for Clients 260
Setting the Maximum Client Connections Accepted by a Server 261
Adjusting the Limits on the Client Connection Pool 261
Optimizing the Client’s Connection Requirements 261
9.7 Performance Considerations 263
9.8 Miscellaneous Server Notes 264
9.9 Socket Configuration 265

Chapter 10. Configuring Multi-site Installations *267*

10.1 Multi-site Caching Overview 268
10.2 Basic Configuration 269
Core Multi-site API 270
Configuring the Gateway Hub 270

Implementing a Write-Behind Cache Listener in Your Hubs 271
Configuring Regions to Send Updates to the Gateway Queue 271
10.3 Gateway Configuration Parameters 272
10.4 Message Processing 274
Configuration Option for Slow Distributed Systems 274
How the Gateway Processes its Queue 275
Gateway Queue Persistence 275
Batch Processing Failure 276
Site-to-Site Queue Conflation 277
10.5 Load Balancing in Multi-site Installations 279
The Multiple-Hub Configuration 279
Event Flow With Multiple Hubs 280
Implementing a Multi-Hub System 281
How Hub Startup Policy Affects Startup Behavior 284
10.6 Failover 285
10.7 Example Configuration 286
Cache Configuration File 287
API 288
10.8 Multi-site Topologies 289
Parallel 289
Serial 289
Hybrid 290
Topologies to Avoid 291

Chapter 11. Handling Events 293

11.1 The GemFire Event Framework 294
Getting Started With Event Handlers 295
Cache Events and Administrative Events 295
11.2 Event Handling in the Caching API 296
Cache Events 297
Implementing Cache Event Handlers 300
Cache Event Listeners 305
Cache Writers 312
CacheCallback 313
Cache Event Handling in a Distributed System 313
Applications Without Local Caching 318
11.3 Event Handling in the Administration API 323
Administrative Event Listeners 324
Universal Membership Listener Adapter 325

Chapter 12. Delta Propagation 327

12.1 How Delta Propagation Works 328
12.2 Supported Topologies and Limitations 329
12.3 Delta Propagation API 331
12.4 Delta Propagation Properties 332
12.5 Implementing Delta Propagation 334

12.6 Errors in Delta Propagation 336
12.7 Performance Considerations and Limitations 337
12.8 Example of Delta Propagation 338

Chapter 13. Querying and Indexing 341

13.1 Executing a Query 342
13.2 Example Data and Example Queries 343
Querying the portfolios Region 344
13.3 Querying Object Data 347
Data Accessibility in a GemFire Cache Query 349
13.4 How Querying Interacts with the Distributed Cache 350
13.5 The Query Language 352
Accessing Cached Data 352
What is a Query String? 356
SELECT Statement Semantics 357
The FROM Clause 358
Helping the Query Engine Resolve Your Query's Attributes 360
The WHERE Clause 363
Joins 364
The SELECT Projection List 364
SELECT Statement Query Results 365
Additional Query Language Elements 367
13.6 Indexes 375
Index Types 375
Creating an Index 376
Configuring Index Maintenance 377
Indexing Guidelines 378
13.7 Querying Partitioned Regions 384
Partitioned Region Query Guidelines 384
Indexing a Partitioned Region 384
The Querying Process 385
13.8 Performance Considerations 386
13.9 The API Interfaces and Classes 387
13.10 Programming Examples 389
Query Management Examples 393

Chapter 14. Continuous Querying 397

14.1 How Continuous Querying Works 398
14.2 The GemFire CQ API 400
14.3 State and Life Cycle of a CQ 401
14.4 Implementing Continuous Queries 402
Writing the Continuous Query 403
Running the Continuous Query Code 406
Executing with Initial Results 407
Managing Your CQs 407

Chapter 15. GemFire Transactions *409*

15.1 The Transactions API 410
15.2 GemFire Transaction Semantics 411
15.3 Cache Design for Transactions 413
15.4 Implementing GemFire Transactions 414
15.5 How GemFire Transactions Work 416
- The Transaction View 416
- Committing 417
- How Transaction Events Are Managed 417
- Application Plug-Ins With Transactions 418
- Transactions in the Distributed Cache 420
- Transactional Versus Non-transactional Operations 426
- Queries and Indexes 428
- Collections and Region.Entry Instances 429
- Eviction and Expiration Operation With Transactions 429
- Monitoring and Troubleshooting Transactions 430
- Coding Examples 431

15.6 Using JTA transactions with GemFire 435

Chapter 16. Function Execution *437*

16.1 Solutions and Use Cases 438
16.2 How Function Execution Works 439
- How Functions are Executed 439

16.3 Executing Functions in GemFire 446
- Writing the Function Code 447
- Registering the Function 449
- Running the Function From the Calling Application 450
- Running the Function in Custom-Partitioned and Colocated Partitioned Regions 451
- Storing and Retrieving Function Results 452

Chapter 17. Managing Member Relationships *453*

17.1 Setting Membership Roles 454
17.2 Setting Membership Requirements on a Region 455
- Membership Attributes 455
- Configuration Examples 459

17.3 Role Loss Scenarios 460
- Missing Role When a Region is Created 460
- Lost Role When the Cache is Inactive 460
- Lost Role During distributed-no-ack Update 460
- Lost Role During distributed-ack Update 461
- Lost Role During a Global Update 461
- Lost Role During a Transaction 461
- Network Failure 462

Chapter 18. Monitoring and Tuning Your Applications *463*
18.1 Statistics 464
Distributed System and Caching Statistics 466
Partitioned Region Statistics 468
Transient Region and Entry Statistics 469
18.2 Performance Controls 470
Controlling Socket Use 471
Management of Slow Receivers 472
Increasing the Ratio of Cache Hits 475
18.3 Maintaining Cache Consistency 476

Chapter 19. Developing System Administration Tools *479*
19.1 The API Interfaces and Classes 480
System Health Monitoring 483
19.2 Programming Examples 485
Accessing Distributed System Processes 488
Starting and Stopping Managed Entities 488
Accessing System Member Information 489
Cache Management 489
Region Management 491
Health Monitoring 493

Appendix A. gemfire.properties Example File *497*

Appendix B. cache.xml File *499*
B.1 Cache XML Basics 500
B.2 Example Files 501
B.3 GemFire Enterprise Cache XML DTD 505

Appendix C. Query Language Grammar and Reserved Words *523*
Language Grammar 524
Language Notes 527

Glossary *529*

Index *541*

List of Figures

Figure 1.1	The Core Components of the GemFire Enterprise Architecture	36
Figure 1.2	Basic Peer-to-Peer Caching Topology	37
Figure 1.3	Client/Server Caching Topology	38
Figure 1.4	Two-Site Caching Topology	38
Figure 1.5	A Single Member Managing a Cache	41
Figure 1.6	Members Managing Data Across Multiple Machines	42
Figure 1.7	A Member Using a Cache Server to Keep a Region Alive	42
Figure 1.8	Using GemFire Enterprise on Clustered Servers	43
Figure 1.9	Customer Loyalty Program: Managing Session State With Request and Response Queues	44
Figure 1.10	Standard Client/Server Installation	45
Figure 1.11	Logical View Versus Physical View of a Partitioned Region	47
Figure 4.1	Dynamic Region Creation	123
Figure 4.2	Dynamic Region Attributes	124
Figure 6.1	Uneven Data Partitioning	147
Figure 6.2	More Balanced Data Partitioning	148
Figure 6.3	Partitioned Region Including a Member With No Local Data Storage	149
Figure 6.4	Reads on a Partitioned Region With Redundant Data	156
Figure 6.5	The Automatic Distribution of Entry Value Modification	161
Figure 6.6	Union Initialization of a Region With a Data Policy of replicate or preloaded	162
Figure 6.7	Data Flow for a Distributed (Non-Replicated) Region	163
Figure 6.8	Data Flow for a Replicated Region	163
Figure 6.9	Regions with Local Scope	168
Figure 7.1	Data Flow Between Persistent Region and Disk Files	171
Figure 7.2	Data Flow Between Overflow Region and Disk Files	171
Figure 7.3	Data Flow Between Persistent/Overflow Region and Disk Files	172
Figure 7.4	Expiration in a Distributed Region	176
Figure 7.5	The Tenured Memory Thresholds	186
Figure 7.6	VSD chart of Resource Manager Statistics	190
Figure 7.7	VSD Chart of Statistics with Late CMS Startup	191
Figure 7.8	Application Plug-ins in a Partitioned Region	194
Figure 7.9	Application Plug-ins in a Distributed Region	195
Figure 7.10	Cache Loaders in a Partitioned Region	196

Figure 7.11	Cache Loader in a Distributed Region	197
Figure 8.1	Standard Client/Server Installation	203
Figure 8.2	Client/Server Installation using Locators	204
Figure 8.3	Client/Server Installation Using Server Groups	205
Figure 8.4	Client-to-Server Entry Update Flow	217
Figure 8.5	Client Notification of Server Events	219
Figure 8.6	Data Update Flow With Multiple Clients	223
Figure 8.7	Client/Server Caching With Very Small Client Cache	224
Figure 9.1	Client Pool Connections for Client-Initiated Messaging	228
Figure 9.2	Subscription Connections for Server-to-Client Event Streaming	229
Figure 9.3	Subscription Queue Messaging	239
Figure 9.4	High Availability Messaging: Server to Client and Primary Server to Secondary Server	243
Figure 9.5	Initialization of a Reconnected Durable Client.	251
Figure 9.6	Async Subscription Queue Conflation	255
Figure 9.7	Configuration Options for Minimizing Connections in a Client/Server Installation	259
Figure 10.1	GemFire Multi-site Topology	268
Figure 10.2	Site-to-Site Queue Conflation	277
Figure 10.3	Two-Hub Configuration with Primaries in Different VMs	279
Figure 10.4	Outgoing Region Event Flow for Two-Hub Configuration	280
Figure 10.5	Example Three-Site Topology	286
Figure 10.6	Parallel Topology	289
Figure 10.7	Serial Topology	289
Figure 10.8	Hybrid Topology: Example 1	290
Figure 10.9	Hybrid Topology: Example 2	290
Figure 10.10	Unsupported Hybrid Topology 1	291
Figure 10.11	Unsupported Hybrid Topology 2	291
Figure 10.12	Unsupported Hybrid Topology 3	291
Figure 11.1	Cache Listeners in a Partitioned Region With Redundant Data	314
Figure 11.2	Cache Listeners in a Distributed Region	315
Figure 11.3	Empty Producer With a Cache Server and an Empty Event Consumer	318
Figure 12.1	Delta Propagation	328
Figure 12.2	Example of Delta Propagation in the Client/Server	338
Figure 13.1	Storage Model Comparison: Database and Cache	347
Figure 13.2	Database and Cache Storage Models With Data	348
Figure 13.3	Data in Scope Before the Query	349
Figure 13.4	Data in Scope After First FROM Expression is Evaluated	349
Figure 13.5	Data in Scope After Second FROM Expression is Evaluated	349
Figure 14.1	Logical Architecture of Continuous Querying	398
Figure 14.2	Data Flow of a Continuous Query	399
Figure 15.1	Transactions Operating on the GemFire Cache	416
Figure 15.2	Transactional View of the Cache	416
Figure 15.3	Transactions on a Partitioned Region	420
Figure 15.4	Transactions on a Partitioned Region with Other Regions	421
Figure 15.5	Transactional Conflict with Local Scope	422
Figure 15.6	Conflicting Transactions in distributed-ack Regions with Multiple Writers	423
Figure 15.7	Conflicting Transactions in distributed-no-ack Regions with Multiple Writers	425

Figure 16.1 Data-Independent Function Invoked from a Client 440
Figure 16.2 Data-Independent Function Invoked on all Members 441
Figure 16.3 Data-Dependent Function Invoked from a Client 442
Figure 16.4 Data-Dependent Function with Filter Invoked from a Client 443
Figure 16.5 Data-Dependent Function with Filter Invoked from a Non-Client Region 444
Figure 16.6 Highly Available Data Dependent Function with Failure on First Execution 445
Figure 17.1 Required Roles Behavior With a Segmented Network 462
Figure 18.1 com.gemstone.gemfire Statistics API 465
Figure 18.2 Async Distributed System Queue Conflation 474
Figure 19.1 GemFire Core Admin Interfaces 482
Figure 19.2 Setting the GemFire Health Configuration 484

List of Tables

Table 2.1	The Core Java API Packages of GemFire Enterprise	53
Table 3.1	Cache Configuration (cache.xml) Attributes	74
Table 3.2	Cache Configuration (cache.xml) Sub-elements	75
Table 4.1	Region Types.	87
Table 4.2	Region Attributes	101
Table 4.3	Data-policy attributes	104
Table 4.4	Partition Attributes	114
Table 5.1	Automated Steps in Entry Retrieval Based on Region Scope - non-Partitioned	137
Table 8.1	Server Configuration Parameters	207
Table 8.2	Client Pool Configuration	210
Table 8.3	ClientCache Updates That Affect the Server	218
Table 8.4	Server Updates Sent to the Client	220
Table 9.1	Server Redundancy Settings	241
Table 10.1	Region Attribute	272
Table 10.2	Gateway Hub Attributes	272
Table 10.3	Gateway Attributes	273
Table 10.4	Gateway Endpoint Attributes	273
Table 10.5	Gateway Queue Attributes	273
Table 10.6	Gateway Hub Startup Policy and Startup Behavior	284
Table 11.1	Hierarchy of Listener Event Handlers for Cache Events	296
Table 11.2	Hierarchy of Writer Event Handlers for Cache Events	297
Table 11.3	Behavior Changes to Local Methods in Empty Regions.	320
Table 11.4	Behavior Changes of Distributed Methods in Empty Regions	321
Table 11.5	Disabled Region Features in Empty Regions	321
Table 11.6	Hierarchy of After-Event Handlers for Administrative Events	323
Table 13.1	Entry Values in Region portfolios	343
Table 13.2	The Effects of Region Scope and Data Policy on the Data Available for Querying and Indexing	351
Table 13.3	Predefined Class Types	362
Table 13.4	Matrix of SelectResults Contents Based on SELECT and FROM Clause Specifications	365
Table 14.1	Query Operation Based on Old and New Entry Values	405

List of Examples

Example 2.1	JNDI-Binding Property Specification	59
Example 2.2	Configuring XAPooledDataSource Connections in cache.xml	60
Example 2.3	Configuring ManagedDataSource Connections in cache.xml	61
Example 2.4	Configuring Non-transactional Pooled Data Source Connections in cache.xml	62
Example 2.5	Configuring a Simple Data Source Connection in cache.xml	63
Example 3.1	cache.xml (Peer/Server)	77
Example 3.2	cache.xml (Client)	77
Example 3.3	Server and Peer Cache Creation with Config File Specification	84
Example 3.4	Client Cache Creation with Modified Properties	84
Example 4.1	Creating a Default Partitioned Region Through the cache.xml	93
Example 4.2	Creating a Default Persistent Partitioned Region Through the cache.xml	93
Example 4.3	Creating a Partitioned Region Through the cache.xml	93
Example 4.4	Creating a Default Replicated Region Through the cache.xml	93
Example 4.5	Creating a Replicated Region with Expiration Through the cache.xml	93
Example 4.6	Creating a Client Region Through the cache.xml	93
Example 4.7	Creating a Global Distributed Region Through the cache.xml	94
Example 4.8	Creating a Partitioned Region Using RegionFactory	94
Example 4.9	Creating a Modified Partitioned Region Using RegionFactory	94
Example 4.10	Creating a Client Region with a Pool Specification Using ClientRegionFactory	94
Example 4.11	Local and Distributed Invalidation of Regions	95
Example 4.12	Local and Distributed Destruction of Regions	95
Example 4.13	Using a Region Shortcut to Define a Region	97
Example 4.14	Using a Region Shortcut with Overrides	97
Example 4.15	Using Stored Region Attributes in the cache.xml	100
Example 4.16	RegionFactory Use of Attributes Template	100
Example 4.17	Retrieving Stored Attributes Through RegionFactory	100
Example 4.18	Attaching the User Attribute to a Region	120
Example 4.19	Implementing a Cache Writer That Reads the User Attribute	120
Example 4.20	Simple Dynamic Regions Factory Declaration	126
Example 4.21	Dynamic Regions Factory Creation Through the API	127
Example 5.1	Simple Declarative Entry Creation	132
Example 5.2	Simple Declarative Entry Creation with Application-Defined Object Type	132

Example 5.3	Declarative Entry Creation with Nested Application-Defined Object Types	133
Example 5.4	Programmatic Entry Creation	134
Example 5.5	Entry Value Retrieval Using get	135
Example 5.6	Retrieving a Distributed Region's Entries from the Local Cache Only	138
Example 5.7	Deciphering the Entry Value (byte array, object type...)	139
Example 5.8	Entry Modification	140
Example 5.9	Making a Safe Change Using CopyHelper.copy	141
Example 5.10	Do Not Make Changes Like This To Your Cached Entries	141
Example 5.11	Alternative Way To Change a Cache Entry	141
Example 5.12	Using putAll	142
Example 5.13	Local and Distributed Invalidation of Entries	144
Example 5.14	Local and Distributed Destruction of Entries	144
Example 6.1	Simulating Partitioned Region Rebalancing for Analysis	150
Example 6.2	Invoking Partitioned Region Rebalancing at Cache Server Startup	151
Example 6.3	Invoking Partitioned Region Rebalancing Programmatically	151
Example 6.4	API - Configuring Redundancy in a Partitioned Region	153
Example 6.5	XML - Configuring Redundancy in a Partitioned Region	154
Example 6.6	Setting a Recovery Delay for Partitioned Region Redundancy	155
Example 6.7	Programming the PartitionResolver's getRoutingObject	158
Example 6.8	XML Custom Partition Resolver Class	158
Example 6.9	Java Custom Partition Resolver Class	158
Example 6.10	XML Data Colocation	159
Example 6.11	Java Data Colocation	159
Example 6.12	XML Colocating Regions with Custom Partitioning	160
Example 6.13	Java Colocating Regions with Custom Partitioning	160
Example 6.14	Locking an Entry	167
Example 6.15	Unlocking an Entry	167
Example 7.1	Creating a Region with Persistence and Overflow Through the API	173
Example 7.2	Creating a Region with Persistence and Overflow in cache.xml	174
Example 7.3	Setting Standard Expiration on an Entry	181
Example 7.4	Programming a Custom Expiration Class	181
Example 7.5	Installing a Custom Expiry Class	181
Example 7.6	XML - LRU Memory Eviction Controller in Non-Partitioned Region	184
Example 7.7	XML - LRU Memory Eviction Controller in Partitioned Region	185
Example 7.8	XML - LRU Heap Eviction Controller	185
Example 7.9	API Entry Count Conviction	185
Example 7.10	Declarative Heap LRU-Based Eviction with Overflow	185
Example 7.11	Configuration for the Resource Manager	188
Example 7.12	cache.xml	192
Example 7.13	API	192
Example 7.14	Declaring a Cache Loader in cache.xml	198
Example 7.15	Implementing a Simple CacheLoader	198
Example 7.16	Creating a Region with a CacheLoader Through the API	198
Example 7.17	Declaring SimpleCacheLoader	199
Example 7.18	Declaring a CacheLoader with a Database Connection	199
Example 8.1	Client cache.xml	209

Example 8.2	Locator and Server GemFire Properties	213
Example 8.3	Server Configuration	213
Example 8.4	Client Pool and Region Configuration	213
Example 8.5	Publisher Client Configuration Through the API	214
Example 8.6	Subscriber Client Configuration Through the API	214
Example 8.7	Server Configuration for Static Server List Configuration	215
Example 8.8	Client GemFire Properties for Running Standalone	215
Example 8.9	Client Pool and Region for Static Server List Configuration	215
Example 8.10	Client Interest List Registration	222
Example 8.11	Client Interest List Registration Using a Regular Expression	222
Example 8.12	Server Cache Configuration File	225
Example 8.13	Client Cache Configuration File	226
Example 9.1	Setting the Client Message Tracking Timeout Through cache.xml	239
Example 9.2	Using One Redundant Server	241
Example 9.3	Using All Available Servers for Redundancy	241
Example 9.4	Setting the Server Message Sync Interval Through cache.xml	244
Example 9.5	Setting the Server Message Sync Interval Through the API	244
Example 9.6	Setting the Server Message Time to Live Through cache.xml	244
Example 9.7	Setting the Server Message Time to Live Through the API	245
Example 9.8	Configuring a Durable Client Through gemstone.properties	246
Example 9.9	Configuring a Durable Client Through the API	247
Example 9.10	API Client Durable Interest List Registration	247
Example 9.11	Durable Client Cache Ready Notification	248
Example 9.12	Durable Client Disconnect Notification	248
Example 9.13	Enabling Conflation Through the Server's cache.xml	256
Example 9.14	Setting the Clients Queue Acknowledgement Interval	256
Example 9.15	Configuring Subscription Queue Overflow to Disk	257
Example 9.16	Setting the Server Maximum Message Count Through the cache.xml	257
Example 9.17	Setting the Server Maximum Message Count Through the API	258
Example 9.18	Setting the Server's Maximum Client Connections Through cache.xml	260
Example 9.19	Setting the Server's Maximum Client Connections Through the API	260
Example 9.20	Setting the Server's Maximum Client Connections Through cache.xml	261
Example 9.21	Setting the Server's Maximum Client Connections Through the API	261
Example 9.22	Setting the Server Socket Buffer Size Through cache.xml	265
Example 9.23	Setting the Server Socket Buffer Size Through the API	265
Example 9.24	Setting the Client Socket Buffer Size Through cache.xml	266
Example 9.25	Setting the Client Socket Timeout Through cache.xml	266
Example 10.1	Basic Gateway Hub Configuration in cache.xml	270
Example 10.2	Basic Gateway Hub Configuration Through the API	271
Example 10.3	Enabling Region Communication with a Gateway Hub in cache.xml	271
Example 10.4	Enabling Region Communication with a Gateway Hub Through the API	271
Example 10.5	Setting Queue Persistence Through cache.xml	275
Example 10.6	Setting Queue Persistence Through the API	275
Example 10.7	Declarative Gateway Conflation Configuration	278
Example 10.8	Gateway Conflation Configuration Through the API	278
Example 10.9	XML: Two-Hub Gateway Hub Configuration	282

Example 10.10 API: Two-Hub Gateway Hub Configuration 282
Example 10.11 Remote Site Configuration Pointing to the Multiple-Hub Site 282
Example 10.12 XML: Enabling Region Communication with Gateway Hubs 283
Example 10.13 API: Enabling Region Communication with a Gateway Hub 283
Example 10.14 Declarative Gateway Hub Configuration 287
Example 10.15 Gateway Hub Configuration Through the API 288
Example 11.1 Event Handler Declaration in cache.xml 301
Example 11.2 Implementation of an After-Event Handler With Input Parameters 301
Example 11.3 XML: Configuring a Transaction Writer and Listeners 302
Example 11.4 Java: Configuring a Transaction Writer and Listeners 302
Example 11.5 API Region Creation with a Cache Listener 303
Example 11.6 Using a Serial Executor in a Callback Method 304
Example 11.7 Simple Implementation of a RegionMembershipListener 306
Example 11.8 Installing Multiple Cache Listeners in a Region Through the API 307
Example 11.9 Defining Multiple Cache Listeners in a cache.xml File 308
Example 11.10 Implementation of BridgeMembershipListener 310
Example 11.11 BridgeMembershipListener Registration 310
Example 11.12 GatewayEventListener Registration 311
Example 11.13 Enabling WBCL Event Queueing 311
Example 11.14 Implementing the Universal Adapter 325
Example 11.15 Using the Universal Adapter for System and Client Membership Events 326
Example 12.1 Disabling Delta Propagation in gemfire.properties 332
Example 12.2 Disabling Delta Propagation Through the API 332
Example 12.3 Enabling Cloning in cache.xml 333
Example 12.4 Enabling Cloning Through the API 333
Example 12.5 Simple Delta Example Implementation 339
Example 13.1 Entry Value Structure for Region portfolios 343
Example 13.2 Retrieve All Portfolios That Are Active 344
Example 13.3 Retrieve All Portfolios That Are Active and Have Type XYZ 344
Example 13.4 Get the ID and Status of All Portfolios with Positions in secId ‘YYY’ 345
Example 13.5 Get Distinct Positions From All Active Portfolios with at Least a $25.00 Market Value 345
Example 13.6 Get Distinct Entry Keys and Positions from Active Portfolios with at Least a $25.00 Market Value 346
Example 13.7 Nested Query 355
Example 13.8 Outer Scope of a Nested Query 355
Example 13.9 Inner Scope of a Nested Query 355
Example 13.10 Query Using Explicit Iterator Variables 360
Example 13.11 Query Using value-constraint for Object Typing 361
Example 13.12 Using Imported Classes 362
Example 13.13 Query Using IMPORT and TYPE for Object Typing 363
Example 13.14 Query Using IMPORT and Typecasting for Object Typing 363
Example 13.15 Query Using IMPORT and Typecasting for Object Typing 363
Example 13.16 Using Query Parameters 373
Example 13.17 Creating an Index in the cache.xml 376
Example 13.18 Creating an Index Through the API 376
Example 13.19 Single Region Example 1 379

Example 13.20 Single Region Example 2 379
Example 13.21 Single Region Example 3 379
Example 13.22 Multi-region Disjoint Example 1 380
Example 13.23 Equi-join Example 1 381
Example 13.24 Equi-join Example 2 381
Example 13.25 Equi-join Example 3 382
Example 13.26 Equi-join Example 4 382
Example 13.27 Index Creation 390
Example 13.28 Retrieving Index Information 391
Example 13.29 Index Removal 392
Example 13.30 Query Execution 393
Example 13.31 Managing Query Results 394
Example 13.32 Output for Simple Query 395
Example 13.33 Output with SelectResults 395
Example 14.1 CqListener Implementation 404
Example 14.2 CQ Creation, Execution, and Close 406
Example 14.3 Handling the CqResults 407
Example 15.1 Running a Transaction 410
Example 15.2 Transaction Run in Multiple Regions 431
Example 15.3 Transactional Function 434
Example 16.1 Coding the Function execute Method 448
Example 16.2 XML Function Registration 449
Example 16.3 Java Function Registration 449
Example 16.4 Running the Function - For Executing Members 450
Example 16.5 Example Using the PartitionRegionHelper in the Function Execute Method 451
Example 17.1 Setting Membership Attributes 455
Example 17.2 Setting Required Roles in the cache.xml File 459
Example 17.3 Setting Required Roles Through the API 459
Example 18.1 Retrieving Region Statistics 469
Example 18.2 Registering a DataSerializable Class Instantiator 470
Example 18.3 Managing Socket Use at the Thread Level 472
Example 19.1 Creating a Distributed System Administration Interface (dedicated to administration only) 486
Example 19.2 Creating a Distributed System Administration Interface (colocated with DistributedSystem connection) 486
Example 19.3 Starting a Distributed System 487
Example 19.4 Stopping a Distributed System 487
Example 19.5 Locating a System Member by Name 488
Example 19.6 Stopping a cacheserver 488
Example 19.7 Listing System Member Details 489
Example 19.8 Retrieving a Member’s Cache 489
Example 19.9 Listing Cache Contents 490
Example 19.10 Creating a Region 491
Example 19.11 Retrieving Region Contents 492
Example 19.12 Configuring Health Objects 493
Example 19.13 Iterate System Members, Registering Health Configuration for the Member’s Machines 494
Example 19.14 Register Health Configuration for Each Member’s Machine 494

Example 19.15	Event Handler for Health-Related Notifications	495
Example A.1	$GEMFIRE/defaultConfigs/gemfire.properties	497
Example B.1	Simple XML File Listing	499
Example B.2	Declarative Cache Initialization: Two Regions	501
Example B.3	Declarative Cache Initialization: Custom Region Attribute Settings	501
Example B.4	Declarative Cache Initialization of a Region with a Data Entry	502
Example B.5	Declarative Cache Initialization Using the RegionAttributes ID Outside a region Element	504
Example B.6	Declarative Cache Initialization Using the RegionAttributes ID Within a region Element	504
Example B.7	cache6_5.dtd	505

Preface

About This Guide

This guide describes the major services and functions in GemStone® GemFire Enterprise®, which is typically used for distributed caching and data distribution. The GemFire Java® API reference ((Javadocs) lists all of the interfaces and methods supplied for building GemFire Enterprise applications. We recommend that you use the reference pages for specific interface and method information.

The document assumes familiarity with basic Java terminology and programming practices.

How This Guide Is Organized

This guide contains the following sections:

- Chapter 1, *Introduction to GemFire Enterprise*, on page 31 provides an overview of the GemFire Enterprise architecture and services.
- Chapter 2, *Programming with GemFire Enterprise*, on page 49 provides specific information for setting up and running your applications with GemFire Enterprise.
- Chapter 3, *GemFire Members and Member Caches*, on page 69 introduces the basic concepts and tools for configuring and starting up a GemFire Enterprise system member and its cache.
- Chapter 4, *Data Regions*, on page 85 covers the nuts and bolts of cache data region configuration.
- Chapter 5, *Data Operations*, on page 129 discusses data entry creation, access, updating, invalidation, and removal.
- Chapter 6, *Configuration and Management by Region Type*, on page 145 covers basic region configuration options and region management particular to each type of configuration.
- Chapter 7, *General Region Management Options*, on page 169 covers region managment options available to all configuration types. The chapter covers data eviction and expiration, persistence and overflow to disk, and application plug-ins.
- Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201 and Chapter 9, *Client/Server Advanced Topics*, on page 227 give guidance on configuring and programming so that your applications and application data can easily scale with your system's need.

- Chapter 10, *Configuring Multi-site Installations*, on page 267 covers horizontal scaling between disparate distributed systems through the loose coupling of multiple sites in peer-to-peer relationships, including information for configuring and tuning installations.
- Chapter 11, *Handling Events*, on page 293 covers all event handling features of the product, including administrative events as well as cache region and data events.
- Chapter 12, *Delta Propagation*, on page 327 describes how to program your cache to distribute just the changes to your entries, rather than the whole entry object, when you update a cached value.
- Chapter 13, *Querying and Indexing*, on page 341 describes how to create indexes and create and run queries against the data in your GemFire Enterprise cache.
- Chapter 14, *Continuous Querying*, on page 397 describes how use active querying in a client/server installation with queries run on the server cache and results sent to client-side listeners.
- Chapter 15, *GemFire Transactions*, on page 409 describes how to use the GemFire Enterprise transaction service to bundle your caching operations.
- Chapter 16, *Function Execution*, on page 437 describes how to use the GemFire Enterprise function execution service to run your functions across your distributed system and on a server system from a client.
- Chapter 17, *Managing Member Relationships*, on page 453 discusses standard membership roles management, and special configuration and handling for specialized system members.
- Chapter 18, *Monitoring and Tuning Your Applications*, on page 463 gives guidance on statistics analysis, information on creating your own statistics, and pointers for tuning your applications.
- Chapter 19, *Developing System Administration Tools*, on page 479, describes the programmatic interface to the GemFire Administration API.
- Appendix A, *gemfire.properties Example File*, on page 497 lists the example GemFire properties file included in the installation in `defaultConfigs/gemfire.properties`.
- Appendix B, *cache.xml File*, on page 499 describes the declarative cache initialization file, provides usage examples, and lists the file's DTD.
- Appendix C, *Query Language Grammar and Reserved Words*, on page 523 lists the reserved words and the grammar of the query language used in GemFire Enterprise.

Typographical Conventions

This document uses the following typographical conventions:

- Methods, types, file names and paths, code listings, and prompts are shown in `Courier New` typeface. For example:

  ```
  gfPut
  ```

- Parameters and variables are shown in italic font. For example,

  ```
  gfConnect(sysDir, connectionName, writeProtectAllowed)
  ```

- In examples showing both user input and system output, the lines you type are distinguished from system output by **boldface** type:

  ```
  prompt> gemfire
  ```

- If you are viewing this document online, the references are hyperlinks to the places they refer to, like this reference to *Technical Support* on page 29. Blue text denotes a hyperlink.

Other Useful Documents

The GemFire Enterprise online Java API documentation can be accessed through the file `index.html` in the GemFire `docs` directory.

The *GemFire Enterprise System Administrator's Guide* describes the installation, configuration and administration requirements and procedures.

The *Visual Statistics Display* guide describes the Visual Statistics Display (VSD) tool, which is used to analyze archived historical data. Contact GemStone Technical Support for instructions about acquiring VSD and its documentation.

Technical Support

GemStone provides several sources for product information and support. The *GemFire Enterprise Developer's Guide*, the *GemFire Enterprise System Administrator's Guide* and the GemFire Enterprise online Java API provide extensive documentation, and should always be your first source of information. GemStone Technical Support engineers will refer you to these documents when applicable. However, you may need to contact Technical Support for the following reasons:

- Your technical question is not answered in the documentation.
- You receive an error message that directs you to contact GemStone Technical Support.
- You want to report a bug.
- You want to submit a feature request.

Questions concerning product availability, pricing, license keyfiles, or future features should be directed to your GemStone account manager.

Preserving Artifacts for Technical Support

If you have a hung VM and you do not have to kill it, leave it while you contact Technical Support. If you cannot leave the VM running, and it is running under Unix, signal it twice with this command, letting five to ten seconds pass between the two signals:

```
kill -QUIT pid
```

This will send the VM's stack dumps into the log file for inspection. For Windows systems, call Technical Support for assistance in obtaining stack dumps.

Don't delete any files until you call Technical Support and find out exactly what data may be useful to Support or Engineering. Save all the artifacts, including:

- Log files. Send the full log to Technical Support, not just the stack. Even at the default logging level, the log contains data that may be important, such as the operating system and license.
- Statistics archive files.
- Core files.
- For Linux, you can use `gdb` to extract a stack from a core file. Call Technical Support if you need assistance.
- Crash dumps.
- For Windows, save the Dr. Watson output.

Contacting Technical Support

The GemStone support site at http://techsupport.gemstone.com provides all the information you need to contact our technical support team. If you are unable to access the website for any reason, you can e-mail technical support at techsupport@gemstone.com.

When contacting GemStone technical support, please be prepared to provide the following information:

- Your name, company name, and GemFire license number
- The GemFire product and version you are using
- The hardware platform and operating system you are using
- A description of the problem or request
- Exact error messages received, if any
- Any artifacts in the preceding list

Your GemStone support agreement may identify specific individuals who are responsible for submitting all support requests to GemStone. If so, please submit your information through those individuals. All responses will be sent to authorized contacts only.

24x7 Emergency Technical Support

GemStone offers, at an additional charge, 24x7 emergency technical support. This support entitles customers to contact us 24 hours a day, 7 days a week, 365 days a year, if they encounter problems that cause their production application to go down, or that have the potential to bring their production application down. Contact your GemStone account manager for more details.

Training and Consulting

Consulting and training for all GemStone products are available through GemStone's Professional Services organization.

- Training courses are offered periodically at GemStone's offices in Beaverton, Oregon, or you can arrange for onsite training at your desired location.
- Customized consulting services can help you make the best use of GemStone products in your business environment.

Contact your GemStone account representative for more details or to obtain consulting services.

Chapter

1 Introduction to GemFire Enterprise

GemStone® GemFire Enterprise® is a distributed memory oriented data management platform that pools memory (and CPU, network and optionally local disk) across multiple processes to manage application objects and behavior. GemFire uses dynamic replication and data partitioning techniques to offer continuous availability, very high performance and linear scalability for data intensive applications without compromising on data consistency even when exposed to failure conditions. Besides being a distributed data container, it is an active data management system that uses an optimized low latency distribution layer for reliable asynchronous event notifications along with highly concurrent data structures for storage.

GemFire at a Glance

These are the main features of the GemFire Enterprise system.

Heterogeneous Data Sharing

C#, C++ and Java applications can share application business objects without going through a transformation layer such as SOAP or XML. The server side behavior, though implemented in Java, provides a unique native cache for C++ and .NET applications. Application objects can be managed in C++ process heap and distributed to other processes using a common “on-the-wire” representation for objects. A C++ serialized object can be directly deserialized as a equivalent Java or C# object. A change to a business object in one language can trigger reliable notifications in applications written in the other supported languages.

Very High Throughput

Throughput is measured in terms of the total number of operations on data that can be accomplished in a unit of time. Use of concurrent main-memory data structures and a highly optimized distribution infrastructure offers 10X or more read and write throughput compared with traditional disk based databases. Applications can dynamically make copies of data in memory through synchronous or asynchronous replication very high read throughput or partition the data across many cluster members to achieve high read and write throughput.

With data partitioning, doubling the capacity would double the aggregate throughput if the data access is fairly balanced across the entire data set. Linear increase in throughput will only be limited by the backbone network capacity (typically the network switch and not the per node full duplex network capacity of GigaBit).

Low and Predictable Latency

GemFire uses a highly optimized caching layer designed to minimize context switches between threads and processes. Contention points are minimized by managing data in highly concurrent structures, all communication to peer members is done synchronously if the receivers are able to keep up, keeping the latency for data distribution to a minimum. A native serialization mechanism reduces the overhead associated with native serialization when working with nested object graphs. Servers manage object graphs in serialized form decreasing the strain on the garbage collector.

High Scalability

Scalability is achieved through dynamic partitioning of data across many member nodes and spreading the data load uniformly across the servers. For “hot” data, the system can be dynamically expanded to have more copies of the data. Application behavior can also be provisioned to run in a distributed manner in close proximity to the data it needs. If the requirement is to support high and unpredictable bursts of concurrent client load, increasing the number of servers managing the data and spreading the data and behavior across them provides uniform and predictable response times. Clients are continuously load balanced to the fabric server farm based on continuous feedback from the servers on their load conditions. With data partitioned and replicated across servers, clients can dynamically move to different servers to uniformly load the servers and deliver the best response times. Scalability is also improved by providing facilities to asynchronously “write behind” changes to the data to external data sources like a database. The bottleneck is avoided by queuing all updates in order and redundantly. Updates can be conflated and also propagated in batch to the database.

Continuous Availability

Besides guaranteeing consistent copies of data in memory, applications also have the choice to synchronously or asynchronously persist data to disk on one or more nodes (using GemFire's “shared nothing disk architecture”) ensuring very high levels of data availability.

All asynchronous events (store-forward events) are always redundantly managed in at least two member nodes such that if one server fails, the redundant one takes over. All clients connect to logical servers and the client automatically fails over to alternate servers in a group during failures or when servers become unresponsive.

Reliable Event Notifications

Publish/subscribe systems offer a data distribution service where new events are published into the system and routed to all interested subscribers in a reliable manner.

Traditional messaging platforms are only concerned with delivery of messages, but often the receiving applications need access to other related data before being able to process the event. This implies connecting to a relational database when the event is delivered making the subscriber ultimately throttled by the speed of the database or availability of the application limited by the database.

GemFire offers data and events through a single platform. Essentially, data is managed as objects in one or more distributed data regions (like tables in a database). Applications simply insert, update or delete objects in data regions and the platform delivers the object changes to the subscribers. The subscriber receiving the event has direct access to the related data in local memory or can fetch the data from one of the fabric member nodes through a single hop.

Continuous Querying

In messaging systems like JMS, clients subscribe to topics and queues. Any message delivered to a topic is sent to the subscriber. GemFire allows applications to express complex interest using the OQL query language and referred to as continuous querying.

Parallelized Application Behavior on Data Nodes

Application business logic can be parallely executed on the data fabric nodes. GemFire's new data-aware function execution service permits execution of arbitrary data-dependent application functions on the data fabric grid nodes—nodes where the data is striped for scale. The fundamental premise is to route the function transparently to the node that carries the data subset required by the application function and avoid moving data around on the network.

Application function can be executed on just one fabric node, executed in parallel on a subset of nodes or in parallel across all the nodes. This programming model is similar to the now popularized Map-Reduce model from Google.

Data-aware function routing is most appropriate for applications that require iteration over multiple data items (such a query or custom aggregation function). By colocating the relevant data and parallelizing the calculation the overall throughput can be dramatically increased. More importantly, the calculation latency is inversely proportional to the number of nodes on which it can be parallelized.

“Shared Nothing” Disk Persistence

Each data fabric member node (cache) manages data on disk files independent of other members. Failures in disks or cache failures in one node has no impact on another cache instance being able to safely operate on its disk files. This “shared nothing” persistence architecture allows applications to be configured such that different classes of data is persisted on different nodes across the cluster dramatically increasing the overall throughput of the application even when disk persistence is configured for application objects.

Unlike a traditional database system, GemFire does not manage data and transaction logs in separate files. All data updates are appended to files that are similar to transactional logs of traditional databases. Disk seek times can be avoided (if the disk is not concurrently being used by other processes) and the only cost incurred is the rotational latency (note that even some of the best disk technology today takes about 2ms to seek to a track).

Wide Area Data Distribution

Peer-to-peer clusters are subject to scalability problems due to the inherent tight coupling between the peers. These scalability problems are magnified if a data center has multiple clusters or if the data sites are spread out geographically across a WAN. GemFire offers a novel model to address these topologies, ranging from a single peer-to-peer cluster all the way to reliable communications between data centers across the WAN. This model allows distributed systems to scale out in an unbounded and loosely-coupled fashion without loss of performance, reliability or data consistency. At the core of this architecture is a gateway hub for connecting to distributed systems at remote sites

Reduced Cost Of Ownership

Caching can be configured in tiers. The client application process can locally host a cache (in memory and overflow to disk) and delegate to a cache server farm on misses. Even a 30% hit ratio on the local cache can translate to significant savings in costs. The total cost associated with every single transaction comes from the CPU cycles spent, the network cost, the access to the database and intangible costs associated with Database maintenance. By managing the data as application objects the additional cost (CPU cycles) associated with mapping SQL rows to objects is avoided.

In this chapter:

- Architectural Overview (page 36)
- Basic System Topologies (page 37)
- Services (page 39)
- Configuration Scenarios (page 41)

1.1 Architectural Overview

The core components in a GemFire Enterprise system are:

- Caches used for storing data. Cached data is stored in VM heaps and managed by the VMs.
- GemFire system members, Java processes that have connected to each other for data and event distribution. Each system member has a unique identity and is aware of the identities of the other members. System members are:
  - Application programs.
  - Long-running, configurable cache servers. These are generally used to serve cached data to other processes.

  All system member caches can be configured via declarative XML files. Applications can also configure their caches through the API.
- The physical machines and underlying transport layer that host the GemFire and client applications and provide distribution of data between machines.

The glue connecting these components is the GemFire Enterprise system. The distributed system defines how the processes communicate and gives access to the APIs that are used to manage the distributed data and related events. Each distributed system is identified by its communication protocol and address information.

These components can be combined in a wide variety of configurations for very simple caching needs, like increasing the performance of database access for a single application, to very complex needs such as for grid computing and widely distributed systems. This chapter lists the features of GemFire Enterprise and describes some of the more common caching scenarios.

This example shows the basic architecture of a single GemFire Enterprise system. The distributed system connects the GemFire Enterprise components and provides access to the distributed data programming APIs.

Figure 1.1 The Core Components of the GemFire Enterprise Architecture

1.2 Basic System Topologies

GemFire Enterprise supports these basic caching topologies, which can be combined to satisfy a wide variety of needs:

Embedded, Peer-to-Peer Caching

In this topology, the cache is embedded within the application process and shares the heap space with the application. Applications with embedded caches become part of their distributed system as peer-to-peer members. Each cache member directly communicates with every other cache member.

In a GemFire distributed system, data is stored in caches in the VM heaps and automatically distributed according to the member's cache configuration.

Figure 1.2 Basic Peer-to-Peer Caching Topology

Client/Server Caching

In a client/server installation, a central cache is managed in one distributed system tier by a number of server members, usually cacheserver processes. In a separate tier, clients maintain their own caches that automatically call upon the server side to satisfy cache misses and update the server with client updates. In addition, the clients can subscribe to server events using both data key specifications and queries. The client/server topology can be extended to any number of tiers.

Figure 1.3 Client/Server Caching Topology

Multi-site Caching

With this topology, distributed systems at different sites are loosely coupled through gateway system members. The distribution of cached data between sites is seamless to the applications within each distributed system. If any system becomes unavailable, the rest of the installation continues to operate.

In a multi-site topology, each site has its own distinct distributed system. In each system, at least one member is chosen to act as a gateway that distributes data to and from the other sites.

Figure 1.4 Two-Site Caching Topology

1.3 Services

Each member of a GemFire Enterprise distributed caching system defines and configures its local cache in its heap. The distributed cache is made up of the combination of all member caches. Data distribution and distributed data management are carried out by the distributed system according to the local cache configurations.

The GemFire Enterprise cache is organized into a hierarchical region structure, like a map of maps. Each data region can be configured with its own set of data and event services. These are the primary services available with the product:

- **Distribution**

  You can choose to keep your region data local, private to the application, or to distribute it. For your distributed data, you can choose whether to distribute asynchronously without acknowledgement (no ACK), synchronously (ACK), or with locking.

- **Data Storage Policy**

  You can choose whether your local cache stores data for the region and, if it does, whether it stores all of the distributed region data (replication) or just the data that the member explicitly creates in the local cache. The replicated region is initialized from all distributed region data and automatically receives updates for all of the distributed region's entry activity. The empty region, with no local cache storage, can be used for such processes as very thin producers that put data into the region only for distribution to other members, and very thin clients that handle distributed region events but have no need for local data storage.

- **Partitioning**

  The partitioned regions feature gives you a highly available, distributed caching option for the GemFire Enterprise caching product. Partitioned regions can manage large quantities of data across many machines by partitioning it into manageable chunks and distributing it efficiently in a balanced way. In case of system failure or planned maintenance, partitioned regions can provide redundancy for high availability. Partitioning provides scalability and reliability for applications where the data set is very large.

- **Loading**

  A cache region can have read-through functionality to a back-end data source by implementing a data loader. The loader is a user-defined plug-in invoked for every cache miss on the region. While loaders are usually used for read-through activities, the plug-in can be programmed to do anything your application requires.

- **Events**

  You can specify write-ahead listeners ("before" events) and write-behind listeners ("after" events) for your region and region data operations. You can configure whether the listener is to receive callbacks only for local cache events or for all distributed region events. Like loaders, event listeners are user-defined and can do whatever is required by your application. Synchronous listeners can be used to implement write-through caching for the region, and asynchronous listeners can be used for write-behind functionality. Synchronous listeners have the ability to abort operations.

- **Persistence**

  You can persist region data to disk and maintain it in memory independent of your applications.

  Persistence to disk is specified in a set of region attributes that include the disk file locations. When a region is configured for persistence, the disk files are automatically kept in sync with the data in memory. A region disk copy outlives the member that holds the region and remains available for loading when the region is recreated.

For in-memory storage, GemFire Enterprise provides a long-lived `cacheserver` utility, independent of your applications, where you can define any region. A region in the cache server lives independently of your application processes. When members crash or exit, the data in the cache server remains. When new applications start they can retrieve the current region data from the cache server. This is handy for systems with transient applications, such as those requiring the same data set for a series of distinct processing activities. For more information on the `cacheserver` utility, see *Configuring and Running the GemFire Cache Server* on page 126 of the *GemFire Enterprise System Administrator's Guide*.

- **Memory**

  The amount of memory a region consumes can be managed through data entry and region expiration settings, and with least recently used (LRU) eviction settings. Entry expiration allows you to remove data that has not been updated or accessed for a specified amount of time. When eviction is configured, GemFire Enterprise monitors region size or entry count and, when capacity is reached, stops region growth by removing or overflowing to disk the stalest LRU entries.

- **Querying**

  You can run SQL-style queries against the cached data in your application heap. You can also define and manage indexes to improve the performance of your queries. In client/server applications, you can create continuous queries that are run actively against server cache events, with relevant querying events sent to listeners on the client side.

- **Transactions**

  Transactions can be executed against the data in the distributed cache. Basic GemFire transactions provide begin, commit, and rollback capabilities. Additionally, GemFire Enterprise provides an implementation of the Java Transaction API (JTA) and, when run inside a J2EE container, is compatible with the container's JNDI and JTA transaction manager.

- **Member Role Requirements**

  Membership roles can be defined for your system members, then requirements can be set at the region level for the roles that must be available in the distributed system for the region to function normally. You can choose whether region access should be interrupted or limited if a required role is missing, and you can configure the action to take once all the roles are restored. This is useful, for example, to keep a producer from putting data into a cache unless there is a consumer to process the put events, or to keep a consumer from getting stale data out of the cache while its producer is down.

1.4 Configuration Scenarios

This section gives an introduction to the basic configurations available with GemFire Enterprise.

The areas covered are:

- Peer-to-Peer Caching (page 41)
- Client/Server Caching (page 45)
- Partitioned Data Caching (page 46)

Peer-to-Peer Caching

This section includes two simple scenarios for peer-to-peer caching, a single-member scenario and one where data is distributed across different machines.

Simple Application Scenarios

The most basic GemFire Enterprise configuration is a single member managing a cache in its heap. Cache size and contents are determined at creation time and through subsequent calls to the caching API.

Application threads in this member use the caching API to manage data in the heap. There are no meaningful distribution activities associated with this configuration, as all data is directly accessed by the threads. The caching API may provide data loading or synchronization with a backend database.

Figure 1.5 A Single Member Managing a Cache

GemFire Enterprise is commonly used to distribute data between different processes and different machines. This situation requires at least one member per machine, each with its own cache. Data distribution between the processes is handled by the distributed system.

For their data to be distributed between the members, these regions must either be partitioned or have distributed scope. Other region attributes may vary to satisfy differing application and system requirements.

Figure 1.6 Members Managing Data Across Multiple Machines

These simple configurations could be "pure Java" systems— those that run without the use of GemFire native code. Pure Java systems are discussed in *Running in Pure Java Mode* on page 52.

Using a Cache Server to Maintain a Region In Memory

Adding a cache server to the single-member scenario in Figure 1.5 allows the member to keep regions alive in the distributed system regardless of the status of the member. The entire member could exit or crash, and it will retrieve the data from the cache server region when it starts back up.

This member uses the region in the cache server to make sure the data outlives any of its threads or even the member itself. The regions' data policy is set to `replicate`, so the members have identical data sets. The application can exit and restart, retrieving the full set of current region data from the cache server.

Figure 1.7 A Member Using a Cache Server to Keep a Region Alive

Clustered Servers

GemFire Enterprise can be used in a clustered server configuration to maintain data that must be current and available on all servers in the cluster. In this scenario, regions are configured as replicas for data duplication, meaning they are initialized with the same sets of data and updated with the same data updates. As a general rule, the distributed system configurations are the same on each machine, however, some settings may vary to reflect varying system resources between the servers. For example, differences in the amount of memory available on each host system might prompt different settings for expiration attributes and eviction controllers. Cache loaders, writers, and listeners would most likely be duplicated between machines.

In this scenario, *n* clustered servers are used for high availability. Each has the same cache and application configurations. The regions are all configured as replicas.

Figure 1.8 Using GemFire Enterprise on Clustered Servers

Managing Session State With Request and Response Queues

This scenario involves a web-based customer loyalty program. Customers receive points that they can use online to manage, trade, and purchase items. The distributed system consists of an application data-warehousing member and a replicated servlet engine. The servlet engine manages HTTP client sessions and forwards requests and responses between the clients and the warehousing member. The warehousing member processes requests using a loyalty points database and other data sources.

The servlet stores client requests in `Request` region entries. These are automatically replicated by the warehousing member's cache and picked up its request region listener. The warehousing member processes the requests and puts responses into `Response` region entries. These are, in turn, replicated by the region in the servlet's cache and processed by the response region listener running there.

The servlet engine stores each client's session state in a region that is configured for replication in the servlet's replica member. If the backup engine is needed, it creates the request and response regions when it is brought online. These replicas are automatically populated from the warehousing cache.

Expiration may be set on the session state region to ensure old session entries are cleared out. The members might destroy requests and responses as they are processed, or a listener for the Session region may destroy all associated Request and Response entries when the client's session entry is expired.

Figure 1.9 Customer Loyalty Program: Managing Session State With Request and Response Queues

Client/Server Caching

In the standard client/server topology, a farm of cache servers provides caching services to a large number of clients:

- Cache servers provide a homogeneous datastore for clients. The server data regions are replicated or partitioned across the server farm.
- Cache servers send their address and load information to the server locator.
- The clients go to the server locator for server connection information. For each request, the locator responds with the address of the least loaded server.
- Client connections are checked periodically for server load balancing and conditioned as needed.
- Clients can subscribe to events at startup. Events are automatically streamed from the servers and sent to client listeners and into the client cache.
- Client API calls, like data updates and data gets, that aren't satisfied by the client cache are forwarded automatically to the servers.

Figure 1.10 Standard Client/Server Installation

Partitioned Data Caching

Partitioning provides scalability and reliability for applications where the data set is very large. When the total size of the data is greater than the heap in any single member, partitioned regions can provide the needed data management.

Architecture of a Partitioned Region

Like any distributed region, you set up a partitioned region by creating region instances with the same name in several members. The only difference is that you enable partitioning during region creation.

> *This document uses the term* partitioned region *to refer to the partitioning distribution model, and the term* distributed region *to refer to the standard GemFire distribution model.*

During operation, a partitioned region looks like one large virtual region, as shown in Figure 1.11 on page 47. The distribution of entries across the region is handled automatically by GemFire.

A distributed system can have multiple partitioned regions, and it can mix partitioned regions with distributed regions and local regions. The usual requirement for unique region names, except for regions with `local` scope, still applies.

A single member can act as a data host for multiple partitioned regions.

Automatic Data Partitioning

GemFire handles the physical location of data in the members that participate in a partitioned region. The first time the application creates a partitioned region, GemFire breaks the data into units of storage known as *buckets*. Buckets are the unit of distribution and replication, and they are distributed in accordance to the region's attribute settings. When an entry is created, it is assigned to a bucket. You can custom partition your region, grouping the data together in logical units. If no bucket has room for the entry, a new bucket is created. Keys are grouped together in a bucket and always remain there, but buckets are not guaranteed to stay in the same member.

The application is responsible for using enough data host members for the region to accommodate storage for all of the data. New storage can be made available on the fly by starting more members. When a new member creates the partitioned region, it takes responsibility for as many buckets as allowed by its configuration.

Operation

A partitioned region operates much like a non-partitioned region with distributed scope. Most of the standard `Region` methods are available, although some methods that are normally local operations become distributed operations, because they work on the entire partitioned region instead of the local cache. For example, a `put` or `create` into a partitioned region may not actually be stored into the cache of the member that did the `put` or `create`. In the simple configuration shown in Figure 1.11 on page 47, you can do what looks like an ordinary `put` of entry Z into member M_1, but the result shows up in the remote cache in member M_3. After the `put` of entry Z into the partitioned region, M_1 can then use a call to `Region.get` to return the value without knowing whether that entry is cached locally or remotely. The retrieval of any entry requires no more than one hop between members.

Partitioned regions support the client/server model, just like other regions. If you need to connect dozens of clients to a single partitioned region, using servers greatly improves performance. For more information, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.

Figure 1.11 Logical View Versus Physical View of a Partitioned Region

Data Redundancy for High Availability

You can configure partitioned regions with redundant storage for fault tolerance. You can specify the creation of as many as three extra copies of the entries. GemFire puts each redundant copy in a different member's cache. If a data host becomes unavailable, the partitioned region still has all its data and continues to operate without interruption. For details, see *Partitioned Regions with Redundancy* on page 190.

You need to start the proper number of members to support the partitioned region's configuration and to configure them as long-running application or cache server processes that can host partitioned data for the lifetime of the partitioned region.

Chapter 2

Programming with GemFire Enterprise

This chapter provides specific information for getting started programming your GemFire Enterprise® system. The topics covered include information for developing and running your GemFire applications, a high-level view of the programming APIs, requirements for your `CLASSPATH` settings and for data distribution, and tools you can use to set your configuration options. This topic also includes information on the programming examples. Other topics of interest to the developer, such the GemFire product tree and information on starting and stopping the components of a GemFire system, are covered in the *GemFire Enterprise System Administrator's Guide*. Throughout this chapter, `productDir` is used to represent the GemFire Enterprise installation directory.

In this chapter:

- Running Your GemFire Applications (page 50)
- The Programming APIs (page 53)
- Configuring and Monitoring Your Applications (page 55)
- Defining Your Own Custom Classes (page 57)
- Configuring Database Connections Using JNDI (page 59)
- Handling Exceptions and System Failures (page 65)
- Considerations for Your Host File Systems (page 66)
- About the Programming Examples (page 67)

2.1 Running Your GemFire Applications

This section provides information for setting up your Java environment for GemFire and for compiling and running GemFire programs.

The sections contained here cover:

- Configuring Your Environment (page 50)
- Running on Solaris and Linux 64-bit Machines (page 52)
- Running in Pure Java Mode (page 52)

Configuring Your Environment

This section describes everything you need in your environment for GemFire.

A Java runtime environment is not provided with GemFire, so you need to have your own compatible Java JDK or runtime JRE installed. For GemFire Enterprise Java requirements, see *System Requirements* on page 20 of the *GemFire Enterprise System Administrator's Guide*.

1. Set the `JAVA_HOME` environment variable to point to your Java runtime installation. There should be a bin directory under `JAVA_HOME`.
2. Set the `GEMFIRE` environment variable to point to your GemFire installation.
3. Update your `PATH` environment variable to include the `bin` directories under `JAVA_HOME` and `GEMFIRE`.

   For example, for Unix Bourne and Korn shells (sh, ksh, bash):

   ```
   > PATH=$PATH:$JAVA_HOME/bin:$GEMFIRE/bin;export PATH
   ```

   and Windows

   ```
   > set PATH=%PATH%;%JAVA_HOME%\bin;%GEMFIRE%\bin
   ```

4. Update your `CLASSPATH` environment variable to include the GemFire classes:

   Unix Bourne and Korn shells (sh, ksh, bash):

   ```
   > CLASSPATH=
   $GEMFIRE/lib/gemfire.jar:$GEMFIRE/lib/antlr.jar:$GEMFIRE/lib/gfSecurityIm
   pl.jar:$CLASSPATH;export CLASSPATH
   ```

   Windows:

   ```
   > set CLASSPATH=
   %GEMFIRE%\lib\gemfire.jar;%GEMFIRE%\lib\antlr.jar;%GEMFIRE%\lib\gfSecurit
   yImpl.jar;%CLASSPATH%
   ```

5. Set a `GF_JAVA` environment variable to point to the `java` executable file under your `JAVA_HOME`.

   Unix Bourne and Korn shells (sh, ksh, bash):

   ```
   > GF_JAVA=$JAVA_HOME/bin/java;export GF_JAVA
   ```

   Windows:

   ```
   > set GF_JAVA=%JAVA_HOME%\bin\java.exe
   ```

Optional GemFire Native Library Settings

For standard GemFire operation, you need to locate the dynamically loadable GemFire native library. (You can run without the native library, but with some functional differences. For more information on this, see *Running in Pure Java Mode* on page 52.)

The native library is expected to reside in the same directory as `gemfire.jar`. If it cannot be loaded from there, the OS-dependent `LD_LIBRARY_PATH` or `PATH` variables are searched. You can update your `PATH` environment variable to include the `bin` directories under `JAVA_HOME` and `GEMFIRE`.

For Unix Bourne and Korn shells (sh, ksh, bash):

```
> LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GEMFIRE/lib;export LD_LIBRARY_PATH
```

and Windows

```
> set PATH=%PATH%;%GEMFIRE%\lib
```

Running on Solaris and Linux 64-bit Machines

You can run your GemFire Enterprise applications on Solaris and Linux 64-bit machines.

Solaris

GemFire Enterprise supports the Sun 64-bit VM on Solaris 9 or later. Each member can be a 64- or 32-bit VM, independent of the mode in which other members of the distributed system are run.

A 64-bit version of the Java HotSpot Virtual Machine is required as part of your Java runtime environment. You install the 64-bit Java VM after first installing the 32-bit VM. To run the 64-bit VM, launch the Java executable with `-d64` as the first command line option.

Linux

Your applications will run in the 32-bit VM on 64-bit machines unless you install and configure the 64-bit VM. For the 64-bit VM, make sure that the 64-bit GemFire shared native library, `libgemfire64.so`, is in the directory referenced by your `LD_LIBRARY_PATH` environment variable. This ensures that the shared native library is loaded automatically.

64-bit VM and Pure Java Mode

See *Running in Pure Java Mode*.

- To avoid running in pure Java mode with the 64-bit VM, make sure that `libgemfire64.so` is in a directory referenced by your `LD_LIBRARY_PATH` environment variable. This ensures that `libgemfire64.so` is loaded automatically.
- To run in pure Java mode, launch the VM with the `-Dgemfire.pureJavaMode=true` system property. Using that system property, `libgemfire64.so` is not loaded even if it is detected.

Running in Pure Java Mode

GemFire Enterprise is designed to run as documented on platforms running Solaris, Linux, and Microsoft Windows. GemFire Enterprise runs with some functional differences in *pure Java mode* without GemFire native code. In pure Java mode GemFire's Java code does not call any of its own native methods. Distributed system members still have access to GemFire's caching and distribution capabilities.

To run in pure Java mode, your application must have the proper `CLASSPATH` settings and must *not* have access to the GemFire native library (see *Optional GemFire Native Library Settings* on page 51). Pure Java mode is enabled automatically when GemFire's Java code cannot load the GemFire native library. Pure Java mode can be forced on platforms that support native code by renaming the native library in the product tree, or by setting the `gemfire.pureJavaMode` system property to `true` on members running user applications.

Note that there are conditions associated with running in pure Java mode on some systems. Details are available at http://techsupport.gemstone.com/gemfire/learning_center/bug_notes/index.htm.

2.2 The Programming APIs

The Java APIs, supplied in *productDir*/lib/gemfire.jar, allow you to configure your distributed system and cache, manage and query cached data, and run administrative tasks. This table contains the primary GemFire Enterprise packages. For a complete list, see the Java API online documentation in the *productDir*/docs directory.

Table 2.1 The Core Java API Packages of GemFire Enterprise

Package	Description
com.gemstone.gemfire	**Statistics, Logging, Data Serialization, and System Failure** The classes, interfaces, and exceptions used for logging and statistics and for data serialization and system failure management. This includes the LogWriter and Statistics classes that allow application-specific logging and statistics gathering. This also includes DataSerializable and SystemFailure.
com.gemstone.gemfire.admin	**Administration API** Tools for administering and monitoring GemFire distributed systems. See Chapter 19, *Developing System Administration Tools*, on page 479.
com.gemstone.gemfire.cache	**Caching** An implementation of distributed object caching that is capable of leveraging GemFire's distribution capabilities. This package includes Cache, Region, and Region.Entry interfaces. SeeChapter 3, *GemFire Members and Member Caches*, on page 69 and Chapter 4, *Data Regions*, on page 85
com.gemstone.gemfire.cache.client	**Client** Used to configure client applications for communication with cache servers. See Chapter 3, *GemFire Members and Member Caches*, on page 69 and Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.
com.gemstone.gemfire.cache.query	**Querying and Indexing** An implementation of OQL querying for local cache data and continuous querying on a server by a client. Supplemented by querying shortcut methods in cache.Region. See Chapter 13, *Querying and Indexing*, on page 341 and Chapter 14, *Continuous Querying*, on page 397.
com.gemstone.gemfire.cache.server	**Server** Used to run GemFire servers and to replace the default GemFire server load probe software with custom software written by you. See *Balancing Server Use* on page 232.
com.gemstone.gemfire.cache.util	**Customized Caching Classes** Utilities for use with the caching classes. These utilities include an object sizer that returns the size of the object passed in, convenience stub implementation classes for GemFire application plug-ins (see *Controlling Data Flow With Application Plug-ins* on page 193) and customized application plug-ins that implement the client side in client/server caching (see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201).

Package	Description
`com.gemstone.gemfire.distributed`	**Distributed System Membership** Common tools for GemFire distributed systems including the `DistributedMember` and `Locator` classes.
`com.gemstone.gemfire.security`	**Security and Authorization** Tools for authenticating and authorizing system and data access. The use of this API is covered in *Security* on page 73 of the *GemFire Enterprise System Administrator's Guide*.

2.3 Configuring and Monitoring Your Applications

In addition to the programming APIs, you can use the following to configure and monitor your GemFire system.

GemFire Configuration Files

The files listed here are used for GemFire deployment and system configuration. Of these files, GemFire only requires the license. If the other files are not available, the system runs with default values.

- **gemfire.properties**—This file contains the settings required to join a distributed system. This file is described in *Configuring GemFire System Properties* on page 45 of the *GemFire Enterprise System Administrator's Guide*.
- **gemfireLicense.zip**—This is the license file provided to you by GemStone that allows you to use the product. This file is described in *GemFire Licenses* on page 24 of the *GemFire Enterprise System Administrator's Guide*.
- **cache.xml**—This is the declarative cache configuration file. Cache configuration is described in Chapter 3, *GemFire Members and Member Caches*, on page 69 and Chapter 4, *Data Regions*, on page 85.

The file names listed are defaults. These files can be deployed in a number of places including the application jar file. For more information on file specifications, deployment options, and file location search order, see *GemFire Configuration Files* on page 42 of the *GemFire Enterprise System Administrator's Guide*.

java.lang.System Properties

A number of configuration properties may be specified in a `java.lang.System properties` object. When you create your cache and connect to a distributed system, the connection references system properties for its input parameters. The system properties referenced by the GemFire Java API methods are detailed in the individual method documentation.

You can set these properties inside your Java applications or at the command line.

> *Check with your GemFire system administrator before changing distributed system properties through the API, including the* `gemfire.properties` *and license files settings. The system administrator may need to set distributed system properties at the command line or in configuration files. Any changes you make through the API would override the other settings.*

GemFire Command-line Utility

GemFire Enterprise includes a command-line utility (`gemfire`) for administering, inspecting, monitoring, and analyzing GemFire Enterprise systems. For details, see *The gemfire Command-line Utility* on page 227 of the *GemFire Enterprise System Administrator's Guide*.

GemFire Statistics

GemFire provides statistics for analyzing system performance. Each application or cache server that joins the distributed system can collect and archive this statistical data: You set the configuration attributes that control statistics collection in the `gemfire.properties` configuration file. For more information, see *System Statistics* on page 235 of the *GemFire Enterprise System Administrator's Guide*.

You can also collect your own application defined statistics. This is described in *Statistics* on page 464.

To view and analyze archived historical data, you can use Visual Statistics Display (VSD). Contact GemStone Technical Support for instructions about acquiring VSD. For documentation, see the *Visual Statistics Display* manual.

2.4 Defining Your Own Custom Classes

This section covers the general requirements for defining your own classes for your data objects and other uses, such as sizers for your cached objects.

CLASSPATH Requirements

Each member's `CLASSPATH` must include classes for all objects the member accesses. This includes objects stored in the cache and objects accessed through events.

- For Java applications, this is the standard Java `CLASSPATH`.
- The `cacheserver` process uses the `CLASSPATH` environment variable, or the cacheserver's `-classpath` parameter. For details, see *Configuring and Running the GemFire Cache Server* on page 126 of the *GemFire Enterprise System Administrator's Guide*.
- Data is sent between clients and servers in serialized form and the server stores client data in serialized form. The server does not need to deserialize data to send it to another client, but it does need to deserialize it to access it. The server `CLASSPATH` must include the classes for:
  - All entry keys
  - Entry values in regions that the server persists to disk
  - Entry values the server accesses for any reason other than transfer of the full entry value to a client

Data Serialization Requirements

Almost every data configuration in GemFire requires serialization.

GemFire serializes data entry keys and values for distribution, so all data that GemFire moves out of the local cache for any reason must be serializable. Additionally, partitioned regions store data in serialized form.

You need serialization if any of these is true about your data region:

- It is partitioned
- It is distributed
- It receives events from other caches
- It is persistent
- It is a server or client region in a client/server installation
- It is distributed between gateways in a multi-site installation

If you use custom data objects in any of these situations, you can use the GemFire interface `com.gemstone.gemfire.DataSerializable` to help create your own data serializers. The interface may also be used to replace standard data serialization for better performance. For information, see *Data Serialization* on page 470.

Requirements for Classes Used as Keys

For the class, override:

- `equals`
- `hashCode` - The default `hashCode` inherited from `Object` uses identity, which is different in every system member. In partitioned regions, hashing based on identity puts data in the wrong place. For details, see the Java API documentation for `java.lang.Object`.

2.5 Configuring Database Connections Using JNDI

Configure JNDI data sources in the `cache.xml`. The `DataSource` object points to either a JDBC connection or, more commonly, a JDBC connection pool. The connection pool is usually preferred, because a program can use and reuse a connection as long as necessary and then free it for another thread to use.

- **XAPooledDataSource**—pooled SQL connections.
- **`ManagedDataSource`**—JNDI binding type for the J2EE Connector Architecture (J2CA) `ManagedConnectionFactory`. For information on the `ManagedConnection` interface, see:

  http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/resource/spi/ManagedConnection.html
- **PooledDataSource**—pooled SQL connections.
- **SimpleDataSource**—a single SQL connection. No pooling of SQL connections is done. Connections are generated on the fly and cannot be reused.

The `jndi-name` attribute of the `jndi-binding` element is the key binding parameter. If the value of `jndi-name` is a `DataSource`, it is bound as `java:/`*`myDatabase`*, where *`myDatabase`* is the name you assign to your data source. If the data source cannot be bound to JNDI at runtime, GemFire logs a warning. For information on the `DataSource` interface, see:

http://java.sun.com/j2se/1.5.0/docs/api/javax/sql/DataSource.html

GemFire supports JDBC 2.0 and 3.0.

> *Include any data source jar files in your* `CLASSPATH`.

You can create multiple data sources of any of the types listed, but all must connect to only one outside resource. You can combine connections from all types of data sources, but the behavior will depend on the underlying resource manager. See the documentation from your database vendor.

For information on transactions, see Chapter 15, *GemFire Transactions*, on page 409.

Configuring JNDI-Binding Properties

The following sections show example `cache.xml` files configured for each of the `DataSource` connection types. You specify the JNDI binding properties through the `config-property` tag, as shown in this example. For other properties, see the vendor documentation for your database. You can add any number of `config-property` elements.

Example 2.1 JNDI-Binding Property Specification

```
<jndi-binding . . .
   <config-property>
      <config-property-name>databaseName</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-value>newDB</config-property-value>
   </config-property>
</jndi-binding>
```

Configuration Examples

The example below shows a `cache.xml` file configured for a pool of `XAPooledDataSource` connections connected to the data resource `newDB`. The log-in and blocking timeouts are set lower than the defaults. The connection information, including `user-name` and `password`, is set in the `cache.xml` file, instead of waiting until connection time. The password is encrypted; for details, see *Encrypting Connection Passwords* on page 64.

Example 2.2 Configuring XAPooledDataSource Connections in cache.xml

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC
   "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<cache lock-lease="120" lock-timeout="60" search-timeout="300">
   <region name="myRegion">
      <region-attributes refid="REPLICATE"
             initial-capacity="16"
             load-factor="0.75" concurrency-level="16"
             statistics-enabled="true">
      . . .
   </region>
   <jndi-bindings>
      <jndi-binding
         type="XAPooledDataSource"
         jndi-name="newDB2trans"
         init-pool-size="20" max-pool-size="100"
         idle-timeout-seconds="20"
         blocking-timeout-seconds="5"
         login-timeout-seconds="10"
         xa-datasource-class="org.apache.derby.jdbc.EmbeddedXADataSource"
         user-name="mitul"
         password="encrypted(83f0069202c571faf1ae6c42b4ad46030e4e31c17409e19a)"
         >
         <config-property>
            <config-property-name>description</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>pooled_transact</config-property-value>
         </config-property>
         <config-property>
            <config-property-name>databaseName</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>newDB</config-property-value>
         </config-property>
         <config-property>
            <config-property-name>vendor_specific_property1
            </config-property-name>
            <config-property-type>type</config-property-type>
            <config-property-value>value</config-property-value>
         </config-property>
      . . .
      </jndi-binding>
   </jndi-bindings>
</cache>
```

`ManagedDataSource` connections for the J2CA `ManagedConnectionFactory` are configured as shown in the example below. This configuration is similar to `XAPooledDataSource` connections, except the type is `ManagedDataSource`, and you specify a `managed-conn-factory-class` instead of an `xa-datasource-class`.

Example 2.3 Configuring ManagedDataSource Connections in cache.xml

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC
       "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<cache lock-lease="120" lock-timeout="60" search-timeout="300">
   <region name="myRegion">
      <region-attributes refid="REPLICATE" initial-capacity="16"
             load-factor="0.75" concurrency-level="16"
             statistics-enabled="true">
      . . .
   </region>
   <jndi-bindings>
      <jndi-binding
         type="ManagedDataSource"
         jndi-name="DB3managed"
         init-pool-size="20" max-pool-size="100"
         idle-timeout-seconds="20"
         blocking-timeout-seconds="5"
         login-timeout-seconds="10"
         managed-conn-factory-class="com.myvendor.connection.ConnFactory"
         user-name="mitul"
         password=
"encrypted(83f0069202c571faf1ae6c42b4ad46030e4e31c17409e19a)"
         >
         <config-property>
            <config-property-name>description</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>pooled_transact</config-property-value>
         </config-property>
         <config-property>
            <config-property-name>databaseName</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>newDB</config-property-value>
         </config-property>
         <config-property>
            <config-property-name>vendor_specific_property1
            </config-property-name>
            <config-property-type>type</config-property-type>
            <config-property-value>value</config-property-value>
         </config-property>
      . . .
      </jndi-binding>
   </jndi-bindings>
</cache>
```

This example shows a `cache.xml` file configured for a pool of `PooledDataSource` connections to the data resource `newDB`. For this connection pool, the log-in and blocking timeouts are set higher than for the transactional connection pools in the two previous examples. The connection information, including `user-name` and `password`, is set in the `cache.xml` file, instead of waiting until connection time. The password is encrypted. See *Encrypting Connection Passwords* on page 64.

Example 2.4 Configuring Non-transactional Pooled Data Source Connections in cache.xml

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC"-//GemStone Systems, Inc.//GemFire Declarative
Caching 6.5//EN" "http://www.gemstone.com/dtd/cache6_5.dtd">
<cache lock-lease="120" lock-timeout="60" search-timeout="300">
   <region name="myRegion">
      <region-attributes refid="REPLICATE" initial-capacity="16"
             load-factor="0.75" concurrency-level="16"
             statistics-enabled="true">
      . . .
   </region>
   <jndi-bindings>
      <jndi-binding
         type="PooledDataSource"
         jndi-name="newDB1"
         init-pool-size="2" max-pool-size="7"
         idle-timeout-seconds="20"
         blocking-timeout-seconds="20"
         login-timeout-seconds="30"
         conn-pooled-datasource-class=
                "org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource"
         user-name="mitul"
         password=
"encrypted(83f0069202c571faf1ae6c42b4ad46030e4e31c17409e19a)" >
         <config-property>
            <config-property-name>description</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>pooled_nontransact</config-property-
value>
         </config-property>
         <config-property>
            <config-property-name>databaseName</config-property-name>
            <config-property-type>java.lang.String</config-property-type>
            <config-property-value>newDB</config-property-value>
         </config-property>
         <config-property>
            <config-property-name>vendor_specific_property1
            </config-property-name>
            <config-property-type>value</config-property-type>
            <config-property-value>value</config-property-value>
         </config-property>
      . . .
      </jndi-binding>
   </jndi-bindings>
</cache>
```

This example shows a configuration in the `cache.xml` file for a `SimpleDataSource` connection to the data resource `oldDB`. You only need to configure a few properties like a `jndi-name` for this connection pool, `oldDB1`, and the `databaseName`, `oldDB`. This password is in clear text.

A simple data source connection does not generally require vendor-specific property settings. If you need them, add `config-property` tags as shown in the earlier examples.

Example 2.5 Configuring a Simple Data Source Connection in cache.xml

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC
      "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<cache lock-lease="120" lock-timeout="60" search-timeout="300">
   <region name="myRegion">
      <region-attributes refid="REPLICATE" initial-capacity="16"
             load-factor="0.75" concurrency-level="16"
             statistics-enabled="true">
      . . .
      </region-attributes>
   </region>
   <jndi-bindings>
      <jndi-binding
         type="SimpleDataSource" jndi-name="oldDB1"
         jdbc-driver-class="org.apache.derby.jdbc.EmbeddedDriver"
         user-name="mitul"
         password="password"
         connection-url="jdbc:derby:newDB;create=true"
      </jndi-binding>
   </jndi-bindings>
</cache>
```

For more information on defining a cache structure in an XML file, see Appendix B, *cache.xml File*.

Encrypting Connection Passwords

The `cache.xml` file accepts passwords in clear text or encrypted text.

To generate an encrypted password, use the `gemfire encrypt-password` command. This shows a sample command invocation and output (assuming *`password`* is the actual password for the datasource):

```
bash-2.05$ $GEMFIRE/bin/gemfire encrypt-password password
Using system directory "/home/users/jpearson/gemfire/defaultSystem".
Encrypted to 83f0069202c571faf1ae6c42b4ad46030e4e31c17409e19a
```

Copy the output from the `gemfire` command to the `cache.xml` file as the value of the password attribute of the jndi-binding tag embedded in `encrypted()`, just like a method parameter. Enter it as encrypted, in this format:

```
password="encrypted(83f0069202c571faf1ae6c42b4ad46030e4e31c17409e19a)"
```

To use a non-encrypted password, put the actual password as the value of the password attribute of the jndi-binding tag, like this:

```
password="password"
```

2.6 Handling Exceptions and System Failures

To see the exceptions thrown by specific methods, refer to the online Java documentation. To handle all the exceptions thrown by GemFire Enterprise, your application needs to catch these two classes:

- **GemFireCheckedException**—This class is the abstract superclass of exceptions that are thrown and declared. Wherever possible, GemFire exceptions are checked exceptions. `GemFireCheckedException` is a GemFire version of `java.lang.Exception`.
- **GemFireException**—This class is the abstract superclass of unchecked exceptions that are thrown to indicate conditions for which the developer should not normally need to check. You can look at the subclasses of `GemFireException` to see all the runtime exceptions in the GemFire system; see the Class Hierarchy in the online Java API documentation. `GemFireException` is a GemFire version of `java.lang.RuntimeException`. You can also look at the method details in the `Region` API javadocs for GemFire exceptions you may want to catch.

  A GemFire system member can also throw exceptions generated by third-party software such as JGroups or `java.lang` *classes. For assistance in handling these exceptions, see the vendor documentation.*

- **SystemFailure**—In addition to exception management, GemFire Enterprise provides a class to help you manage catastrophic failure in your distributed system, particularly in your JVM. The Javadocs for this class provide extensive guidance for managing failures in your system and your JVM. See `SystemFailure` in the `com.gemstone.gemfire` package.

2.7 Considerations for Your Host File Systems

Depending on your system configuration, your GemFire processes may consume a considerable amount of disk space. It is important that you work with your system administrator to make sure your files are in the proper locations on your host systems. This section describes the files your GemFire processes might produce.

Logging

These files contain information about system startup, operation, and shutdown. You can specify whether, how, and where these files are written with `gemfire.properties` file settings. Depending on the log level you choose, these files can grow very quickly. You can limit the size of each individual file and of all of the files for the process, past and current. The system deletes oldest files first to keep within the specified limits. For more information, see *GemFire System Logging* on page 185 of the *GemFire Enterprise System Administrator's Guide*.

Statistics

These files contain system and cache statistics for your processes. You can specify whether, how, and where these files are written with `gemfire.properties` file settings. Depending on the statistics sample rate, these files can grow quickly. You can limit the size of each individual file and of all of the files for the process, past and current. For more information, see *Configuring the System* on page 41 of the *GemFire Enterprise System Administrator's Guide*. The system deletes oldest files first to keep within the specified limits.

Data Overflow and Persistence

Overflow and persistence are stored to disk in GemFire disk stores. Make sure you have enough disk space for everything you intend to store to disk. For information about the file specifications, and about specifying overflow and persistence for your regions and queues, see *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.

2.8 About the Programming Examples

The GemFire Enterprise programming and configuration examples are available in a `zip` download. Instructions for downloading and running the examples are provided on the GemStone Wiki at https://community.gemstone.com/display/gemfire/Getting+Started..

Chapter 3

GemFire Members and Member Caches

GemFire Enterprise® is well-suited to a wide range of caching scenarios, from simple data caching within a single member to multi-platform, multi-cache systems composed of many diverse members. Many configuration options in GemFire have common-sense default settings. You can configure applications through static file declarations, command-line options, and programming API calls. To aid in system tuning decisions, you can use the GemFire Enterprise logging and statistics to create your own custom logging and statistics.

The GemFire cache provides the framework for connecting to the distributed system and storing, managing, and distributing your application data. Whether you are defining a simple peer-to-peer system, a client/server system, or creating a multisite system spread out over several geographic locations, cache configuration, along with cached data region configuration, defines how your data is stored and distributed.

The configuration properties for GemFire Enterprise are divided into two main categories, distributed system and cache. These properties cover virtually every aspect of distributed system and caching behavior.

In this chapter:

- Distributed System Member Configuration (page 70)
- Cache Management and Functionality (page 72)
- Configuring the Cache (page 74)
- Creating and Closing a Cache (page 78)

3.1 Distributed System Member Configuration

Distributed system members are programs that connect to a GemFire distributed system. These programs are also just called members. The connection to the system is done implicitly, when the member creates its GemFire data cache.

You usually configure distributed system membership in the `gemfire.properties` file. You can also provide configuration at the command line and through the API. The properties in the file are referred to as GemFire properties. For full information on the `gemfire.properties` file and the properties specifications, see *Configuring GemFire System Properties* on page 45 of the *GemFire Enterprise System Administrator's Guide*.

GemFire properties specify all necessary information for system member startup, initialization, and communication. You can't change the properties while the system is running.

Use the properties to define:

- How to find and communicate with other system members
- How to perform logging and statistics activities
- Where to find the license file for the system
- What `cache.xml` file to use for cache initialization
- Other options, including event conflation, how to handle network loss, and security settings.

Distributed System Membership and System Topologies

Every GemFire process is a member of a distributed system, even if the distributed system is defined as standalone, with just one member. You can run an individual distributed system in isolation or you can combine systems for vertical and horizontal scaling.

Peer-to-Peer Distributed Systems

Members that define the same member discovery properties belong to the same distributed system and are peers to one another. To configure distributed system discovery and membership, see *Configuring Member Discovery and Communication* on page 59 of the *GemFire Enterprise System Administrator's Guide*.

Client/Server Installations

The client/server topology uses relationships that you configure between members of multiple distributed systems. You configure some or all of the peers in one distributed system to act as cache servers to clients connecting from outside the system. Each server can host many client processes, managing cache access for all in an efficient, vertically hierarchical cache configuration. You configure the client applications to connect to the servers, using a client cache configuration. Clients run as members of standalone GemFire distributed systems, with no peers, so all data updates and requests go to the servers. For information on client/server systems, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.

Multisite Installations

The multi-site topology uses relationships that you configure between members of multiple distributed systems. Through this configuration, you loosely couple two or more distributed systems for automated data distribution. This is usually done for sites at geographically separate locations. You program a subset of peers in each distributed system site as gateways to the other distributed system sites. For information on multi-site configurations, see Chapter 10, *Configuring Multi-site Installations*, on page 267.

Local and Remote Caching and Membership

Discussions of GemFire membership and caching activities often differentiate between local and remote. Local caching always refers to the central member under discussion, if there is one such obvious member, and remote refers to other members. If there is no clear, single local member, the discussion assigns names to the members to differentiate. Operations, data, configuration, etc., that are “local to member Q” are running or resident inside the member Q process. Operations, data, configuration, and so on, that are “remote to member Q” are running or resident inside some other member.

The local cache is the cache belonging to the local member. All other caches are remote, whether in other members of the same distributed system or in different distributed systems.

In the context of a single distributed system, unless otherwise specified, remote refers to other members of the same distributed system. In client/server and multisite installations, remote generally refers to members in other distributed systems. For example, all servers are remote to the clients that connect to them. Each client runs standalone, with connections only to the server tier, so all servers and their other clients are remote to the individual client. All gateway hubs are remote to the gateways that connect to them from other distributed systems and to those gateways’ peers.

3.2 Cache Management and Functionality

The GemFire cache is the entry point to GemFire caching management. The cache provides in-memory storage and management for your data.

- You organize your data in the cache into *data regions*, each with its own configurable behavior.
- You store your data into your regions in key/value pairs called *data entries*.

The cache also provides features like transactions, data querying, disk storage management, and logging.

You configure caches generally using a combination of XML declarations and API calls. GemFire loads and processes your XML declarations when you first create the cache.

GemFire has one cache type for managing server and peer caches and one for one for managing client caches. The `cacheserver` process automatically creates its server cache at startup. In your application process, the cache creation returns an instance of the server/peer or client cache. From that point on, you manage the cache through API calls in your application.

Caching APIs

com.gemstone.gemfire.cache

- **`RegionService`**—Generally, you use the `RegionService` functionality through instances of `Cache` and `ClientCache`. You only specifically use instances of `RegionService` for limited-access users in secure client applications that service many users. See *Creating and Closing RegionService Instances in a Client with Multiple Secure Users* on page 82.

  The `RegionService` API provides:

  - Access to existing cache data regions.
  - Access to the standard query service for the cache. For client caches, queries are sent to the server tier. For server and peer caches, queries are run in the current cache and any available peers.

  `RegionService` is implemented by `GemFireCache`, which is implemented by `Cache` and `ClientCache`.

- **`GemFireCache`**—You do not specifically use instances of `GemFireCache`, but you use `GemFireCache` functionality in the instances of `Cache` and `ClientCache`.

  `GemFireCache` extends `RegionService` and adds these general caching features:

  - Region attributes.
  - Disk stores for region persistence and overflow.
  - Cache read behavior.
  - Logging.
  - Resource manager for memory use control.
  - Access to the underlying distributed system.

  `GemFireCache` is implemented by `Cache` and `ClientCache`.

- **Cache**—Use the `Cache` interface to manage server and peer caches. You have one `Cache` per server or peer process.

  The `Cache` extends `GemFireCache` and adds these server/peer caching features:

  - General communication between the local cache and remote caches in the same distributed system.
  - Region creation.
  - Transactions.
  - Locking for global regions.
  - Querying.
  - Cache servers. Service clients connecting to the servers' distributed system.
  - Multi-site gateway hubs. These are used to distribute events between distributed systems.

com.gemstone.gemfire.cache.client

- **`ClientCache`**—Use the `ClientCache` interface to manage the cache in your clients. You have one `ClientCache` per client process.

  The `ClientCache` extends `GemFireCache` and adds these client-specific caching features:

  - Client region creation.
  - Subscription keep-alive management for durable clients.
  - Querying on server and client tiers.
  - `RegionService` creation for secure access by multiple users.

3.3 Configuring the Cache

You need to configure the cache of each client, server, and peer before it can participate in GemFire's distributed data caching.

You configure a cache using:

- Declarations in the cache configuration file, `cache.xml`
- The GemFire API

For your applications, you can use the XML file and API calls alone or in combination. Exactly how you configure your cache is up to you, and depends on what your applications need and what is easiest for you to write and maintain:

The GemFire `cacheserver` process requires a `cache.xml` file, because the process is not programmable.

The `cache.xml` file has two main elements, `<cache>` and `<client-cache>`. You use the `<cache>` element to configure the `Cache` instance in servers and peers. Use the `<client-cache>` element to configure the `ClientCache` instance in client processes. See *Cache Management and Functionality* on page 72.

The API uses the same attributes as the `cache.xml` file, with the settings managed through `GemFireCache`, `Cache`, and `ClientCache`, depending on their functional area. Attribute and element names correspond between XML and API. For example, `copy-on-read` in the XML corresponds to `setCopyOnRead` in the `GemFireCache` API, and `<region>` in the XML corresponds to the `createRegionFactory` method in the `Cache` API and the `createClientRegionFactory` method in the `ClientCache` API. In this manual, the attributes are generally referred to by the names in the `cache.xml` file.

For a detailed list of all cache configuration properties, see Appendix B, *cache.xml File*, on page 499.

The next two tables describe the attributes and sub-elements of the `<cache>` and `<client-cache>` elements. All attributes and sub-elements belong to `<cache>`. The ones that also belong to `<client-cache>` are marked.

Table 3.1 Cache Configuration (cache.xml) Attributes

<cache> attribute	Also under <client-cache>	Description	Default
copy-on-read	**X**	Whether entry value retrieval methods return direct references to the entry value objects (`false`) or copies of the objects (`true`). See *Safe Entry Modification* on page 140.	`false`
is-server		Whether this member is a cache server. Cache servers are described in the *Configuring and Running the GemFire Cache Server* on page 126 of the *GemFire Enterprise System Administrator's Guide*.	`false`
lock-timeout		Timeout, in seconds, for implicit object lock requests. The setting affects automatic locking only, and does not apply to manual locking. If a lock request does not return before the specified timeout period, it is canceled and returns with a failure. See *Locking* on page 205 and *Setting Cache Timeouts* on page 471.	`60`

lock-lease		Timeout, in seconds, for implicit and explicit object lock leases. The setting affects both automatic locking and manual locking. Once a lock is obtained, it may remain in force for the lock lease time period before being automatically cleared by the system. See *Locking* on page 205 and *Setting Cache Timeouts* on page 471.	120
message-sync-interval		Frequency (in seconds) at which the primary server sends messages to its secondary servers to remove queued events that have already been dispatched to the clients. Used for client subscription queue synchronization when the member acts as a server to clients. See *Highly Available Subscription Delivery* on page 241.	1
search-timeout		Number of seconds a `netSearch` operation waits for data before timing out. Change this setting according to network load or other factors. `netSearch` operation is described in *Accessing an Entry* on page 135.	300

Table 3.2 Cache Configuration (cache.xml) Sub-elements

<cache> element	Also under <client-cache>	Description	Default
cache-server		Configures the cache to serve data to clients. See Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.	None
cache-transaction-manager		Manages GemFire cache transactions and manages the installation of the transaction writer and listeners for the cache. See Chapter 15, *GemFire Transactions*, on page 409.	None
disk-store	**X**	Defines a pool of one or more disk stores, which can be used by regions, client subscription queues, and WAN gateway queues. The cache default disk store is used when no disk store is named. See *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.	cache default disk store
dynamic-region-factory	**X**	Configures the cache for dynamic region management. When this setting is enabled, this cache can distribute dynamic region creations and destructions to other caches with a factory defined and can receive modifications to dynamic regions from other caches. See *Dynamic Region Management* on page 122.	None
function-service	**X**	Configures the behavior of the function execution service. See Chapter 16, *Function Execution*, on page 437.	None
gateway-hub		Makes this process a hub of a WAN gateway. For details on caching across a WAN, see Chapter 10, *Configuring Multi-site Installations*, on page 267.	None
jndi-bindings		The binding for a data-source. See *Configuring Database Connections Using JNDI* on page 59.	None
pool	**X**	Defines a pool used to communicate with server VMs in a different distributed system. See Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.	None

<cache> element	Also under <client-cache>	Description	Default
`region`	**X**	A data region in the cache. See *Data Regions* on page 85.	None
`region-attributes`	**X**	A region configuration that can be stored, by `id`, and referenced later, by `refid`, in the `cache.xml` file and through the API. See *Region Shortcuts and Custom Named Region Attributes* on page 97.	None
`resource-manager`	**X**	A memory monitor that tracks cache size as a percentage of total tenured heap and controls size by restricting access to the cache and prompting eviction of old entries from the cache. Used in conjunction with settings for VM memory and Java garbage collection. See *Controlling Heap Use With the Resource Manager* on page 186	None
`serialization-registration`	**X**	Set of serializer or instantiator tags to register customer `DataSerializer` extensions or `DataSerializable` implementations respectively. See *Data Serialization* on page 470.	None

Cache XML Declaration File Requirements

Your `cache.xml` must be formatted according to the dtd listed in *GemFire Enterprise Cache XML DTD on page 505*. These are the two high-level XML file formats:

Example 3.1 cache.xml (Peer/Server)

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">

<cache>
    ...
</cache>
```

Example 3.2 cache.xml (Client)

```
<?xml version="1.0"?>
<!DOCTYPE client-cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">

<client-cache>
    ...
</client-cache>
```

3.4 Creating and Closing a Cache

Your system configuration and cache configuration are initialized when you start your member processes and create each member's GemFire cache.

The steps in this section use `gemfire.properties` and `cache.xml` file examples, except where API is required. You can configure your distributed system properties and cache through the API as well, and you can use a combination of file configuration and API configuration.

The XML examples listings may not include the full `cache.xml` file listing. All of your declarative cache configuration must conform to the DTD listed in Appendix B, *cache.xml File*, on page 499.

For all of your GemFire applications:

1. Create your `Cache`, for peer/server applications, or `ClientCache`, for client applications. This connects to the GemFire system you have configured and initializes any configured data regions.

   Use your cache instance to access your regions and perform your application work.

2. Close your cache when you are done. This frees up resources and disconnects your application from the distributed system in an orderly manner.

The rest of this section provides instructions for creating and closing your cache for specific GemFire application types.

You may need to combine more than one of these sets of instructions. For example, to create a client cache in a system with security, follow the instructions for Creating and Closing a Client Cache (page 80) and Creating and Closing a Cache in a Secure System (page 81).

These are the instructions for specific application types:

- Creating and Closing a Server or Peer Cache (page 79)
- Creating and Closing a Client Cache (page 80)
- Creating and Closing a Cache in a Secure System (page 81)
- Creating and Closing RegionService Instances in a Client with Multiple Secure Users (page 82)
- Changing gemfire.properties in the Cache Creation Call (page 84)

Creating and Closing a Server or Peer Cache

GemFire peers are members of a GemFire distributed system that do not act as clients to another GemFire distributed system. GemFire servers are peers that also listen for and process client requests.

> *For information on options for client/server configuration, see* *Client/Server Architecture and Configuration Basics* on page 201.

1. Create your cache:

   1.1 In your `cache.xml`, use the `cache DOCTYPE` and configure your cache inside a `<cache>` element. Example:

```
<!DOCTYPE cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<cache>
    // Use the <cache-server> element only for server processes
    <cache-server port="40404"/>

    <region name="customerRegion" refid="REPLICATE" />
    <region name="ordersRegion" refid="PARTITIONED" />
</cache>
```

   1.2 Create the `Cache` instance:

   - In your Java application, use the `CacheFactory create` method:

     ```
     Cache cache = new CacheFactory().create();
     ```

   - The `cacheserver` process automatically creates the cache and connection at startup and closes both when it exits.

     See *The cacheserver Command-Line Utility* on page 127 of the *GemFire Enterprise System Administrator's Guide*.

   The system creates the distributed system connection and initializes the cache according to your `gemfire.properties` and `cache.xml` specifications.

2. Close your cache when you are done using the `close` method of your `Cache` instance:

```
cache.close();
```

Creating and Closing a Client Cache

GemFire clients are processes that send most or all of their data requests and updates to a GemFire server system. Clients run as standalone processes, without peers of their own.

For information on options for client/server configuration, see *Client/Server Architecture and Configuration Basics* on page 201.

> *GemFire automatically configures the distributed system for your* `ClientCache` *as standalone, which means the client has no peers. Do not try to set the* `gemfire.properties` `mcast-port` *or* `locators` *for a client application or the system will throw an exception. See* *Standalone Members* on page 67 of the *GemFire Enterprise System Administrator's Guide*.

1. Create your client cache:

   1.1 In your `cache.xml`, use the `client-cache` `DOCTYPE` and configure your cache inside a `<client-cache>` element. Configure your server connection pool and your regions as needed. Example:

```
<!DOCTYPE client-cache PUBLIC
 "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<client-cache>
   <pool name="serverPool">
      <locator host="host1" port="44444"/>
   </pool>
   <region name="exampleRegion" refid="PROXY"/>
</client-cache>
```

   1.2 If you use multiple server pools, configure the pool explicitly for each client region. Example:

```
<pool name="svrPool1">
   <locator host="host1" port="40404"/>
</pool>
<pool name="svrPool2">
   <locator host="host2" port="40404"/>
</pool>
<region name="clientR1" refid="PROXY" pool-name="svrPool1"/>
<region name="clientR2" refid="PROXY" pool-name="svrPool2"/>
<region name="clientsPrivateR" refid="LOCAL"/>
```

   1.3 In your client application, use the `ClientCacheFactory` `create` method:

```
ClientCache clientCache = new ClientCacheFactory().create();
```

   This creates the server connections and initializes the client's cache according to your `gemfire.properties` and `cache.xml` specifications.

2. Close your cache using the `close` method of your `ClientCache` instance. Example:

```
clientCache.close();
```

   If your client is durable and you want to maintain your durable queues while the client cache is closed, use:

```
clientCache.close(true);
```

   See Durable Subscription Queues on page 246.

Creating and Closing a Cache in a Secure System

This section applies to peers, servers, and clients that are joining a secure system.

When you create your cache in a secure system, you provide credentials to the connection process, for authentication by already-running, secure members. For example, peers can be authenticated by peers that are already members of the system and clients can be authenticated by servers to operate in the secure client/server system.

Follow these steps in addition to the steps in the appropriate section for your process type: either *Creating and Closing a Server or Peer Cache* on page 79 or *Creating and Closing a Client Cache* on page 80.

1. To create your cache.

   1.1 Add any necessary security properties to the `gemfire.properties`, to configure GemFire for your particular security implementation.

   Examples:

   ```
   security-client-auth-init=mySecurity.UserPasswordAuthInit.create

   security-peer-auth-init=myAuthPkg.myAuthInitImpl.create
   ```

   See Security on page 73 of the *GemFire Enterprise System Administrator's Guide*.

   1.2 When you create your cache, pass any properties required by your security implementation to the cache factory `create` call by using one of these methods:

   - `ClientCacheFactory` or `CacheFactory` `set` methods. Example:

     ```
     ClientCache clientCache = new ClientCacheFactory()
         .set("security-username", username)
         .set("security-password", password)
         .create();
     ```

   - `Properties` object passed to the `ClientCacheFactory` or `CacheFactory` `create` method. Example:

     ```
     Properties properties = new Properties();
     properties.setProperty("security-username", username);
     properties.setProperty("security-password", password);
     Cache cache = new CacheFactory(properties).create();
     ```

     These are usually properties of a sensitive nature that you do not want to put inside the `gemfire.properties` file.

     Properties passed to a cache creation method override any settings in `gemfire.properties`.

2. Close your cache using the `close` method of your `Cache` or `ClientCache` instance. Example:

   ```
   cache.close();
   ```

Creating and Closing RegionService Instances in a Client with Multiple Secure Users

You can create multiple, secure connections to your servers from a single client. The most common use case is a GemFire client embedded in an application server that supports data requests from many users. Each user may be authorized to access a subset of data on the servers. For example, customer users may be allowed to see and update only their own orders and shipments.

The authenticated users all access the same `ClientCache` through instances of the `RegionService` interface. Because there are multiple users, with varying authorization levels, access to cached data is done through the servers, where each user's authorization levels can be managed.

> *See Security on page 73 of the GemFire Enterprise System Administrator's Guide.*

Follow these steps in addition to the steps for the `ClientCache` in *Creating and Closing a Cache in a Secure System* on page 81:

1. Create your cache and `RegionService` instances:

   1.1 Configure your client's server pool for multiple secure user authentication. Example:

   ```
   <pool name="serverPool" multiuser-authentication="true">
      <locator host="host1" port="44444"/>
   </pool>
   ```

   This enables access through the pool for the `RegionService` instances and disables it for the `ClientCache` instance.

   1.2 After you create your `ClientCache`, from your `ClientCache` instance, for each user call the `createAuthenticatedView` method, providing the user's particular credentials. These are `create` method calls for two users:

   ```
   Properties properties = new Properties();
   properties.setProperty("security-username", cust1Name);
   properties.setProperty("security-password", cust1Pwd);
   RegionService regionService1 =
      clientCache.createAuthenticatedView(properties);

   properties = new Properties();
   properties.setProperty("security-username", cust2Name);
   properties.setProperty("security-password", cust2Pwd);
   RegionService regionService2 =
      clientCache.createAuthenticatedView(properties);
   ```

   For each user, do all of your caching and region work through the assigned `RegionService` instance. Access to the server cache will be governed by the server's configured authorization rules for each individual user.

2. Close your cache by closing the `ClientCache` instance as described in *Creating and Closing a Client Cache* on page 80. Do not close the `RegionService` instances first. This is especially important for durable clients.

Requirements and Caveats for RegionService

For each region, you can perform operations through the `ClientCache` instance or the `RegionService` instances, but not both.

> *You can use the* `ClientCache` *to create a region that uses a pool configured for multi-user authentication, then access and do work on the region using your* `RegionService` *instances.*

To use `RegionService`:

- Regions must be configured as `EMPTY`. Depending on your data access requirements, this configuration might affect performance, because the client goes to the server for every `get`.
- If you are running durable client queues (CQs) from the `RegionService` instances, stop and start the offline event storage for the client as a whole. The server manages one queue for the entire client process, so you need to request the stop and start of durable CQ event messaging for the cache as a whole, through the `ClientCache` instance. If you closed the `RegionService` instances, event processing would stop, but the events from the server would continue, and would be lost.

  Stop with:

  - `clientCache.close(true);`

  Start up again in this order:

  a. Create `ClientCache` instance.

  b. Create all `RegionService` instances. Initialize CQ listeners.

  c. Call `ClientCache` instance `readyForEvents` method.

Changing gemfire.properties in the Cache Creation Call

Check with your GemFire system administrator before changing distributed system properties through the API, including the `gemfire.properties` *file specification and license files settings. The system administrator may need to set distributed system properties at the command line or in configuration files. Changes you make through the API override the other settings.*

When you create your cache, you can change `gemfire.properties` settings through the `set` method in the cache factories.

The example code in *Creating and Closing a Cache in a Secure System* on page 81 uses the `set` method to provide the properties, `security-username` and `security-password`.

This call specifies a log file and `cache.xml`:

Example 3.3 Server and Peer Cache Creation with Config File Specification

```
Cache cache = new CacheFactory()
    .set("log-file", "./test2/myLog")
    .set("cache-xml-file", "myCache.xml")
    .create();
```

This call specifies a `name` identifier for the process:

Example 3.4 Client Cache Creation with Modified Properties

```
ClientCache clientCache1= new ClientCacheFactory()
    .set("name", "myClientProcess")
    .create();
```

Chapter 4

Data Regions

A GemFire data region is a logical grouping within a cache for a single data set. You can define any number of regions within your cache. Each region has its own configurable settings governing such things as the data storage model, local data storage and management, data event distribution, and data persistence.

This chapter covers the basics of GemFire data region management, including your general options for region configuration and how to create and remove regions from your cache. It also includes the full listing of region attributes settings.

Each of the main region configuration types has its own specific options and features for configuration and management. For detailed information on those, see Chapter 6, *Configuration and Management by Region Type*, on page 145.

All region types use the same configuration and management of features like disk persistence, eviction and expiration, event handling, data loading from an outside data source, and memory management. For detailed information on these common features, see Chapter 7, *General Region Management Options*, on page 169.

In this chapter:

- Region Overview (page 86)
- Region Types (page 87)
- Data Hosts and Data Accessors (page 90)
- Region API (page 91)
- Creating and Managing Data Regions (page 92)
- Region Shortcuts and Custom Named Region Attributes (page 97)
- Region Attributes List (page 101)
- Creating Custom User Attributes for Regions and Entries (page 120)
- Dynamic Region Management (page 122)

4.1 Region Overview

Within a cache, you use regions to group data and configure data management.

- You store data in your regions in key/value pairs, with the values being any object type your application needs. You do all of your data puts, gets, and querying activities against your data regions. The region interface, `Region`, implements `java.util.Map` for easy and familiar access and management.
- You configure your region to customize how your data is stored, distributed, and managed in the cache.

The following list shows the main features that you can configure for your region. (Some features do not apply to client regions.) For the full list, see *Region Attributes List* on page 101. Also see the online Java documentation for `RegionFactory` and `ClientRegionFactory`.

- Overall storage and distribution model of the region
- Object type constraints for the region's entry keys and values
- Partitioning behavior, for partitioned regions
- Application plug-ins for data loading and event handling
- Data persistence and overflow
- Region and entry expiration
- Event subscriptions from peer caches
- Communication settings for peer-to-peer distribution
- Region participation in multi-site event distribution
- Connection as client region to server tier

4.2 Region Types

You configure the storage and distribution model for your region data when you create the region. The main region types are described in the table below and in the sections that follow. These region options define region behavior within a single distributed system. Within a distributed system, you can define distributed regions and non-distributed regions, and you can define regions whose data is spread across the distributed system, and regions whose data is entirely contained in a single member.

Your choice of region type is governed in part by the type of application you are running. In particular, you need to use specific region types for your servers and clients for effective communication between the two tiers:

- Server regions are created inside a `Cache` by servers and are accessed by clients that connect to the servers from outside the server's distributed system. Server regions must have region type partitioned or replicated.
- Client regions are created inside a `ClientCache` by clients and are configured to distribute data and events between the client and the server tier. Client regions must have region type local.
- Peer regions are created inside a `Cache`. Peer regions may be server regions, or they may be regions that are not accessed by clients. Peer regions can have any region type.

Table 4.1 Region Types.

Region type	Description	Best suited for...
Partitioned	System-wide setting for the data set. Data is divided into buckets across the members that define the region. For high availability, configure redundant copies, so that each data bucket is stored in more than one member, with one member holding the primary.	Server regions or peer regions ▸ Very large data sets ▸ High availability ▸ Write performance ▸ Partitioned event listeners and data loaders
Replicated (distributed)	Holds all data from the distributed region. The data from the distributed region is copied into the member replica region. Can be mixed with non-replication, with some members holding replicas and some holding non-replicas.	Server regions or peer regions ▸ Read heavy, small datasets ▸ Asynchronous distribution ▸ Query performance
Distributed (not replicated)	Data is spread across the members that define the region. Each member holds only the data it has expressed interest in. Can be mixed with replication, with some members holding replicas and some holding non-replicas.	Peer regions, but not server regions and not client regions ▸ Asynchronous distribution ▸ Query performance
Local (not distributed)	The region is visible only to the defining member.	Client regions or peer regions ▸ Data that is not shared between applications

For information on using regions in client/server and multi-site (WAN) installations, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201 and Chapter 10, *Configuring Multi-site Installations*, on page 267.

Partitioned Regions

Partitioning is a good choice for very large server regions. Partitioned regions are ideal for data sets in the hundreds of gigabytes and beyond.

Use partitioning for:

- **Large data sets**. Store data sets that are too large to fit into a single member, and all members will see the same logical data set. Partitioned regions divide the data into units of storage called buckets that are split across the members hosting the partitioned region data, so no member needs to host all of the region's data. GemFire provides dynamic redundancy recovery and rebalancing of partitioned regions, making them the choice for large-scale data containers. More members in the system can accommodate more uniform balancing of the data across all host members, allowing system throughput (both `gets` and `puts`) to scale as new members are added.
- **High availability**. Partitioned regions allow you configure the number of copies of your data that GemFire should make. If a member fails your data will be available without interruption from the remaining members. Partitioned regions can also be persisted to disk for high availability.
- **Scalability**. Partitioned regions can scale to large amounts of data because the data is divided between the members available to host the region. Increase your data capacity dynamically by simply adding new members. You can also scale your processing capacity. Because your entries are spread out across the members hosting the region, reads and writes to those entries are also spread out across those members.
- **Good write performance**. You can configure the number of copies of your data. The amount of data transmitted per write does not increase with the number of members. By contrast, with replicated regions, each write must be sent to every member that has the region replicated, so the amount of data transmitted per write increases with the number of members.

You cannot control where your data resides with partitioned regions. You can collocate keys within buckets and across multiple partitioned regions, but you cannot control which members store which data buckets. If you need to control data location, use a replicated or distributed region.

For more information on partitioned regions, see *Partitioned Regions* on page 146.

Replicated Regions

Replication is a good choice for small to medium size server regions.

Replicated regions provide the highest performance in terms of throughput and latency.

Use replication for:

- **Small amounts of data required by all members of the distributed system**. For example, currency rate information and mortgage rates.
- **Data sets that can be contained entirely in a single VM**. Each replicated region holds the complete data set for the region.
- **High performance data access**. Replication guarantees local access from the heap for application threads, providing the lowest possible latency for data access.
- **Asynchronous distribution**. All distributed regions, replicated and non-replicated, provide the fastest distribution speeds.

For more information on replicated regions, see *Replicated Regions and Other Distributed Regions* on page 161.

Distributed, Non-Replicated Regions

Distributed regions provide the same performance as replicated regions, but each member only stores data it has expressed interest in, either by subscribing to events from other members or by defining the data entries in its cache.

Use distributed, non-replicated regions for:

- **Peer regions, but not server regions or client regions**. Server regions must be either replicated or partitioned. Client regions must be local.
- **Data sets where individual members only need notification and updates for changes to a subset of the data**. In non-replicated regions, each member only receives update events for the data entries it has defined in the local cache.
- **Asynchronous distribution**. All distributed regions, replicated and non-replicated, provide the fastest distribution speeds.

For more information on distributed regions, see *Replicated Regions and Other Distributed Regions* on page 161.

Local Regions

> *When created using the* `ClientRegionShortcut` *settings, client regions are automatically defined as local, since all client distribution activities go to and come from the server tier.*

The local region has no peer-to-peer distribution activity.

Use local regions for:

- **Client regions**. Distribution is only between the client and server tier.
- **Private data sets for the defining member**. The local region is not visible to peer members.

For more information on local regions, see *Local Regions* on page 168.

4.3 Data Hosts and Data Accessors

In most cases, when you define a data region in a member's cache, you also specify whether the member is a data host. Members that host data for a region are referred to as data hosts. Members that do not host data are referred to as accessor members, or empty members. Any member, host or accessor, that defines a region can access it, put data into it, and receive events from other members. To configure a region so the member is a data accessor, you use configurations that specify no local data storage for the region. Otherwise, the member is a data host for the region.

4.4 Region API

The GemFire Java region API allows you to manage your region and its attributes in detail. This section lists the core classes and interfaces. For detailed information on the API, start with the online Java API documentation for these:

Peer/Server

Use these methods, interfaces, and classes for peer/server region creation. These correspond to declarations in the `<cache>` element for creating and configuring regions.

com.gemstone.gemfire.cache

- **`Cache.createRegionFactory`**. This method takes a `RegionShortcut` enum, to initiate region configuration, and returns a `RegionFactory`.
- **`RegionFactory`**. Provides methods to further configure the region before the `create` call, which returns `Region`.
- **`RegionShortcut`**. Common region configurations. Can be retrieved through `Cache createRegionFactory` and with the region attribute, `refid`.

Client

Use these methods, interfaces, and classes for client region creation. They correspond to declarations in the `<client-cache>` element for creating and configuring regions.

com.gemstone.gemfire.cache.client

These are client versions of the Peer/Server API listed above for the `com.gemstone.gemfire.cache` package. These provide similar functionality, but are tailored to the needs and behaviors of client regions.

- **`ClientCache.createClientRegionFactory`**. This method takes a `ClientRegionShortcut` enum, to initiate region configuration, and returns a `ClientRegionFactory`.
- **`ClientRegionFactory`**. Provides methods to further configure the region before the `create` call, which returns `Region`.
- **`ClientRegionShortcut`**. Common region configurations. Can be retrieved via `ClientCache createClientRegionFactory` and with the region attribute, `refid`.

Peer/Server/Client

These interfaces and classes are used universally for region management.

com.gemstone.gemfire.cache Package

- **`Region`**. Interface for managing your regions and their entries.
- **`RegionAttributes`**. Object holding region configuration settings.
- **`AttributesFactory`**. Can be used to create `RegionAttributes` to pass to `RegionFactory`.

4.5 Creating and Managing Data Regions

In this section:

- Creating a Region (page 92)
- Invalidating a Region (page 94)
- Clearing a Region (page 95)
- Destroying a Region (page 95)
- Closing a Region (page 96)
- Determining a Region's Status (page 96)
- Tracking Instances of a Distributed Region (page 96)

Creating a Region

You create each data region with `cache.xml` declarations or through `RegionFactory`. The API and the `cache.xml` file use slightly different names for many of the attributes and attribute settings. In this manual, the attributes are called by the names in the `cache.xml` file. For the API names, see *Region Attributes List* on page 101.

To create a region:

1. Start with your region's name and a region attributes shortcut setting, if one applies—find the one that fits your needs most closely. See *Shortcut Attribute Options* on page 97.

   This shortcut setting defines a replicated region in the `cache.xml`:

   ```
   <region name="myRegion" refid="REPLICATE"/>
   ```

2. Add other attributes as needed to customize the region's behavior. See *Region Attributes List* on page 101.

   This adds a cache events listener to a replicated region:

   ```
   <region name="myRegion" refid="REPLICATE">
      <region-attributes>
         <cache-listener>
            <class-name>myPackage.MyCacheListener</class-name>
         </cache-listener>
      </region-attributes>
   </region>
   ```

Region creation is subject to attribute consistency checks, both internal to the cache and, if the region is not local in scope, between all caches where the region is defined. The requirements for consistency between attributes are detailed in the online Java API documentation.

You can change some attributes after creation, if necessary, with `com.gemstone.gemfire.cache.AttributesMutator` obtained from the `Region`. The attributes that take application plug-ins and some attributes governing expiration and eviction are mutable. Only do this if necessary, as it is slower than setting them as you need them from the start. For information, see the online Java documentation.

Region Naming Conventions

The safest approach when naming your regions is to use only alphanumeric characters and the underscore character and to keep your region names short. This should permit you the full range of GemFire region configurations and allow you to perform all available GemFire operations on the region.

- Do not use the separator slash character '/'.
- Do not begin region names with two underscore characters "__". This is reserved for internal GemFire use.

Creating a Region Through the cache.xml

In the `cache.xml`, declare your region as a subelement of the `<cache>` or `<client-cache>` element. All attribute settings you do not provide are set to the defaults.

The examples listed here are partial `cache.xml` file contents. The file must conform to the DTD in *productdir*/dtd/cache6_5.dtd (listed in *GemFire Enterprise Cache XML DTD* on page 505).

Example 4.1 Creating a Default Partitioned Region Through the cache.xml

```
<region name="partition_reg4" refid="PARTITION"/>
```

Example 4.2 Creating a Default Persistent Partitioned Region Through the cache.xml

```
<region name="testPR" refid="PARTITION_PERSISTENT"/>
```

Example 4.3 Creating a Partitioned Region Through the cache.xml

```
<region name="partition_reg2" refid="PARTITION_REDUNDANT">
   <region-attributes>
      <partition-attributes local-max-memory="512" />
   </region-attributes>
</region>
```

Example 4.4 Creating a Default Replicated Region Through the cache.xml

```
<region name="partition_reg4" refid="REPLICATE"/>
```

Example 4.5 Creating a Replicated Region with Expiration Through the cache.xml

```
<region name="myRegion" refid="REPLICATE">
   <region-attributes statistics-enabled="true">
      <entry-time-to-live>
         <expiration-attributes timeout="60" action="destroy"/>
      </entry-time-to-live>
   </region-attributes>
</region>
```

Example 4.6 Creating a Client Region Through the cache.xml

```
<region name="clientRegion" refid="CACHING_PROXY"/>
```

Example 4.7 Creating a Global Distributed Region Through the cache.xml

```
<region name="A">
   <region-attributes scope="global"/>
</region>
```

Creating a Region Using RegionFactory

In your code, create your regions in peers and servers using the `RegionFactory` class in the `com.gemstone.gemfire.cache` package and in clients using the `ClientRegionFactory` class in the `com.gemstone.gemfire.cache.client` package.

The region creation creates the region according to your specifications and, if it does not already exist, also creates your cache.

Example 4.8 Creating a Partitioned Region Using RegionFactory

```
RegionFactory rf =
   cache.createRegionFactory(RegionShortcut.PARTITION);
rf.addCacheListener(new LoggingCacheListener());
custRegion = rf.create("customer");
```

Example 4.9 Creating a Modified Partitioned Region Using RegionFactory

```
PartitionAttributesFactory paf = new PartitionAttributesFactory<CustomerId,
String>();
paf.setPartitionResolver(new CustomerOrderResolver());

RegionFactory rf =
   cache.createRegionFactory(RegionShortcut.PARTITION);
rf.setPartitionAttributes(paf.create());
rf.addCacheListener(new LoggingCacheListener());
custRegion = rf.create("customer");
```

Example 4.10 Creating a Client Region with a Pool Specification Using ClientRegionFactory

```
ClientRegionFactory<String,String> cRegionFactory =
   cache.createClientRegionFactory(PROXY);
Region<String, String> region =
   cRegionFactory.setPoolName("Pool3").create("DATA");
```

Accessing a Region

The APIs provide methods to retrieve information such as the region's name, its assigned membership roles, and the list of all regions in the cache. See the online Java documentation for `Cache` and `ClientCache`. A region can be accessed from the cache using its name:

```
this.currRegion = this.cache.getRegion("regionName");
```

Invalidating a Region

Region invalidation does not work with partitioned regions.

You can invalidate a region directly in the local cache through API calls and through automated expiration activities. In both cases, you can choose whether the invalidation is a local or a distributed operation:

- **Distributed invalidate**. Marks all entries in the region as invalid. Distributes the invalidation according to the region's scope. This is the proper choice when the region is no longer valid for any application in the distributed system.
- **Local invalidate**. Marks all entries in the region as invalid but does not distribute the operation. Local region invalidation is only supported for regions that are not configured as replicas. For a discussion of this, see *Replication and Local Destroy and Invalidate Operations* on page 200.

For information specific to entries, see *Invalidating and Destroying an Entry* on page 143.

Example 4.11 Local and Distributed Invalidation of Regions

```
/* Distributed Region Invalidation */
   thisRegion.invalidateRegion();
/* Local Region Invalidation */
   thisRegion.localInvalidateRegion();
```

Clearing a Region

Partitioned regions cannot be cleared.

The `clear` method removes all region entries, leaving the region empty. If the region `scope` (page 116) is not `local`, the clear is distributed throughout the distributed cache.

Destroying a Region

Region destruction can be performed directly in the local cache through API calls and through automated expiration activities. In either case, destruction can be performed as a local or a distributed operation:

- **Distributed destroy**. Removes the region completely from the cache. All entries are destroyed as well. Destroy actions are distributed according to the region's distribution settings. Use this option when the region is no longer needed for any application in the distributed system.
- **Local destroy**. Removes the region from the local cache, but does not distribute the removal operation to remote members.

For information specific to entries, see *Invalidating and Destroying an Entry* on page 143.

Example 4.12 Local and Distributed Destruction of Regions

```
/* Distributed Region Destruction */
   thisRegion.destroyRegion();

/* Local Region Destruction */
   thisRegion.localDestroyRegion();
```

Closing a Region

The `Region.close` operation works similarly to the `Region.localDestroyRegion` operation with this significant difference:

> *The* `close` *method is called for every callback installed on the region. No events are invoked. Most notably, the entry events,* `beforeDestroy` *and* `afterDestroy`*, and the region events,* `beforeRegionDestroy` *and* `afterRegionDestroy`*, are not invoked. See* Chapter 11, *Handling Events*, on page 293 *for more information about events.*

Closing a region removes it from the cache. Use this method with persistent regions and with all partitioned regions.

If the data is persisted, the data remains on disk. So this is a way of retaining the disk copies for the region without closing the entire cache. You can use the data set from these files to initialize an instance of the region at a later time.

- For persistent replicas, all region data remains on disk.
- For persistent partitioned regions, all local bucket data remains on disk.

> *With partitioned regions, use the* `close` *operation carefully, particularly if no redundancy or persistence is configured.*

Determining a Region's Status

Use the `Region isDestroyed` method to determine whether a region has been destroyed in the local cache. Similarly, the `isEmpty` method returns `true` if the region contains no entries, as would be the case immediately after a region is cleared.

Determine whether a region is invalid in the local cache by iterating through all entries in the region. For distributed regions, use `Region.Entry.getValue` so that you access the local value only.

In partitioned regions, `Region.Entry.getValue` returns a valid value found anywhere in the region. Although you cannot fetch only local values, you can identify which keys are local by using `Region.Entry.isLocal` along with `getValue`. Because partitioned regions can be very large, iterating over the entire entry set to discover what is held in primary storage in the local cache can be expensive. See the online Java documentation for `Region.Entry`.

Tracking Instances of a Distributed Region

To track which members host a distributed region, you can implement a region membership listener. When the listener is installed in a region, it receives notification when a region of the same name has been created in another member or when another member using the region joins or leaves the distributed system. See *RegionMembershipListener* on page 305.

4.6 Region Shortcuts and Custom Named Region Attributes

GemFire provides a number of predefined, shortcut region attributes settings for your use. You can also create custom region attributes and store them with an identifier for later retrieval. Both types of stored attributes are referred to as named region attributes. You can create and store your attribute settings in the `cache.xml` file and through the API.

Retrieve named attributes by providing the ID to the region creation. This example uses the shortcut `REPLICATE` attributes to create a region:

Example 4.13 Using a Region Shortcut to Define a Region

```
<region name="testREP" refid="REPLICATE"/>
```

You can modify named attributes as needed. For example, the following region declaration retrieves the persistent partitioned region shortcut and sets the disk store name used for persistence:

Example 4.14 Using a Region Shortcut with Overrides

```
<disk-store name="testDiskStore" >
   <disk-dirs>
      <disk-dir>PRPersist1</disk-dir>
      <disk-dir>PRPersist2</disk-dir>
   </disk-dirs>
</disk-store>
<region name="testPR" >
   <region-attributes
      refid="PARTITION_PERSISTENT" disk-store-name="testDiskStore"/>
</region>
```

Shortcut Attribute Options

You can select the most common region attributes settings from GemFire's predefined named region attributes in these classes:

- For peers and servers: `com.gemstone.gemfire.cache.RegionShortcut`
- For clients: `com.gemstone.gemfire.cache.client.ClientRegionShortcut`

Shortcut attributes are a convenience only. They are just named attributes that GemFire has already stored for you. You can override their settings by storing new attributes with the same `id` as the predefined attributes.

> *For complete listings and descriptions, including information on the underlying* `RegionAttributes` *settings for each predefined attribute, see the Javadocs for* `RegionShortcut` and `ClientRegionShortcut`.

These sections provide an overview of the primary options available in the region shortcut settings. The names listed appear in the shortcut identifier alone or in combination, like “PARTITION” in `PARTITION`, `PARTITION_PROXY`, and `PARTITION_REDUNDANT`.

RegionShortcuts for Peers and Servers

These are the primary shortcut options available for peer and server regions.

For full details on the options, see the online Java documentation for `RegionShortcut`.

Cache Data Storage Model

- `PARTITION`. Creates a partitioned region. This is a data store for the region.

  You can also specify:

  - `PROXY`. Data is not stored in the local cache and the member is a data accessor to the region. This requires other members to create non-proxy copies of the region, so the data is stored somewhere.
  - `REDUNDANT`. The region stores a secondary copy of all data, for high availability.
- `REPLICATE`. Creates a replicated region. This is a data store for the region.

  You can also specify:

  - `PROXY`. Data is not stored in the local cache and the member is a data accessor to the region. This requires other members to create non-proxy copies of the region, so the data is stored somewhere.
- `LOCAL`. Creates a region private to the defining member.

Data Eviction

- `HEAP_LRU`. Causes least recently used data to be evicted from memory when the GemFire resource manager determines that the cache has reached configured storage limits.

Disk Storage

You can specify these alone or in combination:

- `PERSISTENT`. Backs up all data to disk, in addition to storing it in memory.
- `OVERFLOW`. Moves data out of memory and on to disk, when memory use becomes too high.

ClientRegionShortcuts for Clients

These are the primary shortcut options available for client regions.

For full details on the options, see the online Java documentation for `ClientRegionShortcut`.

Communication with Servers and Data Storage

- `PROXY`. Does not store data in the client cache, but connects the region to the servers for data requests and updates, interest registrations, and so on. The client is a data accessor to the region.
- `CACHING_PROXY`. Stores data in the client cache and connects the region to the servers for data requests and updates, interest registrations, and so on.
- `LOCAL`. Stores data in the client cache and does not connect the region to the servers. This is a client-side-only region. Note that this is not the same as setting the `scope` to `LOCAL`.

Data Eviction

- `HEAP_LRU`. Causes least recently used data to be evicted from memory when the GemFire resource manager determines that the cache has reached configured storage limits.

Disk Storage

With the `LOCAL` and `CACHING` data storage shortcut options, you can also specify disk storage options, alone or in combination:

- `PERSISTENT`. Backs up all data to disk, in addition to storing it in memory.
- `OVERFLOW`. Moves data out of memory and on to disk, when memory use becomes too high.

Storing and Retrieving Named Attributes

You create named region attributes by setting the attributes and storing them with a unique identifier. You retrieve named attributes:

- In the API, by using the value of the identifier in the region factory creation.
- In the `cache.xml`, by using the value of the identifier in the `<region>` or `<region-attribute>` `refid` setting. The `refid` is available in both elements for convenience.

Named attributes are `GemFire`'s predefined attributes plus any you have created and stored. When retrieving named attributes, any attribute settings you specify override the retrieved settings.

Storing and Retrieving Attributes in the cache.xml

Named attributes created in the cache configuration file can be used within the file and within the application. Attributes are stored in the order they appear in the file. You can retrieve them farther down in the file, after you have named them. Predefined attributes are available everywhere.

The next example shows partitioned region creation. An explanation follows.

Example 4.15 Using Stored Region Attributes in the cache.xml

```
<!-- Retrieving and storing attributes -->
<region-attributes id="myPartition" refid="PARTITION_REDUNDANT">
   <partition-attributes local-max-memory="512"/>
</region-attributes>

<!-- Two partitioned regions, one colocated with the other -->

<!-- Attributes are retrieved and applied in the first region -->
<region name="PartitionedRegion1" refid="myPartition"/>

<!-- Same stored attributes, modification for this region-->
<region name="PartitionedRegion2" refid="myPartition">
   <region-attributes>
      <partition-attributes colocated-with="PartitionedRegion1" />
   </region-attributes>
</region>
```

- The first region-attributes declaration starts with the predefined `PARTITION_REDUNDANT` attributes, modifies the `local-max-memory` setting, and stores the resulting attributes in the user-defined `myPartition` attributes.
- The region declarations use the new stored attributes, but each has its own interest policy, which is specified in the region creation.

Storing and Retrieving Attributes in the API

You can store and use stored attributes through the API the same as in the `cache.xml` file. The cache interfaces provide setter, getter, and list methods for attributes.

The example uses `RegionFactory` to create a region based on the predefined `PARTITION` attributes:

Example 4.16 RegionFactory Use of Attributes Template

```
final Region diskPortfolios =
   new RegionFactory("PARTITION").create("Portfolios");
```

This example retrieves an attributes template and passes it to the region creation with a modified pool specification:

Example 4.17 Retrieving Stored Attributes Through RegionFactory

```
ClientRegionFactory<String,String> regionFactory =
   cache.createClientRegionFactory(PROXY);
Region<String, String> region = regionFactory
   .setPoolName("publisher")
   .create("DATA");
```

4.7 Region Attributes List

In the XML file, the names need to be in the order listed in the DTD (see Appendix B, *cache.xml File*, on page 499). API `set` methods are in `RegionFactory` and `ClientRegionFactory` unless noted.

Table 4.2 Region Attributes

XML Configuration File Property Name	Region Factory API Setter Method
`cache-listener` (page 102)	`addCacheListener`
`cache-loader` (page 102)	`setCacheLoader`
`cache-writer` (page 103)	`setCacheWriter`
`cloning-enabled` (page 103)	`setCloningEnabled`
`concurrency-level` (page 103)	`setConcurrencyLevel`
`data-policy` (page 104)	`setDataPolicy`
`disk-store-name` (page 106)	`setDiskStoreName`
`disk-synchronous` (page 106)	`setDiskSynchronous`
`enable-async-conflation` (page 106)	`setEnableAsyncConflation`
`enable-gateway` (page 107)	`setEnableGateway`
`enable-subscription-conflation` (page 107)	`setEnableSubscriptionConflation`
`entry-idle-time` (page 108)	`setEntryIdleTimeout`
`entry-time-to-live` (page 108)	`setEntryTimeToLive`
`eviction-attributes` (page 108)	`setEvictionAttributes`
`hub-id` (page 109)	`setGatewayHubId`
`id` (page 109)	`Cache.setRegionAttributes`
`ignore-jta` (page 110)	`setIgnoreJTA`
`index-update-type` (page 110)	`setIndexMaintenanceSynchronous`
`initial-capacity` (page 110)	`setInitialCapacity`
`is-lock-grantor` (page 111)	`setLockGrantor`
`key-constraint` (page 111)	`setKeyConstraint`
`load-factor` (page 112)	`setLoadFactor`
`membership-attributes` (page 112)	`setMembershipAttributes`
`multicast-enabled` (page 113)	`setMulticastEnabled`
`partition-attributes` (page 113)	`setPartitionAttributes`
`pool-name` (page 115)	`setPoolName`
`refid` (page 116)	Passed to Cache.`createRegionFactory` and `ClientCache.createClientRegionFactory`
`region-idle-time` (page 116)	`setRegionIdleTimeout`
`region-time-to-live` (page 116)	`setRegionTimeToLive`
`scope` (page 116)	`setScope`
`statistics-enabled` (page 117)	`setStatisticsEnabled`
`subscription-attributes` (page 118)	`setSubscriptionAttributes`
`value-constraint` (page 119)	`setValueConstraint`

cache-listener

Event handler plug-in that receives asynchronous, after-event callbacks for modifications to the region and its entries.

The plug-in is written by you, as an implementation of one of the listener interfaces offered by GemFire. You program your implementation to do whatever your application needs to do in response to region and entry events, and then specify it as a `cache-listener` on your region.

You can define any number of cache listeners for a region in any member.

For distributed regions, depending on the region configuration, a cache listener may receive only events that originate in the local cache, or it may receive those events along with events that originate remotely. For more information, see the options in `RegionShortcut` and `ClientRegionShortcut` online Java documentation.

For partitioned regions, listeners are only fired in the member hosting the primary copy of the data.

This declaration assigns a listener to the region:

```
<region-attributes>
   <cache-listener>
      <class-name>quickstart.SimpleCacheListener</class-name>
   </cache-listener>
</region-attributes>
```

For more information on cache listeners, see *Implementing Cache Event Handlers* on page 300.

cache-loader

Event handler plug-in that receives callbacks for cache misses, when a requested key is not present or has a null value in the local cache.

You write the plug-in as an implementation of `com.gemstone.gemfire.cache.CacheLoader`. Generally, a cache loader is programmed to retrieve the missing entry value from an outside data source, but you can program it to do whatever you need.

At most, one cache loader can be defined in each member for the region.

For distributed regions, a cache loader may be invoked remotely from other members that have the region defined. When an entry `get` results in a cache miss in a region with a cache loader defined, the loader's `load` method is called. This method is usually programmed to retrieve data from an outside data source, but it can do anything required by your application.

For partitioned regions, install the cache loader in every data host. Partitioned regions support partitioned loading, where each cache loader loads only the data entries in the local member. If data redundancy is configured, data is loaded only if the local member holds the primary copy. This declaration assigns a loader to the region:

```
<region-attributes>
   <cache-loader>
      <class-name>quickstart.SimpleCacheLoader</class-name>
   </cache-loader>
</region-attributes>
```

For more information on cache loading, see *Loading Entries Through a Data Loader* on page 196.

cache-writer

Event handler plug-in that receives synchronous, before-event callbacks for modifications to the region and its entries.

You write the plug-in as an implementation of `com.gemstone.gemfire.cache.CacheWriter`. You program your implementation to do whatever your application needs to do in response to region and entry events, and then specify it as a `cache-writer` on your region. You can abort the operation from the cache writer, if needed.

At most one cache writer can be defined in each member for the region.

- In distributed regions, a cache writer may be invoked remotely from other members that have the region defined.
- In partitioned regions, the system only uses the cache writer in the primary data host for the event.

This declaration assigns a writer to the region:

```
<region-attributes>
   <cache-writer>
      <class-name>quickstart.SimpleCacheWriter</class-name>
   </cache-writer>
</region-attributes>
```

For more information on cache writers, see *Implementing Cache Event Handlers* on page 300.

cloning-enabled

Determines how `fromDelta` applies deltas to the local cache for delta propagation.

When `true`, the updates are applied to a clone of the value and then the clone is saved to the cache. When false, the value is modified in place in the cache. The default value is false.

This declaration enables cloning:

```
<region-attributes cloning-enabled="true">
</region-attributes>
```

For more information, see *Delta Propagation Properties* on page 332.

concurrency-level

Gives an estimate of the maximum number of application threads that will concurrently access a region entry at one time. This attribute does not apply to partitioned regions.

This attribute helps optimize the use of system resources and reduce thread contention. It sets an initial parameter on the underlying `java.util.ConcurrentHashMap` used for storing region entries. Before you start, read the concurrency level description, then see the Java API documentation for `java.util.ConcurrentHashMap`.

This declaration sets the region's concurrency level to 20 threads:

```
<region-attributes concurrency-level="20">
</region-attributes>
```

Specification (default in bold)	Definition
positive integer **default=16**	Concurrency level is set to the specified value.

data-policy

Specifies how the local cache handles the data for a region.

This setting controls behavior such as local data storage and region initialization. Configure the most common options with GemFire's stored region attributes. See *Shortcut Attribute Options* on page 97 and the Javadocs for `RegionShortcut` and `ClientRegionShortcut`.

The default `data-policy` of `normal` specifies local cache storage. The `empty` policy specifies no local storage. In the region shortcuts, `empty` corresponds to the settings that include the string, `PROXY`. You can use an empty region for event delivery to and from the local cache without the memory overhead of data storage.

> *If you use the* `empty` *policy and want to receive events on the region, you must set the* `subscription-attributes interest-policy` *to* `all`.

The data policy and the `interest-policy` subscription attribute work together. For details, see `subscription-attributes` on page 118. This declaration sets `data-policy` to `replicate`:

```
<region-attributes data-policy="replicate" >
</region-attributes>
```

This does the same thing:

```
<region-attributes refid="REPLICATE" />
```

The `replicate` and `persistent-replicate` policies both specify local cache storage. In addition, if the region has a distributed `scope` (page 116), these two policies cause the region to be initialized from region instances in other caches if any are available when the region is created. The `persistent-replicate` policy stores region data to disk.

Data Policy for a Partitioned Region

For partitioned regions, the data policy must be `partition` or `persistent-partition`. These are automatically set for the `PARTITIONED RegionShortcut`s. You must use the same partitioned settings for a region across the distributed system. You cannot mix `partition` and `persistent-partition` for different members hosting the same partitioned region.

Data Policy for a Server Region

To provide a coherent cache view to their clients, server regions must have one of the partitioned or replicate data-policy settings. Replicated server regions must have `distributed-ack` or `global scope` (page 116). The region shortcut `REPLICATED` settings use `distributed-ack`.

Table 4.3 Data-policy attributes

Specification (default in bold)	Definition
`empty`	Prevents data from ever being stored in local memory. The region always appears empty. Use this for zero-footprint producers that only distribute data to others and for zero-footprint consumers that only see events. To receive events with this policy, you must set the `subscription-attributes` (page 118) `interest-policy` to `all`. This policy applies to distributed regions. For partitioned regions, see the `partition` policy below.
normal	Causes data for the region to be stored in local memory. The `normal` policy allows the contents in the cache to differ from other caches.

Specification (default in bold)	Definition
`replicate`	Causes all data for a distributed region to be stored in local memory. The region is initialized with the data from other caches. Operations that would cause the contents to differ with other caches are not allowed. This is compatible with local scope as well, behaving the same as for `normal`.
`partition`	Data is partitioned across local and remote caches using the automatic data distribution behavior of partitioned regions. You can configure the amount of data held in this cache in the partition attribute `local-max-memory` (page 114). *To duplicate the behavior of the `empty` data policy in partitioned regions, set the region attributes `refid` to one of the `PARTITION_PROXY` settings `RegionShortcut` settings.* The `data-policy` for partitioned regions is `partition`. This data policy setting makes the region a partitioned region.
`persistent-partition`	Behaves the same as `partition`, with the addition of storing the member's buckets to disk and possibly initializing the region from disk. For a discussion, see *Using Disk for Backup and Overflow* on page 170. Data is stored according to the settings in the `DiskStore` associated with the region through the region attribute `disk-store-name`.
`persistent-replicate`	Behaves the same as `replicate`, with the addition of storing region data to disk and possibly initializing the region from disk. For a discussion, see *Using Disk for Backup and Overflow* on page 170. Data is stored according to the settings in the `DiskStore` associated with the region through the region attribute `disk-store-name`.
`preloaded`	Causes the region to quickly establish a working set of data by using cached data from other region instances. Behaves the same as a replicated region at startup, after which it behaves the same as a `normal` distributed region.

disk-store-name

Assigns the region to the disk store with this name from the disk stores defined for the cache.

Store region data to disk by defining the region as persistent using the *Shortcut Attribute Options* on page 97 or the `data-policy` (page 104) or by implementing LRU `eviction-attributes` (page 108) with an action of overflow to disk.

Each disk store defines the file system directories to use, how data is written to disk, and other disk storage maintenance properties. In addition, the `disk-synchronous` region attribute specifies whether writes are done synchronously or asynchronously. By default, the disk store name is not defined, which causes all disk activities to be done through the system's default disk store.

See *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.

disk-synchronous

For regions that write to disk, specifies whether disk writes are done synchronously for the region.

By default, disk writes are asynchronous. Used in conjunction with the disk store specified in `disk-store-name`.

enable-async-conflation

For TCP/IP distributions between peers, specifies whether to allow aggregation of asynchronous messages sent by the producer member for the region. This is a special-purpose attribute that applies only when asynchronous queues are used for slow consumers.

To configure this type of queueing, see the `async* gemfire.properties` settings in *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

A `true` value allows conflation. A `false` value disables conflation so that all asynchronous messages are sent individually. The default value is `true`.

This attribute gives you extra control over peer-to-peer communication between distributed regions using TCP/IP. It does not apply to client/server communication, multi-site communication, or to communication using the UDP unicast or IP multicast protocols.

To use this attribute, both `multicast-enabled` in `cache.xml` and `disable-tcp` in `gemfire.properties` must be `false` (the default).

This `cache.xml` file snippet disables conflation for a region:

```
<region-attributes enable-async-conflation="false">
</region-attributes>
```

The attribute can also be set through the `AttributesFactory` interface.

enable-gateway

In caches where a gateway hub is defined, specifies whether events in the region are forwarded to the gateway.

Gateways are used:

- In multisite installations to define a connection and distribute messages between sites. See Chapter 10, *Configuring Multi-site Installations*, on page 267.
- Within the distributed system to manage asynchronous delivery of batched cache events to a write-behind event listener. See Chapter 11, *Handling Events*, on page 293.

If set to `true`, events are delivered to the gateway queue for delivery to other sites or to the listener. A false value (the default) disables gateway events for the region. For a list of the events that are put into the queue, see *Message Processing* on page 274.

This setting does not affect events coming in through a gateway from other sites. If a gateway is defined, all incoming events are processed.

> *This parameter is valid only for event delivery to the gateway. It has no effect on peer-to-peer communication.*
>
> *This attribute must be consistent for the region across all members in the distributed system.*
>
> *If you enable this in a multi-site system with multiple hubs, you must also set* hub-id (page 109).

This `cache.xml` file snippet enables gateway messaging for the region:

```
<region-attributes enable-gateway="true">
</region-attributes>
```

enable-subscription-conflation

For server-to-client communication, specifies whether conflation is enabled for sending messages from a server to its clients.

A `true` value enables conflation. A `false` value disables conflation (default).

This `cache.xml` file snippet enables bridge conflation for a region:

```
<region-attributes enable-subscription-conflation="true">
</region-attributes>
```

It has no effect on peer-to-peer communication.For information on client/server installations, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.

The `conflate-events` gemfire property may override this setting for the client. See *Configuring GemFire System Properties* on page 45 of the *GemFire Enterprise System Administrator's Guide* for a description of the `conflate-events` gemfire property.

entry-idle-time

Specifies how long the region's entries can remain unexpired in the cache without anyone creating, updating, or accessing them.

See *Keeping Your Data Current With Expiration* on page 176.

This declaration sets the entry idle-time to one minute, with an action of local invalidate:

```
<region-attributes statistics-enabled="true">
   <entry-idle-time>
      <expiration-attributes timeout="60" action="local-invalidate"/>
      </expiration-attributes>
   </entry-idle-time>
</region-attributes>
```

entry-time-to-live

Specifies how long the region's entries can remain unexpired in the cache wtihout anyone creating or updating them.

See *Keeping Your Data Current With Expiration* on page 176.

This declaration sets the entry time-to-live to one minute, with an action of local invalidate:

```
<region-attributes statistics-enabled="true">
   <entry-time-to-live>
      <expiration-attributes timeout="60" action="local-invalidate"/>
      </expiration-attributes>
   </entry-time-to-live>
</region-attributes>
```

eviction-attributes

Specifies whether and how to control a region's size. Size is controlled by removing least recently used (LRU) entries to make space for new ones.

Eviction attributes have two components:

- The eviction algorithm, which specifies what to track and what limit to place on its size.
- The eviction action, which specifies how to evict entries when the specified limit has been reached for the algorithm.

  Entries are not evicted while they are part of a transactional operation.

The settings apply to all of the space used for data in the region or cache, depending on the algorithm. For partitioned regions, this includes redundant copies of data, in addition to the primary copies. The size setting can vary between different members hosting the same region.

Eviction Algorithm	Definition
`lru-entry-count`	Considers the number of entries in the region before performing an eviction.
`lru-memory-size`	Considers the amount of memory, in megabytes, used by the region.

Eviction Algorithm	Definition
`lru-heap-percentage`	Runs evictions when the GemFire resource manager says to. The manager orders evictions when the total cache size is over the heap percentage limit specified in the manager configuration. See *Controlling Heap Use With the Resource Manager* on page 186

The eviction action determines how the controller removes entries from the cache.

In partitioned regions, the action must be the same across the distributed system.

Eviction Action (default in **bold**)	Definition
`local-destroy`	The LRU region entry is locally destroyed. This cannot be used in replicated regions. For an explanation, see *Replication and Local Destroy and Invalidate Operations* on page 200.
`overflow-to-disk`	The LRU region entry value is written to disk and set to null in the member to reclaim heap space. See also `disk-store-name` (page 106).

This declaration causes the region to evict entries when the entry count reaches 1000. When that capacity is reached, the least-recently-used region entry is written to disk and then nulled-out in the member.

```
<region-attributes>
   <eviction-attributes>
      <lru-entry-count maximum="1000" action="overflow-to-disk"/>
   </eviction-attributes>
</region-attributes>
```

You can configure your region for `lru-heap-percentage` with an eviction action of `local-destroy` using GemFire's stored region attributes. See *Shortcut Attribute Options* on page 97 and the Javadocs for `RegionShortcut` and `ClientRegionShortcut`.

See *Controlling Memory Use With Eviction and Overflow* on page 182.

hub-id

Specifies the destination hub for region events. Not set by default.

For multiple-hub systems, in regions with `enable-gateway` (page 107) set, this must be specified. If it is not, region events go to all available hubs, which results in duplicate sends to remote sites.

See *Load Balancing in Multi-site Installations* on page 279.

id

Stores the region attribute settings in the cache with this identifier.

Once stored, the attributes can be retrieved using the region attribute `refid`. For details, see *Region Shortcuts and Custom Named Region Attributes* on page 97.

ignore-jta

Determines whether operations on this region participate in active JTA transactions or ignore them and operate outside of the transactions.

This setting is primarily used for cache loaders, writers, and listeners that need to perform non-transactional operations on a region, such as caching a result set. See *Using JTA transactions with GemFire* on page 435.

This declaration enables the `ignore-jta` attribute:

```
<region-attributes ignore-jta="true">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
`true`	Cache operations execute outside of any JTA transaction.
false	Cache operations automatically become part of any active JTA transaction.

index-update-type

Specifies whether region indexes are maintained synchronously with region modifications, or asynchronously in a background thread.

In the `cache.xml` file, it is set as a value, `asynchronous` or `synchronous`, assigned to the `index-update-type` region attribute. The API sets this property through a boolean passed to the `setIndexMaintenanceSynchronous` method.

This declaration sets the index maintenance to asynchronous:

```
<region-attributes index-update-type="asynchronous">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
`synchronous` (in `cache.xml`)	Indexes are maintained synchronously with region updates. This provides the greatest consistency between the region data and the index.
`asynchronous` (in `cache.xml`)	Indexes are maintained asynchronously with region updates. This provides the best index maintenance performance.

initial-capacity

Together with the `load-factor` region attribute, sets the initial parameters on the underlying `java.util.ConcurrentHashMap` used for storing region entries. This attribute does not apply to partitioned regions.

See the Java API documentation for `java.util.ConcurrentHashMap`.

This declaration sets the region's initial capacity to 20:

```
<region-attributes initial-capacity="20">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
positive integer **default=16**	Initial capacity for the underlying hash map.

interest-policy

See *subscription-attributes* on page 118.

is-lock-grantor

Determines whether the member defines itself as the lock grantor for the region at region creation time.

This attribute only specifies whether the member becomes lock grantor at region creation and does not reflect the current state of the member's lock grantor status. The member's lock grantor status may change if another member subsequently defines the region with `is-lock-grantor` set to `true`.

The attribute is only relevant for regions with `global` scope, as only they allow locking. It affects implicit and explicit locking. Implicit locking is performed for every operation that updates a global region. Explicit locking can be accomplished through the `Region` API.

This declaration creates the region with the member as the lock grantor:

```
<region-attributes is-lock-grantor="true">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
`true`	The member is defined as the lock grantor when the region is created. This may remove lock grantor status from any other member that holds it.
`false`	The member is not defined as the lock grantor.

For more information, see *Locking* on page 205.

key-constraint

Defines the type of object to be allowed for the region entry keys.

The attribute ensures that the keys for the region entries are all of the same class. If `key-constraint` is not used, the region's keys can be of any class. This attribute, along with `value-constraint`, is useful for querying and indexing because it provides object type information to the query engine.

> *A region being queried should contain only homogeneous objects of the same type. See* *The Query Language* on page 352.

This declaration constrains the keys to `String`:

```
<region-attributes>
   <key-constraint>java.lang.String</key-constraint>
</region-attributes>
```

Specification (default in **bold**)	**Definition**
not specified	There is no constraint on the object stored in the entry key.
class	This must be a fully-qualified class name. The object stored in the entry key must be of the specified type.

load-factor

Together with the `initial-capacity` region attribute, sets the initial parameters on the underlying java.util.ConcurrentHashMap used for storing region entries. This attribute does not apply to partitioned regions.

Before you set this attribute, read the discussion of initial capacity and load factor, then see the Java API documentation for `java.util.ConcurrentHashMap`.

This declaration sets the region's load factor to 0.85:

```
<region-attributes load-factor="0.85">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
floating point number **default=0.75**	Load factor is set to a specified value from `0` to `1`.

membership-attributes

Establishes reliability requirements and behavior for a region. Does not apply to partitioned regions.

The `membership-attributes` setting provides options for configuring a region to require one or more membership roles to be present in the system for reliable access to the region. You can set up your own roles, such as `producer` or `backup`, and specify each role as a string.

This declaration configures the region to require a `producer` member in the distributed system:

```
<region-attributes>
   <membership-attributes loss-action="no-access"
            resumption-action="none">
      <required-role name="producer">
      </required-role>
   </membership-attributes>
</region-attributes>
```

Membership attributes have no effect unless one or more membership roles are specified. The attributes are immutable after the region has been created. For additional information, including detailed option descriptions, see *Membership Attributes* on page 455.

multicast-enabled

Specifies whether distributed operations on the region should use multicast.

Multicast must also be enabled in the distributed system for the member using the `mcast-port` gemfire property setting. See *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

This declaration creates the region with multicasting enabled:

```
<region-attributes multicast-enabled="true">
</region-attributes>
```

Specification	Definition
`true`	Region operations will use multicast for messaging with other distributed system members.
`false`	Region operations will not use multicasting.

See also *Peer-to-Peer Messaging and Distribution* on page 65 of the *GemFire Enterprise System Administrator's Guide* and *Tuning Multicast Communication* on page 161 of the *GemFire Enterprise System Administrator's Guide*.

partition-attributes

Defines the partitioning attributes for the region. This is set during the region creation in the first data host for the partitioned region.

Partitioned regions have attributes that control partitioning behavior, such as storage redundancy. The partition attributes are specified before region creation using the `partition-attributes` region attribute, either in the `cache.xml` file or through the Java API. For more information, see the online Java API documentation for `PartitionAttributesFactory`.

You can configure the most common of these options with GemFire's stored region attributes. See *Shortcut Attribute Options* on page 97 and the Javadocs for `RegionShortcut` and `ClientRegionShortcut`.

All members defining a partitioned region must have the same partition attribute settings. The only exception is `local-max-memory`, which can be different for each member. When a member creates the region, if any of the other attribute settings are different from those already defined by prior members, an exception is thrown and the creation fails.

This snippet from `cache.xml` configures a partitioned region with one extra copy of data entries and a maximum of 613 buckets:

```
<region-attributes>
   <partition-attributes redundant-copies="1" total-num-buckets="613"/>
</region-attributes>
```

This does the same thing:

```
<region-attributes refid="PARTITION_REDUNDANT" >
   <partition-attributes total-num-buckets="613"/>
</region-attributes>
```

This table shows the partition attributes. All of these attributes are immutable.

Table 4.4 Partition Attributes

Partition Attribute	Default Setting	Description
`colocated-with`		Name of another partitioned region to collocate with this region. Used to group like data within hosts across multiple partitioned regions. See *Colocating regions with custom partitioning* on page 186.
`local-max-memory`	90% of local heap (Local heap is set on the JVM command line.)	Maximum megabytes of memory set aside for this region in the local member for primary and redundant copies. This value must be smaller than the Java settings for the initial or maximum VM heap. When the memory use goes above this value, GemFire issues a warning, but operation continues. Besides setting the maximum memory to use for the member, this setting also tells GemFire how to balance the load between members where the partitioned region is defined. For example, if one member sets this value to twice the value of another member's setting, GemFire works to keep the ratio between the first and the second at two-to-one, regardless of how little memory the region consumes. *This is a local parameter that applies only to the region in the local member.* A value of 0 turns off local data caching, so the member is a data accessor. See *Data Hosts and Data Accessors* on page 90.
`recovery-delay`	`-1`	Used if `redundant-copies` is not 0. Number of milliseconds to wait after a member failure before recovering redundancy. If set to -1, no recovery is done. Valid settings are long integers greater than or equal to -1. See *Redundancy Recovery* on page 193.
`redundant-copies`	`0`	Number of redundant data copies that the partitioned region must maintain for each entry. Range: 0-3. If you specify 1, this partitioned region maintains the original and one copy, for a total of two copies altogether. A value of 0 disables redundancy. See *Partitioned Regions with Redundancy* on page 190.
`startup-recovery-delay`	`0`	Used if `redundant-copies` is not 0. Number of milliseconds to wait after a member joins before recovering redundancy. If set to -1, no recovery is done. Valid settings are long integers greater than or equal to -1. See *Redundancy Recovery* on page 193.
`total-max-memory`		Maximum combined megabytes of memory to be used by all processes hosting this region for all copies, primary and redundant.

`total-num-buckets`	`113`	Total number of *buckets* or data storage areas allotted for the entire partitioned region. As data moves from one member to another, the entries in a bucket move as one unit. This should be a prime number larger than the total number of members hosting the region. It should be at least four times the number of host members. More buckets increases overhead, however, especially when `redundant-copies=2` or `3`.

pool-name

For client regions, specifies the server pool the region is to use.

If your client cache only has one pool, you do not need to specify the `pool-name`, providing you indicate the region is a client region through the `ClientRegionShortcuts`. The system automatically assigns the single pool to all regions that are configured with one of the `PROXY` `ClientRegionShortcuts`. See *Shortcut Attribute Options* on page 97.

This declaration creates the region as a client region with a server pool named `DatabasePool`. This `pool-name` specification is required, as there are multiple pools in the client cache:

```
<client-cache>
   <pool name="DatabasePool" subscription-enabled="true">
      ...
   </pool>
   <pool name="OtherPool" subscription-enabled="true">
      ...
   </pool>
   <region ...
      <region-attributes pool-name="DatabasePool">
      </region-attributes>
      ...
```

This declaration creates the region as a client region assigned the single pool that is defined for the client cache. Here the `pool-name` specification is implied:

```
<client-cache>
   <pool name="publisher" subscription-enabled="true">
      ...
   </pool>
   <region name="myRegion" refid="CACHING_PROXY">
   </region>
</client-cache>
```

Specification (default in **bold**)	**Definition**
not specified	If the region is otherwise identified as a client region, no `pool-name` indicates to the system to use the one single pool that is defined in the client cache. If multiple pools are defined, the region creation returns an exception.
String corresponding to a `Pool` *instance's* name (page 211) *attribute*	The client region will use the specified Pool to satisfy data requests and to forward data updates to the server side.

For details on client/server installations, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201 and Chapter 9, *Client/Server Advanced Topics*, on page 227.

refid

Retrieves named region attributes for attributes initialization.

You can also use the region shortcut or the refid to initialize the region factory through the methods `Cache.CreateRegionFactory` and `ClientCache.CreateClientRegionFactory`.

See *Region Shortcuts and Custom Named Region Attributes* on page 97.

region-idle-time

Specifies how long the region can remain unexpired in the cache without anyone accessing or updating it. Does not apply to partitioned regions.

For general information on expiration, see *Keeping Your Data Current With Expiration* on page 176.

This declaration sets the region idle-time to one hour, with an action of local destroy:

```
<region-attributes statistics-enabled="true">
   <region-idle-time>
      <expiration-attributes timeout="3600" action="local-destroy"/>
   </region-idle-time>
</region-attributes>
```

region-time-to-live

Specifies how long the region can remain unexpired in the cache without anyone updating it. This attribute does not apply to partitioned regions.

For general information on expiration, see *Keeping Your Data Current With Expiration* on page 176.

This declaration sets the region time-to-live to one hour, with an action of local destroy:

```
<region-attributes statistics-enabled="true">
   <region-time-to-live>
      <expiration-attributes timeout="3600" action="local-destroy"/>
   </region-time-to-live>
</region-attributes>
```

scope

Determines how updates to region entries are distributed to the other caches where the region and entry are defined. Does not apply to partitioned regions.

Scope also determines whether to allow remote invocation of some of the region's event application plug-ins.

You can configure the most common of these options with GemFire's stored region attributes. See *Shortcut Attribute Options* on page 97 and the Javadocs for `RegionShortcut` and `ClientRegionShortcut`.

Server regions that are not partitioned must be replicated with `distributed-ack` or global scope. The region shortcuts specifying `REPLICATE` have `distributed-ack` scope.

This `cache.xml` file snippet sets a region's scope to `distributed-ack`:

```
<region-attributes scope="distributed-ack">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
`local`	No distribution. The region is visible only to threads running inside the member.
`distributed-no-ack`	Events are distributed to remote caches with no acknowledgement required.
`distributed-ack`	Events are distributed to remote caches with receipt acknowledgement required.
`global`	Events are distributed to remote caches with global locking to ensure distributed cache consistency.

The effects of scope on the distribution of data and events is explored in greater detail in *Choosing the Level of Distribution* on page 204 and Chapter 11, *Handling Events*, on page 293.

statistics-enabled

Specifies whether statistics are to be gathered on the region. This does not apply to partitioned regions.

GemFire Enterprise provides a standard set of statistics for cached regions and region entries. Statistics are required for expiration and they also provide information for fine-tuning your distributed system. Unlike other GemFire statistics, region statistics for local and distributed regions are not archived and cannot be charted. Statistics for partitioned regions are always archived, however, and can be charted.

By default, statistics gathering is disabled for local and distributed regions. This declaration enables statistics for the region:

```
<region-attributes statistics-enabled="true">
</region-attributes>
```

Specification (default in **bold**)	**Definition**
`true`	Statistics are gathered. This is required for the automatic expiration of cached data. Ssee *Keeping Your Data Current With Expiration* on page 176.
`false`	Statistics are not gathered.

Statistics are kept in instances of `com.gemstone.gemfire.cache.CacheStatistics` and made available through the region and its entries through the `Region.getStatistics` and `Region.Entry.getStatistics` methods.

Statistics are discussed in Chapter 18, *Monitoring and Tuning Your Applications*, on page 463.

subscription-attributes

Specifies subscriber requirements and behavior for the region. Currently there is one subscription attribute, interest-policy, that defines which entry operations are delivered to the region from remote caches.

The `interest-policy` options are:

- `cache-content`. Default. Registers interest in receiving events only for entries that are already in the local region. For partitioned regions, the local data host must hold the primary copy of the entry's data.
- `all`. Registers interest in receiving events for all entries that are anywhere in the distributed or partitioned region.

This declaration sets the policy to `all`:

```
<region-attributes>
   <subscription-attributes interest-policy="all"/>
</region-attributes>
```

The effect of the `interest-policy` setting varies with the `data-policy` (page 104), as shown in the table below. The interest policy determines which events are delivered, but the data policy determines how the events are applied to the cache.

Data Policy	Interest Policy	
	`cache-content`	`all`
`normal`	**Default: Remote cache events for entries that already exist locally are received, and the local entries are updated. No other remote events are received.**	All cache events are received, and the data is stored in the local region. Behaves the same as the `replicate` data policy, except it does not initialize the region with data from other caches. The region adds entries gradually as events are received.
`replicate` or `persistent-replicate`	All cache events are received, and their data is stored in the local region. If persistent, locally-cached data is also stored to the disk store. *The interest policy setting is ignored.*	
	All cache events are received. Their data is stored in the local region and on disk. *Interest policy setting is ignored.*	
`empty`	Remote region events are received, but not remote entry events. No entry data is stored. Otherwise, normal distribution rules apply.	All cache events are received, including remote entry events. Otherwise, normal distribution rules apply.
`partition` or `persistent-partition`	Cache events for data stored in the local member are received and applied*. Events for data stored in other members are not delivered to cache listeners, even if the operation originates in the member containing the listener.	All cache events are received. An event is applied to the local cache only if its entry belongs there. If persistent, locally-cached data is also stored to the disk store. In some situations, a partitioned region with an interest policy of `all` can create a network bottleneck. The `cache-content` policy creates less traffic to and from each member.

For details on event-driven processing in GemFire, see Chapter 11, *Handling Events*.

value-constraint

Defines the type of object to be allowed for the region entry values.

The attribute ensures that the values for the region entries are all of the same class. If a value constraint is not used, the region's values can be of any class. This attribute, like the `key-constraint` (page 111) attribute, is useful for querying and indexing because it provides object type information to the query engine.

> *A region being queried should contain only homogeneous objects of the same type. See* *The Query Language* on page 352.

The value-constraint is only checked in the cache that does the entry `put` or `create` operation. When the entry is distributed to other caches, to avoid deserializing the object, the constraint is not checked. This means that you should set the constraint in every cache where you create or update the region entries. For client/server and multi-site installations, your constraints should match between client and server and between the disparate distributed systems.

This declaration constrains the values in the region to a user-defined object. The object class must be available through the application `CLASSPATH`.

```
<region-attributes>
   <value-constraint>cacheRunner.Portfolio</value-constraint>
</region-attributes>
```

Specification (default in **bold**)	**Definition**
not specified	No constraint on the object stored in the entry value.
object specification	The object stored in the entry value must be of the specified type.

4.8 Creating Custom User Attributes for Regions and Entries

GemFire allows applications to define custom *user attributes* for regions or for individual data entries. Unlike the other configuration settings in this chapter, these optional user attributes are used by the application only, not by GemFire. For more information, see the online Java API documentation for `Region` and `Region.Entry`.

An application can use these attributes to store data associated with the region or entry and to get the data back later.

To set up user attributes:

1. Create a Java object with your attribute definitions.
2. Attach the object to the region or to an entry:

```
Region.setUserAttribute(value)
Region.getEntry(key).setUserAttribute(value)
```

3. Get the attribute value:

```
Region.getUserAttribute()
Region.getEntry(key).getUserAttribute()
```

User Attributes Example

The following examples show how to store the names of database tables associated with the region, so that a cache writer can access them and use them to update the database. This data storage model is discussed in *Querying Object Data Versus Relational Data* on page 347.

This example shows how to create a region with a user attribute object `myAttribute`, which specifies the database table to be updated from the region, and a cache writer `PortfolioDBWriter`, which takes changes to the region entries and writes them to the database table. The database table's name is `portfolio`.

Example 4.18 Attaching the User Attribute to a Region

```
Object myAttribute = "portfolio"
final Region portfolios =
   new RegionFactory().setCacheWriter(new PortfolioDBWriter())
   .create("Portfolios");
Portfolios.setUserAttribute(myAttribute);
```

This example shows the implementation of the cache writer that takes changes to the region entries and writes them to the database table named in the user attribute.

Example 4.19 Implementing a Cache Writer That Reads the User Attribute

```
//Implement a cache writer that reads the user attribute setting
public class PortfolioDBWriter
extends CacheWriterAdapter {
   public void beforeCreate(RegionEvent event) {
      table = (String)event.getRegion().getUserAttribute();
      // update database table using name from attribute
      . . .
   }
}
```

Limitations and Alternatives

User attributes are not distributed to other processes. In a distributed region, you must create an attribute object in every instance of the region. When you update the attribute, the update isn't distributed. You need to update every instance of the region separately.

User attributes are not stored to disk for region persistence or overflow, so they cannot be recovered to reinitialize the region.

If your application requires features that are not supported by user attributes, an alternative is to create another region to hold this data, instead of putting it into attributes. For instance, an `AttributesRegion` could use region names as keys and the "user attributes," such as the table names in the previous example, as values. Changes to `AttributesRegion` would be distributed to other processes, and you could configure the region for persistence or overflow if needed.

4.9 Dynamic Region Management

Dynamic regions are regions whose creation and destruction is distributed between caches. You can distribute dynamic regions between peers in a distributed system, between clients and servers in a client/server caching topology, and between distributed systems in a multi-site installation. Any cache can be configured to participate in dynamic region management.

Partitioned regions cannot be managed dynamically.

Dynamic region creation and destruction requires the use of a special factory that tracks the regions in the cache and handles the distribution of dynamic region events. Once created, however, dynamic regions behave the same as the other regions in your cache.

Dynamic regions can have any of the configuration settings that are available to normal cache regions. The only caveat is that each dynamic region assumes the attributes of its parent region, so you must begin with non-dynamic parent regions in place in your caches with the attribute settings you need for your dynamic regions. Dynamic regions cannot be created as root regions.

To use dynamic regions, you must:

1. Configure every cache that participates in dynamic region management with a dynamic region factory. This factory creates and destroys dynamic regions, distributes the operations, and processes dynamic region operations received from remote factories.
2. In all participating caches, create parents for your dynamic regions with the attributes that you want their children to inherit. These parent regions must be declared in your `cache.xml` file, not created through the API.
3. Create your dynamic regions in the appropriate caches. The creation operation first creates the region locally and then distributes the creation. All connected caches configured with a dynamic region factory automatically receive dynamic region creations. Each dynamic region is created in every remote cache where a dynamic region factory exists, the parent region exists, and the region itself does not already exist.

The following figure illustrates the basic concept for a peer-to-peer installation. When a cache has a dynamic region factory, it accepts dynamic region creation from connected caches. Here, member M_1 creates the dynamic region, `C`, as a child of region `/A/B`. The dynamic region creation is distributed to M_2 because it has a dynamic region factory instantiated.

Figure 4.1 Dynamic Region Creation

Do not use standard `Region` *methods to create or destroy dynamic regions.*

The region created dynamically must not exist locally before the create method call. The dynamic region is created locally as well as remotely.

When a dynamic region is created, it inherits all attributes from the parent region. Application plug-ins belonging to the parent are provided to the dynamic region by reference, so care must be taken when creating the plug-ins for static parent regions. The plug-ins must be thread-safe and must continue to function after a close is called for a specific region.

This figure illustrates how the factory assigns attributes to dynamic regions.

Figure 4.2 Dynamic Region Attributes

Region `/A/B/C` inherits all attributes from its parent region, `/A/B`. The three application plug-ins listed above are inherited by reference while other attributes (including the capacity controller) are copied.

The rest of this section covers the following:

- The Dynamic Region Interfaces and Classes (page 124)
- Managing the Dynamic Region Factory (page 125)
- Creating and Destroying Dynamic Regions (page 127)

The Dynamic Region Interfaces and Classes

This section gives an overview of the primary classes and interfaces used for dynamic region management. For complete information, see the online Java API documentation.BT:

com.gemstone.gemfire.cache Class

- **DynamicRegionFactory**. Interface that is the entry point to dynamic region management. When you create an instance of this factory, you open your cache for dynamic region creation by other caches. Use this factory to create and destroy dynamic regions in the local cache and in other caches.
- **DynamicRegionFactory.Config**. Object that holds the dynamic region factory configuration. For information about configuration options, see *Managing the Dynamic Region Factory* on page 125.
- **DynamicRegionListener**. Interface that handles dynamic region events. The methods are invoked synchronously.

  You should use either *the API or the* `cache.xml` *file to configure the dynamic region factory. The* `cache.xml` *declaration overrides any dynamic region factory configuration done through the API. This is different from the standard* `cache.xml` *loading behavior.*

Managing the Dynamic Region Factory

The system stores dynamic region meta-information in a *reserved* root region, `/__DynamicRegions`, inside your cache. This region is reserved for system use only. When you configure your dynamic region factory, most of the settings affect how `/__DynamicRegions` is managed, rather than how the dynamic regions themselves are managed.

> *Do not modify* `/__DynamicRegions`*. This reserved root region contains all meta-data pertaining to dynamic region management. Directly altering this region may give unexpected results and could compromise your data. Only alter your dynamic region factory settings through the factory methods.*

Configuration Options

This section lists the API configuration options for the dynamic region factory.

diskDir

This option is relevant only if the dynamic region factory attribute, `persistBackup`, is `true`. It specifies where to store the persistence files for `/__DynamicRegions`. If not specified, the files are stored in the current working directory. The default value is `null`.

> *You must specify different directories for the different members that have a dynamic region factory. You can do this either by explicitly specifying different* `diskDir` *parameters or by starting your members in different current working directories.*

If you use the same `diskDir` specification for a dynamic region factory in multiple members, the first member succeeds in creating the factory, but subsequent members fail with a `DiskAccessException` saying that "`The file...is being used by another process.`"

poolName

This is used to pass dynamic region meta-information between this member and a server in a client/server installation. The default value is `null`. This property must be `null` for any non-client process. It must be set a valid pool for clients, with the proper server and configuration information. For information on configuring the pool, see client/server caching in the *GemFire Enterprise Developer's Guide*.

> *The* `poolName` *provided for this dynamic region factory must be configured with* `queue-enabled` *set to* `true`*.*

This specification is not applied to your dynamic regions. The pool for a dynamic region, if any, is inherited from its parent region.

persistBackup

A `true` setting for this option causes `/__DynamicRegions` to be persisted to disk. The default value is `true`. For general information on persistence, see `data-policy` on page 104 and *Using Disk for Backup and Overflow* on page 170.

If you persist the dynamic region factory data, all dynamic regions that were present when the factory was last closed are automatically recreated when the factory is opened again. These creations fail if their parent regions are not present when the factory is created, at cache creation time.

> *If you persist your dynamic region factory, all non-dynamic parents to your dynamic regions must be declared in the application's* `cache.xml` *file.*

registerInterest

This is a boolean that applies only to clients in a client/server installation. A `true` setting causes every dynamic region created by the factory to register interest in all keys in the corresponding server cache region. A `false` setting causes no interest registration. The default value is `true`. If enabled, this uses the default `InterestResultPolicy` of `KEYS_VALUES`.

Client/Server Settings for Dynamic Regions

To configure a client for dynamic region management, define the dynamic region factory with the poolName set to point to a Pool instance. The pool is used to communicate dynamic region operations between the client and server.

Configure the server side with the standard configuration. If you disable `registerInterest` for any client, the client will not receive update notifications any of its dynamic region entries.

The Dynamic Regions Factory Listener

The dynamic region listener has `before` and `after` callbacks for the creation and destruction of dynamic regions in the local cache. `DynamicRegionListener` can be registered before open is called and before cache creation so that the listener will be called if dynamic Regions are created during cache creation.

You can register any number of `DynamicRegionListener` instances against your factory. To get all region callbacks, you should register the listener before opening the factory.

You can program the callbacks to do anything your application requires. Be aware, however, that the calling process waits for the listener to respond. If your listener has a lot of work to perform, spawn a separate thread to perform the work and allow the main thread to return.

Configuring the Factory

You can configure the dynamic region factory through the API or through the `cache.xml` file. If you use the API, you must create the factory before creating the cache.

> *You should use either the API or the* `cache.xml` *file. The* `cache.xml` *declaration overrides any configuration done through the API.*

cache.xml

You can configure your cache for dynamic regions through the `cache.xml` file by adding a dynamic region factory to the cache element. The `cache.xml` file section in This example shows a basic dynamic region factory declaration. The factory uses a non-default disk directory.

Example 4.20 Simple Dynamic Regions Factory Declaration

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC "-//GemStone Systems, Inc.//GemFire Declarative
Caching 6.5//EN" "http://www.gemstone.com/dtd/cache6_5.dtd">
<cache>
   <dynamic-region-factory disable-persist-backup="false">
      <disk-dir>/lucy/users/testSystem2/dynamicRegionMetaData</disk-dir>
   </dynamic-region-factory>
      . . .
</cache>
```

With the dynamic region factory declared in the `cache.xml` file, the program can create the cache and then retrieve the dynamic region factory with the method:

```
DynamicRegionFactory dynRegFactory = DynamicRegionFactory.get();
```

API

This example creates a dynamic region factory with the default configuration:

Example 4.21 Dynamic Regions Factory Creation Through the API

```
// Connect to the GemFire distributed system
   ...
DistributedSystem ds = DistributedSystem.connect(props);

// Configure the dynamic region factory
DynamicRegionFactory dynRegFactory =
      DynamicRegionFactory.get().open();

// Create the cache
Cache cache = CacheFactory.create(ds);
```

If the `DynamicRegionFactory open` method is called before cache creation and the `cache.xml` has a `dynamic-region-factory` element, the `cache.xml` overrides the `open` call's configuration.

Activating the Factory

Cache creation activates the configured dynamic region factory, making it ready to receive distributed events from remote factories. At this time, the dynamic region factory automatically creates any dynamic regions that are already defined through persistence or in remote caches.

You must have parent regions defined in your `cache.xml` file for all regions that might be created automatically at dynamic region factory activation. Automatic region creation can come from two sources:

- When you activate a persistent dynamic region factory (`Config.persistBackup` set to `true`) the persisted dynamic region meta-information is loaded and those regions created. The parents to those regions must already exist in the cache or the load will fail.
- When your dynamic region factory is activated, all remote factories send their dynamic region meta-information to it. This creation requires parent regions in the local cache.

Creating and Destroying Dynamic Regions

Once you've opened the dynamic region factory and activated it by creating your cache, you can use it to create and destroy your dynamic regions with the methods, `DynamicRegionFactory.createDynamicRegion` and `DynamicRegionFactory.destroyDynamicRegion`.

Do not create or destroy dynamic regions with the standard `Region` *methods.*

The create method takes a parent region and the new, dynamic region name. Destruction takes the fully qualified name of the region to be destroyed.

So you would create the region `/expenses/travel/flights` by calling:

```
createDynamicRegion(/expenses/travel, flights);
```

And you would destroy the same region by calling:

```
destroyDynamicRegion(/expenses/travel/flights);
```

The creation and destruction are distributed to all connected members with a dynamic region factory. There is no way to know whether the dynamic region creation or destruction succeeded in a remote cache.

You cannot create dynamic regions in your `cache.xml` file. Dynamic regions must be created using the API.

Dynamic Region Attributes

When a dynamic region is created, it is given the attributes of its local parent region. So for the dynamic region, `/expenses/travel/flights`, all attributes assigned to `/expenses/travel` are assigned to`/expenses/travel/flights`.

User-defined application plug-ins for the parent region are assigned to the new region *by reference*, rather than by creating new instances of the plug-ins.

> *To use dynamically-created regions, you must ensure thread safety in their parent regions' application plug-ins (*`CacheLoader`*,* `CacheWriter`*, and* `CacheListener`*). Additionally, you must ensure that the plug-ins'* close *implementations do not stop them from functioning.*

The GemFire classes provided for region attributes use are safe for dynamic regions.

Chapter 5

Data Operations

This chapter covers the mechanics of cached data storage and retrieval in GemFire Enterprise®. In the GemFire cache, your application data is stored in key/value pairs as data region entries. When an entry is created, a new object is instantiated in the region containing:

- **The entry key**—This key may be constrained to a specific object type with the region's `KeyConstraint` attribute.
- **The entry value**—This is the application data object. The entry value may be set to `null` (equivalent to an invalid value). It may also be constrained by the region's `ValueConstraint` attribute.

In addition to the key and value, the entry can provide a statistics object (if statistics are enabled for the region) and can set and retrieve a user-defined attribute.

A cache region data entry is created, updated, invalidated and destroyed through the following:

- API calls made through the `Region` and `Entry` interfaces
- Automated data management based on region attribute settings such as eviction, cache loaders and writers, replication, and expiration (see *Region Attributes List* on page 101)
- Operations distributed from other caches

In this chapter:

- The Entry Interfaces and Classes (page 130)
- Adding an Entry to the Cache (page 131)
- Accessing an Entry (page 135)
- Updating an Entry (page 140)
- Invalidating and Destroying an Entry (page 143)

5.1 The Entry Interfaces and Classes

This section gives an overview of the primary entry management API for GemFire Java. For complete information on the classes and interfaces described here, see the online Java API documentation.

com.gemstone.gemfire.cache

- **Region**—This interface provides methods for managing the cached data stored inside the region. The methods in this interface provide local and distributed entry operations. The methods allow you to do the following entry management functions:
  - Region operations that affect entries, such as `regionDestroy` and `regionClear`.
  - Create, update, invalidate and destroy region entries.
  - Query, individually or as entire sets, the region's entry keys, entry values and `Region.Entry` objects. See Chapter 13, *Querying and Indexing*, on page 341.
  - Obtain a distributed lock on a `Region.Entry`. This is available only for distributed regions with `global` scope.
  - Manage your entries through a familiar `Map` interface. The GemFire `Region` interface extends the `java.util.Map` interface, which provides a standard set of map methods for basic data management in addition to the specialized GemFire Enterprise cache methods. For information on region-specific implementation of `Map` APIs, see the GemFire online Java API documentation at `docs/index.html`. The interface `Map` specification is also available on the Sun web site at http://java.sun.com/j2se/1.5.0/docs/api/java/util/Map.html.

    CAUTION: Great care must be exercised if mutable objects are used as entry keys. The behavior of a `Region` *is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the* `Region`*. A special case of this prohibition is that it is not permissible for a* `Region` *to contain itself as a key. While a* `Region` *can contain itself as a value, extreme caution is advised. The* `equals` *and* `hashCode` *methods are no longer well defined on such a* `Region`*.*

  For partitioned regions, `Region` methods that are distributed operations work on all the buckets that make up the partitioned region, whether in the local member or elsewhere. Of the `Region` methods that are normally local operations, only `close` and `localDestroyRegion` remain local in partitioned regions. Other methods become distributed, and some are not supported by partitioned regions. Some of these differences are described in this chapter. For the rest, check the Java API documentation for each method.
- **Region.Entry**—This extends the familiar `java.util.Map.Entry` interface. It contains the key and value for the entry and provides local and distributed entry operations. Except for `setValue`, this object's operations are not distributed, do not acquire any locks, and do not affect statistics.
- **CacheEvent**—This interface defines common event methods; it is extended by `EntryEvent`.
- **EntryEvent**—This interface extends `CacheEvent` for entry events. It provides all of the information available for the `EntryEvent` and also provides entry-specific information such as the old and new entry values. It includes methods for retrieving serialized olds and new values.
- **CacheStatistics**—This interface defines common statistics methods. `Region.Entry` has a method that returns a `CacheStatistics` object for accessing and resetting statistics counts.

For information on creating your own classes for data objects and keys, see *Defining Your Own Custom Classes* on page 57.

5.2 Adding an Entry to the Cache

To store data in your local cache, use a region `refid` with a `RegionShortcut` or `ClientRegionShortcut` that has local state. Under the covers, these set the region `data-policy` (page 104) to a non-`empty` policy. The regions without storage can send and receive event distributions without storing anything in your VM heap. With the other settings, all entry operations received are stored locally. For more information on event management, see Chapter 11, *Handling Events*, on page 293.

A region can be populated with cached entries in a number of ways.

- Declaratively, as `cache.xml` file declarations that are loaded at cache creation time. For examples, see *Creating Entries in XML* on page 132.
- Programmatically, when an application executes a `create` or a `put` operation for an entry that does not already exist in the cache. For examples, see *Creating Entries Through the API* on page 134 and *Updating an Entry* on page 140.
- Implicitly, when an application `gets` an entry that does not already exist in the cache. In this case, the entry value is automatically retrieved from remote caches or through loading operations. See *The Distributed Region.get Operation* on page 135 and *Loading Entries Through a Data Loader* on page 196.
- Automatically, for replicate or partitioned regions or with `data-policy` of `normal` and a `subscription-attributes` (page 118) `interest-policy` of `all`. With any of these configurations, when an entry is created anywhere in the distributed region, it is automatically distributed to the local region as well. For replicates and preloaded regions, the region is automatically initialized at creation time from the entire distributed region. With a persistent data-policy, the region might be initialized from the data stored on disk or from the distributed region.

Regardless of how an entry is created, if any `cache-writer` (page 103) is available in the distributed region, it is called and may abort the creation process. If the region does not store data in the local cache, data added may anyway be forwarded to the rest of the distributed cache.

The rest of this section covers explicit entry creation in the following:

- Creating Entries in XML (page 132)
- Creating Entries Through the API (page 134)

Creating Entries in XML

Entry keys and values created through the `cache.xml` file must either be `String` or must be a user-defined, declarable object type. This `cache.xml` file snippet creates an entry with `String` key and value:

Example 5.1 Simple Declarative Entry Creation

```
<region name="sampleRegion">
   <region-attributes statistics-enabled="true" scope="distributed-ack">
   </region-attributes>
   <entry>
      <key><string>MyKey</string></key>
      <value><string>MyValue</string></value>
   </entry>
</region>
```

Application-defined classes must implement the `com.gemstone.gemfire.cache.Declarable` interface and the object must be declared within a `declarable` element. This is a simple example of the use of an application-defined object type:

Example 5.2 Simple Declarative Entry Creation with Application-Defined Object Type

```
<region name="sampleRegion">
   <region-attributes statistics-enabled="true" scope="distributed-ack">
   </region-attributes>
   <entry>
      <key><string>MyKey</string></key>
      <value>
         <declarable>
            <class-name>com.company.data.MyDeclarable</class-name>
         </declarable>
      </value>
   </entry>
</region>
```

The application `MyDeclarable.java` must include these lines:

```
import com.gemstone.gemfire.cache.Declarable;
. . .
public class MyDeclarable implements Declarable . . .
```

The next example is a little more complex. The declarations listed are for a `Portfolio` class that contains a `Map` of `Position` objects, so the file has nested `declarable` elements. The initialization of `Portfolio` and `Position` take parameters, which are provided by the XML.

Example 5.3 Declarative Entry Creation with Nested Application-Defined Object Types

```
<entry>
   <key><string>1</string></key>
   <value>
      <declarable>
         <class-name>cacheRunner.Portfolio</class-name>
         <parameter name="id">
            <string>1</string>
         </parameter>
         <parameter name="type">
            <string>type1</string>
         </parameter>
         <parameter name="status">
            <string>active</string>
         </parameter>
         <parameter name="position1">
            <declarable>
               <class-name>cacheRunner.Position</class-name>
               <parameter name="secId">
                  <string>SUN</string>
               </parameter>
               <parameter name="qty">
                  <string>34000</string>
               </parameter>
               <parameter name="mktValue">
                  <string>24.42</string>
               </parameter>
            </declarable>
         </parameter>
         <parameter name="position2">
            <declarable>
               <class-name>cacheRunner.Position</class-name>
               <parameter name="secId">
                  <string>IBM</string>
               </parameter>
               <parameter name="qty">
                  <string>8765</string>
               </parameter>
               <parameter name="mktValue">
                  <string>34.59</string>
               </parameter>
            </declarable>
         </parameter>
      </declarable>
   </value>
</entry>
```

Creating Entries Through the API

Explicit programmatic entry creation may be performed with just the entry key (and a `null` value), or with a valid key and value. Adding a new entry key into the cache with a `null` value creates an invalid, cached entry. This example shows the creation of an entry with a value.

Example 5.4 Programmatic Entry Creation

```
String name = ...
String value = ...
this.region.create(name, value);
```

You can provide a third parameter to the `Region.create` method. This parameter is passed as-is as a callback argument to listeners, writers, and other event handlers. See *Handling Events* on page 293.

5.3 Accessing an Entry

The caching API provides a number of methods for the retrieval of the entry keys and values. The methods provide individual entry access, including access to the `Region.Entry` object itself, and setter operations that return information on all data available in the region. For a complete listing of all available access methods, please refer to the API documentation.

If you are accessing an entry for update, read Safe Entry Modification on page 140.

This section covers the following:

- The Distributed Region.get Operation (page 135)
- Batching get Operations (page 137)
- Accessing Only the Local Cache (page 138)
- Getting Different Object Types From the Cache (page 139)

The Distributed Region.get Operation

The standard region entry `get` method, `Region.get`, considers all caches and all applicable loaders in the distributed system in an attempt to return a valid entry value to the calling application. The primary region attribute settings affecting this entry retrieval are `scope` (page 116), `data-policy` (page 104), and `cache-loader` (page 102).

This example shows the use of the `Region.get` method.

Example 5.5 Entry Value Retrieval Using get

```
void myGet(String command) throws CacheException {
    String name = parseName(command);
    if (name != null) {
        Object valueBytes = this.currRegion.get(name);
        printEntry(name, valueBytes);
    }
}
```

`Region.get` calls a number of operations in order to retrieve a valid entry value. The operations used depend on the region's attribute settings—in particular the scope setting if the region is non-partitioned—and on the state of the entry itself.

Partitioned Region

In a partitioned region, the entry value is fetched from any member hosting a copy of the entry, with the local cache preferred. So sometimes the fetch is done from the primary and sometimes from a redundant copy.

If the region is persisted to disk, and the only available entry is on disk, the `get` operation returns a `PartitionOfflineException`. This situation occurs when every member hosting the entry is offline. To retrieve the entry, you must either start the member that hosts the data, which allows the system to access the disk data, or remove the data from the system by revoking the offline member from the disk store. See *Handling Missing Disk Stores* on page 120 of the *GemFire Enterprise System Administrator's Guide*.

If the entry is not present in any member, in memory or on disk, or has an invalid value, and a cache loader is configured, GemFire loads the entry into the cache.

Distributed Region

In a distributed region, the entry value is first requested from the local cache. If it is not present or has an invalid value, GemFire retrieves the value using a combination of the locally-defined cache loader (if one exists), and the `netSearch` and `netLoad` operations.

Local Region

In a region with scope set to `local`, the entry value is either retrieved from the local cache or fetched by the region's locally-defined cache loader.

netSearch

The `netSearch` operation searches for the entry's value throughout the distributed region.

> *Because expiration attributes may vary between caches, before retrieving a value from a remote cache,* `netSearch` *applies the local cache's expiration attributes. Entries that would have already expired in the local cache are passed over.*

The `netSearch` operation is available publicly as a method of the `LoaderHelper` interface in `com.gemstone.gemfire.cache`. The method accepts a `doNetLoad` boolean parameter which, if true, causes automatic invocation of `netLoad` if the search fails to retrieve a value.

The `netSearch` operation is not used in replicated regions, except when called explicitly through the API. This is because the replicated region already contains all distributed region data.

Cache Loader

A cache loader is an application plug-in used to load data into the region on cache misses. You can use the loader to provide read-through functionality for your region data. When an entry is requested that is unavailable in the region, a cache loader may be called upon to load it. A cache loader is generally used to retrieve data from an outside source such as a database, but it may perform any operation required by your application.

The `com.gemstone.gemfire.cache.CacheLoader` interface provides one method, `load`, for customizing entry loading. It takes `LoaderHelper` in input. Among other things, the helper class allows direct invocation of `netSearch` to aid in finding the entry value.

A distributed region may have cache loaders defined in any or all caches where the region is defined. When loading an entry value, a locally defined cache loader is always used before a remote loader. Remote loaders are run using the `netLoad` operation described below.

In addition, the `netSearch` operation, available through the `CacheLoader`'s `LoaderHelper` parameter, can be instructed to run `netLoad` automatically if the search fails to find a value.

netLoad

The `netLoad` operation runs all remotely defined cache loaders for the distributed region until the entry value is returned or all loaders have been tried. The `netLoad` operation cannot be directly invoked. It may be invoked implicitly through the `netSearch` operation, as described above, and is automatically invoked for entry value retrieval as outlined in this table.

Scope and Entry Value Retrieval

For non-partitioned regions, the automated ordering of the entry retrieval processes varies according to the region's scope, as shown in this table.

Table 5.1 Automated Steps in Entry Retrieval Based on Region Scope - non-Partitioned

Region Scope Attribute Setting		
`local`	**`distributed-no-ack`** or **`distributed-ack`**	**`global`**
1. local cache search 2. server pool* 3. local load	1. local cache search 2. server pool* 3. local load 4. netSearch** 5. netLoad	1. local cache search 2. server pool* 3. netSearch** 4. local load 5. netLoad

*server pool used only for client regions.

`**netSearch` is not used for replicated regions.

For regions with `global` scope, the entry value is requested from all remote caches before any loaders are run. For non-global distributed regions, the local load is favored over remote cache values.

The distributed system only allows `netSearch` and `netLoad` to be run once for any `get` operation.

How the get Operation Affects the Local Entry Value

For any distributed region that stores data in the local cache, when a `get` operation retrieves an entry value from outside the local cache, it automatically `put`s the value into the local cache for future reference. If the entry is obtained via a `netSearch`, the value is `put` without calling any cache writer. If the entry is obtained through any other means, however, the `put` invokes a `netWrite` that can, in turn, abort the `put` operation. Regardless of the outcome of the local `put` operation, the `get` always returns the entry value to the calling application. In this way, it is possible for a value that is returned from a `get` to remain unavailable in the local cache even when the region is configured to store data locally.

With the `empty` data policy, the `get` operation never modifies the local cache. It always retrieves from outside the local cache and never stores the result locally.

With partitioned regions, the `get` operation never modifies the local cache.

Batching get Operations

The `Region.getAll` method gets values for all keys in a specified collection. The method returns a map of values for the input keys. For details about using `getAll`, see the Java API documentation for `Region`.

Accessing Only the Local Cache

The main entry retrieval method, `Region.get`, does everything it can to return a value to the calling process. If the value is not in the local cache, it accesses all available loaders and remote region instances before returning empty-handed. You may sometimes want to access only the local cache. You can do this by using the objects and methods described in this section.

Retrieving Local Values in Distributed Regions

A region's entry keys and `Region.Entry` objects are obtained directly from the local cache. In addition, you can directly access the local cache's stored entry value through the `Region.Entry.getValue` method. In distributed regions, the `getValue` method returns the value if a valid value is present in the local cache, or `null` if the value is not valid locally. This method does no data loading in a distributed region, nor does it look elsewhere in the region for a valid value.

The code in this example accesses the `Region.Entry` object. This code iterates over the set of `Region.Entry`s for each entry's `key` and `value`.

Example 5.6 Retrieving a Distributed Region's Entries from the Local Cache Only

```
Set nameSet = this.currRegion.entrySet();
for (Iterator itr = nameSet.iterator(); itr.hasNext();) {
   Region.Entry entry = (Region.Entry) itr.next();
   String name = (String)entry.getKey();
   Object valueBytes = entry.getValue();
// Call method to print the entry key and value to screen
   printEntry(name, valueBytes);
   System.out.println("\t\t" + this.currRegion.get(name));
}
```

If you wanted the code in this example to retrieve the value from the distributed system when it isn't present in the local cache, you could rewrite it to retrieve the set of entry keys via the `Region.keySet` method and iterate over it, retrieving each entry's value through the `Region.get` method and retrieving entry statistics from `Region.Entry`.

In partitioned regions, `Region.Entry.getValue` returns a valid value found anywhere in the region. You cannot fetch just local values, but you can identify which keys are local by using `Region.Entry.isLocal` along with `getValue`. Since partitioned regions can be very large, iterating over the entire entry set to discover what is held in primary storage in the local cache can be expensive.

Getting Different Object Types From the Cache

The method listed in this example interprets the object value and prints the key/value pair. The method interprets values as byte array, `String`, `Integer`, or a non-trivial object type, `ExampleObject`.

Example 5.7 Deciphering the Entry Value (byte array, object type...)

```
void printEntry(String key, Object valueBytes) {
    String value = "null";
    if (valueBytes != null) {
        if (valueBytes instanceof byte[]) {
            value = "byte[]: \"" + new String((byte[]) valueBytes) + "\"";
        }
        else if (valueBytes instanceof String) {
            value = "String: \"" + valueBytes + "\"";
        }
        else if (valueBytes instanceof Integer) {
            value = "Integer: \"" + String.valueOf(valueBytes) + "\"";
        }
        else {
            value = String.valueOf(valueBytes);
        }
    } // null CHECK
    else {
        value = "No value in cache.";
    }
    System.out.println(" " + key + " -> " + value);
}
```

5.4 Updating an Entry

A cached entry may be updated in several ways:

- Explicitly, when the application invokes a `put` operation on an existing entry.
- Implicitly, when a `get` is performed on an entry with no value in the cache. Note that even a successful `get` may not result in an update to the local cached entry. For information about this, see *How the get Operation Affects the Local Entry Value* on page 137.
- Automatically, when a new entry value is distributed from another cache.

As with entry creation, all of these operations can be aborted by a cache writer.

`Region.put` creates an entry if the specified key is not present in the cache. If the key exists, its value is updated. So you could use `put` for entry creation when you don't care if the entry already exists. If you want the creation to fail if the entry is already present in the cache, use the `create` method instead. As with the `create` method, `put` accepts a callback argument to be passed to any applicable listener invocations.

Example 5.8 Entry Modification

```
String name = .. // entry key
String value = .. // entry value
this.currRegion.put(name,value);
```

Safe Entry Modification

When you get an entry value from the cache, by default, the retrieval methods return a direct reference to the cached object. This provides the value as quickly as possible, but also opens the cache to direct, in-place changes.

> *Do not directly modify cached values. Modifying a value in place bypasses the GemFire Enterprise distribution framework, including cache writers and listeners, expiration activities, and transaction management, and can produce undesired results.*

Always change your entries using copies of the retrieved objects—never directly modify the returned objects. You can do this in one of two ways:

1. Change the entry retrieval behavior for your cache by setting the `cache` attribute, `copy-on-read` (page 74), to `true` (the default is `false`).

   ```
   <cache copy-on-read="true">
   </cache>
   ```

   When `copy-on-read` is `true`, the entry access methods return copies of the entries. This protects you from inadvertently modifying in-place, but negatively impacts performance and memory consumption when copying is not needed.

   These entry access methods return an entry reference if `copy-on-read` is `false` and a copy of the entry if `copy-on-read` is `true`:

   - `Region.get`
   - `Region.values`
   - `Query.select`
   - result of `Region.put`
   - `Region.Entry.getValue`
   - `EntryEvent.getNewValue`
   - `EntryEvent.getOldValue`

2. Create a copy of the returned object and work with that. For objects that are cloneable or serializable, you can copy the entry value to a new object using `com.gemstone.gemfire.CopyHelper.copy`.

Example 5.9 Making a Safe Change Using CopyHelper.copy

```
Object o = (StringBuffer)region.get("stringBuf");
StringBuffer s = (StringBuffer) CopyHelper.copy(o);
s.append("Changes to value, added using put.");
region.put("stringBuf", s);
```

Example Comparison of Cache Modification Inside a Transaction

This section gives an example of an incorrect way and correct way to change an entry inside a transaction. In a transaction, any direct change to stored region values is immediately visible outside of the transaction.

This listing shows what *not* to do:

Example 5.10 Do Not Make Changes Like This To Your Cached Entries

```
   Region<Integer, StringBuilder> r1 = ...;
   CacheTransactionManager txmgr = ...;
   try {
      txmgr.begin();
      Integer key = Integer.valueOf(1);
// DO NOT DO THIS!!!  This change is not isolated from other transactions!
      StringBuilder v = r1.get(key);
      v.append(" added stuff");
// DO NOT DO THIS!!!  This change is not isolated from other transactions!
      r1.put(key, v);
      txmgr.commit();
   } catch (CommitConflictException conflict) {
   txmgr.rollback();
   }
```

This is one correct way to make the change:

Example 5.11 Alternative Way To Change a Cache Entry

```
Region<Integer, StringBuilder> r1 = ...;
CacheTransactionManager txmgr = ...;
try {
   txmgr.begin();
   Integer key = Integer.valueOf(1);
   StringBuilder v = r1.get(key);  // assume v is not null
   StringBuilder newV;
   synchronized (v) {
      newV = new StringBuilder(v.toString());
   }
   newV.append(" added stuff");
   r1.put(key, newV);
   txmgr.commit();
} catch (CommitConflictException conflict) {
   txmgr.rollback();
}
```

There are two important improvements in the second example:

1. The new value is `put` into the `Region`, so it will will be staged in the transactional context and committed if there are no conflicts.
2. The code now synchronizes on the `StringBuilder` call, ensuring a consistent view of the contents when making the copy. The `StringBuilder` interface itself is not synchronized in this example—you would provide synchronization as needed.

This code synchronizes on the instance for protection, so writes must observe the same usage.

Behavior of the put Operation in Partitioned Regions

In a partitioned region, `Region.put` creates an entry only when the specified key is not present in any member that participates in the partitioned region. When an entry is created, it may be placed in the local cache or in a remote member cache. For more information, see *Partitioned Data Caching* on page 46.

Batching put Operations

Using `Region.putAll`, you can batch up multiple `put` operations into the cache in a single distributed operation without needing to handle multiple round trips for message distribution. A `putAll` message consists of a map that can contain many key-value entry pairs, with some consisting of `create` events and others as `update` or other events. When the messages are received they are individually inserted into the cache and processed for interest registrations. Map entries from `putAll` retain their original order when they are processed by the cache server.

Multiple events are sent by `putAll` as a single message to the primary buckets for a partitioned region, then the events are distributed to the secondary buckets.

When processing very large maps, system performance and the possibility of cache update timeouts may need to be considered. One situation that may affect performance is if the overflow feature for high availability is enabled, since it involves writing to and reading from the disk. Timeouts may occur if many large entries are added to the cache using `putAll`.

This example demonstrates `Region.putAll`.

Example 5.12 Using putAll

```
LinkedList list = new LinkedList();
parseCommand(command, list);
String name = (String) list.get(1);
String size_str = (String) list.get(2);
int size = Integer.valueOf(size_str).intValue();
int start_value = 0;
if (list.size() > 3) {
   String start_str = (String) list.get(3);
   start_value = Integer.valueOf(start_str).intValue();
}
Map map = new LinkedHashMap();
for (int i=0; i<size; i++) {
   map.put(name+i, new ExampleObject(i+start_value));
}
this.currRegion.putAll(map);
```

5.5 Invalidating and Destroying an Entry

Entry invalidation sets the entry value to `null`. Entry destruction removes the entry from the region. These operations may be carried out directly in the local cache in the following ways:

- Through API calls
- Through the region's eviction controller
- Through expiration activities based on the entry's statistics and the region's attribute settings

Whether carried out explicitly or through expiration activities, destruction and invalidation cause event notification activities. The `CacheEvent` object that is sent has an `getOperation.isExpiration` flag set to `true` for events resulting from expiration activities and false otherwise.

This section covers the following:

- Local and Remote Operations (page 143)
- Destroy and Invalidate in Partitioned Regions (page 144)
- Determining an Entry's Status (page 144)

Local and Remote Operations

Invalidation and destruction may be performed as a local or a distributed operation. For replicate and partitioned regions, only the dsitributed operations are allowed:

- **Distributed destroy/remove**—Removes the entry completely from the cache and distributes the operation according to the region's distribution settings. This can be an explicit API call or a result of a region destroy or clear operation.
- **Local destroy/remove**—Removes the entry from the local cache but does not distribute the operation. This can be an explicit API call or a result of a local region destroy or clear operation. This action is not valid if the region is configured as a replicated or partitioned region.
- **Distributed invalidate**—Sets the entry value to `null` and distributes the operation according to the region's configuration. When an entry is invalid, a `get` causes the cache to retrieve the value according to the steps described on *The Distributed Region.get Operation* on page 135. This can be an explicit API call or a result of a region invalidate operation.
- **Local invalidate**—Sets the entry value to `null` in the local cache but does not distribute the operation. This can be an explicit API call or a result of a local region invalidate operation. Local entry invalidation is supported only for regions that are not configured as replicas. This action is not valid if the region is configured as a replicated or partitioned region.

A user-defined cache writer can abort an entry destroy operation. If a writer is defined for the region anywhere in the distributed system, it is invoked before an entry is explicitly destroyed.

This example shows local and distributed entry invalidation for an entry with key, `entryKey`.

Example 5.13 Local and Distributed Invalidation of Entries

```
/* Distributed Entry Invalidation */
   this.currRegion.invalidate(entryKey);

/* Local Entry Invalidation */
   this.currRegion.localInvalidate(entryKey);
```

This example shows remote and local entry destruction for the same entry.

Example 5.14 Local and Distributed Destruction of Entries

```
/* Distributed Entry Destruction */
   this.currRegion.destroy(entryKey);

/* Local Entry Destruction */
   this.currRegion.localDestroy(entryKey);
```

Destroy and Invalidate in Partitioned Regions

For partitioned regions, a distributed invalidate or destroy of an entry through `Region.invalidate(key)` or `Region.destroy(key)` affects every copy of the entry with the specified key in all the members hosting the partitioned region. You cannot do a local invalidate or local destroy of an entry.

`Region.remove(key)` removes the mapping for every instance of the specified key in all the members hosting the partitioned region.

Determining an Entry's Status

You can tell whether an entry has been invalidated or destroyed from the `Region` and `Region.Entry` objects.

- If the entry has been destroyed in the local cache:
  - `Region.containsKey` returns `false` for the entry's key.
  - `Region.Entry.isDestroyed` returns `true`.
  - `Region.Entry.getValue` returns an `EntryDestroyedException`.
- If the entry is invalid in the local cache:
  - `Region.containsValueForKey` returns `false` for the entry's key.
  - `Region.Entry.getValue` returns `null`.

When using `getValue` *in partitioned regions, to be assured of getting the entry's current status, call* `getEntry` *each time before calling* `getValue`*.Calling* `getValue` *on entries that are not in the local cache produces a snapshot of the state of the entry. The content of such an entry will not change to reflect modifications in other processes.*

Chapter 6

Configuration and Management by Region Type

This chapter covers the GemFire region management options that are specific to the high-level region configuration types, partitioned, replicated, distributed, and local.

For information on the basics of GemFire data region management, including your general options for region configuration, how to create and remove regions from your cache, and a full listing of region attributes settings, see Chapter 4, *Data Regions*, on page 85.

All region types use the same configuration and management of features such as disk persistence, eviction and expiration, event handling, data loading from an outside data source, and memory management. For detailed information on these common features, see Chapter 7, *General Region Management Options*, on page 169.

In this chapter:

- Partitioned Regions (page 146)
- Replicated Regions and Other Distributed Regions (page 161)
- Local Regions (page 168)

6.1 Partitioned Regions

This section covers GemFire data partitioning.

In this section:

- *Managing Partitioned Regions* on page 146
- *Rebalancing Partitioned Regions* on page 150
- *Partitioned Regions with Redundancy* on page 152
- *Custom Partitioning and Data Colocation* on page 157

Managing Partitioned Regions

Partitioned regions handle large volumes of data across many members. GemFire automatically partitions the data in a partitioned region across all available host members.

These are the main aspects of partitioned region behavior and management:

- You can configure high availability for your partitioned regions by using in-memory redundancy and by persisting the region to disk. You must persist all members hosting data. Everywhere you configure your region partitioned with data storage, configure it as persistent as well.
- Partitioned regions never run asynchronously. If you use redundancy, operations in partitioned regions always wait for acknowledgement from the caches containing the original data entry and any redundant copies.
- You must install a cache loader in every cache where the partitioned region is configured to host data (`local-max-memory` > 0). See *Plug-ins in a Partitioned Region* on page 193.
- Data in partitioned regions is managed in buckets of entries. GemFire may transfer buckets from one host member to another for load balancing. See *Rebalancing Partitioned Regions* on page 150**.**
- You do not control in which members your entries are stored. You can, however, group entries together into buckets using a custom partitioning resolver. You can also group entries from multiple regions, so the bucket for one data set in one region and the bucket for a corresponding data set in another region are kept together in a single member. For example, you might group a set of customers in one region, group those customers' orders in another region, and group the data from the two regions together. In this way, for a particular customer or set of customers, you can have all customer information and all customer order information hosted by the same member. See *Custom Partitioning and Data Colocation* on page 157.

Deciding on the Number of Buckets for Your Region

GemFire distributes buckets as evenly as possible across all members hosting the partitioned region. The number of buckets allotted for the partitioned region determines the granularity of data storage and thus how evenly the data can be distributed. The partition attribute `total-num-buckets` (page 115) controls the total for the entire partitioned region across all the participating members.

For even distribution, `total-num-buckets` should be a prime number. This value must be larger than the total number of members hosting the region, as a rough guideline, at least four times as many buckets as members. When you plan the number of buckets, you need to consider the trade-off between load balancing and overhead, however. Managing a bucket introduces significant overhead, especially with higher levels of redundancy.

The larger the ratio of buckets to host members, the more evenly the load can be spread across the members. You are trying to avoid the situation where some members have more data entries than others. For example, compare the next two figures. This figure shows a region with three host members and seven buckets. If all the entries are accessed at about the same rate, this configuration creates a hot spot in the distributed system. Member M_3 is likely to be a slow receiver and potential point of failure.

Figure 6.1 Uneven Data Partitioning

Configuring more buckets gives you fewer entries in a bucket and a more balanced data distribution. This figure keeps the same members and data as before but increases the number of buckets to 13. Now the data entries are distributed evenly.

Figure 6.2 More Balanced Data Partitioning

You can also initiate a rebalancing of partitioned region buckets to optimize data distribution. See *Rebalancing Partitioned Regions* on page 150 for more information.

Defining a Partitioned Region With No Local Data Storage

A partitioned region can be defined with no local data storage for some members. A member that defines the region in this way can do all the usual operations on the partitioned region without the overhead of data caching. GemFire's automatic data partitioning handles the caching, and it is not visible at the application level. This type of member does not affect the load balancing between the data host members.

> *Create the partitioned region with data storage in at least one member before you create the region in the member with no local data storage.*

To create a partitioned region without local data storage, set the `local-max-memory` partition attribute to 0:

```
<partition-attributes local-max-memory="0" />
```

In this figure, member M_1 has no local data storage, so GemFire has distributed all the data to M_2 and M_3. M_1 has the same logical view of the partitioned region as the other members, so it can manage data and handle events in exactly the same way.

Figure 6.3 Partitioned Region Including a Member With No Local Data Storage

Rebalancing Partitioned Regions

Rebalancing migrates data buckets to maintain a fair balance of data and behavior distribution across the distributed system. In a distributed system running with minimal contention to the concurrent threads reading or updating from the members, rebalancing is used to dynamically increase or decrease the data and processing capacity.

Rebalancing is used to even out the load in these situations:

- Data host members join or depart the system. You can manually invoke a rebalance and you can also configure the system to automatically rebalance for you.
- You have configured your system to require manual redundancy satisfaction. If you set the `startup-recovery-delay` and `recovery-delay` attributes on a partitioned region to `-1`, GemFire does not restore redundancy after a data host fails or after a new host joins, until you invoke a rebalance. For more information about these attributes, see *Redundancy Recovery* on page 154.
- You have uneven hashing of data. Uneven hashing can occur if your keys do not have a hashcode method which ensures uniform distribution, or if you choose custom partitioning (using a `PartitionResolver` to colocate data). In either case, some buckets may receive more data than others. Rebalancing can be used to even out the load between host members by putting fewer buckets on members that are hosting large buckets.

You manually invoke a rebalance operation on a specific host through the GemFire API. The member where rebalancing is invoked directs the operation by instructing other members to move buckets. The operation rebalances every partitioned region defined in the member where rebalancing is invoked, even if that member does not host data for that partitioned region. Each partitioned region is rebalanced separately.

> *You can continue to use your partitioned regions normally while rebalancing is in progress.* `Read` *operations,* `write` *operations, and function executions continue while data is moving. If a function is executing on a local data set, you may see a performance degradation if that data moves to another node during function execution. Future function invocations will be routed to the correct member.*

In this section:

- Simulating Partitioned Region Rebalancing (page 150)
- Invoking Partitioned Region Rebalancing (page 151)
- How Partitioned Region Rebalancing Works (page 151)

Simulating Partitioned Region Rebalancing

You can simulate the rebalancing operation to analyze its results before actually executing the rebalance, as shown in the following API code snippet:

Example 6.1 Simulating Partitioned Region Rebalancing for Analysis

```
ResourceManager manager = cache.getResourceManager();
RebalanceOperation op = manager.createRebalanceFactory().simulate();
RebalanceResults results = op.getResults();
System.out.println("Rebalance would transfer " +
results.getTotalBucketTransferBytes() +" bytes ");
System.out.println(" and create " + results.getTotalBucketCreatesCompleted()
+ " buckets.\n");
```

Invoking Partitioned Region Rebalancing

Partitioned region rebalancing can be invoked when you start a cache server by passing a `rebalance` command-line argument, as shown in the following example:

Example 6.2 Invoking Partitioned Region Rebalancing at Cache Server Startup

```
bin/cacheserver start -rebalance
```

You can invoke partitioned region rebalancing programmatically using the API.

> *For efficiency, when starting multiple data hosts, add all the hosts and then trigger the* `rebalance` *operation once.*

Example 6.3 Invoking Partitioned Region Rebalancing Programmatically

```
ResourceManager manager = cache.getResourceManager();
RebalanceOperation op = manager.createRebalanceFactory().start();
//Wait until the rebalance is complete
RebalanceResults results = op.getResults();
System.out.println("Took " + results.getTotalTime() + " milliseconds\n");
System.out.println("Transfered " + results.getTotalBucketTransferBytes()+
"bytes\n");
```

How Partitioned Region Rebalancing Works

For each partitioned region, the rebalancing operation first tries to restore redundancy for any buckets that are not at the configured redundancy level. For details about restoring redundancy, see *Redundancy Recovery* on page 154.

Then GemFire performs the relancing work, to evenly distribute the data over all data hosts.

> *The rebalance operation only works on the partitioned regions defined in the host where the rebalancing is invoked. It may move data to or from any of the members in the distributed system that host the same partitioned region.*

GemFire tries to ensure that each member has the same percentage of its available space used for each partitioned region. You configure the available space for each data host using the `local-max-memory` (page 114) setting. For example, if you configure a partitioned region in one member with `local-max-memory` set to twice that of the region in another member, the first member will receive roughly twice the data.

If `local-max-memory` is not set, the rebalancing capacity defaults to 90% of the VM maximum heap size. As a result, members with a higher maximum heap size receive more data. For details on configuring VM heap size, see *JVM Memory Settings* on page 141 of the *GemFire Enterprise System Administrator's Guide*.

Partitioned region rebalancing honors the following constraints:

1. Redundant copies of the same bucket are not placed on the same IP address, if it can be avoided.
2. The `local-max-memory` setting for a member is not exceeded during the rebalancing process.

   There is one exception to this. If the member has LRU eviction configured with overflow to disk, rebalancing is allowed to exceed the member's `local-max-memory` setting. Eviction and expiration information are not retained during bucket rebalancing, so `entry-time-to-live` and `entry-idle-time` statistics for entries are reset during the migration of partitioned data.

Rebalancing Colocated Partitioned Regions

Single, non-colocated partitioned regions are rebalanced one at a time. GemFire works on colocated regions as a group. Colocated buckets are treated like one bucket, with a load equal to the sum of the loads of the component buckets. The weight on each member is the sum of the weights for each colocated partitioned region.

The decision to move a colocated bucket is based on whether moving any of the component buckets make the partitioned region on that member exceed its `local-max-memory` threshold.

Partitioned Regions with Redundancy

Highly available partitioned regions provide reliability by maintaining redundant data. When you configure a partitioned region for redundancy, each entry in the region is stored in at least two members' caches. If one of its members fails, operations continue on the partitioned region with no interruption of service. Recovering redundancy can be configured to take place immediately, or delayed a configurable amount of time until a replacement system is started.

Without redundancy, failure of any of the members hosting the partitioned region results in partial loss of data. Typically redundancy is not used when applications can directly read from another data source, or when write performance out weighs read performance. The addition of a redundant copy ensures high availability of your data in the partitioned region.

Even with high availability, you can lose data if too many applications and cache servers fail at the same time. You can also lose access to all copies of your data through network failure. Any lost data is recreated by the application as it returns to active work. To estimate the number of members you should run for your partitioned region, see *Planning for Enough Members to Support Redundancy* on page 153.

In this section:

- Performance (page 152)
- Redundancy Recovery (page 154)
- Load Balancing With Redundant Partitioned Regions (page 156)

Performance

Redundancy requires GemFire to distribute the data to one or more backup copies and ensures that the copies have received the data before returning from a write operation (`put` or `create`). Redundant copies are not just standby nodes. They are active members of the distributed system. Every member of the distributed system that defines a region gets some primary copies and some redundant copies. Write operations always go to the designated primary and get distributed to the redundant node. However, read operations are directed to primary as well as redundant copies, allowing a read intensive system to scale much better and handle higher load.

From a performance standpoint, a GemFire cache server must:

- Allow writes and reads with the lowest latency to the applications that are using the cache server
- Accept subscriptions and fulfill them when any data changes pertaining to those subscriptions happen in the GemFire distributed system
- Provide high availability of data and a way to scale up the usage and data volume based on application needs

As discussed in the previous section, providing high availability requires `write` operations to be distributed to the redundant copies, which is more expensive than not distributing it (no redundancy). Handling subscriptions requires notifications to be sent to the subscribers who may be remote to the cache server where the data change occurred. Again, in this case, distributing the data change to a remote subscriber is more expensive than notifying a subscriber that is local to the cache server.

An application designer must assess the afore-mentioned factors as a part of the performance considerations for the system while configuring an application to use GemFire.

Planning for Enough Members to Support Redundancy

Most of the setup for redundancy is in figuring out what you need to provide adequate capacity for your region data. If you need more space at run time, you can start a new member host. New members are populated as described in *Recovery After Startup* on page 155.

Make sure you start enough members to provide the level of reliability you require. To calculate the number of members you need for redundant data storage, assuming that all your members are on separate machines, use this formula as a general guideline:

```
data_store_members = redundant_copies + 1 + concurrent_member_failures
```

where:

- `data_store_members =` members with caching enabled that participate in the partitioned region
- `redundant_copies =` value of the `redundant-copies` partition attribute
- `concurrent_member_failures =` number of inaccessible members that your partitioned region must be able to tolerate at any given time without data loss

> *This calculation tells you how many members with caching you need. If you are also using accessor members with caching disabled, those are additional.*

Use the calculation as the beginning point in your planning process, not the final answer. This formula doesn't always work. Where `redundant_copies` is less than `concurrent_member_failures`, for example, no number of members can guarantee that you won't have data loss.

When redundancy is compromised, `PartitionedRegionStorageException`s are logged at a warning level, and application threads receive `PartitionedRegionStorageException`s. If you begin seeing `PartitionedRegionStorageException`s, you may not have enough resources to support your redundant data configuration. Then you need to evaluate whether to start more members. Look at the overall installation, since this exception can also reflect conditions arising outside the partitioned region. For example, other operations at the application level could be set to a higher priority than the GemFire processes, or the network could be saturated by traffic from other sources.

Configuring Redundancy

Set the partition attribute `redundant-copies` (page 114) to the number of extra copies you need. The `RegionShortcut` configurations that specify `PARTITION` and `REDUNDANT` all set the level at 1. By default, `redundant-copies` is 0, which disables redundancy.

> *The redundant copy is a backup. Setting redundant copies to one, as in the following examples, creates a total of two copies of every entry, the primary and one backup.*

Set redundant copies in one of these ways:

Example 6.4 API - Configuring Redundancy in a Partitioned Region

```
PartitionAttributes pa =
    new PartitionAttributesFactory()
        .setRedundantCopies(1)
        .create();
```

The next example shows the same configuration by setting the `redundant-copies` parameter in the `cache.xml` file. Again, this region is configured to maintain the maximum data redundancy.

Example 6.5 XML - Configuring Redundancy in a Partitioned Region

```
<region name="PR1" refid="PARTITION_REDUNDANT">
</region>
```

Redundancy Recovery

When the redundancy settings are not met, GemFire recovers redundancy by copying data buckets from members holding the primary data copies to other host members. It can do this when members providing redundancy leave the system and when members join the system.

Recovery With Persistent Regions

If the partitioned region is persisted to disk, recovery of redundancy for a bucket follows the same guidelines as general data recovery for the region. When GemFire attempts to recover redundancy, it does not recover to a new member when another, stopped member still has data persisted to disk.

So, if you are restarting members in a system with persistent partitioned regions:

1. If there are any stopped member with valid disk stores, start them with the same region configuration.
2. If there are any stopped members with missing or corrupt disk stores, revoke them so the running members can recover redundancy to other members.

For information about disk stores, including validating disk stores and revoking disk stores, see *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.

Recovery After Departure

When a member hosting partitioned data departs the distributed system, by default GemFire does not attempt to re-establish the configured redundancy level for the partitioned regions.

This is based on the assumption that, typically, the departed member will be restarted within a short time, so the most efficient response is to wait instead of temporarily moving the data around to other members.

You can change this behavior with the `recovery-delay` attribute:

recovery-delay	Effect following a member failure
-1	No automatic recovery of redundancy following a member failure. This is the default.
`long greater than or equal to 0`	Number of milliseconds to wait after a member failure before recovering redundancy.

Recovery After Startup

When a member hosting a partitioned region joins the distributed system, by default, GemFire immediately attempts to recover redundancy if redundancy was not satisfied before.

You can change this behavior with the `startup-recovery-delay` attribute:

startup-recovery-delay	Effect following a member startup
`-1`	No automatic recovery of redundancy after a new member comes online. The user must manually execute rebalancing to recover redundancy.
`long greater than or equal to` **0**	Number of milliseconds to wait after a member comes online before recovering redundancy.

Tuning Redundancy Recovery

GemFire does not automatically satisfy redundancy when `startup-recovery-delay` and `recovery-delay` are both set to `-1`. You can have GemFire attempt to satisfy redundancy immediately in both situations by setting both attributes to `0`. A manual rebalance can be done later to even out any imbalances in load.

With the default settings, when you simultaneously start two or more partitioned region host members, you may end up with only one of them taking on buckets to satisfy redundancy when it would be possible for both to host buckets. This happens when the `startup-recovery-delay` is short enough that only one member creates the partitioned region before recovery is done. To avoid this, set the `startup-recovery-delay` to delay redundancy recovery long enough to allow you to start both members and for them to recreate the partitioned region. They will both take on new buckets, if possible, because they are both there and recognize each other during the startup delay window.

Note that there is a distinction between satisfying redundancy and adding capacity. Once redundancy is satisfied, any new members do not take buckets until you invoke a rebalance.

For more information about partitioned region rebalancing, see *Rebalancing Partitioned Regions* on page 150.

Configuring Timed Recovery in cache.xml

The following `cache.xml` example sets `recovery-delay` to `3600000`, which sets a delay of one hour, until the lost capacity is restored. If the partitioned region is not created in a new data host within an hour, the existing data hosts make copies of the buckets with low redundancy. If a new data host is started, setting the `startup-recovery-delay` value to `0` causes redundancy recovery to occur immediately.

Example 6.6 Setting a Recovery Delay for Partitioned Region Redundancy

```
<cache>
   <region name="pRoot" refid="PARTITION_REDUNDANT">
      <region-attributes>
         <partition-attributes
            recovery-delay="3600000"
            startup-recovery-delay="0"/>
      </region-attributes>
   </region>
</cache>
```

Load Balancing With Redundant Partitioned Regions

Operations are spread across members for better load balancing. Writes are always sent to the primary bucket, processed, then forwarded to secondaries and any other VMs that have subscribed to the region events. Reads use the local cache if there is a copy of the data there. If no copy exists, the read is forwarded at random to one of the VMs hosting the data. So reads use secondaries as well as primaries, spreading the load across the system.

Figure 6.4 Reads on a Partitioned Region With Redundant Data

Here, redundant data is distributed across a three-member partitioned region. The entries are grouped: Entries W and X were created in the same storage area, so as they move through the region, they always stay together in the same bucket. Likewise, entries Y and Z are grouped together. Each group of entries contains either primary or secondary data, but not both.

Custom Partitioning and Data Colocation

By default, GemFire partitions each data point into a bucket using a hashing policy on the key. The physical location of the key-value pair is abstracted away from the application. Additionally, there is no default relation between data stored in different partitioned regions.

To run transactions on your partitioned regions, you must colocate all data that you access in any single transaction on the same data host.

Additionally, in many cases, you get better performance if you colocate similar data within a region and between regions.

- A query run on a patient, her health records, and her insurance and billing information is more efficient if all of the data is grouped in a single VM.
- A financial risk analytical application runs more quickly if all trades, risk sensitivities, and reference data associated with a single instrument are together.

Colocation generally improves the performance of data-intensive operations. You can reduce network hops for iterative operations on related datasets. You can usually significantly increase overall throughput for compute-heavy, data-intensive applications.

You specify colocation through configuration attributes and custom coding. You can:

- Add custom partitioning code to a region to route logically connected data into the same buckets.
- Colocate multiple regions, so buckets with the same number in each region are stored in the same VM.

The two mechanisms are independent, but to colocate data between regions, either use the same code for all regions or make sure to write code that results in like data getting the same routing objects across regions.

These are the options:

- Custom Partitioning to Colocate Like Data Within a Region (page 157)
- Colocating Data Between Multiple Regions (page 159)
- Combining Custom Partitioning with Multiple Region Colocation (page 159)

Custom Partitioning to Colocate Like Data Within a Region

For custom partitioning, you provide code to replace GemFire's default hashing policy.

1. Implement `com.gemstone.gemfire.cache.PartitionResolver` in one of these places, listed here in the search order used by GemFire:
   1.1 Custom class
   1.2 Entry key
   1.3 Cache callback argument. This restricts you to using only entry methods that accept a cache callback argument. For a full list of the methods that take a callback argument, see the `Region` Javadocs. Among the methods that you cannot use with cache callback are `putAll`, `getAll`, `containsKey`, and `getDistributedLock`. Further, the GemFire function execution service cannot take advantage of data colocated using this argument.
2. Program the resolver's `getName` method, if you need that.
3. Program the resolver's `getRoutingObject` method to return the routing object for each entry, based on how you want to group the entries. Give the same routing object to entries you want to group together - they will be placed in the same bucket. This is an example implementation on a region key object that groups the entries by month and year:

Example 6.7 Programming the PartitionResolver's getRoutingObject

```
Public class TradeKey implements PartitionResolver
{
    private String tradeID;
    private Month month;
    private Year year;
    public TradingKey(){ }
    public TradingKey(Month month, Year year)
    {
        this.month = month;
        this.year = year;
    }
    public Serializable getRoutingObject(EntryOperation opDetails)
    {
        return this.month + this.year;
    }
}
```

4. Make sure GemFire finds your resolver for every operation you perform on the region's entries. GemFire locates the entries for every operation, so the resolver must be the same every time.

   4.1 Custom class - Define the class for the region at creation. The resolver will be used for every entry operation. Use one of these methods:

Example 6.8 XML Custom Partition Resolver Class

```
<region name="trades">
    <region-attributes>
        <partition-attributes>
            <partition-resolver="TradesPartitionResolver">
                <class-name>myPackage.TradesPartitionResolver
                </class-name>
            </partition-resolver>
        <partition-attributes>
    </region-attributes>
</region>
```

Example 6.9 Java Custom Partition Resolver Class

```
PartitionResolver resolver = new TradesPartitionResolver();
PartitionAttributes attrs =
   new PartitionAttributesFactory()
   .setPartitionResolver(resolver).create();
Region Trades =
   new RegionFactory().setPartitionAttributes(attrs).create("trades");
```

   4.2 Entry key - Use the key object with the resolver implementation for every entry operation.

   4.3 Cache callback argument - Provide the argument to every call that accesses an entry.

5. If your colocated data is in a server system, add the `PartitionResolver` implementation class to the `CLASSPATH` of your Java clients. The resolver is used for single hop access to partitioned region data in the servers. See `pr-single-hop-enabled` on page 211.

Colocating Data Between Multiple Regions

You can colocate any number of partitioned regions by linking them at creation time.

1. Set the same values for these partition attributes across all of the colocated regions:
   - `recovery-delay`
   - `redundant-copies`
   - `startup-recovery-delay`
   - `total-num-buckets`
2. Identify one region as the central region, and explicitly colocate the other regions with it.
   - 2.1 Define the central region first. GemFire will verify its existence when the others are created, returning `IllegalStateException` if it is not there. Do not add colocation specifications to this central region.
   - 2.2 For other regions, set the partition attributes `colocated-with` to the name of the first region.
3. If you need to persist in any of these regions:
   - 3.1 Persist the central region. This is required.
   - 3.1 Persist the other regions as needed.
   - 3.2 Use the same disk store for all of the colocated regions.

Use one of these methods to colocate your regions:

Example 6.10 XML Data Colocation

```
<cache>
    <region name="trades" refid="PARTITION"/>
   <region name="trade_history" refid="PARTITION">
        <region-attributes>
            <partition-attributes colocated-with="trades">
                ...
            <partition-attributes>
        </region-attributes>
    </region>
</cache>
```

Example 6.11 Java Data Colocation

```
PartitionAttributes attrs = ...
Region trades = new
RegionFactory().setPartitionAttributes(attrs).create("trades");
...
attrs = new
PartitionAttributesFactory().setColocatedWith(trades.getFullPath()).create()
;
Region trade_history = new
RegionFactory().setPartitionAttributes(attrs).create("trade_history");
```

Combining Custom Partitioning with Multiple Region Colocation

When you colocate regions, you combine colocation with a single custom resolver used in all regions.

Example 6.12 XML Colocating Regions with Custom Partitioning

```
<cache>
   <region name="trades">
      <region-attributes>
         <partition-attributes>
            <partition-resolver="TradesPartitionResolver">
               <class-name>myPackage.TradesPartitionResolver
               </class-name>
            </partition-resolver>
         </partition-attributes>
      </region-attributes>
   </region>
   <region name="trade_history">
      <region-attributes>
         <partition-attributes colocated-with="trades">
            <partition-resolver="TradesPartitionResolver">
               <class-name>myPackage.TradesPartitionResolver
               </class-name>
            </partition-resolver>
         </partition-attributes>
      </region-attributes>
   </region>
</cache>
```

Example 6.13 Java Colocating Regions with Custom Partitioning

```
PartitionResolver resolver = new TradesPartitionResolver();
PartitionAttributes attrs =
   new PartitionAttributesFactory()
   .setPartitionResolver(resolver).create();
Region trades = new
RegionFactory().setPartitionAttributes(attrs).create("trades");
attrs = new PartitionAttributesFactory()

.setColocatedWith(trades.getFullPath()).setPartitionResolver(resolver).creat
e();
Region trade_history = new
RegionFactory().setPartitionAttributes(attrs).create("trade_history");
```

6.2 Replicated Regions and Other Distributed Regions

How Distributed Regions Work

A distributed region—replicated or not—automatically sends entry value updates to remote caches and receives updates from remote caches. These are the primary aspects of distribution:

- Entry update events—from `Region put` and `create` operations—are sent to all caches where the entry is already defined. (The creation of an entry with a non-null value is a value update in remote caches that already have the entry defined.)
- The distributed region shares any cache loader and writer application plug-ins across the distributed system. For more information, see *Controlling Data Flow With Application Plug-ins* on page 193.

This figure shows the general data flow that accompanies entry updates in a distributed region. In a distributed region, new and updated entry values are automatically distributed to remote caches that already have the entries defined. If the region in member M_2 did not already contain entry X, it would not be updated with the new value.

Figure 6.5 The Automatic Distribution of Entry Value Modification

Distribution causes a region's cache writer and cache loader to be available to other caches where the region is defined. For example, a cache loader may be defined only in one cache for a region that is distributed between multiple caches on multiple machines. This remote invocation is performed by the distributed system's `netLoad` and `netWrite` operations. For more information, see *The Distributed Region.get Operation* on page 135.

How Replicated Regions Work

Replication causes your local cache region to reflect the state of all entries in the distributed region. A replicated region is initialized from the distributed region or from disk backup.

This figure shows a replicate being initialized from all the data in the distributed region. Region A is created as a replicate in member M_3.

Figure 6.6 Union Initialization of a Region With a Data Policy of replicate or preloaded

If a union of regions is used for initialization, as in the figure, and one of the individual source regions goes away during the initialization (due to cache closure, member crash, or region destruction), the new region may contain a partial data set from the crashed source region. When this happens, there is no warning logged or exception thrown. The new region still gets a complete set of the remaining members' regions. If the region is being initialized from a replicated or preloaded region and the source region crashes, the initialization starts over using the regions that remain in the distributed system.

If the application data can be divided into data sets small enough to fit in a single cache, putting a replicated region on multiple cache servers provides highly reliable long-term data caching.

You can also define a region as preloaded. Preloading initializes the region like a replica, but then acts like a normal distributed region thereafter. The preloaded region gives newly started VMs a view of the distributed region without the ongoing event deliveries that come with true replication.

Replication and Local Destroy and Invalidate Operations

Of all the operations that affect local cache state, only local region destroy is allowed in a replicated region. The other operations throw an exception or are not configurable (like local destroy as an expiration action). Local operations such as entry invalidate, entry destroy, and region invalidate remove entry data from the local cache only (see *Invalidating and Destroying an Entry* on page 143). A

replicated region would no longer be complete if entries were removed or modified locally but were left intact in the distributed region as a whole.

The Difference Between Distributed and Replicated

Distributed regions share data between members that have the region defined. If a member has a data entry defined in its local cache, distribution causes data updates to the entry, from anywhere in the distributed system, to be copied into the member's cache. This is commonly known as the pull distribution model. The existance of the entry in the local cache pulls all updates from remote caches.

Figure 6.7 Data Flow for a Distributed (Non-Replicated) Region

Replicated regions are distributed regions that also receive all new entry creations from the distributed system. This is commonly known as the push distribution model, as anyone defining data for the region will push it to the replicate.

Replicated regions hold all entries that are available anywhere in the distributed region. This figure shows the data flow of a distributed replicated region.

Figure 6.8 Data Flow for a Replicated Region

You can define empty distributed and replicated regions, that serve only to send and possibly receive data events from other members.

Creating a Replicated Region

Replicates are configured using the `RegionShortcut` settings that start with `REPLICATE`. You can configure replicated regions manully by setting your region `data-policy` to `replicate` or `persistent-replicate`. To create a preloaded region, set your region `data-policy` to `preloaded`. This `XML` declaration creates a replicate with a scope of `distributed-ack`:

```
<region refid="REPLICATE"/>
```

When a replicated or preloaded region is created, it is initialized from one of these sources, in this order of preference:

- another replicate
- for `preloaded` only, other preloaded regions
- the union of all remaining regions in the distributed cache

Creating a Distributed Region

To configure a distributed region, or a non-standard replicated region, use a combination of the region attributes, `data-policy`, `subscription-attributes`, and `scope`:

- Scope—The `scope` (page 116) setting determines how data distribution is done. Scope settings range from `local`, with no distribution, to `global`, with the distribution of entry values guaranteed to be consistent across the distributed system. For local regions, see *Local Regions* on page 168. For all other settings, see *Choosing the Level of Distribution* on page 165.
- Interest policy—The `subscription-attributes` (page 118) `interest-policy` setting specifies which of the distributed region's entry operations are sent to the local cache from remote caches. The options are `cache-content`, which sends operations only for entries already defined in the local region, and `all`, which sends everything.
- Data policy—A `data-policy` (page 104) of `empty` causes no local data storage. A `normal` policy provides storage only for those entries that this member has explicitly expressed interest in. The `replicate` policies create replicated regions.

  If you want to store no data locally, but get events from the distributed cache or from servers, set `data-policy` *to* `empty` *and the* `subscription-attributes interest-policy` *to* `all`.

  An `interest-policy` *of* `all` *and a* `data-policy` *of* `normal` *effectively gives you a replicated region, with the same push model of distribution as in the normally configured replicate.*

This creates a distributed region with asynchronous event delivery:

```
<region name="dNoAck"
   <region-attributes scope="distributed-no-ack">
   </region-attributes>
</region>
```

This creates a replicate with a no-ack scope:

```
   <region refid="REPLICATE">
      <region-attributes scope="distributed-no-ack"/>
   </region>
```

Choosing the Level of Distribution

Each distributed region must have the same scope throughout the distributed system. Distributed scope is provided at three levels:

- **distributed-no-ack**— This setting dictates that distribution operations return without waiting for a response from other caches. This setting provides the best performance, but is also most prone to race conditions. A temporary disruption of the network transport layer, for example, could cause failure of a distribution to a cache on a remote machine.
- **distributed-ack**—This setting causes distribution to wait for acknowledgement from other caches before continuing. This is slower than `distributed-no-ack`, but covers simple communication problems such as the one described above. Race conditions are still possible, however, as there is no guarantee that updates sent from multiple caches arrive in the proper time order. Distribution operations may arrive out of order due to network latency and other differences between caches. For example, some caches may reside on the same machine while others may communicate over large networks. Because there is no locking, with either `distributed-ack` or `distributed-no-ack` scope it is possible for two members to replace an entry value simultaneously.

  The distributed system has a configurable time to wait for acknowledgment to any `distributed-ack` message. If the time elapses, the system sends alerts to the logs about a possible problem with the unresponsive node. The distributed system attribute governing this is `ack-wait-threshold`. No matter how long the wait, the sender keeps waiting in order to honor the `distributed-ack` region setting.

  In systems where there are many `distributed-no-ack` operations, it is possible for `distributed-ack` operations to take a long time to complete. For more information, see *Tuning to Reduce Slow distributed-ack Messages* on page 150 of the *GemFire Enterprise System Administrator's Guide*.
- **global**—With `global` scope, entries and regions are automatically locked across the distributed system during distribution operations (see *Locking* on page 166). All load, create, put, invalidate, and destroy operations on the region and its entries are performed with a distributed lock. This is the only scope that absolutely guarantees consistency across the distributed system. It is also the slowest. In addition to the implicit locking performed by distribution operations, regions with `global` scope and their contents can be explicitly locked through the application APIs. This allows applications to perform atomic, multi-step operations on regions and region entries. For details, see the online Java documentation for the `Region` class.

  > *The distributed scopes are not compatible with one another. A distributed region's* `scope` *setting must be the same across the distributed system.*

The choice of distribution level affects the data policy as well. For details, see `data-policy` on page 104.

Statistics monitoring can help with the task of fine-tuning distribution settings. For more on distribution level and system performance, see Chapter 18, *Monitoring and Tuning Your Applications*, on page 463.

Locking

This section applies to regions with `global` scope. Locking happens in two ways:

- **Implicit locking**—GemFire uses locking automatically during most operations that update the global region. Region invalidation and destruction do not acquire locks.
- **Explicit locking**—You can manually lock the region and its entries through the API. This allows you to guarantee atomicity in tasks involving multiple-step distributed operations. For details, see the online Java documentation for the `Region` class.

  You must use the `Region` *API to lock regions and region entries. Do not use the* `DistributedLockService` *in the* `com.gemstone.gemfire.distributed` *package. That service is available only for locking in arbitrary distributed applications. It is not compatible with the* `Region` *locking methods.*

Locking involves obtaining the lock using a lock service instance, and releasing the lock. Implicit and explicit locking operations are affected by timeouts set as cache attributes and also by parameters you provide through the API:

- The lock time-out setting specifies how long to wait to obtain a lock. The `lock-timeout` (page 74) that is set as a cache attribute governs implicit lock requests. For explicit locking, you specify the wait time through your calls to the locking API. You can decide whether to wait a specific amount of time, to return immediately either with or without the lock, or to wait indefinitely.
- For all locking, the cache attribute, `lock-lease` (page 75), determines how long a lock may be held before it is automatically released by the distributed system. A timed lock allows the application to recover from a situation where the entity that obtained a lock goes away without releasing it.

This `cache.xml` snippet sets the lock timeout for implicit lock requests to one minute and the lock lease for all locks in the cache to two minutes.

```
<cache lock-timeout="60" lock-lease="120" >
</cache>
```

The Lock Grantor

Each region with `global` scope has its own lock service. The lock service receives lock requests from system members, queues them as needed, and grants them in the order received. The process responsible for running the lock service is known as the lock grantor. The lock grantor is always one of the members that have a region with `global` scope defined. The lock grantor is assigned as follows:

- Any member that requests lock grantor status is assigned it. Thus at any time, the most recent member to make the request is the lock grantor.
- If no member requests lock grantor status for a region, or if the current lock grantor goes away, the system assigns a lock grantor from the members that have the region defined in their caches.

Any member with a region defined can ask to be the lock grantor. You can make the request at region creation through the `is-lock-grantor` attribute. After region creation, you can retrieve this attribute through the region method, `getAttributes`, to see whether you requested to be lock grantor for the region.

The `is-lock-grantor` *attribute does not change after region creation.*

To become the lock grantor after region creation, call the region `becomeLockGrantor` method. Changing lock grantors should be done with care, however, as doing so takes cycles from other operations. In particular, be careful to avoid creating a situation where you have members vying for lock grantor status.

The lock grantor grants locks in the order the requests are received, giving a first-come, first-served fairness to the locking process. The grantor itself, however, is at a slight advantage as it is the only member that does not have to send a message to request a lock. The grantor's requests cost the least for the same reason. Thus, you can optimize locking in a region by assigning lock grantor status to the member that acquires the most locks. This may be the member that performs the most `put`s and thus requires the most implicit locks or this may be the member that performs many explicit locks.

Locking Examples

These two examples, taken from the `cacheRunner` example program, show entry locking and unlocking. Note how the entry's `Lock` object is obtained and then its `lock` method is invoked to actually set the lock. The example program stores the entry lock information in a hash table for future reference. The first two lines listed in this example are the program's variable definitions pertinent to this example code.

Example 6.14 Locking an Entry

```
/* variable definitions used in this method */
Lock lock;
HashMap lockedItemsMap = new HashMap();

void lock(String command) { . . .
    String name = parseName(command);
    if (name != null) {
        if (!lockedItemsMap.containsKey(name)) {
            Lock lock = this.currRegion.getDistributedLock(name);
            lock.lock();
            lockedItemsMap.put(name, lock);
        }
    }  . . .
}
```

Example 6.15 Unlocking an Entry

```
void unlock(String command) { . . .
    String name = parseName(command);
    if ((name != null) && (lockedItemsMap.containsKey(name))) {
        Lock lock = (Lock) lockedItemsMap.remove(name);
        lock.unlock();
    }
    else . . .
}
```

6.3 Local Regions

Local regions are most easily configured using the `RegionShortcut` and `ClientRegionShortcut` settings starting with `LOCAL`. These create regions with a `scope` (page 116) set to `LOCAL`.

> *This is the most common client region configuration. It allows distribution between client and server, but isolates the region from any client-tier distribution.*

A local region a private data set for the member, invisible to peer members in the distributed system. All region configuration options, like expiration and eviction, and cache listeners and loaders, affect only the local cache.

This `cache.xml` file snippet creates a local region:

```
<region name="clientCopy" refid="LOCAL"/>
```

Locally scoped regions cannot be configured to receive any events from remote regions in the same distributed system. Data is not distributed between a locally scoped region and other caches in the system through peer-to-peer communication.

This figure depicts two distinct regions with local scope. The arrows show who can access the regions' data in the distributed system.

Figure 6.9 Regions with Local Scope

Chapter 7

General Region Management Options

This chapter tells you how to use the general region management features provided by GemFire. All GemFire region types use the same configuration and management for features like disk persistence, eviction and expiration, event handling, and using an applicatin plug-in to load data from an outside data source.

For information on the basics of GemFire data region management, including your general options for region configuration, how to create and remove regions from your cache, and a full listing of region attributes settings, see Chapter 4, *Data Regions*, on page 85.

Each of the main region configuration types has its own specific considerations for configuration and management. For detailed information on those, see Chapter 6, *Configuration and Management by Region Type*, on page 145.

In this chapter:

- Using Disk for Backup and Overflow (page 170)
- Keeping Your Data Current With Expiration (page 176)
- Controlling Memory Use With Eviction and Overflow (page 182)
- Controlling Heap Use With the Resource Manager (page 186)
- Controlling Data Flow With Application Plug-ins (page 193)

7.1 Using Disk for Backup and Overflow

You can copy region data to disk for backup purposes (persistence) and to satisfy region capacity restrictions without destroying the cache-local data (overflow).

All persistence and overflow uses GemFire disk stores. See *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.

The persistence and overflow mechanisms use disk files to hold entry data. For persistence , the entry keys and values are copied to disk. For overflow, only the entry values are copied. Other data, such as statistics and user attributes, are retained in memory only.

> *Persistent data outlives the member where the region resides and can be used to initialize the region at creation. Overflow acts only as an extension of the region in memory.*
>
> *Persistent regions may load data from disk at startup.*

In this section:

- How Region Persistence and Overflow Work (page 171)
- Programming Example for Persistence and Overflow (page 173)
- Performance Considerations (page 175)
- Entry Retrieval in Overflow and Persistent Regions (page 175)

How Region Persistence and Overflow Work

For information on configuration for the disk stores used for persistence and overflow, see *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.

Persistence and overflow can be used individually or in combination.

- **Persistence**—Provides a disk copy of region entry data. With persistence, the keys and values of all entries are saved to disk, like having a replicate of the region on disk. Region entry operations such as `put` and `destroy` are carried out in memory and on disk. When the member stops (either from a normal shutdown or a crash), the region data on disk remains. In partitioned regions, where data buckets are divided among members, this can result in some data only on disk and some on disk and in memory. When the region is created again in the member, it can be populated from disk. Persistence is specified using the region shortcut settings with `PERSISTENT` in their names. This figure shows the data flow for persistence.

Figure 7.1 Data Flow Between Persistent Region and Disk Files

- **Overflow**—Allows you to keep the region within a user-specified size in memory by relegating the values of least recently used (LRU) entries to disk. Overflow basically uses disk as a swap space for entry values. When the region size reaches the specified threshold, entry values are moved from memory to disk. If an entry is requested whose value is only on disk, the value is copied back up into memory, possibly causing the value of a different LRU entry to be removed to disk. As with persisted entries, overflowed entries are maintained on disk just as they are in memory. Overflow is specified as the eviction option in `eviction-attributes` (page 108) specification.

  In this figure the value of LRU entry X has been moved to disk to recover space in memory. The key for the entry remains in memory. From the distributed system perspective, the value on disk is as much a part of the region as the data in memory. A `get` performed on region B looks in memory and then on disk as part of the local cache search.

Figure 7.2 Data Flow Between Overflow Region and Disk Files

- **Persistence and overflow in combination**—Used together, persistence and overflow keep all entry keys and data on disk and only the most active entry values in memory. The removal of an entry value from memory due to overflow has no effect on the disk copy as all entries are already present there.

Figure 7.3 Data Flow Between Persistent/Overflow Region and Disk Files

Overflow is specified by using an eviction controller with an eviction action of `overflow-to-disk`.

A cache disk store defines where and how the data is written to disk. Each region is assigned a disk store through the region attribute, `disk-store-name` (page 106).

Programming Example for Persistence and Overflow

To specify writes through the API, use the `DiskWriteAttributesFactory` to create a `DiskWriteAttributes` object and use the object in your region creation. For asynchronous writes, set the attributes as shown in the following example. For synchronous writes, do the same thing except use `setSynchronous(synchronous)` and remove the `setTimeInterval` and `setBytesThreshold` lines.

The Java program, `cacheOverflow/CacheOverflow.java`, creates a region with persistence and overflow settings determined by user input.

Example 7.1 Creating a Region with Persistence and Overflow Through the API

```
Cache cache;
try {
      cache = new CacheFactory(p).create();

      DiskStoreFactory dsf = cache.createDiskStoreFactory();
      dsf.setDiskDirs(dirs.toArray(new File[dirs.size()]));
      dsf.setTimeInterval(1000);
      //Setting the queue size to 0 indicates that we will allow
      //an unlimited number of entries to be queued to be written to disk
      dsf.setQueueSize(0);
      dsf.setAutoCompact(!disableCompaction);
      dsf.setMaxOplogSize(maxOplogSize);
      dsf.create(DiskStoreFactory.DEFAULT_DISK_STORE_NAME);

      RegionFactory<String, long[]> regionFactory =  null;
      if(backup) {
          regionFactory =
             cache.createRegionFactory(RegionShortcut.LOCAL_PERSISTENT);
} else {
        regionFactory = cache.createRegionFactory(RegionShortcut.LOCAL);
      }

      regionFactory.setDiskSynchronous(synchronous);
      regionFactory.setEvictionAttributes(EvictionAttributes
          .createLRUMemoryAttributes((int) overflowThreshold, null /* sizer
*/,
                EvictionAction.OVERFLOW_TO_DISK));
      region = regionFactory.create("CacheOverflow");
    } catch (CacheException ex) {
      err.println("While creating the Cache and Region:");
      ex.printStackTrace(err);
      System.exit(1);
      return;
    }
```

A similar region, configured as a replicate, is declared in the following XML example. In this case, the persistence and overflow settings are hard-coded.

Example 7.2 Creating a Region with Persistence and Overflow in cache.xml

```
<disk-store name="DEFAULT" auto-compact="true" max-oplog-size="20" queue-
size="10000000" time-interval="15000">
        <!-- Write buffer data to disk every 10 million bytes or 15 seconds,
             whichever comes first -->
      <disk-dirs>
        <disk-dir>backupDirectory1</disk-dir>
      </disk-dirs>
  </disk-store>

  <region name="root" refid="REPLICATE_PERSISTENT">
    <region-attributes disk-synchronous="false">
      <eviction-attributes>
        <!-- Overflow to disk when 100 megabytes of data reside in the
             region -->
        <lru-memory-size maximum="100" action="overflow-to-disk"/>
      </eviction-attributes>
    </region-attributes>

    <region name="partitioned_region" refid="PARTITION_PERSISTENT">
      <region-attributes>
        <partition-attributes redundant-copies="0" total-num-buckets="10"/>
      </region-attributes>
    </region>
```

Performance Considerations

While the performance of accessing data on disk has been shown to be quite high, there are a couple of considerations to take into account when using disk storage.

- When an entry is overflowed, its value is written to disk but its key and entry object remain in the member. This scheme allows the region to receive updated entry values, but also incurs some memory overhead in the member. For more information, see *Memory Overhead Introduced by the Cache API* on page 135 of the *GemFire Enterprise System Administrator's Guide*.
- When a persistent disk store is restored into a replicated or partitioned region, the region or region buckets are populated with the entry data found on disk. Some of this data may be out of date with respect to the other members of the distributed system. To ensure that the region has the most recent data values, the cache requests the latest value from the other members of the distributed system after the disk store is loaded. This operation could be quite expensive for a region that contains a lot of data. For more information, see *Recovering Data From Disk* on page 219 of the *GemFire Enterprise System Administrator's Guide*.
- There are other performance considerations that your system administrator takes into account when setting up the disk stores you use. See *Managing Memory* on page 135 of the *GemFire Enterprise System Administrator's Guide*.

Entry Retrieval in Overflow and Persistent Regions

For entry retrieval in a region with an overflow eviction controller installed, memory and disk are treated as the "local cache." When a `get` is performed in a region with overflow enabled, memory and then disk are searched for the entry value as part of the local cache search. If the value is not found, the distributed system runs the standard operations listed in Table 5.1 on page 137.

With persistence enabled and overflow disabled, the disk is not accessed for entry retrieval as it does not hold any values that are not also in memory. When you persist a region, the entry data on disk outlives the region in memory. If your member exits or crashes, the data remains in the data directories specified by the region's disk-store. If the same region is created again using the same disk store, this saved disk data can be used to initialize the region. For details on recovering data from disk, see *Recovering Data From Disk* on page 219 of the *GemFire Enterprise System Administrator's Guide*.

7.2 Keeping Your Data Current With Expiration

Use expiration to configure automated removal of unused data and data regions from the cache. Expiration helps keep data current by removing outdated entries, prompting a reload the next time they are requested. Expiration may also be used to recover space in the cache by clearing out entries and regions that are not being used.

> *When a* `netSearch` (page 136) *is used to retrieve an entry value from a remote cache, the local cache's expiration attributes are compared to the remote entry's last accessed and last updated statistics. Entries that would have already expired in the local cache are passed over.*

You can expire a region and its entries according to last accessed time and last modified time. You can also define your own custom expiration based on the entry keys, using the `CustomExpiry` interface. You can use this on its own or as an override to region-wide settings.

This figure shows two expiration scenarios in a distributed region. This is a system where a producer application provides data to a number of consumer applications.

- The producer populates the region from a database and the data is automatically distributed throughout the system. The data is valid only for one hour, so the producer performs a distributed destroy on entries that are an hour old. This causes the entries to be destroyed in producer and consumer.
- All other applications are consumers. The consumers free up space in their individual caches by locally removing entries for which there is no local interest (idle-time expiration). This way, they only retain the entries they are actively interested in. If a consumer needs an expired entry, it will automatically go to the producer for the current value.

Figure 7.4 Expiration in a Distributed Region

You can choose to mark expired objects as invalid or remove them from the distributed cache. Expiration actions vary according to region type:

- In local regions, expiration affects only the local cache and you configure local expiration actions.
- In distributed regions, you configure distributed or local actions. Thus, one cache could control expiration for a number of caches in the system or could just remove its own copies of the data.
- In partitioned regions, you configure either distributed invalidate or distributed destroy.

If you use expiration, set the region's `statistics-enabled` (page 117) attribute to `true`. Expiration is measured by comparing the statistics for access and modification to the expiration attributes, so statistics are required. These statistics are available directly to the application through the `CacheStatistics` object returned by the `Region` and `Region.Entry` `getStatistics` methods. The `CacheStatistics` object also provides a method for resetting the statistics counters.

The following `cache.xml` declaration causes the region's entries to be invalidated in the local cache after they have not been accessed for one minute. Note that statistics are enabled, as required for expiration activities:

```
<region-attributes statistics-enabled="true">
   <entry-idle-time>
      <expiration-attributes timeout="60" action="local-invalidate"/>
   </entry-idle-time>
</region-attributes>
```

By default, there are no expiration activities configured for the region.

Custom Expiration

Typically, one expiration setting applies to all region entries. You can also set custom expiration attributes for individual region entries that override the region expiration settings.

The following `cache.xml` snippet defines the overall region expiration settings with a timeout of `60` seconds and an expiration action of `local-invalidate`. Using the `custom-expiry` attribute with a custom class named `MyClass`, custom expiration limits are also set.

```
<region-attributes statistics-enabled="true">
   <entry-idle-time>
      <expiration-attributes timeout="60" action="local-invalidate">
         <custom-expiry>
            <class-name>com.megaconglomerate.mypackage.MyClass
              </class-name>
         </custom-expiry>
      </expiration-attributes>
   </entry-idle-time>
</region-attributes>
```

In the previous XML example, `MyClass` contains the custom expiration settings you configured for the specified region entries.

This `MyClass` code snippet shows how the class uses the key for a region entry to set an expiration timeout of `10` seconds for even-numbered keys. Odd-numbered keys get a shorter timeout of `5` seconds. At the end of the timeouts, a `DESTROY` action removes the entries from the cache.

```
public class MyClass implements CustomExpiry, Declarable {

   private static final ExpirationAttributes LONG_EXPIRY =
      new ExpirationAttributes(10, ExpirationAction.DESTROY);

   private static final ExpirationAttributes SHORT_EXPIRY =
      new ExpirationAttributes(5, ExpirationAction.DESTROY);

   public ExpirationAttributes getExpiry(Entry entry) {
      // Use key to determine expiration.
      // If odd, expire quickly; if even, expire slowly
      int key = (Integer) entry.getKey();
      return key % 2 == 0 ? LONG_EXPIRY : SHORT_EXPIRY;
   }
}
```

Using the `custom-expiry` attribute, the default expiration timeout of `60` seconds and eviction action of `local-invalidate` are overridden by the custom expiration settings defined in `MyClass`.

Expiration Types

Region and region entry expiration attributes are set at the region level. By default, regions and entries do not expire. In partitioned regions, expiration is carried out only by the primary data hosts, and affects the primary and all secondaries.

The attributes cover two types of expiration:

- **Time to Live (TTL)**. Amount of time, in seconds, the object may remain in the cache after the last creation or update. For entries, the counter is set to zero for `create` and `put` operations. Region counters are reset when the region is created and when an entry has its counter reset. Thus, an update to an entry causes the TTL counters to be reset for the entry and its region.

  The TTL expiration attributes are:

  - `region-time-to-live` (page 116)
  - `entry-time-to-live` (page 108)

- **Idle Timeout**. Amount of time, in seconds, the object may remain in the cache after the last access. The idle timeout counter for an object is reset any time its TTL counter is reset. In addition, an entry's idle timeout counter is reset any time the entry is accessed through a `get` operation or a `netSearch`. The idle timeout counter for a region is reset whenever the idle timeout is reset for one of its entries. For partitioned regions, see *Partitioned Region Note for Idle Time Expiration* on page 179.

  Idle timeout expiration attributes are:

  - `region-idle-time` (page 116)
  - `entry-idle-time` (page 108)

Expiration Actions

For each attribute, an expiration action must be specified. The options are:

- `destroy`
- `local destroy`
- **`invalidate`** (default)
- `local invalidate`

For descriptions of these options, see *Invalidating a Region* on page 94, *Destroying a Region* on page 95, and *Invalidating and Destroying an Entry* on page 143. Destruction and invalidation cause the same event notification activities whether carried out explicitly or through expiration activities. The `CacheEvent` object includes an `getOperation.isExpiration` flag that is `true` for expiration activities and `false` otherwise.

Partitioned Region Note for Idle Time Expiration

For overall region performance reasons, idle time expiration in partitioned regions may expire some entries sooner than expected.

Expiration in partitioned regions is executed in the primary copy, based on the primary's last accessed and last updated statistics.

- Entry updates are always done in the primary copy, resetting the primary copy's last updated and last accessed statistics.
- Entry retrieval uses the most convenient available copy of the data, which may be one of the secondary copies. This provides the best performance at the cost of possibly not updating the primary copy's statistic for last accessed time.

When the primary expires entries, it does not request last accessed statistics from the secondaries, as the performance hit would be too great. It expires entries based solely on the last time the entries were accessed in the primary copy.

Interaction Between Expiration Settings and netSearch

Before `netSearch` (page 136) retrieves an entry value from a remote cache, it validates the remote entry's statistics (see *Transient Region and Entry Statistics* on page 469) against the local region's expiration settings. This ensures that an entry that would have expired locally is not brought into the local cache. Once validated, the entry is brought into the local cache and the local access and update statistics are updated. The last accessed time is reset and the last modified time is updated to the time in the remote cache, with corrections made for system clock differences. Thus the local entry is assigned the true last time the entry was modified in the distributed system. The `netSearch` operation has no effect on the expiration counters in remote caches.

Configuring Expiration

These steps do not have to be done in sequential order.

1. Set the region's statistics-enabled attribute to true.
2. Set the expiration attributes by expiration type, with the max times and expiration actions. For partitioned regions, use the time to live attributes, not the idle time attributes, to ensure reliable read behavior.

Example 7.3 Setting Standard Expiration on an Entry

```
<region-attributes statistics-enabled="true">
  <entry-idle-time>
    <expiration-attributes timeout="60" action="local-invalidate"/>
  </entry-idle-time>
</region-attributes>
```

3. If you need to, override the region-wide settings for specific entries:

   3.1 Program a custom expiration class.

Example 7.4 Programming a Custom Expiration Class

```
// Use the key for a region entry to set entry-specific expiration timeouts
of
//   10 seconds for even-numbered keys with a DESTROY action on the expired
entries
//   Leave the default region setting for all odd-numbered keys.
public class MyClass implements CustomExpiry, Declarable
{
    private static final ExpirationAttributes CUSTOM_EXPIRY =
            new ExpirationAttributes(10, ExpirationAction.DESTROY);
    public ExpirationAttributes getExpiry(Entry entry)
    {
        int key = (Integer)entry.getKey();
        return key % 2 == 0 ? CUSTOM_EXPIRY : null;
    }
}
```

   3.2 Define the class inside the expiration attributes settings for the region.

Example 7.5 Installing a Custom Expiry Class

```
<!-- Set default entry idle timeout expiration for the region -->
<!-- Pass entries to custom expiry class for expiration overrides -->
<region-attributes statistics-enabled="true">
   <entry-idle-time>
      <expiration-attributes timeout="60" action="local-invalidate">
         <custom-expiry>
            <class-name>com.megaconglomerate.mypackage.MyClass</class-name>
         </custom-expiry>
      </expiration-attributes>
   </entry-idle-time>
</region-attributes>
```

7.3 Controlling Memory Use With Eviction and Overflow

Eviction settings cause GemFire to work to keep a region's resource use under a specified level by removing least recently used (LRU) entries to make way for new. Using `eviction-attributes` (page 108), you configure for eviction based on one of these:

- Entry count
- Absolute memory use
- Percentage of available VM heap.

Eviction management is controlled in different ways depending on the algorithm used:

- Entry count and absolute memory use are managed completely at the region level by the GemFire eviction controller. The controller uses the algorithm and the maximum size to determine when to evict entries. When eviction is needed, the eviction controller works to make space in the region by removing the least-recently-used (LRU) entries from the cache. For a partitioned region, GemFire evicts the least recently used entry *in the local bucket* where the new entry operation is being performed.
- Heap percentage is managed by the GemFire resource manager on the cache level. The maximum setting is provided for the cache in the manager configuration. When the manager determines that eviction is required, the manager orders the eviction controller to start evicting from all regions where the eviction algorithm is set to `lru-heap-percentage`. Eviction continues until the manager calls a halt. GemFire evicts the least recently used entry hosted by the VM for the region. For information on the manager, see *Controlling Heap Use With the Resource Manager* on page 186.

The `eviction-attributes` also control the action to take when the limit is reached:

- Locally destroy the entry.
- Overflow the entry to disk and null its value in the member. For more information on writing entries to disk, see *Using Disk for Backup and Overflow* on page 170.

An eviction controller must be set at region creation, and cannot be removed from a region once the controller is established. However, the entry count, region size, or percentage of heap memory used can be changed through the API using the `EvictionAttributesMutator`.

When GemFire determines that adding or updating an entry would take a region over the specified eviction level, it removes as many older entries as it needs to make room. For entry count eviction, this means a one-to-one trade of an older entry for the newer one. For the memory settings, the number of older entries removed to make space depends on the relative sizes of the older and newer entries.

Partitioned Region Note for Eviction

In partitioned regions with entry count and memory eviction, the eviction behavior is kicked off when a member's data hits the limit for combined primary and redundant copies in the local cache. For heap eviction, it is kicked off by the manager.

GemFire maintains LRU entry information on a bucket-by-bucket bases, as the cost of maintaining information across the partitioned region or across buckets in the member would be too great. In addition, applying evictions across all buckets of a partitioned region could contribute to region imbalance.

Because of this, eviction in partitioned regions may leave older entries for the region in other buckets in the local data host as well as in other hosts in the distributed system. It may also leave entries in a primary copy that it evicts from a secondary copy or vice-versa.

LRU eviction is done in each bucket separately:

- For memory and entry count eviction, LRU eviction is done *in the bucket where the new entry operation is being performed* until the overall size of the combined buckets in the member has dropped enough to perform the operation without going over the limit.
- For heap eviction, each partitioned region bucket is treated as if it were a separate region, with each eviction action only considering the LRU for the bucket, and not the partitioned region as a whole.

Configuring Eviction

Use the `eviction-attributes` region attribute settings to keep your region within a specified limit using eviction controllers.

These steps do not have to be done in sequential order.

1. Decide whether to evict based on:
   1.1 Entry count (useful if your entry sizes are relatively uniform).
   1.2 Total bytes used. In partitioned regions, set this using `local-max-memory`. In non-partitioned regions, set it in `eviction-attributes`.
   1.3 Percentage of VM heap in use. This uses the GemFire resource manager. If you use this, see the additional configuration information in Controlling Heap Use With the Resource Manager (page 186). The heap can be consumed by other threads, classes or regions in the member, so the percentage can vary even if the region entries do not change.
2. Decide the action to take when the limit is reached:
   2.1 Locally destroy the entry.
   2.2 Overflow the entry data to disk. See *Using Disk for Backup and Overflow* on page 170.
3. Decide on the maximum amount of data to allow in the member for the eviction measurement indicated. This is the maximum for all storage for the region in the member. For partitioned regions, this is the total for all buckets stored in the member for the region - including any secondary buckets used for redundancy.
4. If you want to provide a custom sizer for your region, program that. If you are able to provide such a class, it might be faster than the standard sizing done by GemFire. Your custom class must follow the guidelines for defining custom classes and, additionally, must implement `com.gemstone.gemfire.cache.util.ObjectSizer`. See *Defining Your Own Custom Classes* on page 57.
5. Configure your eviction settings and anything else you need, such as resource manager and custom sizer installation.

Examples

Example 7.6 XML - LRU Memory Eviction Controller in Non-Partitioned Region

```
<region-attributes>
  <eviction-attributes>
    <lru-memory-size maximum="1000" action="overflow-to-disk">
      <class-name>com.myLib.MySizer</class-name>
      <parameter name="name">
        <string>Super Sizer</string>
      </parameter>
    </lru-memory-size>
  </eviction-attributes>
</region-attributes>
```

Example 7.7 XML - LRU Memory Eviction Controller in Partitioned Region

```
<region name="demoPR">
  <region-attributes>
    <partition-attributes local-max-memory="512" total-num-buckets="13" />
    <eviction-attributes>
       <lru-memory-size action="destroy"/>
       <class-name>com.gemstone.gemfire.cache.util.ObjectSizerImpl
       </class-name>
    </eviction-attributes>
  </region-attributes>
</region>
```

Example 7.8 XML - LRU Heap Eviction Controller

```
<region-attributes>
  <eviction-attributes>
    <lru-heap-percentage maximum="90" action="overflow-to-disk">
  </eviction-attributes>
</region-attributes>
```

Example 7.9 API Entry Count Conviction

```
Region lruDiskRegion = new RegionFactory()
    .setKeyConstraint(String.class)
    .setDiskStoreName(myOverflowDiskStore.getName))
    .setEvictionAttributes(EvictionAttributes.createLRUEntryAttributes
       (250, EvictionAction.OVERFLOW_TO_DISK))
    .create("LRUDiskRegion");
```

Example 7.10 Declarative Heap LRU-Based Eviction with Overflow

```
<cache>
    <disk-store name="overflowDiskStore1"/>
       <disk-dirs>
          <disk-dir>overflowDir</disk-dir>
       </disk-dirs>
    </disk-store>
    <resource-manager critical-heap-percentage="80" eviction-heap-percentage=
"60"/>
    <region name="demoPR" refid="PARTITION_REDUNDANT">
       <region-attributes disk-store-name="overflowDiskStore1"
                 disk-synchronous="true">
          <partition-attributes total-num-buckets="13"/>
          <eviction-attributes>
             <lru-heap-percentage action="overflow-to-disk"/>
          </eviction-attributes>
       </region-attributes>
    </region>
</cache>
```

.

7.4 Controlling Heap Use With the Resource Manager

The GemFire resource manager works with your VM's tenured garbage collection (GC) to control heap use and protect your VM from hangs and crashes due to memory overload. The manager prevents the cache from consuming too much memory by evicting old data and, if the collector is unable to keep up, by refusing additions to the cache until the collector has freed an adequate amount of memory.

> *Resource manager behavior is closely tied to the triggering of GC activities, the use of concurrent GCs in the VM, and the number of parallel GC threads used for concurrency. These recommendations for using the manager assume you have a solid understanding of your Java VM's heap management and garbage collection service.*

The resource manager has two threshold settings, each expressed as a percentage of the total tenured heap, and both of which are disabled by default.

- Eviction threshold. Above this, the manager orders evictions for all regions with `eviction-attributes` (page 108) set to `lru-heap-percentage`. This prompts dedicated, background evictions, independent of any application threads and it also tells all application threads adding data to the regions to evict at least as much data as they add. The VM garbage collector removes the evicted data, reducing heap use. The evictions continue until the manager determines that heap use is again below the eviction threshold.
- Critical threshold. Above this, all activity that might add data to the cache is refused. This threshold is set above the eviction threshold and is intended to allow the eviction and GC work to catch up. This VM, all other VMs in the distributed system, and all clients to the system receive `LowMemoryException` for operations that would add to this critical member's heap consumption. Activities that fetch or reduce data are allowed. For a list of refused operations, see the Javadocs for `ResourceManager.setCriticalHeapPercentage`.

> *Avoid passing the critical threshold. It might be better than a hang or an OME, but the GemFire member that becomes a critical VM is a read-only member that refuses cache updates for all of its regions, including incoming distributed updates.*

Figure 7.5 The Tenured Memory Thresholds

When heap use passes the eviction threshold, in either direction, the manager logs an info-level message. When it passes the critical threshold, the manager logs an error-level message.

How Background Eviction is Performed

When the manager kicks off evictions:

1. From all regions in the local cache that are configured for heap LRU eviction, the background eviction manager creates a randomized list containing one entry for each partitioned region bucket (primary or secondary) and one entry for each non-partitioned region. So each partitioned region bucket is treated the same as a single, non-partitioned region.
2. The background eviction manager starts ((# of processors on local machine) * 4) evictor threads, and passes each thread its share of the bucket/region list. The manager divides the bucket/region list as evenly as possible by count, and not by memory consumption.
3. Each thread iterates round-robin over its bucket/region list, evicting one LRU entry per bucket/region until the resource manager sends a signal to stop evicting.

Configuring the Resource Manager

You can use the resource manager in any GemFire member, but you may not want to use it everywhere. For some members it might be better to occasionally restart after a hang or OME crash than to evict data and/or refuse distributed caching activities. Also, members that do not risk running past their memory limits would not benefit from the overhead required to run the resource manager. Cache servers are often configured to use the manager because they generally host more data and have more data activity than other members, requiring greater responsiveness in data cleanup and collection.

For the members where you want to implement the resource manager functionality:

1. Configure GemFire for heap LRU management
2. Set the VM GC tuning parameters to handle heap and garbage collection in conjunction with the GemFire manager
3. Monitor and tune heap LRU configurations and your GC configurations:
   - Before going into production, run your system tests with application behavior and data loads that approximate your target systems so you can tune as well as possible for production needs
   - In production, keep monitoring and tuning to meet changing needs

Configuring GemFire for Heap LRU Management

The configuration terms used here are `cache.xml` elements and attributes, but you can also configure through the `ResourceManager` and `Region` APIs.

1. Set the `resource-manager` `critical-heap-percentage` threshold. This should be as close to 100 as possible while still low enough so the manager's response can prevent the VM from hanging or getting `OutOfMemoryError`. The threshold is zero (no threshold) by default.
2. Set the `resource-manager` `eviction-heap-percentage` threshold to a value lower than the critical threshold. This should be as high as possible while still low enough to prevent your VM from reaching the critical threshold. The threshold is zero (no threshold) by default.
3. Decide which regions will participate in heap eviction and set their `eviction-attributes` (page 108) to `lru-heap-percentage`. These regions should have enough data activity for the evictions to be useful and should contain data your application can afford to delete or offload to disk.

Example 7.11 Configuration for the Resource Manager

```
<resource-manager critical-heap-percentage="80" eviction-heap-percentage=
"60"/>
<region refid="REPLICATE_HEAP_LRU" />
```

Setting Your VM's GC Tuning Parameters

See your JVM documentation for all VM-specific settings that can be used to improve GC response.

At a minimum:

1. Set the initial and maximum heap switches, `-Xms` and `-Xmx`, to the same values.
2. Configure your VM for concurrent mark-sweep collector garbage collection.
3. If your VM allows, configure it to initiate concurrent mark-sweep collection before the GemFire resource manager thresholds are reached. You want the collector to be working when GemFire is evicting or the evictions will not give you more free memory.

VM	Conc mark-sweep switch flag	CMS initiation (begin at heap % N)
Sun HotSpot	`-XX:+UseConcMarkSweepGC`	`-XX:CMSInitiatingOccupancyFraction=N`
JRockit	`-Xgc:gencon`	`-XXgcTrigger:N`
IBM	`-Xgcpolicy:gencon`	N/A

For the `cacheserver`, pass these settings with the `-J` switch, like `-J-XX:+UseConcMarkSweepGC`.

Monitoring and Tuning Heap LRU Configurations

In tuning the resource manager, your central focus should be keeping the VM below the critical threshold. The critical threshold is provided to avoid VM hangs and crashes, but because of its exception throwing behavior for distributed updates, the time spent in critical negatively impacts the entire distributed system. To stay below critical, tune so that the GemFire eviction and the VM's GC respond adequately when the eviction threshold is reached.

Use the statistics provided by your VM to make sure your memory and GC settings are sufficient for your needs.

The GemFire `ResourceManagerStats` provide information about memory use and the manager thresholds and eviction activities. For general information on GemFire statistics, see to *System Statistics* on page 235 of the *GemFire Enterprise System Administrator's Guide*.

If you are spiking above the critical threshold on a regular basis, try lowering the eviction threshold. If you never go near critical, you might raise the eviction threshold to gain more usable memory without the overhead of unneeded evictions or GC cycles.

The settings that will work well for your system depend on a number of factors, including:

- The size of the data objects you store in the cache. Very large data objects can be evicted and garbage collected relatively quickly. The same amount of space in use by many small objects takes more processing effort to clear and might require lower thresholds to allow eviction and GC activities to keep up.
- Application behavior. Applications that quickly put a lot of data into the cache can more easily overrun the eviction and GC capabilities. Applications that operate more slowly may be more easily offset by eviction and GC efforts, possibly allowing you to set your thresholds higher than in the more volatile system.
- Your choice of VM. Each VM has its own GC behavior, which affects how efficiently the collector can operate, how quickly it kicks in when needed, and other factors.

In this sample statistics chart in VSD, the manager's evictions and the VM's GC efforts are good enough to keep heap use very close to the eviction threshold. The eviction threshold could be increased to a setting closer to the critical threshold, allowing the VM to keep more data in tenured memory without the risk of overwhelming the VM. This chart also shows the blocks of times when the manager was running cache evictions.

Figure 7.6 VSD chart of Resource Manager Statistics

In this chart, it looks like the manager's evictions are kicking in at the right time, but the concurrent mark sweep GC is not starting soon enough to keep memory use in check. It might be that it is not configured to start as soon as it should be - it should be started just before the eviction threshold is reached - or perhaps there is some other issue with the garbage collection service.

Figure 7.7 VSD Chart of Statistics with Late CMS Startup

Resource Manager Configuration Example

These examples set the critical threshold to 85 percent of the tenured heap and the eviction threshold to 75 percent. The region `bigDataStore` is configured to participate in the resource manager's eviction activities.

Example 7.12 cache.xml

```
<cache>
   <resource-manager critical-heap-percentage="85" eviction-heap-percentage=
"75"/>
   <region name="bigDataStore" refid="PARTITION_HEAP_LRU"/>
</cache>
```

Example 7.13 API

```
Cache cache = CacheFactory.create();

ResourceManager rm = cache.getResourceManager();
rm.setCriticalHeapPercentage(85);
rm.setEvictionHeapPercentage(75);

RegionFactory rf =
   cache.createRegionFactory(RegionShortcut.PARTITION_HEAP_LRU);
Region region = rf.create("bigDataStore");
```

Use Case Scenario

This is one possible scenario for the configuration used in the examples:

- A 64-bit Sun Java VM 1.6 VM, with 8 Gb of heap space on an 4 CPU system running Linux.
- The data region `bigDataStore` has approximately 2-3 million small values with average entry size of 512 bytes. So approximately 4-6 Gb of the heap is for region storage.
- The VM hosting the region also runs an application that may take up to 1 Gb of the heap.
- The application must never run out of heap space and has been crafted such that data loss in the region is acceptable if the heap space becomes limited due to application issues, so the default `lru-heap-percentage` action `destroy` is suitable.
- The application's service guarantee makes it very intolerant of `OutOfMemoryException`s. Testing has shown that leaving 15% head room above the critical threshold when adding data to the region gives 99.5% uptime with no `OutOfMemoryException`s, when configured with the CMS garbage collector using `-XX:CMSInitiatingOccupancyFraction=70`.

7.5 Controlling Data Flow With Application Plug-ins

This section covers the following:

- Data Distribution Through Application Plug-ins (page 193)
- Loading Entries Through a Data Loader (page 196)

Data Distribution Through Application Plug-ins

A region can act as a front to an outside data source and write back to the data source, using the `cache-loader` (page 102) and `cache-writer` (page 103) application plug-ins. A third plug-in, the `cache-listener` (page 102), provides after-event notification of changes to the region and its entries.

- **Cache writer**—In partitioned regions, a cache writer is only called if it is present on the primary data host, so install writers everywhere you host data.

  In distributed regions, cache writers are usually defined in only a subset—often only one—of the caches holding the region. Before performing any operation on a region entry, if any cache writers are defined for the region in the distributed system, the system invokes the most convenient writer.

- **Cache loader**—In partitioned regions, the cache loader must be installed in every data host. Partitioned regions support partitioned loading, where each cache loader loads only the data entries in the local partitioned region data host. If data redundancy is configured, data is loaded only if the local host holds the primary copy.

  In distributed regions, cache loaders are usually defined in only a subset of the caches. If a loader is needed, the system calls the available loaders one at a time, in order of convenience, until the data is loaded or all loaders have been tried.

- **Cache listener**—The cache listener must be installed in every cache that needs notification of changes. When a region's `interest-policy` is `cache-content`, the listener in that member is only notified of changes to primary data entries in that member's local cache. These local operations may be direct, cache-local API calls or they may arrive at the cache through data or event distribution. When the `interest-policy` is `all`, the listener in that member is notified of changes to primary data entries in the entire distributed region. In a partitioned region, a cache listener only fires in the member hosting the primary data copy. Listeners on secondaries are not fired.

The plug-ins defined in a region with `local` scope apply only to the cache where they are defined.

For more information on cache loaders, see *Loading Entries Through a Data Loader* on page 196. For more information on cache writers and listeners, see Chapter 11, *Handling Events*.

Plug-ins in a Partitioned Region

The members in this figure are members of one distributed system that participate in the same partitioned region. These members may be running on different machines. Because this is a partitioned region, every member must have a cache loader and every member must have a cache writer.

Every member can also have an optional cache listener. The listener receives events according to the partitioned region's `interest-policy`. So in this example, where M_3 has no primary data entries, its listener would see no events if its `interest-policy` is set to the default `cache-content`. If its `interest-policy` is set to `all`, however, its listener receives all events for the region, no matter where the primary data entries reside.

Figure 7.8 Application Plug-ins in a Partitioned Region

Plug-ins in a Distributed Region

The members in this figure, members of one distributed system, may be running on different machines. Loading for the distributed region is performed from member M_1 while cache writer functions are split between two members. The listener receives callbacks only for modifications that occur (either via direct API calls or distribution) in its local cache. In this configuration, operations in M_3 do not trigger events, because there is no local cache listener.

Figure 7.9 Application Plug-ins in a Distributed Region

Loading Entries Through a Data Loader

By default, a region has no data loader defined. You can plug an application-defined loader into any region by setting the region attribute `cache-loader` (page 102). The loader is called on cache misses during the `get` operation, and it can populate the cache with the new entry value in addition to returning the value to the calling thread. A loader can be configured to load data from an outside data store. The reverse operation, writing data from the cache to an outside data store, is done by a cache writer. For details, see *Implementing Cache Event Handlers* on page 300.

The placement and behavior of cache loaders depends on the type of region.

Data Loading in Partitioned Regions

Because of the huge amounts of data they can handle, partitioned regions support *partitioned loading*. Each cache loader loads only the data entries in the local cache where the loader is defined. If data redundancy is configured, data is loaded only if the member holds the primary copy. Partitioned loading requires a cache loader to be installed in every member that allows data caching.

If you depend on a JDBC connection, every data host must have a connection to the data source. Here the three members require three connections, compared to only one connection shared by the three members in the distributed example, Figure 7.11 on page 197.

> *Partitioned regions generally require more JDBC connections than other region types.*

Figure 7.10 Cache Loaders in a Partitioned Region

Data Loading in Distributed Regions

When associated with a non-partitioned distributed region, the cache loader plug-in is available to all other members that have the region defined. Loaders are usually defined in only a subset—often only one—of the caches holding the region. If a loader is needed, all available loaders for the region are invoked, starting with the most convenient loader, until the data is loaded or all loaders have been tried.

In the following figure, these members of one distributed system can be running on different machines. Loading for the distributed region is performed from M_1. The event listener only receives callbacks for modifications that occur (either via direct API calls or distribution) in its local cache.

Figure 7.11 Cache Loader in a Distributed Region

Data Loading in Local Regions

For local regions, the cache loader is available only in the member where it is defined. If a loader is defined, it is called whenever a value is not found in the local cache.

Implementing a Cache Loader

To create your loader, implement `com.gemstone.gemfire.cache.CacheLoader`. To declare the loader in a `cache.xml` file, also implement `com.gemstone.gemfire.cache.Declarable`. The `CacheLoader` interface has a single callback method, `load`, which you can program to do whatever your application requires for retrieving the value from outside the cache. For more information on the `load` method, see the online Java documentation.

> *You must not make any direct calls to* `Region` *methods from your load method implementation. Doing so could cause the cache loader to block, which would hurt the performance of the distributed system. Instead, spawn separate threads to perform* `Region` *API calls.*

This example lists a `cache.xml` that declares a cache loader named `myCacheLoader`.

Example 7.14 Declaring a Cache Loader in cache.xml

```
<region name="exampleRegion">
   <region-attributes>
      <cache-loader>
         <class-name>myCacheLoader</class-name>
      </cache-loader>
      ...
   </region-attributes>
</region>
```

Alternately, the cache loader can be defined through the API.

Example 7.15 Implementing a Simple CacheLoader

```
public class SimpleCacheLoader implements CacheLoader, Declarable {
   public Object load(LoaderHelper helper) {
      String key = (String) helper.getKey();
      System.out.println(" Loader called to retrieve value for " + key);
   // Create a value using the suffix number of the key (key1, key2, etc.)
      return "LoadedValue" + (Integer.parseInt(key.substring(3)));
   }
   public void close() { // do nothing }
   public void init(Properties props) { // do nothing }
}
```

This example sets a cache loader for a new region:

Example 7.16 Creating a Region with a CacheLoader Through the API

```
RegionFactory<String,Object> rf = cache.createRegionFactory(REPLICATE);
rf.setCacheLoader(new QuoteLoader());
rf.setEntryTimeToLive(new ExpirationAttributes(20));
if ( aListener != null ) {
   rf.addCacheListener(aListener);
}

//create the region
quotes = rf.create("NASDAQ Quotes");
```

Declaring a Cache Loader

The cache loader can be declared in the `cache.xml` file or through the API, and it runs inside the defining member. This declaration specifies the loader, `quickstart.SimpleCacheLoader`, for a region.

Example 7.17 Declaring SimpleCacheLoader

```
<region-attributes>
   <cache-loader>
      <class-name>quickstart.SimpleCacheLoader</class-name>
   </cache-loader>
</region-attributes>
```

This declaration specifies a loader with a JDBC connection to a database. For the full XML file, see Example B.2 on page 501.

Example 7.18 Declaring a CacheLoader with a Database Connection

```
<cache-loader>
   <class-name>com.company.data.DatabaseLoader</class-name>
   <parameter name="URL">
      <string>jdbc:cloudscape:rmi:MyData</string>
   </parameter>
</cache-loader>
```

Chapter 8

Client/Server Architecture and Configuration Basics

GemFire Enterprise® provides mechanisms for scaling vertically between disparate distributed systems by defining client/server relationships between two distinct distributed system tiers. This chapter introduces vertical scaling using GemFire Enterprise, giving basic information on systems and cache configuration and options for managing data flow between your clients and servers. Chapter 9, *Client/Server Advanced Topics*, on page 227 continues the discussion with higher-level topics such as connection and resource management, customized load balancing, and configuring for high availability.

In this chapter:

- Client/Server Caching Features (page 202)
- Standard Client/Server Deployment (page 203)
- The Client/Server API (page 206)
- Configuration Properties and Examples (page 207)
- Client/Server Data Flow (page 216)
- Example Scenario: Very Small Client Caches (page 224)

8.1 Client/Server Caching Features

GemFire Enterprise client server caching provides these core features:

- **Dynamic server discovery**—The GemFire server locator utility dynamically tracks server processes and directs clients to new servers, giving clients indirection from explicit server information. Clients only need to know how to connect to the locator services. They do not need to know where servers are running or how many servers are available at any time.
- **Server groups**—You can assign your servers to logical groups that your clients can refer to in their connection configurations. For example, you might use groups to manually partition your data, with one group of servers hosting one set of data and another hosting another set. Or you might use a group to direct all database-centric traffic to the subset of servers that are directly connected to a backend database. Servers can belong to multiple groups. Your clients only need to specify the group to use and are isolated from having to know which servers belong to which groups.
- **Server load balancing** —The GemFire server locator tracks current load information for all servers, directing new client connections to the servers with the least load. GemFire provides a default load probe for your servers, which you can replace if with your own customized plug-in.
- **Server connection conditioning**—Client connections can be configured to transparently time out and be replaced with new connections, which allows overall server use to be rebalanced after new servers are started. This helps speed conditioning in situations such as adding servers or recovery from server crashes and other downtimes.
- **Automated data and query updates**—Your clients can subscribe to events in the server caches. These events may include cached data updates and updates to results for continuos queries that the client has registered with the server. The server uses subscription queues to send the updates asynchronously.
- **Server failover and high availability**—When servers crash, the client connections automatically fail over to the remaining servers. If the servers are sending automated updates to the clients, the update requests also automatically fail over. You can configure redundancy in your server subscription queues so that the failover does not interrupt the stream of cache events from the server side.

8.2 Standard Client/Server Deployment

In the client/server topology, a farm of cache servers provides caching services to a large number of clients:

- Cache servers provide a homogeneous datastore for clients. The server data regions are replicated or partitioned across the server farm.
- Cache servers send their address and load information to the server locator.
- The clients go to the server locator for server connection information. For each request, the locator responds with the address of the least loaded server.
- Client connections are checked periodically for server load balancing and conditioned as needed.
- Clients can subscribe to events at startup. Events are automatically streamed from the servers and sent to client listeners and into the client cache.
- Client API calls, like data updates and data gets, that aren't satisfied by the client cache are forwarded automatically to the servers.

Figure 8.1 Standard Client/Server Installation

Simple Configuration for Server Discovery

With server locators, clients are configured with a locator list and locators maintain a dynamic server list. The locator listens at an address and port for connecting clients and gives the clients server information. The clients are configured with locator information and have no configuration specific to the servers.

This figure shows the GemFire processes. Only one locator is shown, but the recommended configuration uses multiple locators for high availability.

Figure 8.2 Client/Server Installation using Locators

The locator and servers have the same peer discovery configured in their `gemfire.properties`:

```
locators=lucy[41111]
```

The servers, run on their respective hosts, have this cache-server configuration in their `cache.xml`:

```
<cache-server port="40404" ...
```

The client's `cache.xml` contains this pool configuration and region attributes assignment:

```
<pool name="PoolA" ...
   <locator host="lucy" port="41111">
<region-attributes pool-name="PoolA" ...
```

Configuration With Server Grouping

You can control which servers your clients use with named server groups. Do this if you want your servers to manage different data sets or to direct specific client traffic to a subset of servers, such as those directly connected to a back-end database.

To split data management between servers, configure some servers to host one set of data regions and some to host another set. Assign the servers to two separate server groups. Then, define two separate server pools on the client side and assign the pools to the proper corresponding client regions.

This figure shows the GemFire processes in a client/server installation where one set of servers hosts a Portfolios region and another hosts a Products region. In this figure, the client use of regions is also split, but you could have both pools and both regions defined in all of your clients.

Figure 8.3 Client/Server Installation Using Server Groups

For grouping, the server and client configurations have an added *server group* specification. This is the `cache-server` declaration for cacheserver 1:

```
<cache-server port="40404" ...
   <group>Portfolios</group>
```

And the `Pool` declaration for `Client1`:

```
<pool name="PortfolioPool" server-group="Portfolios"...
   <locator host="lucy" port="41111">
<region-attributes pool-name="PortfolioPool" ...
```

`Server2` and `Client2` are similarly configured, with group and pool names for the `Products` group.

8.3 The Client/Server API

The main client/server configuration can be done through the `cache.xml` or through the API. Clients use the API to interface with servers during runtime. This section lists the core API components. Most of these can be defined in the `cache.xml` or through the `Cache`, `ClientCache`, and `Region` APIs. The `BridgeMembershipListener` must be defined through the API.

The XML DTD is listed in Appendix B, *cache.xml File*, on page 499.

com.gemstone.gemfire.cache.server

- **CacheServer**—The server instance defined at the cache level for the server member. Multiple server instances can be defined for a single cache. For each, you must specify a unique port on the member's host machine where the server listens for client connection communication.
- **ServerLoadProbe**—Application plug-in used to provide server load information to the server locators for use in load balancing and conditioning. The `getLoad` method of this interface returns a `ServerLoad` object with information on current connection loads as well as estimates of how much additional load a new connection would add to the pool connection load or the subscription connection load. GemFire provides a default load probe, which you can override with this plug-in.

com.gemstone.gemfire.cache.client

- **ClientCache**—The interface for client caching. This is like the Cache API, but tailored for client activities.
- **Pool**—The interface the client uses to access its servers. The `Pool` is created at the cache level, then is named in the `RegionAttributes` for every region that uses it. This pool defines the server locators that maintain the server connection lists for client applications. It also defines the parameters for communication between the clients and servers. Instances of Pool are configured, created, and managed using the `PoolFactory` and `PoolManager` interfaces in this package.

com.gemstone.gemfire.cache

- **RegionAttributes**—This interface must name a server pool in the `pool-name` region attribute. Setting the pool name defines the region as a client region. The pool must be defined for the region to access the server tier.

com.gemstone.gemfire.cache.util

- **BridgeMembershipListener**—A listener for client/server connection events defined for the client or server member. Clients can receive membership events regarding their servers, and servers can receive membership events regarding their clients. For more information on this and related interfaces and classes, see Chapter 11, *Handling Events*, on page 293.

  To handle cache events in a client or server region, use a `CacheListener`*, not a* `BridgeMembershipListener`.

8.4 Configuration Properties and Examples

You can configure your clients through the API or in the `cache.xml`. If you run your servers as GemFire applications, you can configure them also through the API or in the `cache.xml`. Servers that are run using the GemFire `cacheserver` process use only `cache.xml` configuration. For information on the `cacheserver`, see *Configuring and Running the GemFire Cache Server* on page 126 of the *GemFire Enterprise System Administrator's Guide*.

Server Configuration

To provide a coherent cache view to their clients, server regions must be partitioned or replicated.

- Replicated server regions must have `distributed-ack` or `global` `scope` (page 116) and every server that defines the region must store data. When you define your region, use any of the `REPLICATE` `RegionShortcut` options except for `REPLICATE_PROXY`. The shortcuts use `distributed-ack` `scope`.
- Partitioned server regions can be defined with local data storage in some servers and no local storage in others. When you define your region, use the `PARTITIONED` `RegionShortcut` options.

  If you do not configure your server regions as partitioned or replicate, you can get unexpected results for calls that check server region contents, like `keySetOnServer` *and* `containsKeyOnServer`*. You may get only partial results and you may get inconsistent responses from two consecutive calls. This is because the servers may not have a complete view of the data, and they only report on their local cache content.*

This table describes the configuration parameters for the server's `cache.xml` `<cache>` and `<cache-server>` elements. The `Cache` and `CacheServer` APIs provide corresponding setter and getter methods, which are detailed in the online Java documentation.

Table 8.1 Server Configuration Parameters

Server Configuration Parameter	Description
`<cache>`	
`message-sync-interval`	The frequency, in seconds, that the primary server sends subscription queue synchronization messages to its secondaries. Default: 1. For details, see *Keeping Redundant Servers Synchronized* on page 242.
`<cache-server>`	
`bind-address`	The hostname or IP address that the server is to listen on for client connections. If null, the server listens on the machine's default address. Default: null.
`custom-load-probe`	Application plug-in used to provide current and predicted server load information to the server locators. If this is not defined, the default GemFire load probe is used. Default: null.

Server Configuration Parameter	Description
`client-subscription`	How to handle client subscription queues. This is used to set a capacity limit on the in-memory queue and to specify where to overflow when capacity is reached. Specified in three parts with `eviction-policy`, `capacity`, and `disk-store-name`. Valid settings are `eviction-policy` of `mem` with `capacity` specifying the megabytes or memory, or `eviction-policy` of `entry` and `capacity` specifying the entry count. The `disk-store-name` must specify a disk store that is already defined in the cache. If `eviction-policy` is not specified, no overflow is used. By default it is not specified. For more information see *Overflowing the Client Queue to Disk* on page 257.
`group`	The group or groups this server belongs to for servicing clients. Every server belongs to the global server group. This property is used to assign the server to more fine-grained groups. This is used to split the servers in the pool into functional groups. In the `cache.xml` file, this is an attribute that accepts multiple values. Through the API it is set as a `String` array. Default: null.
`hostname-for-clients`	The hostname or IP address to pass to the client as the location where the server is listening. When the server connects to the locator it passes the host/port where it listens for client connections. If the host the server uses by default is one that the client can't translate into an IP address, the client will have no route to the server's host. For this situation, you must supply the server's alternate hostname for the locator to pass to the client. If null, the server's `bind-address` (page 207) setting is used. Default: null.
`load-poll-interval`	The frequency, in milliseconds, to poll the load probe for load information on the server. Default: 5000 (5 seconds).
`max-connections`	The maximum total number of client connections for the server. When the maximum is reached, the server refuses additional client connections. Default: 800. For details, see Setting the Maximum Client Connections Accepted by a Server (page 261). *This property should be set at least as high as* `max-threads`.
`max-threads`	The maximum number of threads allowed in this server to service client connections. When the limit is reached, server threads begin servicing multiple connections. A zero setting causes the server to use a thread for every client connection. Default: 0. See Controlling the Number of Server Threads Used for Clients (page 260). *This property should be set no higher than* `max-connections`. *This property is ignored on Windows if you are running with the JRockit VM and/or using internet version 6 protocol IPv6.*

Server Configuration Parameter	Description
`maximum-message-count`	Available only if `client-subscription` (page 208) is not configured. The maximum number of messages allowed in a subscription queue. When the queue reaches this limit, messages block. Default: `230000`.
`maximum-time-between-pings`	The maximum time, in milliseconds, the server allows to pass between messages or pings indicating a client is healthy. Default: 60000 (1 minute). *A setting of `0` or a negative number turns off client health monitoring. Be careful not to do this accidentally.*
`message-time-to-live`	Setting used for highly available subscription queues. The expiration time, in seconds, for non-durable messages in the secondary server's client subscription queue. The system removes non-durable messages that have been in the queue beyond this time. If set to 0 (zero), the messages are never removed. Default: 180 (3 minutes). *Set this high enough to avoid removing messages that are still valid, or you might lose messages to your clients on server failover.*
`port`	The port the server listens on for client communication. Default: `40404`.
`socket-buffer-size`	The size for socket buffers from the server to the client. For details, see Socket Buffer Size (page 265). Default: `32768`.

Client Cache Configuration

You configure a client using the `<client-cache>` XML and `ClientCache` API.

In the `cache.xml` file, use the `client-cache` DOCTYPE and element, according to the dtd listed in *GemFire Enterprise Cache XML DTD* on page 505.

Example 8.1 Client cache.xml

```
<?xml version="1.0"?>
<!DOCTYPE client-cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">

<client-cache>
    ...
</client-cache>
```

Client Pool Configuration

This table describes the configuration parameters for the client's `cache.xml` `<pool>` element. The `PoolFactory` and `Pool` APIs provide corresponding setter and getter methods, which are detailed in the online Java documentation. Except where noted, all parameters are optional.

Table 8.2 Client Pool Configuration

Client Pool Attribute	Description
`free-connection-timeout`	The amount of time a thread will wait to get a pool connection before timing out with an exception. This timeout keeps threads from waiting indefinitely when the pool's `max-connections` has been reached and all connections in the pool are in use by other threads. Default: `10000`.
`idle-timeout`	The maximum time, in milliseconds, a connection can stay open without being used when there are more than `min-connections` open. Pings do not count as connection use. If set to -1, there is no idle timeout. Default: `5000`
`load-conditioning-interval`	The amount of time, in milliseconds, a pool connection can remain open before being eligible for silent replacement to a less-loaded server. Default: 300000 (5 minutes).
`locator`	The addresses and ports of the locators to connect to. You can define multiple locators for the pool. You must provide a locator list or the `server` (page 211) list, but not both.
`max-connections`	The maximum number of pool connections that the pool can create. If all of the connections are in use, an operation requiring a client to server connection blocks until a connection becomes available or the `free-connection-timeout` is reached. If set to -1, there is no maximum. The setting must indicate a cap greater than `min-connections`. Default: `-1`. *If you need to use this to cap your pool connections, you should disable the pool attribute* `pr-single-hop-enabled`. *Leaving single hop enabled can increase thrashing and lower performance.*
`min-connections`	The minimum number of connections to keep available. This is used to establish the initial connection pool. If set to 0 (zero), no connection is created until an operation requires it. This is the starting point, with more connections added later as needed, up to the `max-connection` setting. The setting must be an integer greater than or equal to 0. Default: `1`.
`multiuser-authentication`	Used for installations with security where you want to accommodate multiple users within a single client. If set to `true`, the pool provides authorization for multiple user instances in the same client application, and each user accesses the cache through its own `RegionService` instance. If `false`, the client either uses no authorization or just provides credentials for the single client process. Default: `false`. See *Creating and Closing RegionService Instances in a Client with Multiple Secure Users* on page 82

Client Pool Attribute	Description
`name`	Name of this pool. Used in the client region `pool-name` to assign this pool to a region in the client cache. This is a required field with no default value.
`ping-interval`	How often to communicate with the server to show the client is alive, set in milliseconds. Pings are only sent when the ping-interval elapses between normal client messages. This must be set lower than the server's `maximum-time-between-pings` (page 209). Default: `10000`.
`pr-single-hop-enabled`	Setting used to improve access to partitioned region data in the servers. Indicates whether to use metadata about the partitioned region data storage locations to decide where to send some data requests. This allows a client to send a data operation directly to the server hosting the key. Without this, the client contacts any available server and that server contacts the data host. This is used only for operations that can be carried out on a server-by-server basis, like `put`, `get`, and `destroy`. Default: `true`. See *Single-Hop Partitioned Region Data Access* on page 236.
`read-timeout`	The amount of time, in milliseconds, for the client to wait for a response from a server. For details, see *Socket Timeout* on page 266. Default: `10000`.
`retry-attempts`	The number of times to retry a client request before giving up. If one server fails, the pool moves to the next, and so on until it is successful or it hits this limit. If the available servers are fewer than this setting, the pool will retry servers that have already failed until it reaches the limit. If this is set to -1, the pool tries every available server once. Default: -1.
`server`	The addresses and ports of the servers to connect to. You can define multiple servers for the pool. You must provide a server list *or* the `locator` (page 210) list, but not both.
`server-group`	The logical named server group to use from the pool. A null value uses the global server group to which all servers belong. This is only used when the `locator` (page 210) list is defined. This specification allows clients some control over the servers they use. Without server groups, the locators can direct the clients to any of the servers available in the server distributed system. Default: null.
`socket-buffer-size`	The size for socket buffers from the client to the server. For details, see *Socket Buffer Size* on page 265. Default: `32768`.
`statistic-interval`	The interval, in milliseconds, at which to send client statistics to the server. If set to -1, statistics are not sent. Default: `-1`.

Client Pool Attribute	Description
`subscription-ack-interval`	Used only when `subscription-redundancy` (page 212) is not equal to ‘0’ (zero). This is the amount of time, in milliseconds, between messages to the primary server to acknowledge event receipt. Default: `500`.
`subscription-enabled`	Configures the client to receive callbacks from the server. If `true`, the server connects back to the client and automatically sends server-side cache update information for all regions. Any bind address information for the client is automatically passed to the server for use in the callbacks. For details, see Server-Initiated Data Flow (page 219). Default: `false`.
`subscription-message-tracking-timeout`	The time-to-live, in milliseconds, for entries in the client’s message tracking list. For information about this list, see *Discarding Duplicate Events* on page 239. Default: `900000` (15 minutes).
`subscription-redundancy`	The number of servers to use as backup to the primary server for subscription queue management. If set to 0, no secondaries are used. If set to -1, all available servers are used as secondaries. For details, see *Highly Available Subscription Delivery* on page 241. Default: `0`.
`thread-local-connections`	Boolean specifying whether connections are sticky. A `true` setting causes the connection to stick to the thread for multiple requests. A `false` setting causes each connection to be returned to the pool after a request finishes. A sticky connection is returned to the pool when the thread releases it through `Pool releaseThreadLocalConnection`, when the `idle-timeout` (page 210) is reached, or when the pool is destroyed. Default: `false`.

Examples

In the standard example configuration, locators and servers use the same distributed system properties file, which lists locators as the discovery mechanism for peer members and for connecting clients. These are the key lines in the `gemfire.properties` file for locator and server:

Example 8.2 Locator and Server GemFire Properties

```
mcast-port=0
locators=localhost[41111]
```

The locator is started with this line. By default the locator provides peer and server discovery.

```
gemfire start-locator -port=41111
```

The server's `cache.xml` declares a cache server element, which defines the VM as a server in the distributed system. Note that these are the same settings as for the static server list configuration example.

Example 8.3 Server Configuration

```
<cache>
   <cache-server port="40404" ... />
   <region . . .
```

Once the locator and server are started, the locator knows the server is a peer in its distributed system and that it is waiting to service client connections at port 40404.

The client's `cache.xml` `client-cache` declaration automatically configures it as a standalone GemFire application. The XML:

- Declares a single connection pool with the locator as the reference for obtaining server connection information.
- Creates `cs_region` with the client region shortcut configuration, `CACHING_PROXY`. This configures it as a client region that stores data in the client cache.

There is only one pool defined for the client, so the pool is automatically assigned to all client regions.

Example 8.4 Client Pool and Region Configuration

```
<client-cache>
   <pool name="publisher" subscription-enabled="true">
      <locator host="localhost" port="41111"/>
   </pool>
   <region name="cs_region" refid="CACHING_PROXY">
   </region>
</client-cache>
```

With this, the client is configured to go to the locator for the server connection location. Then any cache miss or put in the client region is automatically forwarded to the server.

For more examples of client cache configuration, see *Creating and Closing a Cache* on page 78.

This API example creates a standalone publisher client and the client pool and region.

Example 8.5 Publisher Client Configuration Through the API

```
public static ClientCacheFactory connectStandalone(String name) {
    return new ClientCacheFactory()
        .set("log-file", name + ".log")
        .set("statistic-archive-file", name + ".gfs")
        .set("statistic-sampling-enabled", "true")
        .set("cache-xml-file", "")
        .addPoolLocator("localhost", LOCATOR_PORT);
}

private static void runPublisher() {
    ClientCacheFactory ccf = connectStandalone("publisher");
    ClientCache cache = ccf.create();
    ClientRegionFactory<String,String> regionFactory =
        cache.createClientRegionFactory(PROXY);
    Region<String, Strini> region = regionFactory.create("DATA");

    //... do work ...

     cache.close();
}
```

This API example creates a standalone subscriber client using the same `connectStandalone` method.

Example 8.6 Subscriber Client Configuration Through the API

```
private static void runSubscriber() throws InterruptedException {
    ClientCacheFactory ccf = connectStandalone("subscriber");
    ccf.setPoolSubscriptionEnabled(true);
    ClientCache cache = ccf.create();
    ClientRegionFactory<String,String> regionFactory =
        cache.createClientRegionFactory(PROXY);
    Region<String, String> region = regionFactory
        .addCacheListener(new SubscriberListener())
        .create("DATA");
    region.registerInterestRegex(".*", // everything
        InterestResultPolicy.NONE,
        false/*isDurable*/);
    SubscriberListener myListener =
        (SubscriberListener)region.getAttributes().getCacheListeners()[0];
    System.out.println("waiting for publisher to do " + NUM_PUTS + "
puts...");
    myListener.waitForPuts(NUM_PUTS);
    System.out.println("done waiting for publisher.");

     cache.close();
}
```

Client/Server with Static Server List

You can specify a static server list in the client configuration instead of a locator list. In this case, the client uses only the specified servers for its connections. With this option, you do not have dynamic server discovery, server load conditioning, or the option of logical server grouping. This model can be useful for small deployments, especially in test systems, where your server pool is stable and you do not want the administrative overhead of running locators. Also, if you are required to use hardware load balancers, you can put the addresses of the load balancers in your server list and allow the balancers to redirect your client connections.

In this configuration, clients explicitly list their servers in their cache configuration. The client's server specification must match the addresses where the servers are listening. This example lists the pertinent settings in the server cache configuration file:

Example 8.7 Server Configuration for Static Server List Configuration

```
<cache>
   <cache-server port="40404" ... />
   <region . . .
```

The client's `gemfire.properties` makes it a standalone GemFire application:

Example 8.8 Client GemFire Properties for Running Standalone

```
mcast-port=0
locators=
```

The client's `cache.xml` declares a connection pool with the server explicitly listed and names the pool in the attributes for the client region.This XML file uses a region attributes template to initialize the region attributes configuration.

Example 8.9 Client Pool and Region for Static Server List Configuration

```
<client-cache>
   <pool name="publisher" subscription-enabled="true">
      <server host="localhost" port="40404"/>
   </pool>
   <region name="cs_region" refid="CACHING_PROXY">
   </region>
</client-cache>
```

With this configuration, the client's server information does not change for the life of the client VM.

8.5 Client/Server Data Flow

Clients initiate data flow when they send gets and cache updates to the server side. In addition, clients can request automated cache updates from the server side, which are sent via asynchronous subscription queues.

This section describes how cached data moves between clients and servers and discusses related configuration options. For information on the mechanisms used to send data between the client and server, see Chapter 9, *Client/Server Advanced Topics*, on page 227.

In this section:

- Client-Initiated Data Flow (page 217)
- Server-Initiated Data Flow (page 219)
- Data Flow Between Clients (page 223)

Client-Initiated Data Flow

Client-initiated data flow uses the client pool connections to communicate data requests and updates to the server. When you name a `Pool` for your region, the client automatically sends data requests to the server for cache misses and forwards updates to the server for relevant client-side entry and region operations.

Data Requests

When a client cache can directly fulfill a client data request, it does so. On cache misses the client region's pool automatically requests the data from a server. If the server cannot satisfy the request, the client uses all other resources available to it to find the data. For the complete order of steps taken to satisfy the data request, see *Automated Steps in Entry Retrieval Based on Region Scope - non-Partitioned* on page 137.

Data Updates

Client cache modifications pass first through a client `CacheWriter`, if one is defined, then to the server through the client pool, and then finally to the client cache itself. A cache writer, either on the client or server side, may abort the operation.

This figure shows the data flow for entry updates that originate on the client. The event in the client is forwarded to the server through the pool. The event is then distributed to the server's peers and other clients according to the server, server region, and other clients' configurations.

Figure 8.4 Client-to-Server Entry Update Flow

This table lists the client-side modifications that propagate to the server.

Table 8.3 ClientCache Updates That Affect the Server

Change in ClientCache	Effect on Server Cache
Entry creation or update	Creation or update of entry.
Distributed entry destruction	Entry destroyed. The destroy call is propagated to the server even if the entry is not in the client cache.
Distributed region destruction	Region destroyed.
Distributed region clear	Region cleared.

Invalidations on the client side are not forwarded to the server.

Distributed entry destruction is forwarded to the server even if the entry is not present in the client's cache. In this way, the client can destroy an entry on the server regardless of the entry's local cache state.

Server-Initiated Data Flow

You can configure your servers to automatically push cache updates to clients. To do this, create your client `Pool` with `subscription-enabled` (page 212) set to `true` and register client interest in events on the server. The server uses a queue to manage synchronous messaging to the client pool.

These are the events the server pushes to the client:

- Changes to continuous query (CQ) results for queries the client has registered with the server. For more information on this, see Chapter 14, *Continuous Querying*, on page 397.
- Changes to region entries for which the client has registered interest. See *Client Interest Registration* on page 220.

  If you have dependencies between CQs and interest subscriptions, so that you want the two types of subscription events to arrive as closely together on the client, use a single `Pool` *for everything. Using different pools can lead to time differences in the delivery of events because the* `Pool`*s might use different servers to process and deliver the event messages.*

This figure shows the complete data flow for interest registrations.

Figure 8.5 Client Notification of Server Events

This table shows how server-side changes are pushed to the client.

Table 8.4 Server Updates Sent to the Client

Change in Server Cache	Effect on ClientCache
Entry create or update	For subscriptions with `receiveValues` set to true, entry create or update. For subscriptions with `receiveValues` set to false entry invalidate if the entry already exists in the client cache; otherwise, no effect. The next client `get` for the entry is forwarded to the server.
Entry invalidate or destroy (distributed only)	Entry invalidate or destroy
Region destroy or clear (distributed only)	Region destroy or local region clear

Region invalidation in the server is not forwarded to the client.

The server-side distributed operations are all operations that originate as a distributed operation in the server or one of its peers.

To maintain a unified set of data in your servers, do not do local entry invalidates in your server regions.

Client Interest Registration

The client's interest list registration is done through the API using the `registerInterest` and the `registerInterestRegex` methods. You can call these methods multiple times. The server maintains the union of all of the specifications in a list. The server maintains the list separate from the region. The list can contain specifications for entries that are not currently in the server region. Once interest is registered, the server sends the client updates for changes to all entries specified in the list.

You cannot register interest for a client region that is configured as a replicate. Replication and register interest are incompatible because, with registered interest, the client's region may be reinitialized during failover to a new server. This would temporarily empty the region and force it to not replicate the full distributed region data set, which violates the replicate contract. For more information, see *Durable Subscription Queues* on page 246.

Configuration

Set the client `subscription-enabled` (page 212) to `true`.

Interest Specification

Register interest by specifying keys or specifying regular expressions to compare to key strings.

- The `registerInterest` method takes one of the following:
  - Single key - registers interest in one key only
  - Key List - registers interest in a List containing key specifications
  - ALL_KEYS - special token used to register interest in the entire server region
- The `registerInterestRegex` method takes a regular expression, which is matched against the keys in the server region using the standard `java.util.regex` methods. Any entry in the server region whose key satisfies the expression is kept updated on the client.
- All methods take an optional `receiveValues` parameter, indicating whether to send values with entry create and update events, or to send invalidations instead. If `receiveValues` is `false`,

invalidations are sent. For invalidations, the client updates the values lazily from the server as needed.

Each interest registration adds to the server's list of registered interest criteria for the client. So if a client registers interest in key 'A', then registers interest in regular expression "B*", the server will send updates for all entries with key 'A' or key beginning with the letter 'B'.

Bulk Loading During Registration

You can request an initial bulk load for your interest registration using the `InterestResultPolicy` parameter. If a bulk load is requested, all available entries that satisfy the interest registered are immediately sent from the server to the client region before control is returned to the calling method. If the server region is partitioned, the entire partitioned region is used in the bulk load. If the server region is distributed, only the server's local cache region is used. The interest results policy options are:

- **`KEYS`**—The client receives a bulk load of all available keys matching the interest registration criteria.
- **`KEYS_VALUES`**—The client receives a bulk load of all available keys and values matching the interest registration criteria. This is the default interest result policy.
- **`NONE`**—The client does not receive any immediate bulk loading.

No events are generated by the register interest calls, even if they load values into the client cache.

The Registration Process

During the register interest call, the system silently destroys entries in the client cache to get rid of any old data. This doesn't generate any events.

- For the `registerInterest` method, the system destroys all of the specified keys, leaving other keys in the client region alone. So if you have a client region with keys A, B, and C and you register interest in the key list A, B, at the start of the `registerInterest` operation, the system destroys keys A and B in the client cache but does not touch key C.
- For the `registerInterestRegex` method, the system silently destroys all keys in the client region.

The interest registration process follows these steps:

1. The entries in the client region that may be affected by this interest registration are silently destroyed. For regular expression interest registration, the region is cleared of entries.
2. The interest specification is sent to the server, where it is added to the client's interest list.
3. If indicated by the interest result policy, the server sends the client a bulk load of the keys or keys and values that currently satisfy the interest specification.

Interest Registration Examples

This example shows use of `registerInterest` requesting notification for a single key, for multiple keys, and for all keys in the region.

Example 8.10 Client Interest List Registration

```
// Register Interest in a Single key and download its entry at this time, if
it is available in the server cache
Region region1 = . . . ;
region1.registerInterest("key-1");

// Register Interest in a List of Keys but do not do an initial bulk load
// do not send values with events - just send the key with invalidation
Region region2 = . . . ;
List list = new ArrayList();
list.add("key-1");
list.add("key-2");
list.add("key-3");
list.add("key-4");
region2.registerInterest(list, InterestResultPolicy.NONE, false);

// Register Interest in All Keys and download all available keys at this
time
Region region3 = . . . ;
region3.registerInterest("ALL_KEYS", InterestResultPolicy.KEYS);
```

The `registerInterestRegex` method registers interest in a regular expression pattern. The server automatically sends the client changes for entries whose keys match the specified pattern. This example shows the interest registration for all keys whose first three characters are a single alphabetic character followed by an underscore followed by an integer.

Example 8.11 Client Interest List Registration Using a Regular Expression

```
// Register Interest in all keys matching a regular expression
Region region1 = . . . ;
region1.registerInterestRegex("[a-zA-Z]+_[0-9]+");
```

Data Flow Between Clients

In a normal installation, with many clients connected to the same server group. Each client's updates are propagated to the servers, and the server-side update is propagated to the other clients that are connected and have subscriptions enabled. The server does not return the update to the sending client.

The update is passed to the server and then passed to every other client that has registered interest in the entry key.

Figure 8.6 Data Update Flow With Multiple Clients

These are the steps in the figure:

1. Entry X is updated or created in Region A through a direct API call on $Client_1$.
2. The update to the region is passed to the pool named in the region.
3. The `Pool` propagates the event to the Cache Server, where the region is updated.
4. The server member distributes the event to its peers and also places it into the subscription queue for $Client_2$ because that client has previously registered interest in entry X.
5. The event for entry X is sent out of the queue to $Client_2$. When this happens is indeterminate.

If the $Client_2$ specified `receiveValue=false`, the data flow is the same except $Client_2$ receives an invalidate for entry X instead of receiving the new value.

8.6 Example Scenario: Very Small Client Caches

This scenario uses a client/server cache configuration to provide directed get and put actions between a tier of clients and a tier of servers.

The scenario involves a number of clients and a number of servers with access to a data source. The clients are in one distributed system, and their region has `local` scope. The servers are in a separate distributed system with the region defined as a replicate. Thus, the clients do not receive distributions from other instances of the client region, while the servers all have the same region data. All servers are configured with the `CacheServer` interface. The client regions use a `Pool` that communicates with the servers. The servers have the most up-to-date data and the clients send data requests and updates to the servers. An empty `data-policy` (page 104) keeps data out of the client regions altogether, ensuring that the clients get all data from their servers.

Figure 8.7 Client/Server Caching With Very Small Client Cache

The next two examples list the cache declarations for the servers and clients depicted here. This first configuration file is used to start servers on two machines, `stanley` and `ollie`.

In this example, note that:

- The region is configured as a replicate. This way the servers all cache the same data for client cache consumption.
- The server has a loader which it uses for its cache misses. A cache writer might also be added to keep the backend data source updated when client updates are forwarded to the server.

Example 8.12 Server Cache Configuration File

```
<cache>
   <cache-server port="40404"/>
   <region name="bridge_region" refid="REPLICATE">
      <region-attributes>
         <cache-loader>
            <class-name>cacheRunner.StringLoader</class-name>
         </cache-loader>
      </region-attributes>
   </region>
</cache>
```

The client-side configuration shows:

- The declarations of the locator running in the server's distributed system. The servers running on `stanley` and `ollie` are identified by the locator and their information is passed to the client, so the client has no need of an explicit server listing.
- No cache storage in the client region because of the `PROXY` client region shortcut setting.

Example 8.13 Client Cache Configuration File

```
<client-cache>
   <pool name="client" subscription-enabled="true">
      <locator host="localhost" port="41111"/>
   </pool>
   <region name="bridge_region" refid="PROXY" />
</client-cache>
```

Chapter 9

Client/Server Advanced Topics

GemFire Enterprise® provides imechanisms for scaling vertically between disparate distributed systems by defining client/server relationships between two distinct distributed system tiers. Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201 introduces vertical scaling using GemFire Enterprise, giving basic information on systems and cache configuration and options for managing data flow between your clients and servers. This chapter continues the discussion with higher-level topics such as connection and resource management, high availability, and optimizations for your client read and write configurations.

In this chapter:

- Client/Server Connections (page 228)
- Balancing Server Use (page 232)
- Client Operations to Server (page 234)
- Single-Hop Partitioned Region Data Access (page 236)
- Subscribing to Server Events (page 238)
- Controlling Thread and Connection Use (page 259)
- Performance Considerations (page 263)
- Miscellaneous Server Notes (page 264)
- Socket Configuration (page 265)

9.1 Client/Server Connections

Client/server communication is done in two distinct ways:

- The client sends individual operations to the server to update cached data, to satisfy a local cache miss, or to run an ad hoc query.
- The server streams cache update and continuous query events to the client based on client subscriptions.

Each kind of communication uses a different type of connection for maximum performance and availability, as described here.

Pool Connection

The pool connection is used by the client to send individual data requests and data updates to servers. Each pool connection goes to a host/port location where a server is listening. The server responds to the request on the same connection. Generally, client threads use a pool connection for an individual operation and then return the connection to the pool for reuse. For details on the kinds of messages sent through pool connections, see *Client-Initiated Data Flow* on page 217. This figure shows pool connections for one client and one server. At any time, a pool may have from zero to many pool connections to any one server.

Figure 9.1 Client Pool Connections for Client-Initiated Messaging

Subscription Connection

This type of connection is used to stream cache and continuous query events from the server to the client. To set this up, configure your client pool with `subscription-enabled` (page 212) set to `true`. The server establishes a queue to send subscription events asynchronously to the pool and the pool establishes a subscription connection to handle the incoming messages. The events sent through the subscription connection are described under *Server-Initiated Data Flow* on page 219.

Figure 9.2 Subscription Connections for Server-to-Client Event Streaming

Creating and Maintaining Connections

At startup, the pool establishes the minimum number of pool and subscription connections specified by your configuration properties. During runtime, the pool works to keep the minimum connection count satisfied and also to open additional connections as needed, up to the maximum connection count. All client and server connections are established by the client pool, which allows the client to connect to the server over a firewall.

Choosing a Server To Connect To

The pool gets server connection information from the server locators or, alternately, from the static server list. The pool's locator choice is random and the pool sticks with a locator until the connection fails.

Server Locator

Server locators maintain information about which servers are available and which has the least load. New connections are sent to the least loaded servers. The pool requests server information from the locator when it needs a new connection.

Static Server List

If you use a static server list, the pool shuffles it once at startup, to provide randomness between clients with the same list configuration, and then runs through the list round robin connecting as needed to the next server in the list. There is no load balancing or dynamic server discovery with the static server list.

Connecting

When a pool needs a new connection, it goes through these steps until either it successfully establishes a connection, it has exhausted all available servers, or the `free-connection-timeout` (page 210) is reached.

1. Requests server connection information from the locator or retrieves the next server from the static server list.
2. Sends a connection request to the server.

If the pool fails to connect while creating a subscription connection or provisioning the pool to satisfy the `min-connections` setting, it logs a fine level message. It then retries after the time indicated by `ping-interval` (page 211). If an application thread calls an operation that needs a connection and the pool can't create it, the operation returns a `NoAvailableServersException`.

How The Pool Manages Pool Connections

There is one connection pool for each of the `Pool` instances in your client. The pool works to keep the minimum number of pool connections open at all times and to respond as efficiently as possible to connection loss and requests for new connections. When you use a pool with the server locator, your pool can quickly respond to changes in server availability, adding new servers and disconnecting from unhealthy or dead servers with little or no impact on your client threads. Static server lists require more close attention as the client pool is only able to connect to servers at the locations specified in the list.

In addition to the information provided here on opening and closing connections, see the information on dynamic conditioning of server connections in *Pool Connection Conditioning* on page 232.

Opening Pool Connections

The `Pool` adds a new pool connection whenever one of these happens:

- The number of open connections is less than the `Pool`'s `min-connections` (page 210) setting.
- A thread needs a connection, all open connections are in use, and adding another connection would not take the open connection count over the pool's `max-connections` (page 210) setting. If the `max-connections` setting has been reached, the thread blocks until a connection becomes available.

Closing Pool Connections

The `Pool` closes a pool connection when one of the following occurs:

- The client receives a connectivity exception from the server.
- The server doesn't respond to a direct request or ping within the timeout period `read-timeout` (page 211). In this case, the `Pool` removes all connections to that server.
- The number of pool connections exceeds the `Pool`'s `min-connections` (page 210) setting and the client doesn't send any requests over the connection for the timeout period `idle-timeout` (page 210).

When it closes a connection that a thread is using, the pool switches the thread to another server connection, opening a new one if needed.

How the Pool Manages Subscription Connections

The pool's subscription connection is established in the same way as the pool connections, by requesting server information from the locator and then sending a request to the server, or, if you are using a static server list, by connecting to the next server in the list. Subscription connections remain open for as long as needed and are not subject to the timeouts that apply to pool connections.

9.2 Balancing Server Use

When the client pool requests connection information from the server locator, the locator returns the least-loaded server for the connection type. This “best server” response is used by the pool both to open new connections and also to condition existing pool connections.

Calculating Server Load

The locator tracks server availability and load for pool and subscription connections using current and estimated load information provided by the servers. The server periodically probes its load metrics and, whenever a change is detected, sends new information to the locator. This information consists of current load levels and estimates of how much load would be added for each additional connection. The locator compares the load information from its servers to determine which servers can best handle more connections.

You can configure how often the servers check their load with the cache server’s `load-poll-interval` (page 208). If you find your server loads fluctuating too much during normal operation, you might want to set this interval lower. The lower you set it, however, the more overhead your load balancing will use.

Between updates from the servers, the locators estimate which server is the least loaded using the estimates it has received from the servers for the cost of additional connections. For example, if the current pool connection load for a server’s connections is 0.4 and the each additional connection would add 0.1 to its load, the locator can estimate that adding two new pool connections will take the server’s pool connection load to 0.6. Locators do not share connection information among themselves, so these estimates just provide rough guidance to the individual locators for the periods between updates from the servers.

The Default Server Load Probe

GemFire provides a default utility that probes the server and its resource usage to give load information to the locators. The default probe returns the following load metrics:

- The pool connection load is the number of connections to the server divided by the server’s `max-connections` (page 208) setting. This means that servers with a lower `max-connections` setting receives fewer connections than servers with a higher setting. The load is a number between 0 and 1, where 0 means there are no connections, and 1 means the server is already at `max-connections`. The load estimate for each additional pool connection is 1/`max-connections`.
- The subscription connection load is the number of subscription queues hosted by this server. The load estimate for each additional subscription connection is 1.

Implementing a Custom Load Probe

You can override the default probe with your own load probe plug-in using the `ServerLoadProbe` or `ServerLoadProbeAdapter` and related interfaces and classes in `com.gemstone.gemfire.cache.server`. For information on how to do this, see the online Java documentation for this package.

Pool Connection Conditioning

If your server pool connections are out of balance for any reason, the `Pool` can silently replace the connections to highly loaded servers with connections to less loaded servers. This automatic conditioning allows very efficient upscaling of your server pool. It is also useful following planned and unplanned server outages, during which time the entire client load will have been placed on a subset of the normal set of servers.

The conditioning work uses server locators and the `Pool`'s `load-conditioning-interval` (page 210) configuration setting. The connection lifetime says how long a connection can exist before being eligible for conditioning. It is usually set in minutes. Once a connection's lifetime is reached, the `Pool` checks with the locator to see if the connection is already using the least-loaded server available. If so, the connection's lifetime counter is reset. If not, the `Pool` establishes a new connection to the least loaded server, silently puts it in place of the old connection, and closes the old connection. This happens behind the scenes and does not affect your application's connection use.

Grouping Your Servers

Using a server group narrows your pool's view to a subset of the running servers, giving you a degree of control over the servers used by any one client or client region. You may want to divide your servers into groups for things like hand partitioning your data or to connect specific regions only to servers with direct connections to a backend database.

To group servers, give them one or more `group` (page 208) designation in the `CacheServer` configuration. Then specify a `server-group` (page 211) for each pool that you want to have connected to a specific subset of servers. Pools without a server-group specification use all available servers.

At runtime, the servers communicate their various group designations to the locators and the pools send the locators their group requirements in their server requests. The locators provide server connection information to the pool using just the subset of servers that belong to the indicated group.

9.3 Client Operations to Server

When a client modifies a region with a pool defined, the operation is performed on the client's local cache and possibly forwarded to a server. Client data requests and updates are forwarded through the connection pool. This section discusses how your clients use the pool connections and how the pool maintains them.

Operations That Go to the Server

Client updates to the local cache are always forwarded to the server. These are:

- Region destruction and clear: `Region.destroyRegion`, `Region.clear`
- Entry destruction: `Region.destroy`, `Region.remove`

  The entry destruction operation is always forwarded. This allows your clients to destroy an entry in the system without having to have the entry in its own cache.

- Entry creation and update.

Client requests are forwarded if the local cache doesn't have the requested data:

- Entry `get` that results in a local cache miss.

When you perform one of these operations in the client application, the operation returns after it has been applied to the client cache and forwarded as needed to the server side.

How Client Threads use Pool Connections

By default, a client thread retrieves a connection from the open connection pool for each forwarded operation and returns the connection to the pool as soon as the request is complete. If your client thread runs a `put` on a client region, for example, that operation grabs a server connection, sends the `put` to the server, and then returns the connection to the pool. This keeps the connections available to the most threads possible.

Threads use dedicated connections when `thread-local-connections` (page 212) is set to `true`. In this case the thread holds its connection either until it explicitly releases it or it is expired based on `idle-timeout` (page 210) or `load-conditioning-interval` (page 210).

If you are using thread-local connections, you should release the connection as soon as your thread finishes its server activities. Do this by calling `releaseThreadLocalConnection` on the `Pool` instance you are using for the region. With the region `myRegion` in hand, this call closes your thread's connection:

```
PoolManager.find(myRegion).releaseThreadLocalConnection();
```

When Operations Don't Return Quickly

Your client operations can be slow to return for a couple of reasons. If you experience a lot of slowing in your client operations, read this section for ways to ameliorate the problem.

No Connection Available

When your client thread makes a call that is forwarded to a server, it must have a server connection to finish the operation. Generally, a thread retrieves a connection from the pool, runs the operation, and returns the connection. If there is no free connection, and the pool already has `max-connections` (page 210) open, the thread blocks until a connection becomes available. You can increase the `max-connections` setting so more connections are generally available generally. Alternately, you can

set `thread-local-connections` to `true` so your threads do not automatically release connections to the pool when an operation finishes. Note, however, that having any of your threads hold connections between operations will leave even fewer connections available for general use.

Retrying Requests

Your call may also return slowly if there is trouble with your servers or server connections. When a thread sends an operation to an unresponsive server, it waits for the length of the pool's `read-timeout` (page 211) setting before trying a different server. If other servers are unresponsive, the thread tries the number of servers indicated in `retry-attempts` (page 211) before giving up and returning an exception to the calling operation. So when your servers are not responsive, even a successful operation could take close to the product of `read-timeout` times `retry-attempts` to before returning.

9.4 Single-Hop Partitioned Region Data Access

With single-hop data access, the client pool maintains information on where a partitioned region's data is hosted in the servers. To access a single entry, the client directly contacts the server that hosts the key—in a single hop.

> *Single hop is only used for these operations that are run on single data points:* `put`, `get`, `destroy`, `putAll`.

To enable single hop:

- Make sure the pool attribute, `pr-single-hop-enabled` (page 211) is not set or is set to `true`. It is `true` by default.
- Include the server's implementation of `PartitionResolver` in the client's `CLASSPATH`. See *Adding custom partitioning to colocate like data within a region* on page 184. Do this for all partitioned regions where you have custom partitioning on the server and want to use single hop access. The server passes the name of the resolver for each custom partitioned region, so the client will use the proper one. If the server does not use a `PartitionResolver`, the default partitioning between server and client will match, so single hop will still work.

Disable single hop behavior for a region by setting its pool's attribute, `pr-single-hop-enabled` (page 211) to `false`.

With single hop, the client connects to every server, so more connections are generally used. This works fine for smaller installations, but is a barrier to scaling. If you have a large installation with many clients, you may want to disable single hop by setting the pool attribute, `pr-single-hop-enabled` (page 211) to false in your pool declarations.

Without single hop, the client uses whatever server connection is available, the same as with all other operations. The server that receives the request determines the data location and contacts the host, which might be a different server. With single hop disabled, more multiple-hop requests are made to the server system.

To perform single-hop data access, clients automatically get metadata from the servers about where the entry buckets are hosted. The metadata is maintained lazily. It is only updated after a single-hop operation ends up needing multiple hops, an indicator of stale metadata in the client. So you may see some multiple hop behavior for single-key operations even when single hop is configured.

Getting the Best Performance with PR Single Hop

1. Do not use single-hop and the pool's `max-connections` (page 210) together if you can avoid it.
   - Setting max connections with single hop can cause connection thrashing, throughput loss, and server log bloat. If you need to limit the pool's connections, either disable single hop or keep a close watch on your system for these negative effects.
   - Setting no limit on connections, however, can result in too many connections to your servers, possibly causing you to run up against your system's file handle limits. Review your anticipated connection use and make sure your servers are able to accommodate it.
2. If possible, partition your region data to accommodate your data use patterns. Single-hop gives the biggest benefits when data access is well balanced across your servers.

   See *Colocating Data in Partitioned Regions* on page 182.

In particular, the loads for client/server connections can get out of balance if you have these in combination:

- servers that are empty data accessors or that do not host the data the clients access through single-key operations
- many single-key operations from the clients

If data access is greatly out of balance, you can end up with clients thrashing trying to get to the data servers, when it might be faster to go through servers that do not host the data.

3. Add single-hop considerations to your overall server load balancing plan. See *Balancing Server Use* on page 232.

   Single-hop uses data location rather than least loaded server to pick the servers for single-key operations. Poorly balanced single-hop data access can affect overall client/server load balancing.

   Some counterbalancing is done automatically. The servers that receive more single-key operations become more loaded and are less likely to be picked for other operations.

9.5 Subscribing to Server Events

You can request automatic updates from the server by setting `subscription-enabled` (page 212) to `true` on the pool. When you do this, the server creates a queue for the pool so it can send the messages asynchronously and the pool establishes a subscription connection to receive the messages. All queueing for a single pool is done by one server.

Subscription maintenance is performed automatically by clients and servers, but may require some tuning for optimal performance. To help with this, GemFire Enterprise provides statistics that track subscription messages queued on the server. The statistics are documented in *System Statistics* on page 235 of the *GemFire Enterprise System Administrator's Guide*.

You can configure your clients with redundant subscriptions, hosted by different servers, for high availability. You can also configure durable subscriptions, so events are not lost if clients disconnect for a period of time.

This section contains the following:

- Events the Client Can Subscribe To (page 238)
- Discarding Duplicate Events (page 239)
- Highly Available Subscription Delivery (page 241)
- Durable Subscription Queues (page 246)
- Failover with Subscription Queues (page 254)
- Limiting Subscription Queue Memory Use (page 255)

Events the Client Can Subscribe To

Server cache updates are sent to the client according to server configuration and client subscriptions. The subscriptions are:

- Region and entry changes. The region events sent are destruction and clear. If the server is configured for subscription, the entry changes are only for entries the client has registered interest in. If the server is not configured for subscription, these are all changes to all entries the pool is assigned to, but for performance reasons, entry values are not sent until explicitly requested by client get operations. For more information on this, see *Server-Initiated Data Flow* on page 219.
- Result updates for continuous queries. These are queries the client registers with the server that run continuously against server cache updates. When a change occurs that changes the query results, the query results change event is sent to the client. For more information on this, see Chapter 14, *Continuous Querying*, on page 397.

  If you have dependencies between CQs and interest subscriptions, so that you want the two types of subscription events to arrive as closely together on the client, use a single `Pool` *for everything. Using different pools can lead to time differences in the delivery of events because the* `Pool`*s might use different servers to process and deliver the event messages.*

When the server cache is updated, the updating thread checks whether the event should be forwarded to any of its clients and, if so, adds the event to the appropriate subscription queues. The server uses one queue for each client pool that requests it, sending the messages asynchronously. A pool may be assigned to multiple regions, so the queue may also be used for one or many regions. Each queue has its own dedicated dispatcher thread in the server.

Discarding Duplicate Events

A single client thread receives and processes messages from the server, tracking received messages to make sure it does not process duplicate sends. It does this using the process IDs from originating threads. Every event in the queue originates in an operation performed by a thread in a client, a server, or an application in the server's or some other distributed system. The event message has a unique identifier composed of the originating thread's ID and its member's distributed system member ID, and the sequential ID of the operation. So the event messages originating in any single thread can be grouped and ordered by time from lowest sequence ID to highest. The servers and clients use this to avoid processing duplicate sends.

This figure shows the stages of queue messaging for one client.

Figure 9.3 Subscription Queue Messaging

The client uses its message tracking list to discard any duplicates it might receive from the server side.

The client's message tracking list holds the highest sequence ID of any message received for each originating thread. The list can become quite large in systems where there are many different threads coming and going and doing work on the cache. After a thread dies, its tracking entry is not needed. To avoid maintaining tracking information for threads that have died, the list has a limit on the amount of time it allows an entry to remain in the list with no activity. You can change this limit with the client's `subscription-message-tracking-timeout` (page 212) property. Entries that have not been modified after this amount of time are expired.

This XML sets the timeout period to `70000` milliseconds (70 seconds):

Example 9.1 Setting the Client Message Tracking Timeout Through cache.xml

```
<pool name="client" subscription-enabled="true"
      subscription-message-tracking-timeout="70000">
   <locator host="localhost" port="33333"/>
</pool>
```

The `subscription-message-tracking-timeout` is especially important in systems where it is vital to avoid or greatly minimize duplicate events. If you detect that duplicate messages are being processed by your clients, increasing this timeout may help.

> *Setting* `subscription-message-tracking-timeout` *may not completely eliminate duplicate entries, but careful configuration can help minimize occurrences.*

You may not be able to completely eliminate duplicate events received by a client. Duplicates are monitored by keeping track of message sequence IDs from the source thread where the operation originated. For a long-running system, you would not want to track this information for very long periods or the information may be kept long enough for a thread ID to be recycled. If this happens, messages from a new thread may be discarded mistakenly as duplicates of messages from an old thread with the same ID. In addition, maintaining this tracking information for old threads uses memory that might be freed up for other things.

To minimize duplicates and reduce the size of the message tracking list, set your client `subscription-message-tracking-timeout` (page 212) value higher than double these combined times.

- The longest time your (originating) threads might wait between operations
- If you are using redundant servers (see *Highly Available Subscription Delivery* on page 241) add:
  - The server's `message-sync-interval` (page 207)
  - The time required for failover (usually 7-10 seconds, including the time to detect failure)

You risk losing live thread tracking records if you set the value lower than this,. This could result in your client processing duplicate event messages into its cache for the associated threads.

It is worth working to set the `subscription-message-tracking-timeout` as low as you reasonably can. The lower it is, the less memory the tracking list requires in your client and the lower the possibility of overlapping management of messaging from an old dead thread and a new live thread with the same thread ID. These situations can occur when inactive entries in the list aren't removed after their originating threads die.

Highly Available Subscription Delivery

You can make your subscription delivery highly available by configuring redundant copies of your subscription queues in secondary servers. When redundancy is enabled, in addition to the primary server pushing events to the client, secondary servers maintain queue backups. If the primary server fails, one of the secondaries steps in as primary to provide uninterrupted event messaging to the client.

With redundancy, each pool has a primary server and some number of secondaries. The primaries and secondaries are assigned on a per-pool basis and are generally spread out for load balancing, so a single client with multiple pools may have primary queues in more than one server.

To use high availability, set the client pool's `subscription-redundancy` (page 212) to the number of secondary servers you want it to use. The client pool automatically connects to the primary and secondary servers.

This table lists the settings for `subscription-redundancy`:

Table 9.1 Server Redundancy Settings

subscription-redundancy	Description
`0`	No secondary servers are configured, so high availability is disabled.
`>0`	This sets the precise number of secondary servers to use for backup to the primary.
`-1`	All available non-primary servers are to be used as secondary servers. For the N servers running at any time, one is the primary and N-1 are secondary.

So, for example, if there are six servers and `subscription-redundancy` is set to two, one server is the primary, two servers are secondary, and the remaining three do not actively participate in HA for the client.

If the primary server fails, the system assigns one of the secondaries as the new primary and attempts to add another server to the secondary pool to retain the initial redundancy level. If no new secondary server is found, then the redundancy level is not satisfied but the failover procedure completes successfully. As soon as another secondary is available, it is added.

This examples creates a pool of servers where one is used as primary and one as secondary. Any remaining servers are not used.

Example 9.2 Using One Redundant Server

```
<pool name="red1" subscription-enabled="true" subscription-redundancy="1">
   <locator host="nick" port="41111"/>
   <locator host="nora" port="41111"/>
</pool>
```

This examples creates a pool of servers where all available secondaries are used.

Example 9.3 Using All Available Servers for Redundancy

```
<pool name="redX" subscription-enabled="true" subscription-redundancy="-1">
   <locator host="nick" port="41111"/>
   <locator host="nora" port="41111"/>
</pool>
```

Keeping Redundant Servers Synchronized

With high availability, the primary server and secondary servers work together to send all server-side events at least once to the client.

This is how event processing works when high availability is enabled:

- The primary server sends event messages to the clients.
- Periodically, the clients send received messages to the server and the server removes the sent messages from its queues.
- In the case of primary server failure, one of the secondaries becomes the primary and begins sending event messages from its queues to the clients. Just after failover, the new primary usually resends some messages that were already sent by the old primary. The client recognizes these as duplicates and discards them.

This figure provides a simple example of high availability messaging and queue maintenance. In stage one, the primary sends an event message to the client and a synchronization message to its secondary. By stage two, the secondary and client have updated their queue and message tracking information accordingly.

If the primary failed at stage two, the secondary would start sending event messages from its queue beginning with message A10. The client would discard the resend of message A10 and then process subsequent messages as usual.

Figure 9.4 High Availability Messaging: Server to Client and Primary Server to Secondary Server

The primary server regularly synchronizes with the secondaries, notifying them of messages that can be removed from their queues. Since there is a lag in notification, the secondary servers remain only roughly

synchronized with the primary. Secondary queues contain all event messages that are contained in the primary queue plus they might contain a few messages that have already been sent to clients.

By default, the primary server sends queue synchronization messages to the secondaries every second. You can change this interval with the configuration setting `message-sync-interval` (page 207).

The ideal setting for this interval depends in large part on your application behavior. These are the benefits of shorter and longer interval settings:

- A shorter interval requires less memory in the secondary servers because it reduces queue buildup between synchronizations. In addition, fewer old messages in the secondary queues means reduced message resends after a failover. These considerations are most important for systems with high update rates in the regions.
- A longer interval requires fewer distribution messages between the primary and secondary, which benefits overall system performance.

The message sync interval affects the performance of the client's `subscription-message-tracking-timeout` (page 212). For information on this, see *Discarding Duplicate Events* on page 239.

The `cache.xml` snippet in the following example sets `message-sync-interval` for the server's cache to `2` seconds.

Example 9.4 Setting the Server Message Sync Interval Through cache.xml

```
<cache lock-lease="120" lock-timeout="60"
   message-sync-interval="2"/>
```

The code in this API example sets the sync interval to 2 seconds.

Example 9.5 Setting the Server Message Sync Interval Through the API

```
cache = CacheFactory.create(system);
cache.setMessageSyncInterval(2);
```

Removing Orphaned Events from Secondary Queues

Usually, all event messages are removed from secondary queues based on synchronization messages sent by the primary (see Figure 9.4 on page 243). Occasionally, however, problems in synchronization can leave some messages orphaned in the secondary queues. To make sure these messages are eventually removed, the secondaries expire all messages that have been enqueued longer than the time specified in the `message-time-to-live` (page 209).

One situation that can leave orphans in secondary subscriptions queues is when a primary fails in the middle of sending a synchronization message to its secondaries. In this case, some secondaries might receive the message and some might not. If the failover goes to a secondary that did receive the message, the result is a system where there are secondary queues holding messages that are no longer in the primary queue. The new primary will never synchronize on these messages. Automatic expiration of old messages allows the secondary queues to clean up after the fact.

This `cache.xml` snippet sets the expiration time, in seconds, to `200`:

Example 9.6 Setting the Server Message Time to Live Through cache.xml

```
<cache-server port="41414"
   message-time-to-live="200"/>
```

The code in this API example creates a server with the same settings as in Example 9.6.

Example 9.7 Setting the Server Message Time to Live Through the API

```
Cache cache = ...;
CacheServer cacheServer = cache.addCacheServer();
cacheServer.setPort(41414);
cacheServer.setMessageTimeToLive(200);
cacheServer.start();
```

Durable Subscription Queues

The client can register any of its interest lists and continuous queries as durable. Durable interest remains even if the client disconnects for a period of time. During the client's down time, the server maintains its durable subscriptions and then, when the client reconnects, plays them back to the client.

For durable messaging, you need to do the following:

- Identify your client as durable.
- Specify which interest lists are durable.
- Specify which continuous queries (CQs) are durable.
- Program your client for durable event storage and replay. The main pieces of this are:
  - Disconnect keepalive request—When a durable client disconnects normally, it should tell the server whether to maintain its subscription queue while it is gone. If the client disconnects abnormally, the server automatically maintains the events.
  - Cache ready message—When it connects and is ready to receive stored events, a durable client must call `ClientCache.readyForEvents` so the server knows it can replay stored events.
  - Cache listener—The durable client's cache listener must be able to handle having events played after the fact. Generally listeners receive events more-or-less as they happen, but the durable client may receive events that occurred minutes before and are not relevant to current cache state.
  - Optional cache listener `afterRegionLive` callback—If you use cache listeners, you can implement the `afterRegionLive` callback method. This is invoked after the client connects to its servers, when it has received all of its stored messages and replayed the events.

This section describes the options for configuring and implementing durable messaging.

Client-Side Configuration

All the configuration specific to durable messaging is on the client side.

Configuring a Durable Client

The durable client can be configured in the `gemfire.properties` file, through the API, or in the `DistributedSystem` connection.

- **Durable client ID**—You indicate that the client is durable by giving it a `durable-client-id`. The servers use this ID to identify the client. For a non-durable client, the `durable-client-id` is an empty string. The ID can be any number that is unique among the clients attached to servers in the same distributed system.
- **Durable client timeout**—The `durable-client-timeout` specifies how long this client's servers should wait after it disconnects before terminating its message queue. During that time, the servers consider the client alive and continue to accumulate messages for it. The default is 300 seconds.

These lines in `gemstone.properties` make the client durable and set the timeout, in seconds, to `200`.

Example 9.8 Configuring a Durable Client Through gemstone.properties

```
durable-client-id=31
durable-client-timeout=200
```

This API example creates a durable client in the configuration of the `DistributedSystem` connection.

Example 9.9 Configuring a Durable Client Through the API

```
Properties props = new Properties();
props.setProperty("name", "DurableClient");
props.setProperty("cache-xml-file", "xml/DurableClient.xml");
props.setProperty("mcast-port", "0");
props.setProperty("durable-client-id", "DurableClientId");
props.setProperty("durable-client-timeout", "" + 300);
DistributedSystem ds = DistributedSystem.connect(props);
```

The `durable-client-timeout` is a tuning parameter. When setting the timeout, take into account the normal activity of your application, the average size of your messages, and the level of risk you can handle. Assuming that no messages are being removed from the queue, how long can the application run before the queue reaches the maximum message count? How long can it run before the queued messages consume all the memory on the client host? How serious is each of those failures to your operation?

To assist with tuning, GemFire Enterprise provides statistics that track message queues for durable clients through the disconnect and reconnect cycles. The statistics are documented in *Event Queue Statistics From Server-to-Client Communication* on page 281 of the *GemFire Enterprise System Administrator's Guide*.

Configuring Durable Interest in Keys

When a durable client disconnects, its servers keep queuing messages for selected keys. The client indicates which keys by registering *durable* interest. This fine-grained control handles the constraints of queue size and memory by saving only critical messages.

You still register interest for other keys, but not durable interest. When the client is connected to its servers, it receives messages for all keys. When the client is disconnected, its non-durable interest registrations are removed but all messages already in the queue remain there.

For durable clients, do all interest registration immediately after the regions are created. This is required whether interest registration is durable or not durable. An extra `registerInterest` parameter specified for durable clients indicates whether the registration is durable (`true`) or not (`false`).

This example registers non-durable interest in `key1` and `key2`, and durable interest in `key3` and `key4`. Here the `InterestResultPolicy` is set to `NONE` to prevent immediate bulk loading of keys or values, but any `InterestResultPolicy` works with durable clients.

Example 9.10 API Client Durable Interest List Registration

```
// Client registering interest with keys key1 & key2 as non-durable and
// key3 & key4 as durable
exampleRegion.registerInterest("key1", InterestResultPolicy.NONE, false);
exampleRegion.registerInterest("key2", InterestResultPolicy.NONE, false);
exampleRegion.registerInterest("key3", InterestResultPolicy.NONE, true);
exampleRegion.registerInterest("key4", InterestResultPolicy.NONE, true);
```

You use the typical methods for client interest registration and configure notification by subscription on the server as usual. For details, see *Server-Initiated Data Flow* on page 219.

> *Changing the interest registration after the durable client connects the first time can cause data inconsistency and is not recommended.*

At restart, if the client doesn't register durable interest in exactly the same keys as before, the entries in the interest list are not copied from the server during the registration. Instead, the client cache starts out empty, and entries are added during updates. If no updates come in for an entry, it never shows up in the client cache.

Configuring Durable Continuous Queries

When a durable client disconnects, its servers keep queuing messages for durable continuous queries (CQs) that are executing at the time of disconnect. Durable CQs that are in a stopped state are not retained. The client indicates which queries to maintain by registering *durable* interest in them at creation. When the client is connected to its servers and the CQs are running, the client receives events for those CQs. When the client is disconnected, its non-durable CQs and CQs that are not currently running are deleted but messages that are already in the queue remain there. For more information, see the online Java documentation for the `QueryService` method `newCq`. For information on continuous querying, see Chapter 14, *Continuous Querying*, on page 397.

Durable Client Operations

Durable clients require additional programming to handle stored event delivery. This section covers the methods and parameters on the `ClientCache` interface that are specific to durable clients.

Sending the ClientCache Ready Message to the Server

After a durable client connects and initializes its cache, regions, any cache listeners, and creates and executes any durable continuous queries, it invokes `ClientCache.readyForEvents` to indicate to the servers that the client is ready to receive any messages accumulated for it. This example creates the client cache and calls the `readyForEvents` method. The region configuration is shown in the code in Example 9.9.

Example 9.11 Durable Client Cache Ready Notification

```
ClientCache clientCache = CacheFactory.create(ds);
// Create region here, if it's not defined in a cache.xml file.
// Create any multi-user RegionService instances here
// Register all durable CQs and subscriptions for clientCache and any
RegionService instances
clientCache.readyForEvents();
```

To make sure the client receives and processes all events, call this method *after* all the regions, listeners, and continuous queries are in place. For more information, see *Reconnection* on page 249.

Disconnecting From the Server

When a durable client closes its cache and disconnects, it tells the servers whether to maintain its queues. For this purpose, use the version of `Pool.close` or `ClientCache.close` with the boolean `keepalive` parameter, as shown in this example. If the setting is `true`, the servers keep the durable client's queues and durable subscriptions and CQs alive for the timeout period. Only the resources and data related to the session are removed, such as port numbers and non-durable subscriptions. If the setting is `false`, the servers do the same cleanup that they would do for a non-durable client.

Example 9.12 Durable Client Disconnect Notification

```
clientCache.close(true);
```

Life Cycle of a Durable Client

This section discusses the high-level operation of a durable client through initial startup, disconnection, and reconnection.

Initial Operation

The initial startup of a durable client is similar to the startup of any other client, except that it specifically calls the `ClientCache.readyForEvents` method when all regions, listeners, and CQs on the client are ready to process messages from the server. See *Sending the ClientCache Ready Message to the Server* on page 248.

Disconnection

While the client and servers are disconnected, their operation varies depending on the circumstances.

- **Normal disconnect—**When a client closes its connection, it can pass a flag stating whether to maintain the client's message queue and durable subscriptions and CQs. The servers stop sending messages to the client and release its connection. See *Disconnecting From the Server* on page 248 for more information.

  If the client requests it, the servers maintain the queues and durable interest list and CQ information until the client reconnects or times out. The non-durable interest list and CQs are discarded. The servers continue to queue up incoming messages for entries on the durable interest list and in the durable CQ list. All messages that were in the queue when the client disconnected remain in the queue, including messages for entries and CQs in the non-durable lists.

  If the client requests not to have its subscriptions maintained, or if there are no durable subscriptions or CQs, the servers unregister the client and do the same cleanup as for a non-durable client.

- **Abnormal disconnect**—If the client crashes or loses its connections to all servers, the servers automatically maintain its message queue and durable subscriptions and CQs until it reconnects or times out.

- **Client disconnected but operational**—If the client operates while it is disconnected, it gets what data it can from the local cache. Since updates are not allowed, the data can become stale. An `UnconnectedException` occurs if an update is attempted.

- **Client stays disconnected past timeout period**—The servers track how long to keep a durable subscription queue alive based on the `durable-client-timeout` setting. If the client remains disconnected longer than the timeout, the servers unregister the client and do the same cleanup that is performed for a non-durable client. The servers also log an alert. When a timed-out client reconnects, the servers treat it as a new client making its initial connection.

Reconnection

During initialization, operations on the client cache can come from three sources:

- Cache operations by the application
- Results returned by the server in response to the client's interest registrations and CQ executions
- Callbacks triggered by replaying old events from the queue

These procedures can act on the cache concurrently, and the client cache is never blocked from doing operations.

GemFire Enterprise handles the conflicts between the application and interest registrations, and the CQ executions do not create conflicts, but you need to prevent the callback problem. Writing callback methods that do cache operations is never recommended, but it is a particularly bad idea for durable clients, as explained in *Implementing Cache Listeners for Durable Clients* on page 253.

When the durable client reconnects, it must go through these steps:

1. Create the cache and regions and create and execute all CQs. This ensures that all listeners are ready.

   At this point, the client can begin cache operations.

2. Call `ClientCache.readyForEvents`, which tells the server that all regions and listeners on the client are ready to process events.

   The cache ready message triggers the queued event replay process on the primary server.

3. Issue its register interest requests. This allows the client cache to be populated with the initial interest registration results.

   Registration must happen immediately after the cache ready message.

   The primary server responds with current state of entries that are present in the server's cache.

For an example using `ClientCache.readyForEvents`, see *Sending the ClientCache Ready Message to the Server* on page 248.

The following figure shows the three concurrent procedures during the initialization process. The application begins operations immediately on the client (step 1), while the client's cache ready message (also step 1) triggers a series of queue operations on the servers (starting with step 2 on the primary server). At the same time, the client registers interest (step 2 on the client) and receives a response from the server.

Message B2 applies to an entry in Region A, so the cache listener handles B2's event. Because B2 comes before the marker, the client does not apply the update to the cache.

Figure 9.5 Initialization of a Reconnected Durable Client.

Only one region is shown for simplicity, but the messages in the queue could apply to multiple regions. Also, the figure omits the concurrent cache updates on the servers, which would normally be adding more messages to the client's message queue.

Durable Event Replay

This section covers event replay for a durable client that reconnects before the timeout period. When the primary server receives the cache ready message, the servers and client execute the following procedure to update the queue and replay the events from the stored messages. To avoid overwriting current entries with old data, the updates are not applied to the client cache. CQ events never modify the client cache, so that behavior remains unchanged for durable CQ message replay.

1. The server finds the subscription queue for this durable client ID and updates its client information, including socket and remote port information.
2. All the servers with a queue for this client place a marker in the queue.

   Messages in the queue before the marker are considered to have come while the client was disconnected. Messages after the marker are handled normally.
3. The server sends the queued messages to the client up to the marker.
4. The client receives the messages but does not apply the usual automatic updates to its cache. If cache listeners are installed, they handle the events. See *Implementing Cache Listeners for Durable Clients* on page 253.
5. The client receives the marker message indicating that all past events have been played back.
6. The server sends the current list of live regions.
7. For every `CacheListener` in each live region on the client, the marker event triggers the `afterRegionLive` callback.

After the callback, the client begins normal processing of events from the server and applies the updates to its cache.

Even when a new client starts up for the first time, the cache ready markers are inserted in the queues. If messages start coming into the new queues before the servers insert the marker, those messages are considered as having happened while the client was disconnected, and their events are replayed the same as in the reconnect case.

Application Operations During Interest Registration

As soon as the client creates its regions, the application hosting the client can start cache operations, even while the client is still receiving its interest registration responses. In that case, application operations take precedence over interest registration responses.

When adding register interest responses to the cache, the following rules are applied:

- If the entry already exists in the cache with a valid value, it is not updated.
- If the entry is invalid, and the register interest response is valid, the valid value is put into the cache.
- If an entry is marked destroyed, it is not updated. Destroyed entries are removed from the system after the register interest response is completed.

If the interest response does not contain any results, because all of those keys are absent from the server's cache, the client's cache can start out empty. If the queue contains old messages related to those keys, the events are still replayed in the client's cache.

Implementing Cache Listeners for Durable Clients

To create cache listeners for durable clients:

- Implement the callback methods to behave properly when stored events are replayed.
- Program the `CacheListener` callback method, `afterRegionLive`.

`CQListener`s also require you to program the callback methods to behave properly when stored events are replayed.

For general instructions on implementing a `CacheListener` or `CqListener` see *Implementing Cache Event Handlers* on page 300 and *Cache Event Listeners* on page 305.

Writing Callbacks for Use With Durable Messaging

Durable clients require special attention to cache callbacks. During the initialization window when a reconnecting client has a functioning cache but is still receiving the stored messages from the queue, the client can replay events that are long past. Cache events are not applied to the cache, but they are sent to the cache listener. CQ events are never applied to the cache, but are always sent to the `CqListener`. If the listener's callbacks invoked by these events make changes to the cache, that could conflict with current operations and create data inconsistencies. So you need to keep your callback implementations lightweight and not do anything in the cache that could produce incorrect results during this window. For details on implementing callbacks for GemFire event handlers, see *Writing Callback Methods* on page 304.

Implementing the afterRegionLive Method

If you are using `CacheListener`s, you can implement the `afterRegionLive` callback method provided for durable clients. This callback is invoked when the client has received all the old messages that were stored in its queue while it was disconnected. Implementing this method enables you to do application-specific operations when the client has replayed all of these old events.

Failover with Subscription Queues

When a server hosting a subscription queue fails, the queueing responsibilities pass to another server. How this happens depends on whether the new server is a secondary server. In any case, all failover activities are carried out automatically by the GemFire system.

Failing Over to a High Availability Secondary Server

If your client pool is configured with redundancy and a secondary server is available at the time the primary fails, the failover is invisible to the client. The secondary server resumes queueing activities as soon as the primary loss is detected. The secondary might resend a few events, which are discarded automatically by the client message tracking activities. See *Discarding Duplicate Events* on page 239.

Message Loss During HA Failover

There is a very small potential for message loss during HA server failover. The risk is not present for failover to secondaries that have fully initialized their subscription queue data. The risk is extremely low in healthy systems that use at least two secondary servers. The risk is higher in unstable systems where servers often fail and where secondaries do not have time to initialize their subscription queue data before becoming primaries. To minimize the risk, the failover logic chooses the longest-lived secondary as the new primary.

Failover for HA Durable Clients

Redundancy management is handled by the client, so when the durable client is disconnected from the server the redundancy of client events is not maintained. Even if the servers fail one at a time, so that running clients have time to fail over and pick new secondary servers, an offline durable client cannot fail over. As a result, the client loses its queued messages.

Failing Over Without High Availability

A client will fail over without high availability in either of these cases:

- If it is not configured for redundancy
- If all secondary servers also fail before new secondaries can be initialized

In either case, as soon as it can attach to a server, the client goes through an automatic reinitialization process. In this process, the failover code on the client side silently destroys all entries the client has registered interest in and refetches them from the new server, essentially reinitializing the client cache from the new server's cache. This does not affect entries in the client that are not in the client's register interest lists.

To reduce failover noise, the events caused by the local entry destruction and refetching are blocked by the failover code and do not reach the client cache listeners. Because of this, your clients could receive some out-of-sequence events during and after a server failover. For example, entries that exist on the failed server and not on its replacement are destroyed and never recreated during a failover. Because the destruction events are blocked, the client ends up with entries removed from its cache with no associated destroy events.

Limiting Subscription Queue Memory Use

This section lists options for limiting the amount of server memory your subscription queues require.

Conflating the Subscription Queue

Conflating the subscription queue can both improve performance and reduce the amount of memory required on the server for queueing. You configure the server regions to conflate queued messages so the client receives only the latest available update in the queue for a particular entry key. Conflation is enabled in the server region, but can be overridden on a per-client basis.

Conflation is particularly useful when a single entry is updated often and the intermediate updates don't require processing by the client. With conflation, if an entry is updated and there is already an update in the queue for its key, the existing update is removed and the new update is placed at the end of the queue. This conflation behavior is shown in the figure below for entry key A.

Conflation can only be done on messages that are not in the process of being sent to the client. For this example, assume none of these messages are being sent while the update for Key A is added.

Figure 9.6 Async Subscription Queue Conflation

This method of conflation is different from gateway-to-gateway conflation, described at Site-to-Site Queue Conflation on page 277*. It behaves the same as peer-to-peer conflation, described in Asynchronous Queueing For Slow Receivers* on page 473*.*

Choosing Whether to Use Conflation

For systems where the clients don't need to see every single message from the server, conflation is a good way to speed event delivery. However, conflation is not always appropriate. These are the main reasons you may not want to use it:

- With conflation, earlier events in the queue are removed in favor of updates sent later in the queue. This is problematic for applications that depend on a specific ordering of entry modifications. For

example, if any client has a `CacheListener` that needs to know about every state change, you should not use conflation.

- If your queue remains in use for a significant period and you have entries that are updated frequently, you could have a series of update message replacements resulting in a notable delay in the arrival of *any* update for some entries. Imagine that update 1, before it is sent, is removed in favor of a later update 2. Then, before update 2 can be sent, it is removed in favor of update 3, and so on. This could result in unacceptably stale data on the client.

Enabling Conflation

To enable conflation, set the server's region attribute `enable-subscription-conflation` (page 107) to `true`. It is false by default. Conflation is enabled in the server region, so by default, all clients interested in updates in a particular region either get the updates conflated or not.

This example snippet of a server `cache.xml` enables conflation for the `trades` region.

Example 9.13 Enabling Conflation Through the Server's cache.xml

```
<region name="trades">
   <region-attributes enable-subscription-conflation="true"/>
</region>
```

You can override the server setting on a per-client basis by setting the `conflate-events` property in the client's `gemfire.properties`. Valid settings are `server`, which uses the server settings, `true`, which conflates everything sent to this client, and `false`, which does not conflate anything sent to this client. For more information on this, see *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

Clearing the Queue With HA

With redundant queues, the primary server waits for client acknowledgement of receipt before notifying secondaries to delete events. The client does not acknowledge each individual message, but instead periodically sends a batch acknowledgement. This speeds message delivery from the server and generally reduces traffic between the server and client.

By default, the client sends an acknowledgement message every 500 milliseconds. You can change the interval of time between these messages by setting the `subscription-ack-interval` (page 212) in the client's `pool` configuration. You might want to lower the interval if you have a very busy system and want to reduce the space required in the servers for the subscription queues. More frequent acknowledgements means fewer events held in the server queues awaiting acknowledgement.

This `cache.xml` snippet sets the client acknowledgement interval to three seconds:

Example 9.14 Setting the Clients Queue Acknowledgement Interval

```
<cache>
  <pool name="client" subscription-enabled="true" subscription-ack-interval=
"3000">
    <server host="localhost" port="40404"/>
  </pool>
```

Limiting Subscription Queue Size

You can limit server queue size either by having the queue overflow to disk when it reaches a limit or by having it block until the queue drains enough to add more messages. You can use one or the other, but not both. These settings help avoid out of memory errors on the server in the case of slow clients. A slow client slows the rate that the server can send messages, causing messages to backup in the queue. Without overflow or blocking, the queue might grow too large and cause out of memory on the server.

Overflowing the Client Queue to Disk

With overflow, the most recently used (MRU) events are written out to disk, keeping the oldest events, the ones that are next in line to be sent to the client, available in memory.

You configure subscription queue overflow using the server's `client-subscription` (page 208) property. This is specified in three parts:

- `eviction-policy`—How the capacity is calculated. The options are `mem` for memory use , `entry` for message count, and `null` for no overflow. The default is `null`.
- `capacity`—Used if `eviction-policy` is not `null`. Specified in megabytes for `mem` and as a positive integer for `entry`. The default is 1.
- `disk-store-name`—Used if `eviction-policy` is not `null`. By default, this is not defined, which causes all disk activities to be done through the system's default disk store. If specified, this is the name of a store in the pool of disk stores serving the cache. See *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.

This XML shows a cache server with queue overflow configuration:

Example 9.15 Configuring Subscription Queue Overflow to Disk

```
<cache-server port="40404">
   <client-subscription eviction-policy="entry"
         capacity="10000" disk-store-name="serverOverflow"/>
</cache-server>
```

Blocking

The server's `maximum-message-count` (page 209) is the maximum number of event messages allowed in any single subscription queue before incoming messages are blocked. Note that you can only limit the message count, not the size allocated for messages.

> *With this setting, one slow client can slow the server and all of its other clients because this blocks the threads that write to the queues.*

All operations that add messages to the queue block until the queue size drops to an acceptable level. If the regions feeding these queues are partitioned or have `distributed-ack` or `global scope`, operations on them remain blocked until their event messages can be added to the queue. If you see stalling on your server region operations, your queue capacity might be too low for your application behavior.

This XML snippet sets the maximum message count to 50000 entries:

Example 9.16 Setting the Server Maximum Message Count Through the cache.xml

```
<cache-server port="41414" maximum-message-count="50000"/>
```

The code in this API example creates a server with the same settings.

Example 9.17 Setting the Server Maximum Message Count Through the API

```
Cache cache = ...;
CacheServer cacheServer = cache.addCacheServer();
cacheServer.setPort(41414);
cacheServer.setMaximumMessageCount(50000);
cacheServer.start();
```

9.6 Controlling Thread and Connection Use

This section discusses things you can do to avoid bottlenecks and avoid resource drains in your client/server installations. It discusses client connection requirements, server handling of client requests, and the server's client and peer-to-peer connection management.

This figure shows the sockets and threads involved in a default client/server architecture and highlights the configuration points for reducing bottlenecks. By default:

- The server dedicates one thread to each client connection
- Each server thread servicing client requests also uses two connections to *each* of its peers to forward data requests and distribute events

Figure 9.7 Configuration Options for Minimizing Connections in a Client/Server Installation

Server Side Options

`conserve-sockets=true`—Lowers the number of connections used for peer communication by having threads share sockets. See *Controlling the Number of Server Connections to Peers* on page 260.

`max-threads`—Limits the number of threads servicing client requests. See *Controlling the Number of Server Threads Used for Clients* on page 260.

`max-connections`—Limits the number of simultaneous client connections allowed by the server. See *Setting the Maximum Client Connections Accepted by a Server* on page 261.

Client Side Options

Client pool configuration—Impacts the number of connections required by the client. See *Adjusting the Limits on the Client Connection Pool* on page 261.

How you architect and program your clients can help minimize the number of connections they require. See *Optimizing the Client's Connection Requirements* on page 261.

For clients, the goal is to keep connections to and from the servers to a minimum without compromising performance. For servers, you have to manage both client connections and peer connections.

There are a number of things that affect resource use. All involve some combination of limiting threads that communicate with other members and limiting the number of sockets through which those threads communicate. To find the best solution for your system, you need to combine a study of your system requirements with careful testing to see which options work best. Since things can change over time, you should monitor your system and adjust settings as needed. For information on system monitoring, see Chapter 18, *Monitoring and Tuning Your Applications*, on page 463.

Controlling the Number of Server Connections to Peers

Servers always maintain two outgoing connections to each of their peers. So for each peer a server has, there are four total connections: two going out to the peer and two coming in from the peer.

The server threads that service client requests also communicate with peers to distribute events and forward client requests. If the server's GemFire connection property `conserve-sockets` is set to `true` (the default), these threads use the already-established peer connections for this communication. If `conserve-sockets` is `false`, each thread that services clients establishes two of its own individual connections to its server peers—one to send and one to receive.

Each socket uses a file descriptor, so the number of available sockets is governed by two operating system settings, the maximum open files allowed on the system as a whole and the maximum allowed for each session. In servers with many threads servicing clients, if `conserve-sockets` is set to `false`, the demand for connections can easily overrun the number of available sockets.

Even with `conserve-sockets` set to `false`, you can cap the number of these connections by setting the server's `max-threads` parameter. See *Controlling the Number of Server Threads Used for Clients* on page 260.

For details on managing the conserve sockets settings, see *Controlling Socket Use* on page 471.

Controlling the Number of Server Threads Used for Clients

Normally, the server dedicates one thread to each client connection. You can limit the total number of server threads used for client connections by setting the `CacheServer` property `max-threads` (page 208). When the maximum is reached, the server threads begin servicing multiple client connections. This scalable solution enables servers to handle many more clients. It uses a server thread pool selector to detect and dispatch new client requests. This option increases performance in systems with many clients by reducing contention for server threads. In systems with few clients, this option decreases performance because of the overhead of using the selector. The default setting of 0 dedicates one thread to each client connection and does not use a thread pool selector.

The threads used for pushing events into the subscription queues are the same threads that update the server cache. The `cache.xml` snippet in this example creates a server with a maximum of 16 threads for servicing client connections.

Example 9.18 Setting the Server's Maximum Client Connections Through cache.xml

```
<cache-server port="41414" max-threads="16"/>
```

The code in this example accomplishes the same thing.

Example 9.19 Setting the Server's Maximum Client Connections Through the API

```
Cache cache = ...;
CacheServer cacheServer = cache.addCacheServer();
cacheServer.setPort(41414);
cacheServer.setMaxThreads(500);
cacheServer.start();
```

Setting the Maximum Client Connections Accepted by a Server

You can put a limit on the total number of client connections for any server using the `CacheServer` property `max-connections` (page 208). This parameter is used to control server thread count when `max-threads` (page 208) is set to 0. When the limit is reached further client connections are refused, so if you regularly go over the limit your clients may suffer noticeably from refused server connections. The default value is 800, which allows 800 *connections*, not 800 clients. A multithreaded client typically makes multiple connections to the server.

The `max-connections` *property should be set at least as high as* `max-threads`.

The `cache.xml` snippet in this example creates a server with a maximum of 1000 client connections.

Example 9.20 Setting the Server's Maximum Client Connections Through cache.xml

```
<cache-server port="41414" max-connections="1000"/>
```

The code in this example accomplishes the same thing.

Example 9.21 Setting the Server's Maximum Client Connections Through the API

```
Cache cache = ...;
CacheServer cacheServer = cache.addcacheServer();
cacheServer.setPort(41414);
cacheServer.setMaxConnections(1000);
cacheServer.start();
```

Adjusting the Limits on the Client Connection Pool

When a client `Pool` instantiated, it establishes the minimum specified number of connections to the available servers. If `min-connections` (page 210) is set to three, the client establishes a total initial pool of three connections, available to requesting threads.

The connections established based on this property are just the starting point. If the threads need more connections than are available, the pool establishes up to the `max-connections` (page 208) limit. If a thread needs a connection and all possible connections are in use, the thread blocks waiting for one to become available.

When you set these limits, make sure to take into account how your threads use connections, including whether you have multiple threads making simultaneous server requests and whether your threads hold connections between requests or return them to the pool, which is determined by the `thread-local-connections` (page 212) setting.

Optimizing the Client's Connection Requirements

Review your code to see if there are any optimizations you can make in how your clients manage and use their connection pool. These are the main areas to consider:

- By default, client connections to servers are returned to the pool after each operation. If you set the pool's `thread-local-connections` (page 212) to `true`, make sure your threads explicitly release connections when they are done with them, instead of letting them time out.
- If you have multiple threads connecting to a server to perform a single job you might be able to centralize server communication and eliminate some connections requirements. In systems with

thousands of clients, each connecting thread that you eliminate from the individual client application can result in thousands fewer server connections overall.

- If possible, share your pools across many regions.
- Set a low `idle-timeout` (page 210), so the number of unused open connections is kept as close as possible to the lower limit of `min-connections` (page 210).
- Do what you can to reduce the server queueing required by your client. Every pool where the client has `subscription-enabled` (page 212) set to true uses a separate thread and connection that is not subject to the client-to-server connection-controlling options like `max-threads` or `max-connections`.
- If you have highly multi-threaded clients, figure out the maximum number of threads you might have running at the same time and set the client's `max-connections` (page 210) to that. This enforces a limit on the number of connections per client. Before you set `max-connections`, be sure to read the section on accessing partitioned region data.

9.7 Performance Considerations

There are a number of factors to weigh when configuring and programming your client/server architecture. This section lists the most significant.

- Number of clients
- Amount of data on the servers
- Number of continuous queries (CQs) and register interest calls registered on the servers by clients
- Number of CQs and register interest calls affected by updates
- Amount of CPU consumed by the servers in processing updates
- Nature of traffic (is the update traffic steady or bursty)
- Failover capacity of the system and redundancy level set by clients
- Size of update objects sent to the clients
- Expected throughput at the server

These factors are very specific to the application under consideration, but in general, if you have a large number of updates going on in the system, each client requires more CPU for overall processing, so you should reduce the number of clients each server is required to handle.

If the client configurations are such that a number of updates affect a large percentage of clients, these updates will generate a lot of messaging traffic. In this case, you may need to consider reducing the number of clients per server.

If you need to ensure a throughput for updates in your system, make sure each server that receives and processes updates can do so quickly without building up a backlog of updates in the queues. These backlogs can slow the server and bring the overall system throughput down.

In general, tuning the system for client/server configurations is an iterative exercise in looking for the best operating parameters for your systems and your applications. You are encouraged to use VSD which captures a large number of useful statistics. This tool can help you analyze the impact of tweaking various input parameters in the iterative process of improving system performance. Once your system is running, you should check periodically that performance continues to meet your requirements and make adjustments as needed.

9.8 Miscellaneous Server Notes

Garbage Collection Recommendations for Servers

Your servers' garbage collection (GC) parameters should be tuned to minimize full GCs. If a client pings a server that is doing a full GC, the client might think the server is dead and close connections to it. For guidelines on expediting your servers' garbage collection activities, see *Garbage Collection* on page 142 of the *GemFire Enterprise System Administrator's Guide*.

Suppressing Server Exception Logging

To reduce the amount of logging output from your server, you can configure your GemFire system to suppress bridge exceptions that are returned when a client or server unexpectedly terminates. To enable this option, add this system property setting to your Java command-line invocation:

```
-Dgemfire.bridge.suppressIOExceptionLogging=true
```

9.9 Socket Configuration

This section covers basic socket configuration for client/server communication.

Server Socket Backlog Size

When a client sends a connection request to a server, the operating system automatically accepts it into a socket backlog buffer. Whenever the server can handle another request, the operating system passes it one from the buffer. When the backlog is full, new requests are rejected.

By default, the GemFire server backlog setting is 1000 connection requests. It is quite high because of the very high client-to-server ratio of most GemFire installations. Depending on your application, you might need to set it higher or lower. A high setting can keep your clients from being needlessly rejected when there are a lot of clients connecting at once. However, queued requests can end up blocked in queue if any earlier request takes a long time to complete. If a client times out while waiting, it has to begin again to establish a server connection, and the server is left with the client's dead request in its backlog.

You can modify the backlog setting by passing this to your application at the command line:

```
-DCacheServer.backlog=newSetting
```

Socket Buffer Size

You can configure the buffer size of the sockets connecting the client and server both through the cache configuration file and through the Java API. The default setting is 32768 bytes. You should make sure the size is at least as large as the largest key and value that you expect to send plus 100 bytes for overhead. For best performance, the sizes should match between the client and server.

Server

To configure the socket buffer size for sockets originating in the server, use either the `socket-buffer-size` (page 209) attribute in the `cache.xml` or the `CacheServer` method `setSocketBufferSize`. This `cache.xml` snippet sets the socket buffer size to 256,000 bytes.

Example 9.22 Setting the Server Socket Buffer Size Through cache.xml

```
<cache-server port="41414" socket-buffer-size="256000"/>
```

The code in this example creates a server with the same settings as in Example 9.22.

Example 9.23 Setting the Server Socket Buffer Size Through the API

```
Cache cache = ...;
CacheServer cacheServer = cache.addCacheServer();
cacheServer.setPort(41414);
cacheServer.setSocketBufferSize(256000);
cacheServer.start();
```

Client

To configure the socket buffer size on sockets originating in the client, use either the pool `socket-buffer-size` (page 211) attribute in the `cache.xml` or the `PoolFactory` method `setSocketBufferSize`. This `cache.xml` snippet sets the socket buffer size to 256,000 bytes for a client.

Example 9.24 Setting the Client Socket Buffer Size Through cache.xml

```
<pool name="client" subscription-enabled="true" socket-buffer-size="256000">
   <locator host="localhost" port="41111"/>
</pool>
```

Socket Timeout

The default socket timeout setting for clients is 10000 milliseconds. This is the longest the client will wait for a server response to any operation. Too low a socket timeout setting might cause the client to get `SocketTimeoutException`s. A client is most likely to get exceptions during long operations, such as when it is registering interest. This tuning parameter should not just be automatically bumped up if there are exceptions. Generally, if this timeout setting is not unreasonably low, exceptions indicate overloading or resource contention on the server side that might be remedied by adding servers or server resources or by modifying server configuration.

To configure the timeout on client sockets, use either the pool `read-timeout` (page 211) attribute in the `cache.xml` or the `PoolFactory` method `setReadTimeout`. The `cache.xml` snippet in this example sets the pool's socket read timeout to 60000 milliseconds.

Example 9.25 Setting the Client Socket Timeout Through cache.xml

```
<pool name="client" subscription-enabled="true" read-timeout="60000">
    <server host="localhost" port="40404"/>
</pool>
```

Chapter 10

Configuring Multi-site Installations

GemFire Enterprise® provides mechanisms for scaling horizontally between disparate distributed systems through the loose coupling of multiple sites in peer-to-peer relationships. This chapter covers horizontal scaling using GemFire Enterprise, including information for configuring and tuning installations. A wide-area network (WAN) is the primary use case for the multi-site topology. A multi-site topology has two basic needs:

- To make systems that are at disparate geographical locations appear as one coherent system at all locations
- To ensure independence of the systems, so if any system is lost from view, the remaining systems continue to operate

The GemFire Enterprise multi-site implementation allows your disparate systems to act as one when they are coupled and as independent systems should communication between sites fail. The coupling is tolerant of weak or slow links between distributed system sites. These are some of the major features:

- Each site manages its own GemFire distributed system, but region data is shared across all sites.
- Data is consistent across two sites, provided the update key sets are unique to each site. If two sites are updating data for the same keys at the same time, there is no guarantee of consistency.
- The queue containing messages to be distributed from one site to another is overflowed to disk as needed. If a link is not available, the sending member stores its updates on disk to avoid running out of memory. In addition, the queue can be persisted to disk, so if the member managing the queue goes down it will pick up where it left off once it is restarted.
- Ordering of events sent between sites is preserved.

In this chapter:

- Multi-site Caching Overview (page 268)
- Basic Configuration (page 269)
- Gateway Configuration Parameters (page 272)
- Message Processing (page 274)
- Load Balancing in Multi-site Installations (page 279)
- Failover (page 285)
- Example Configuration (page 286)
- Multi-site Topologies (page 289)

10.1 Multi-site Caching Overview

A multi-site installation consists of two or more distributed systems that are loosely coupled. Each site is its own distributed system and has one or more logical connections to other sites over one or more physical connections. Those logical connections are known as *gateway hubs*.

In a client/server installation, the gateway hubs are configured in the server layer.

The gateway hubs are defined at startup in the distributed system member caches. Each hub is usually hosted by more than one system member, with one member hosting the primary hub instance and the others hosting backup secondaries. The primary is the only instance that communicates with remote sites. Hubs are identified within the distributed system by their hub-id. For any single hub, the configuration must be consistent across the distributed system.

This figure shows the high-level topology of a simple two-site installation with one logical gateway hub per site.

Figure 10.1 GemFire Multi-site Topology

10.2 Basic Configuration

To configure a multi-site installation:

1. Plan the topology of the connections among the multiple sites.

   For guidelines and caveats, see *Multi-site Topologies* on page 289.

2. Decide the gateway hub startup policy.
3. Define the gateway hubs.
4. Define the gateways for each hub, including:
   - Endpoints
   - Queue attributes
5. Enable regions to communicate updates to the gateways:
   - Enable communication
   - Specify hub ID

   For this hub to receive communication from other sites, the host on which this member is running and the port specified by this gateway hub must correspond to the host and port specified in at least one remote site's `Gateway.Endpoint`. For this hub to send communication to other sites, there must be `Gateway.Endpoint`s defined in its `Gateway`s with host and port pairs corresponding to the host and port of remote hubs.

   Except for ensuring that each hub is listening at a different host and port location and specifying whether the hub should start as primary or secondary, the gateway hub configuration should be identical between members in the same distributed system. The primary and its secondaries must interface in the same way with other sites.

Core Multi-site API

The `com.gemstone.gemfire.cache.util` package provides the interfaces and classes for multi-site communication.

com.gemstone.gemfire.cache.util

- **GatewayHub**—The gateway hub is defined at the cache level and represents this distributed system in the multi-site installation. The first member to start with a gateway hub defined generally becomes the primary hub member for the distributed system. The hub manages the gateways that send messages to other distributed system sites.
- **Gateway**—A gateway represents a single *remote* distributed system site in the multi-site installation and manages communication with the remote site. The gateway belongs to a gateway hub and acts as a proxy for communication with a remote site. The gateway might have multiple endpoints assigned to a single remote site. The gateway also specifies queue management parameters for its endpoints.
- **Gateway.Endpoint**—A gateway endpoint represents a single outgoing communication channel to a remote site. The gateway endpoint belongs to a gateway and communicates with a specific hub at a remote site.
- **GatewayQueueAttributes**—This specifies the communication behavior of the gateway. The gateway queue attributes govern the amount of memory allowed for the queue and what to do when the queue reaches the maximum. It also governs how the messages batches are managed and the frequency with which they are sent.

Configuring the Gateway Hub

This is a sample `cache.xml` for a gateway hub that listens on port 22222 and connects to two other hubs:

Example 10.1 Basic Gateway Hub Configuration in cache.xml

```
<cache>
   <gateway-hub id="Site1" port="22222" socket-buffer-size="125000">
      <gateway id="Site2" socket-buffer-size="125000">
         <gateway-endpoint id="Site2Lucy" host="lucy" port="22222"/>
         <gateway-endpoint id="Site2Ricky" host="ricky" port="22222"/>
         <gateway-queue batch-conflation="true"/>
      </gateway>
   </gateway-hub>
   ...
```

For compatibility, `Site2` hubs and endpoints must also define their `socket-buffer-size` as `125000`.

This example shows the API equivalent of the `cache.xml` declarations in the previous example:

Example 10.2 Basic Gateway Hub Configuration Through the API

```
// Create or obtain the GemFire cache
Cache cache = ... ;

// Create the Gateway Hub
GatewayHub site1Hub = cache.addGatewayHub("Site1", 22222);
site1Hub.setSocketBufferSize(125000);

// Add the Site2 gateway
Gateway site2Gateway = site1Hub.addGateway("Site2");
site2Gateway.setSocketBufferSize(125000);

// Create the Site2 endpoints
site2Gateway.addEndpoint("Site2", "lucy", 22222);
site2Gateway.addEndpoint("Site2", "ricky", 22222);
```

Implementing a Write-Behind Cache Listener in Your Hubs

To implement a write-behind cache listener, configure a gateway hub with:

1. no incoming port specification
2. a gateway with
   2.1 no endpoints
   2.2 one or more gateway event listeners

The gateway hub will pass events to the gateway's listeners. See *GatewayEventListener* on page 311.

Configuring Regions to Send Updates to the Gateway Queue

For a region to send its updates to the defined gateways, it must have the `enable-gateway` (page 107) region attribute set to `true`. Set this the same for all caches where the region is defined, regardless of whether a hub is running in the cache. This example shows the `cache.xml` settings for a region:

Example 10.3 Enabling Region Communication with a Gateway Hub in cache.xml

```
<region name="gatewayRegion">
   <region-attributes ... enable-gateway="true"/>
   ...
</region>
```

Here are the same settings through the API.

Example 10.4 Enabling Region Communication with a Gateway Hub Through the API

```
AttributesFactory factory = new AttributesFactory();
factory.setEnableGateway(new Boolean(true));
multiSiteRegion = cache.createRegion("gatewayRegion", factory.create());
```

10.3 Gateway Configuration Parameters

You can configure the gateways through the API or you can declare them in the `cache.xml` file as attributes and subelements of the `<gateway-hub>` element. The following tables describe the configuration parameters for the `cache.xml` file. The API provides setter methods for these parameters, which are detailed in the online Java documentation.

Table 10.1 Region Attribute

region attribute	Description	Default
`hub-id`	Specifies the destination hub for region events. *For multiple-hub systems this must be specified. If it is not, region events go to all available hubs, which results in duplicate sends to remote sites.*	`""`

Table 10.2 Gateway Hub Attributes

attribute	Description	Default
`id`	Identifier for the hub, usually an identifier associated with a physical location. *A primary hub and its secondary hubs must have the same ID, as they are the same logical thing.*	`none`
`manual-start`	Used only in the `cache.xml`. When `true`, the hub does does not start at cache startup and you need to start the hub through the API using one of the `GatewayHub` `start` methods. This lets you control when your system starts communicating with remote sites without while still using the XML declaration. Useful for involved startup procedures, such as those required when recovering from a crash, or for any situation in which you want to ensure that cache initiation finishes before remote site operations are allowed in.	`false`
`maximum-time-between-pings`	The maximum time in milliseconds the gateway allows to pass between messages or pings indicating the remote site is healthy. *A setting of 0 or a negative number turns off monitoring of the remote site's health. Be careful not to do this accidentally.*	`60000`
`port`	The port the hub is to listen on for communication from other sites. *If possible, use a single port number for all hubs in your WAN installation. This helps distinguish your gateway hubs from other services. If you have two hubs running on one machine, you must use different port numbers.*	`none`
`socket-buffer-size`	The size of the socket buffer used to receive messages from remote sites. This socket buffer size should match the size of the socket buffers on all remote gateways that have an endpoint specifying this hub's host and port. For details, see *Socket Buffer Size* on page 265.	`32768`
`startup-policy`	Specifies whether the hub should attempt to start as primary or secondary hub. Used for load balancing. Valid values are `primary`, `secondary`, and `none`.	`none`

The rest of the tables in this section list the configuration parameters of each subelement of `<gateway-hub>`.

Table 10.3 Gateway Attributes

attribute	Description	Default
`id`	Identifier for the gateway, usually the name of the remote hub represented by this gateway. Each gateway must be assigned a unique ID within the hub.	none
`socket-buffer-size`	The size of the socket buffers used to send messages to remote sites. For more information, see *Socket Buffer Size* on page 265.	`32768`
`socket-read-timeout`	The amount of time in milliseconds that this gateway will block waiting to receive an acknowledgement from a remote site.	`10000`

Table 10.4 Gateway Endpoint Attributes

attribute	Description	Default
`host`	The host to connect to. This must correspond to a host where a hub is running.	none
`id`	Identifier for the gateway, usually the name of the remote hub represented by this gateway. Each gateway must be assigned a unique ID within the hub.	none
`port`	The port to connect to. This must correspond to the port on which the hub is listening.	none

Table 10.5 Gateway Queue Attributes

attribute	Description	Default
`alert-threshold`		
`batch-conflation`	Boolean specifying whether to conflate messages.	`false`
`batch-size`	Limit on the number of messages a batch can contain.	`100`
`batch-time-interval`	Limit on the milliseconds that can elapse between sending batches.	`1000`
`disk-store-name`	Used if `enable-persistence` is `true`. Assigns this queue to the disk store with this name. By default this is not defined, which causes all disk activities to be done through the system's default disk store. If specified, this is the name of a store in the pool of disk stores serving the cache. See *Managing Disk Stores* on page 95 of the *GemFire Enterprise System Administrator's Guide*.	
`enable-persistence`	Boolean specifying whether to persist the queue to disk for high availability. If enabled, the queue is persisted to the disk store specified in `disk-store-name`. *This causes the queue to be managed like a persistent replicated region in the gateway's cache. .*	`false`
`maximum-queue-memory`	Maximum memory in megabytes that the queue can occupy before overflowing to disk.	`100`

10.4 Message Processing

The multi-site installation is designed so that disparate distributed systems can share data modifications with a minimal impact on distributed system performance. To this end, only the farthest-reaching entry operations are distributed between sites. Also, the queues used for messaging are designed in a way to minimize impact on distributed system performance. This section describes the information that is distributed between sites and how the distribution occurs. It also provides guidance for fine-tuning your gateway hubs for the most efficient and reliable message processing.

Operations That are Propagated Between Sites

The following operations are propagated from one site to another:

- entry create
- entry put
- entry distributed destroy, providing the operation is not an expiration action

These operations are not propagated:

- get
- invalidate
- local destroy
- expiration actions of any kind
- region operations

Configuration Option for Slow Distributed Systems

By default, when your gateway hub receives a message that should be propagated to remote sites, it synchronously deposits it into the gateway queue for processing. Depending on your system, this may noticeably affect the distributed system's performance as each update to a region waits for the gateway operations to finish before continuing. This has the severest performance impact for regions with `distributed-ack` scope.

If your distributed system is running slowly, it may be because it is waiting for the gateway queues. You can try queueing asynchronously to see if that alleviates the problem. To switch to asynchronous queueing, add the following to your Java command when starting your gateway hub:

```
-Dgemfire.asynchronous-gateway-distribution-enabled=true
```

When running in asynchronous mode, the gateway hub asynchronously places messages in a temporary queue rather than placing them directly into the gateway queues. The hub's temporary queue processor removes entries from this queue and delivers them to each gateway for distribution to remote sites. This protects the local distributed system's performance in case of a slow queue, as the region operation can continue without waiting for processing by the gateways. The following issues are associated with the temporary hub queue:

- The temporary queue is unprotected and unbounded, so it could cause the member to run out of memory. This could occur if the hub's queue processor either fails or does not get scheduled due to CPU issues.
- While the gateway queue can be persisted to disk (see *Gateway Queue Persistence* on page 275), the temporary queue cannot. If the member crashes or exits for any reason while data is in transit, the temporary queue contents are lost.

How the Gateway Processes its Queue

Each primary gateway contains a processor thread that reads messages from the queue and distributes them to the connected site. This thread batches the messages to be sent to the other site. To process the queue, the thread:

- Reads messages from the queue
- Creates a batch of the messages
- Synchronously distributes the batch to the other site and waits for a reply
- Removes the batch from the queue once the other site has successfully replied

Since the batch is not removed from the queue until after the other site has replied, the message cannot get lost. On the other hand, in this mode a message could be processed more than once. If a site goes offline in the middle of processing a batch of messages, then that same batch will be sent again once the site is back online. For more information, see *Gateway Queue Persistence* on page 275.

The batch is configurable via the batch size and batch time interval settings (for details, see *Configuring the Gateway Hub* on page 270). A batch of messages is processed by the queue when either the batch size or the time interval is reached. In an active network, it is likely that the batch size will be reached before the time interval. In an idle network, the time interval will most likely be reached before the batch size. This may result in some network latency that corresponds to the time interval.

Gateway Queue Persistence

The gateway queue can be persisted to disk for high availability. If a gateway hub with persistence enabled exits for any reason, when the hub is restarted (with the same queue persistence settings) it automatically reloads the queue and resumes sending messages. The queue is persisted to the disk store specified in `disk-store-name` (page 273).

Persisted gateway queues are configured like persistent replicated regions.

Persistence is enabled through the `cache.xml` by setting `enable-persistence` to `true` and specifying the disk store to use:

Example 10.5 Setting Queue Persistence Through cache.xml

```
<gateway-queue enable-persistence="true" disk-store-name="wanQueue"/>
```

Through the API, persistence is enabled by calling `setEnablePersistence` on the `GatewayQueueAttributes`, as in this example:

Example 10.6 Setting Queue Persistence Through the API

```
Gateway gateway = hub.addGateway("US");
GatewayQueueAttributes queueAttributes = gateway.getQueueAttributes();
queueAttributes.setEnablePersistence(true);
```

Batch Processing Failure

This section describes the exceptions that may occur while processing a batch. GemFire supports “at least once” delivery of messages in multi-site installations. Normally, when a receiving gateway receives a batch of messages, it processes them and responds with a success acknowledgement for the batch.

- **The receiving gateway fails with acknowledgement**—If processing fails while the receiving gateway is processing a batch, it replies with a failure acknowledgement containing the exception, including the identity of the message that failed, and the ID of the last message successfully processed. The sending gateway removes the successfully processed messages and the failed message from the queue and logs the exception and the failed message information. The sender then continues processing the messages remaining in the queue.
- **The receiving gateway fails without acknowledgement**—If the receiving gateway does not acknowledge a sent batch, the sender does not know which messages were successfully processed, so it resends the entire batch.
- **No remote gateways are available**—If a batch processing exception occurs because there are no remote gateways available, the batch remains on the queue. The sending gateway waits for a time, then attempts to re-send the batch. The time periods between attempts is five seconds. The existing server monitor continuously attempts to connect to the remote gateway, so eventually a connection will be made and queue processing will continue. Messages will build up in the queue and possibly overflow to disk in the meantime.

Site-to-Site Queue Conflation

Entry updates between remote distributed systems in a multi-site installation (see *Multi-site Topologies* on page 289) can be optionally conflated so only the latest value is sent for a particular key. Conflation is particularly useful when a single entry is updated often and all the individual updates don't require processing by the client. When an update is being added to a queue that has conflation enabled, if there is already an update in the queue for the entry key, the existing update assumes the value of the new update and the new update is dropped. This is shown in this figure for entry key A.

Figure 10.2 Site-to-Site Queue Conflation

Entry updates in the current in-process batch are not eligible for conflation. For this example, assume all of these messages are outside of the batch that is currently being sent to the remote site.

> *This method of conflation is different from the one used for server-to-client subscription queue conflation and peer-to-peer distribution within a distributed system. Compare to Conflating the Subscription Queue* on page 255 *and Asynchronous Queueing For Slow Receivers* on page 473.

Conflation is enabled on each gateway, so for any `gateway-hub`, each remote site either gets the updates conflated or does not.

With conflation, earlier entry updates in the queue are modified from updates sent later in the queue. This is problematic for applications that depend on a specific ordering of entry modifications. For example, if any remote gateway has a `CacheListener` that needs to know about every state change, you should disable conflation.

Declarative Configuration

Configuration of update conflation is done with the `batch-conflation` attribute of the `<gateway-queue>` node in the gateway-hub's cache configuration file. The attribute is `false` by default. This means that update conflation is disabled for that particular region. If the value is set to `true`, then updates for the region are conflated before they are sent to interested clients. This example enables conflation for the `EU` gateway:

Example 10.7 Declarative Gateway Conflation Configuration

```
<gateway id="EU" socket-buffer-size="256000">
   <gateway-queue overflow-directory="/home/wan_queues"
          maximum-queue-memory="50"
          batch-size="100"
          batch-time-interval="1000"
          batch-conflation="true"/>
</gateway>
```

Configuration Through the API

You can configure gateway update conflation through the Java API. This example also enables conflation for the `EU` gateway:

Example 10.8 Gateway Conflation Configuration Through the API

```
// Create the Gateway Hub
GatewayHub usHub = ...

// Add the EU gateway
Gateway euGateway = usHub.addGateway("EU");
euGateway.setSocketBufferSize(256000);

// Create the EU queue attributes
GatewayQueueAttributes euQueueAttributes =
new GatewayQueueAttributes("/home/wan_queues", 50, 100, 1000, true);
euGateway.setQueueAttributes(euQueueAttributes);
```

10.5 Load Balancing in Multi-site Installations

You can balance the load of gateway communication between multiple VMs. The basic approach is to define multiple gateway hubs and then split outgoing region event messaging between them.

With this functionality, you:

- Specify multiple gateway-hubs in your cache
- Indicate for each VM whether a hub should start as primary or secondary
- Point each gateway-enabled region to a specific gateway-hub

Only the primary hub instances send region events to remote sites. By splitting your hub primaries between multiple VMs and splitting your region events between hubs, you can achieve a level of load balancing in your outgoing hub messaging.

The Multiple-Hub Configuration

This figure shows a typical multiple-hub configuration. There may be any number of VMs in this distributed system in addition to the ones that host the gateways hubs.

In this configuration:

- There are two VMs hosting gateway hubs.
- There are two gateway hubs in each hub VM. One hub is primary in VM1 and the other hub is primary in VM2. Only the primaries send data to remote sites.
- Each hub has some of the region traffic pointing to it.

Figure 10.3 Two-Hub Configuration with Primaries in Different VMs

Event Flow With Multiple Hubs

This figure shows event flow for region A in the multiple-hub configuration.

- Region events originating in VM1 are stored in the local secondary Hub Site1a and are also distributed to VM2. Then in VM2, the events arriving in the local Region are sent to the VM's primary Hub Site1a, which then forwards them to remote sites.
- Region events originating in VM2 are sent directly out of the local primary Hub Site1a. They are also distributed to VM1 for cache update and storage in the secondary Hub Site1a.
- HubSite1b never sees this region's outgoing events.

Figure 10.4 Outgoing Region Event Flow for Two-Hub Configuration

If you have multiple hubs, make sure you explicitly assign one hub to each distributed region using the `hub-id` *region attribute. If you do not specify a hub for a region, its events will go to all hubs and each hub's primary will send the message to the remote sites, resulting in duplicate sends.*

Implementing a Multi-Hub System

Determine the configuration of the connections between the different distributed system sites.

For each site:

1. Identify the members that will host your gateway hubs, including which machines they will run on and the addresses they will use for gateway communication.
2. Configure your multi-hub gateways as you do your single hubs, with these additions:
   2.1 Set the `startup-policy` (page 272) for each hub. See *Configuring the Multi-Hub Gateway* on page 282.
   2.2 On all remote sites, define the gateways to the multi-hub gateway by pointing to every hub instance that you defined. See *Configuring to Connect to a Multi-Hub Gateway* on page 282.
3. Decide which regions will point to which hubs. Enable regions to communicate updates to the gateways, specifying:
   - `enable-gateway` (page 107)
   - `hub-id` (page 109)

   See *Configuring your Regions for Multi-Hub* on page 283.
4. Start all host VMs in the distributed system at the same time. This allows the primaries to assume their roles before any secondaries default to primary.

Configuring the Multi-Hub Gateway

Configure your multi-hub member by defining multiple gateway hubs, and specifying the startup policy for each hub:

Example 10.9 XML: Two-Hub Gateway Hub Configuration

```
<cache>
   <gateway-hub id="Site1a" port="22222" startup-policy="secondary">
      ...
   </gateway-hub>
   <gateway-hub id="Site1b" port="22223" startup-policy="primary">
      ...
   </gateway-hub>
```

Example 10.10 API: Two-Hub Gateway Hub Configuration

```
// Create or obtain the GemFire cache
Cache cache = ... ;

// Create the first Gateway Hub
GatewayHub site1aHub = cache.setGatewayHub("Site1a", 22222);
site1aHub.setStartupPolicy("STARTUP_POLICY_SECONDARY");

// Create the second Gateway Hub
GatewayHub site1bHub = cache.setGatewayHub("Site1b", 22223);
site1bHub.setStartupPolicy("STARTUP_POLICY_PRIMARY");

// Define gateways and endpoints for the gateway hubs ...
```

Configuring to Connect to a Multi-Hub Gateway

Specify all of your hubs as gateway endpoints on your remote site. The remote site will send each event to only one endpoint. This shows the XML for connecting to a multi-hub member:

Example 10.11 Remote Site Configuration Pointing to the Multiple-Hub Site

```
<cache>
   <gateway-hub id="Some remote gateway hub ...
      <gateway id="Site1a" ...
         <gateway-endpoint id="Site1aLucy" host="lucy" port="22222"/>
         <gateway-endpoint id="Site1aRicky" host="ricky" port="22222"/>
         <gateway-endpoint id="Site1bLucy" host="lucy" port="22223"/>
         <gateway-endpoint id="Site1bRicky" host="ricky" port="22223"/>
      </gateway>
   </gateway-hub>
```

The region hub specifications are only for outgoing events. For incoming events, hubs process every event received regardless of their destination regions.

Configuring your Regions for Multi-Hub

Split region events between your hubs as evenly as you can. If you have two hubs and three regions, for example, with one region having the bulk of the events, put that region on one hub and the others on the other hub.

> *When you use multiple hubs, you must point each gateway-enabled region at one of the hubs. If you do not specify a hub, region events are delivered to every hub and then sent to remote sites by each hub's primary instance. The region configuration must be consistent across the distributed system, even in caches where there are no hubs running.*

Assign regions to their hubs using one of these methods:

Example 10.12 XML: Enabling Region Communication with Gateway Hubs

```
<region name="A">
   <region-attributes ... enable-gateway="true" hub-id="Site1a"/>
   ...
</region>
<region name="B">
   <region-attributes ... enable-gateway="true" hub-id="Site1b"/>
   ...
</region>
```

Example 10.13 API: Enabling Region Communication with a Gateway Hub

```
// Region A
AttributesFactory factory = new AttributesFactory();
factory.setEnableGateway(new Boolean(true));
factory.setGatewayHubId("Site1a");
Region multiSiteRegionA = cache.createRegion("A", factory.create());

// Region B
AttributesFactory factory = new AttributesFactory();
factory.setEnableGateway(new Boolean(true));
factory.setGatewayHubId("Site1b");
Region multiSiteRegionB = cache.createRegion("B", factory.create());
```

How Hub Startup Policy Affects Startup Behavior

Every hub must have one primary instance running to maintain communication with remote sites. The other instances are secondaries that act as backups to the primary.

You can specify which hub instances should start as primary and which as secondary in your distributed system. When a hub is initialized, it attempts to assume the role specified, but will start up even if it must assume a different role.

This table describes startup behavior for the three policies.

Table 10.6 Gateway Hub Startup Policy and Startup Behavior

Startup Policy	Startup Behavior
`none` (default)	If no primary is running for the id, the hub starts as primary. Otherwise it starts as secondary.
`primary`	Same behavior as for `none` with the addition that the hub logs a warning if it starts as secondary.
`secondary`	If a primary is present, the hub starts as secondary. If there is no primary, the hub waits for up to one minute for a primary to start. If no primary starts in that time, the hub starts as primary and logs a warning.

10.6 Failover

This section describes possible failover scenarios.

- **The primary gateway hub goes offline**—If the primary gateway hub goes offline, a secondary gateway hub must take over primary responsibilities. The existing secondary gateway hubs detect that the primary gateway hub has gone offline, and one of them becomes the new primary gateway hub. Since the queue is distributed, its contents are available to all gateways. So when a secondary gateway becomes primary, it is able to start processing the queue where the previous primary left off with no loss of data.
- **A connection to a remote gateway hub goes offline**—If a connection to a remote gateway hub goes offline, a connection to another remote gateway hub is established so that messages can continue to be sent from the local gateway to the remote gateway.

10.7 Example Configuration

This section gives a hub configuration example for the US Site in this three-site installation:

Figure 10.5 Example Three-Site Topology

Cache Configuration File

This example XML file section shows the `US` gateway hub declaration with European and Asian gateway specifications in a WAN configuration:

Example 10.14 Declarative Gateway Hub Configuration

```
<cache>
   <gateway-hub id="US" port="22222" socket-buffer-size="256000">
      <gateway id="EU" socket-buffer-size="256000">
         <gateway-endpoint id="EU" host="poe" port="22222"/>
         <gateway-queue overflow-directory="/home/wan_queues"
            maximum-queue-memory="50"
            batch-size="100"
            batch-time-interval="1000"/>
      </gateway>
      <gateway id="ASIA" socket-buffer-size="256000">
         <gateway-endpoint id="ASIA-1" host="camus" port="22222"/>
         <gateway-endpoint id="ASIA-2" host="newton" port="22222"/>
      </gateway>
   </gateway-hub>
   ...
   <region name="multiSiteRegion">
      <region-attributes ... enable-gateway="true"/>
      ...
   </region
   ...
</cache>
```

Each `gateway` and `gateway-endpoint` must be assigned a unique ID.

API

A site can also be configured using GemFire API. This example accomplishes the same configuration as the declarative configuration in the previous example:

Example 10.15 Gateway Hub Configuration Through the API

```
// Create or obtain the GemFire cache
Cache cache = ... ;

// Create the Gateway Hub
GatewayHub usHub = cache.setGatewayHub("US", 22222);
usHub.setSocketBufferSize(256000);

// Add the EU gateway
Gateway euGateway = usHub.addGateway("EU");
euGateway.setSocketBufferSize(256000);

// Create the EU endpoint
euGateway.addEndpoint("EU", "poe", 22222);

// Create the EU queue attributes
GatewayQueueAttributes euQueueAttributes =
new GatewayQueueAttributes("/home/wan_queues", 50, 100, 1000);
euGateway.setQueueAttributes(euQueueAttributes);

// Add the ASIA gateway
Gateway asiaGateway = usHub.addGateway("ASIA");
asiaGateway.setSocketBufferSize(256000);

// Create the ASIA endpoints
asiaGateway.addEndpoint("ASIA-1", "camus", 22222);
asiaGateway.addEndpoint("ASIA-2", "newton", 22222);

Region multiSiteRegion;
try {
   AttributesFactory factory = new AttributesFactory();
   factory.setEnableGateway(new Boolean(true));
   ...
   multiSiteRegion = cache.createRegion("multiSiteRegion",
factory.create());

} catch (RegionExistsException ex) {
   multiSiteRegion = ex.getRegion();
}
```

Each `gateway` and `gateway-endpoint` must be assigned a unique ID.

10.8 Multi-site Topologies

This section describes GemFire's support for various topologies. Depending on your application needs, there may be several topologies that work. These are considerations to keep in mind:

- When a GemFire site receives a message, it forwards it to the other sites it knows about, excluding sites it knows have already seen the message. The message contains the initial sender's ID and the ID of each of the sites the initial sender sent to, so no site forwards to those sites. The messages do not pick up the ID of the sites they pass through, however, so it is possible in certain topologies for more than one copy of a message to be sent to one site (see *Topologies to Avoid* on page 291).
- In some configurations, the loss of one site affects how other sites communicate with one another.

Parallel

A parallel network is one where all sites know about each other. This is a robust configuration, where one of the sites can go down without disrupting communication between the other sites. This configuration also guarantees that no site receives multiple copies of the same message.

The parallel topology for three sites is shown in this figure. In this scenario, if site 1 sends an update to site 2, site 2 forwards to site 3. If site 1 sends an update to sites 2 and 3, neither forwards to the other. Likewise for any other initiating site. If any site is removed, the other two maintain a parallel topology.

Figure 10.6 Parallel Topology

Serial

A serial network is one where each site only knows about one other site. Data is passed from one site to the next serially. This figure shows the topology for three sites. In this scenario, if site 1 sends updates to site 2, site 2 forwards to site 3. Site 3 does not send the updates back to site 1.

This topology guarantees that every site receives one copy of each message sent by any site. With a serial installation, every site must stay up to maintain the topology. The failure of any site breaks the serial link. If site 2 went down, for example, site 3 could send to site 1, but site 1 could not send to site 3.

Figure 10.7 Serial Topology

Hybrid

A hybrid network is one contains both parallel and serial connections. There are numerous such configurations.

This figure shows a hybrid topology composed of a serial topology with an additional link directly from site 1 to site 3, creating a parallel topology between sites 1 and 3. With this topology, no site can receive the same update twice.

With this hybrid topology, if site 2 went down, it would not affect communication between sites 1 and 3 as both could still send to the other. If site 3 went down, however, site 2 would not be able to send to site 1.

Figure 10.8 Hybrid Topology: Example 1

In the figure below, sites 2 and 3 are isolated from one another. This topology might work for a situation where there is one producer (site 1) and multiple consumers that have nothing to gain from being connected. This topology also guarantees that no site receives the same update twice.

Figure 10.9 Hybrid Topology: Example 2

Topologies to Avoid

This section lists topologies that do not work and are unsupported.

In the topology shown in this figure, some sites might receive multiple copies of the same message. Assume, for example, that site 1 sends a message to site 2. Site 2 forwards the message to both sites 3 and 4. Site 4 will also send the message to site 3 as it does not know that site 3 has already been sent the message. Site 3 will receive the message twice. Additionally, sites 3 and 4 are dependent on site 2 for updates from site 1.

Figure 10.10 Unsupported Hybrid Topology 1

The figure below shows another hybrid topology that you should not use. In this configuration, if site 1 sends a message to site 2, site 2 forwards to site 3 and site 3 forwards to site 4. Site 4 does not know that site 3 has already been sent the message (the message only reports the original sender and receivers, in this case, site 1 and site 2, respectively). Site 4 also sends the message to site 3, and site 3 receives the message twice.

Figure 10.11 Unsupported Hybrid Topology 2

In the configuration in this figure, if site 1 sends a message to sites 2 and 3, sites 2 and 3 will both forward to site 4.

Figure 10.12 Unsupported Hybrid Topology 3

Chapter 11

Handling Events

This chapter covers the event handling features of the GemFire Enterprise® Cache. GemFire allows you to monitor system membership and caching activities throughout the distributed system. A subset of the capabilities is available for remote distributed systems, as between clients and servers and between distributed systems in a multi-site (WAN) installation.

In this chapter:

- The GemFire Event Framework (page 294)
- Event Handling in the Caching API (page 296)
- Event Handling in the Administration API (page 323)

11.1 The GemFire Event Framework

GemFire operations produce events that enable you to be notified of the work done by an application and unexpected situations like a losing a member. The events reflect changes to members, caches, regions, or data entries. You can implement event handlers to receive the events with callback methods that specify the action to take in response to each event.

The Event Cycle

The basic event cycle includes these steps:

1. An operation begins, such as closing the cache.
2. Executing the operation generates these objects:
   - An object of type `Operation`, which describes the method that triggered the event
   - An event object, which describes the event, such as the member and region where it originated
3. If the right kind of event handler is installed, the event is forwarded to it.

   Different event types require different handler types in different locations. If there is no matching event handler, that does not change the effect of the operation, which happens as usual.
4. When the handler receives the event, it triggers the handler's callback method for this event.

   The callback method can hand off the event object as input to another method.

   Depending on the type of event handler, the callbacks can be triggered before or after the operation. The timing depends on the event handler, not on the event itself.

   For transactions, after-operation listeners receive the events after the transaction has committed.
5. If the operation is distributed, so that it causes follow-on operations in other members, those operations generate their own events, which can be handled by their listeners in the same way.

   Even though the distributed operation usually causes the same method to be run on each member, they produce separate event objects that are not identical.

The Event Object

Event objects come in several types, depending on the operation. Some operations generate multiple objects of different types.

All event objects contain data describing the event, and each event type carries slightly different kinds of data appropriate to its matching operation. An event object is stable. For example, its content does not change if you pass it off to a method on another thread.

Getting Started With Event Handlers

Each event handler has a collection of methods that are called before or after different events. You can program the callback methods to do anything your application requires.

For details on writing event handlers, see *Implementing Cache Event Handlers* on page 300.

Cache Events and Administrative Events

GemFire provides two categories of events and event handlers:

- **Cache events** in the caching API are used by applications with a cache. Cache events provide detail-level notification, and they are the primary way for applications to handle events.
- **Administrative events** in the administration API are used by administrative applications without caches.

Both kinds of events can be generated in the same member. An operation can produce administrative events, cache events, or both, depending on the implementation of the method.

Your applications can use either type of event handlers to trigger callbacks, but you should choose only one type for each member. Because GemFire maintains the order of administrative events and the order of cache events separately, using cache events and administrative events in a single process can cause unexpected results.

For the cache event options, continue to *Event Handling in the Caching API* on page 296. For administrative events, go to *Event Handling in the Administration API* on page 323.

11.2 Event Handling in the Caching API

This section introduces the classes related to events and event handling in the caching API. For complete information, see the online Java API documentation for each class.

Listener Event Handlers

Any number and combination of the cache listeners in this table can be installed in the cache and its regions.

Table 11.1 Hierarchy of Listener Event Handlers for Cache Events

Handler	Events Received	Description
`CacheCallback` (page 313)		Superinterface of all cache event listeners except `BridgeMembershipListener`. The only function of `CacheCallback` is to clean up resources that the callback allocated. For more information, see the online Java API documentation.
`CacheListener` (page 305)	`RegionEvent` (page 298) `EntryEvent` (page 298)	Tracks changes to region and its data entries, responds synchronously. Extends `CacheCallback` interface. Installed in region.
`RegionMembershipListener` (page 305)	`RegionEvent` (page 298)	Provides after-event notification when a region with the same name has been created in another member and when other members hosting the region join or leave the distributed system. Extends `CacheCallback` and `CacheListener`. Installed in region as a `CacheListener`.
`RegionRoleListener` (page 306)	`RoleEvent` (page 298) `RegionEvent` (page 298) `EntryEvent` (page 298)	Tracks the presence of essential membership roles in the distributed system, plus changes to member, region, and data entries. Extends `CacheCallback`, `CacheListener`, and `RegionMembershipListener`. Installed in region as a `CacheListener`.
`TransactionListener` (page 308)	`TransactionEvent` (page 298) with embedded list of `EntryEvent` (page 298)	Tracks the outcome of transactions and changes to data entries in the transaction.Works with region-level listeners if needed. Installed at the cache level.
`BridgeMembershipListener` (page 309)	`BridgeMembershipEvent` (page 299)	Tracks changes to client/server connections. Static utility, installed at the process level using `BridgeMembership`.
`CqListener` (page 310)	`CqEvent` (page 299)	Tracks changes in a server cache that satisfy a client-specified query. Extends `CacheCallback` interface. Installed in `CqQuery` in the client.

Handler	Events Received	Description
GatewayEventListener (page 311)	GatewayEvent (page 299)	Tracks changes in a region for write-behind processing. Extends CacheCallback interface. Installed in a gateway.

Writer Event Handlers

Only one CacheWriter instance can be installed for each Region instance.

Table 11.2 Hierarchy of Writer Event Handlers for Cache Events

Handler	Events Received	Description
CacheCallback (page 313)		Superinterface of CacheWriter. The only function of CacheCallback is to clean up resources that the callback allocated.
CacheWriter (page 312)	RegionEvent (page 298) EntryEvent (page 298)	Installed in region. Responds to pending changes to the region and its data entries.
TransactionWriter (page 313)	TransactionEvent (page 298) with embedded list of EntryEvent (page 298)	Provides synchronous transaction handling, for use primarily with an outside data source. This handler can abort the transaction if needed. Installed at the cache level.

Cache Events

The caching API provides six event types. RegionEvent and EntryEvent, which extend the CacheEvent interface, are the general-purpose events created by changes to a region or a data entry. RoleEvent extends RegionEvent for reliability events. In addition, three special-purpose events, TransactionEvent, BridgeEvent, and CqEvent represent the complex events generated by operations within a transaction, by operations between a server and client, and by cache events that satisfy user-specified continuous queries.

All cache events are synchronous. To make them asynchronous, you can hand them off to another thread for the callback. For details on implementing callbacks, see *Writing Callback Methods* on page 304.

CacheEvent

The CacheEvent interface defines common event methods. The event object contains data needed to diagnose the circumstances of the event. It provides information about the event, including a description of the operation being performed on the cache, what member and region the event originated in, whether the event originated in a cache remote to the event handler, and the callback argument passed to the method that generated this event. This interface is extended by RegionEvent and EntryEvent.

Cache listeners and cache writers receive a CacheEvent, either a RegionEvent or an EntryEvent. The same event can be delivered to more than one event handler, and the handlers can take different actions in response. Cache listeners trigger their callbacks after the event happens, and cache writers take action before the event. The event handed to each of them can be the same. Before or after is not a property of the event itself, but of the event handler.

RegionEvent

The `RegionEvent` interface extends `CacheEvent` for region events. It provides information about operations that affect the whole region, such as whether the region is being reinitialized after being destroyed.

`RegionEvent`s are passed to `CacheListener`, `RegionMembershipListener`, `RegionRoleListener`, and `CacheWriter`.

EntryEvent

The `EntryEvent` contains information about an event affecting a data entry in the cache, including the key, the value before this event, and the value afterwards.

`EntryEvent.getNewValue` returns the current value of the data entry. `EntryEvent.getOldValue` returns the value before this event or `null` in cases such as empty regions where there is no data entry. For a partitioned region, you get the old value if the local cache holds the primary copy of the affected entry or `null` otherwise.

Note that `getOldValue` returns `null` for "entry not available" and also for "no value." If you need to know which, use `EntryEvent.isOldValueAvailable`. Entry not available can happen when the cost of obtaining the entry is deemed too high, like when the entry has been written out to disk. No value means there is no value that could be obtained for the entry.

You can retrieve serialized values from `EntryEvent`. This is useful if you get values from one region's events just to put them into a separate cache region. The standard `getOldValue` and `getNewValue` functions deserialize the values, which would then be reserialized when you `put` them into the cache. There is no counterpart `put` function as the `put` recognizes that the value is serialized and bypasses the serialization step.

`EntryEvent` also provides the transaction ID. This is the GemFire transaction ID, not the JTA ID. If the operation happened outside a transaction, the ID is `null`.

`EntryEvent`s are passed to `CacheListener`, `RegionMembershipListener`, `RegionRoleListener`, and `CacheWriter`. In addition, `TransactionListener` receives a list of `EntryEvent` instances in a `TransactionEvent`, as explained in *TransactionEvent* on page 298.

RoleEvent

The `RoleEvent` interface extends `RegionEvent` for region reliability events. Contains information about an event where membership roles are lost or gained, thus affecting a region's reliability.

`RoleEvent`s are passed to `RegionRoleListener`.

TransactionEvent

A `TransactionEvent` describes the work done inside a transaction. Depending on the event handler, this work is either about to be committed, just committed, or abandoned by a rollback or by a commit that has failed or been aborted.

The work is represented by an ordered list of `EntryEvent` instances. The entry events are listed in the order in which the operations were performed during the transaction.

As the transaction operations are performed, the events are conflated, with only the last event for each entry remaining in the list. So if entry A is modified, then entry B, then entry A, the list will contain the event for entry B followed by the second event for entry A.

`TransactionEvent` has methods that get the `TransactionId` and return the list of the `CacheEvents` for this transaction in chronological order of the operations.

`TransactionEvent`s are passed to `TransactionWriter` and `TransactionListener`.

BridgeMembershipEvent

A `BridgeMembershipEvent` provides information about changes in a client or a server. The event methods allow you to get a unique ID for the other process and determine if the event affects a connection to a client of this process.

`BridgeMembershipEvent`s are passed to `BridgeMembershipListener`.

CqEvent

A `CqEvent` provides information about a single event in a server. The `CqEvent` is associated with a continuous query, created by a client and run on its server to filter server events. The server runs the query against its events and forwards the ones that make it through the filter. In addition to information from the original cache event, the server adds information about the continuous query operation itself.

`CqEvent`s are passed to `CqListener`.

GatewayEvent

A `GatewayEvent` provides information about a single event in the cache. The `GatewayEvent` is delivered to a gateway queue for batched, asynchronous delivery to a `GatewayEventListener`. These are a subset of the entry events in a region. See *Message Processing* on page 274

Implementing Cache Event Handlers

You can specify event handlers for your region and region entry operations. These handlers register their interest in one or more events and are notified when the events occur.

For event handlers with more than one callback, GemFire provides a convenience class with empty stubs for every callback method in the interface. You can extend the class or the event handler interface, then define the callback methods to do whatever is required by your application in response to the events.

1. Decide which events your application needs to recognize.

   Make this decision for each region and for each cache. Event tracking generally happens on a region-by-region basis and, for transaction events, at the cache level.

2. Decide which handler needs to handle each event.

   Look at the `*Listener` and `*Adapter` classes in the `com.gemstone.gemfire.cache.util` package to see what your options are. If you need to respond to losing a region, for example, `CacheListenerWriter` has this callback method:

   - `beforeRegionDestroy(RegionEvent event)`

   And `CacheListenerAdapter` has this callback method:

   - `afterRegionDestroy(RegionEvent event)`

   The event is exactly the same, but the cache writer lets you prepare before the region is destroyed, while the cache listener lets you clean up afterwards.

   If you want to respond to losing this region in other members as well as the local one, then you want `RegionMembershipListenerAdapter`. This listener inherits the `afterRegionDestroy` callback method from the `CacheListener` interface and also provides these callback methods:

   - `afterRemoteRegionCrash(RegionEvent event)`
   - `afterRemoteRegionDeparture(RegionEvent event)`

3. Implement each event handler:

   3.1 Extend its adapter class or implement the interface. The adapter class has all methods already implemented as no-ops, so you only need to program the ones you want.

   3.2 Implement the callback method for each event you need to handle. If your event handler can be called by different threads at the same time, program for thread safety.

   3.3 If you intend to declare your handler in a `cache.xml` file, program it to implement the `Declarable` interface.

   3.4 Define the event handler for your region or cache, using either the API or the `cache.xml` file. For additional information on specific handlers, see the subsections under *Cache Event Listeners* on page 305 and *Cache Writers* on page 312.

4. Start the member. The event handlers are initialized automatically during cache and region creation.

GemFire provides several event handlers. Listeners are after-event handlers and writers are before-event handlers. In a single region, you can have different kinds of handlers, including single cache writer and more than one cache listener. On the cache, you can have a single transaction writer and multiple transaction listeners. In a gateway, you can have multiple gateway event listeners.

Declaring Event Handlers

This example shows how to declare an event handler for a region in `cache.xml`. This handler is a cache listener designed to communicate changes to a DB2 database. The declaration includes the listener's parameters: the database path, username, and password.

Example 11.1 Event Handler Declaration in cache.xml

```
<region name="exampleRegion">
   <region-attributes>
      . . .
      <cache-listener>
         <class-name>JDBCListener</class-name>
         <parameter name="url">
            <string>jdbc:db2:SAMPLE</string>
         </parameter>
         <parameter name="username">
            <string>gfeadmin</string>
         </parameter>
         <parameter name="password">
            <string>admin1</string>
         </parameter>
      </cache-listener>
   </region-attributes>
</region>
```

This shows part of the implementation of the `JDBCListener` declared in the previous example. When an entry is created, this listener's `afterCreate` callback updates the database. The listener's properties, which were declared in `cache.xml`, are passed into the `Declarable.init` method and used to create a database connection.

Example 11.2 Implementation of an After-Event Handler With Input Parameters

```
   . . .
public class JDBCListener
extends CacheListenerAdapter
implements Declarable {
   public void afterCreate(EntryEvent e) {
   . . .
      // Initialize the database driver and connection using input
parameters
      Driver driver = (Driver) Class.forName(DRIVER_NAME).newInstance();
      Connection connection =
         DriverManager.getConnection(_url, _username, _password);
      System.out.println(_connection);
         . . .
   }
         . . .
   public void init(Properties props) {
      this._url = props.getProperty("url");
      this._username = props.getProperty("username");
      this._password = props.getProperty("password");
   }
}
```

These examples install cache-wide transaction event handlers.

Example 11.3 XML: Configuring a Transaction Writer and Listeners

```
<cache search-timeout="60">
   <cache-transaction-manager>
      <transaction-listener>
         <class-name>com.company.data.MyTransactionListener</class-name>
         <parameter name="URL">
            <string>jdbc:cloudscape:rmi:MyData</string>
         </parameter>
      </transaction-listener>
      <transaction-listener>
      . . .
      </transaction-listener>
      <transaction-writer>
         <class-name>com.company.data.MyTransactionWriter</class-name>
         <parameter name="URL">
            <string>jdbc:cloudscape:rmi:MyData</string>
         </parameter>
         <parameter
         ...
         </parameter>
      </transaction-writer>
   </cache-transaction-manager>
   . . .
</cache>
```

Example 11.4 Java: Configuring a Transaction Writer and Listeners

```
CacheTransactionManager manager = cache.getCacheTransactionManager();
manager.addListener(new LoggingTransactionListener());
manager.addListener(new NotificationListener());
manager.setWriter(new SyncTransactionWriter());
```

Defining an Event Handler Through the API

Alternately, an event handler can be defined through the Java API. The next example shows how to define a cache listener through the `AttributesFactory` when creating a region. Because there can be multiple `CacheListener`s in a `Region`, this example uses `addCacheListener` to add the new listener to those already on the list.

Example 11.5 API Region Creation with a Cache Listener

```
// get attributes factory AttributesFactory fac = ...
fac.addCacheListener(new SimpleCacheListener());
Region nr=cache.createRegion(name, fac.create());
```

You can create a cache writer the same way, except using the `AttributesFactory` method `setCacheWriter`:

```
fac.setCacheWriter(new SimpleCacheWriter());
```

Writing Callback Methods

Callback methods in listeners, writers, and loaders must not call Java `Region` methods directly. Doing so could cause synchronous event handlers to block, hurting distributed system performance.

Instead, consider using a solution such as the `java.util.concurrent.Executor` interface in JDK version 1.5 and above. The following example shows a serial `Executor`, where the callback creates a `Runnable` that can be pulled off a queue and run by another object. This use of the `Executor` preserves the ordering of events. For details, see the `SerialExecutor` example on the Sun web site at http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executor.html.

Example 11.6 Using a Serial Executor in a Callback Method

```
public void afterCreate(EntryEvent event) {
    final Region otherRegion = cache.getRegion("/otherRegion");
    final Object key = event.getKey();
    final Object val = event.getNewValue();

    serialExecutor.execute(new Runnable() {
        public void run() {
            try {
                otherRegion.create(key, val);
            }
            catch (com.gemstone.gemfire.cache.RegionDestroyedException e) {
            ...
            }
            catch (com.gemstone.gemfire.cache.EntryExistsException e) {
            ...
            }
        }
    });
}
```

Cache Event Listeners

The cache event listeners allow you to receive after-event notification of changes to the region and its entries. Any number of listeners can be defined for the same region in the same member.

For descriptions of the callback methods these listeners provide, see the online Java documentation.

CacheListener

Partitioned region events only call listeners in the primary data host. Secondary data host events do not call listeners.

For distributed regions, listeners are invoked only for events in the local cache. These operations may be direct, cache-local API calls, or they may arrive at the cache through distribution or replication. Each listener has a collection of methods to be called after different events. Thus, when an entry is destroyed in a remote cache, the `CacheListener` is not invoked locally. If that destroy is distributed, however, this causes the local `CacheListener`'s `afterDestroy` method to be called.

Cache listener callbacks are called synchronously, so they will cause the cache modification operation to block if the callback blocks. This is particularly likely if you perform cache operations from a callback. To avoid this, use the `Executor` interface discussed in *Writing Callback Methods* on page 304.

Applications can subclass the `CacheListenerAdapter` class and override only the methods for the events you need. In addition to the inherited close method, the listener has callback methods for entry and region create, update, destroy, and invalidate, and one callback, `afterRegionLive`, for coordinating startup of durable clients.:

If you have durable clients, you invoke the `afterRegionLive` callback method after a region is initialized. For details on using cache listeners with durable clients, see *Implementing Cache Listeners for Durable Clients* on page 253.

A cache listener behaves a little differently with transactional operations. For details, see *How Transaction Events Are Managed* on page 417.

RegionMembershipListener

The `RegionMembershipListener` provides after-event notification when a region with the same name has been created in another member. You can also be notified when other members hosting the region join or leave the distributed system. This listener handles region events and extends `CacheListener` and `CacheCallback`.

Applications can subclass the `RegionMembershipListenerAdapter` class and override only the methods for the events you need.

`RegionMembershipListener` inherits the callback methods from `CacheCallback` and `CacheListener` and adds callback methods that allow you to track all other instances of the region in the distributed cache. :

In addition, The listener in this example responds to the loss of a local or remote region.

Example 11.7 Simple Implementation of a RegionMembershipListener

```
public class MyLostRegionListener
extends RegionMembershipListenerAdapter {
   public void afterRegionDestroy(RegionEvent event) {
      // customer code
   }
   public void afterRemoteRegionCrash(RegionEvent event) {
      // customer code
   }
   public void afterRemoteRegionDeparture(RegionEvent event) {
      // customer code
   }
}
```

RegionRoleListener

The `RegionRoleListener` handles region role membership events. These are membership events that are specific to the loss or gain of membership roles as defined by the region's `membership-attributes` configuration.

> *Partitioned regions do not support membership roles.*

Notification is given when joining or departing members that match one of the required membership roles for the region cause a change in reliability. Any number of region role listeners may be registered or removed. Notification of role loss only occurs after all instances of that role have left the system membership.

Applications can subclass the `RegionRoleListenerAdapter` class and override only the methods for the events you need.

`RegionRoleListener` inherits the callbacks from `CacheCallback`, `CacheListener` and `RegionMembershipListener`.

Adding Multiple CacheListeners to a Region

You can install as many `CacheListener`s on your region as you need, including instances of the subinterfaces `RegionMembershipListener` and `RegionRoleListener`. You can have different implementations doing different things, or multiple instances of the same listener.

> *This section does not apply to* `TransactionListeners` *or* `BridgeMembershipListeners`.

The order in which the listeners are added is important, because it is the order in which they are called. GemFire maintains an ordered list of the regions's listeners. You can execute `getCacheListeners` on the `AttributesFactory` or `AttributesMutator` to fetch the current ordered list. The `AttributesMutator.removeCacheListener` method takes a specified listener off the list.

This example uses the API to specify three cache listeners for a region, where `listener1`, `listener2`, and `listener3` are instances of `CacheListener` that you have already created.

Example 11.8 Installing Multiple Cache Listeners in a Region Through the API

```
CacheListener listener1 = new myCacheListener1;
CacheListener listener2 = new myCacheListener2;
CacheListener listener3 = new myCacheListener3;
attribFactory.initCacheListeners(new CacheListener[] { listener1, listener2,
listener3});
```

The `initListeners` method replaces any existing listeners. If you want to add more listeners to those that already exist, use `addListener` instead.

The cache listeners are sequential: `listener1` must finish its work before `listener2` begins, and `listener2` must finish before `listener3` begins. GemFire guarantees the order, as long as the same thread calls each listener. Because the listeners are synchronous, they will execute one at a time, as required.

This example adds the same cache listeners to the region in the same order, using a `cache.xml` file.

Example 11.9 Defining Multiple Cache Listeners in a cache.xml File

```
<region name="exampleRegion">
   <region-attributes>
         . . .
      <cache-listener>
         <class-name>myCacheListener1</class-name>
      </cache-listener>
      <cache-listener>
         <class-name>myCacheListener2</class-name>
      </cache-listener>
      <cache-listener>
         <class-name>myCacheListener3</class-name>
      </cache-listener>
   </region-attributes>
</region>
```

TransactionListener

The `TransactionListener` is a synchronous event listener for handling transaction-related events after the commit or rollback operation completes. `TransactionListener` methods cannot make transactional changes or cause a rollback. They can start a new transaction.

For partitioned region transactions, only the listeners in the primary data host are fired.

When the `TransactionListener` method executes, the transaction that caused the `TransactionListener` to be called no longer exists, but the thread that performed the transaction operation does. Multiple transactions on the same cache can cause concurrent invocation of `TransactionListener` methods.

> *Multiple transactions on the same cache can cause concurrent invocation of* `TransactionListener` *methods, so you must implement methods that do the appropriate synchronizing of the multiple threads for thread-safe operation.*

The `TransactionListener` is configured as an attribute in the `CacheTransactionManager` interface. Transaction listeners are set for the cache, not at the region level like most cache event listeners. Each transaction listener handles all the threads in all the regions of the cache for the application that set the listener.

Applications can subclass the `TransactionListenerAdapter` class and override only the methods for the events you need. The listener has callback methods for after commit or failed commit and for after an explicit transaction rollback.

BridgeMembershipListener

The `BridgeMembershipListener` allows clients and servers to receive notification of changes to client/server connections. Clients can receive membership events regarding their servers, and servers can receive membership events regarding their clients.

You can define one or more `BridgeMembershipListener`s for a single member. A `BridgeMembershipListener` is set for the member, like the administrative *SystemMembershipListener* on page 324, instead of at the region or cache level, like the other cache event listeners. You can call a `BridgeMembershipListener` from anywhere in your program.

Each listener receives all client/server membership events for the member. In a client, for every region that has a loader or writer defined, all client/server membership events feed to every listener that is registered. For details on client/server communications, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.

Applications can subclass the `BridgeMembershipListenerAdapter` class and override only the methods for the events you need.

BridgeMembership in com.gemstone.gemfire.cache.util

The API for client/server membership listeners is provided in the `com.gemstone.gemfire.cache.util` package.

- **BridgeMembershipEvent**—An event interface reporting a change in a client/server connection. The event methods allow you to get the member ID of the other process and to determine if the event regards a connection to a client of this process.
- **BridgeMembershipListener**—A listener whose callback methods are invoked when this process detects connection changes to servers or clients. The listener's callback methods are invoked when changes occur in the connection between this process and the other tier of the client/server cache. If this is a client, the methods are invoked for changes in connections to the server side. If this is a server, the methods are invoked for changes in connections to the client side. The listener has these callback methods:
  - `memberJoined(BridgeMembershipEvent event)`—Invoked when a connection is established between this process and a process in the other tier of the client/server cache.
  - `memberLeft(BridgeMembershipEvent event)`—Invoked for an orderly disconnect between this process and a process in the other tier of the client/server cache.
  - `memberCrashed(BridgeMembershipEvent event)`—Invoked for an unexpected disconnect between this process and a process in the other tier of the client/server cache.
- **BridgeMembership**—Provides utility methods for registering and deregistering `BridgeMembershipListeners` in this process.
- **BridgeMembershipListenerAdapter**—Provides empty implementations for all methods in `BridgeMembershipListener`. You can subclass this class and override the methods you need.

Implementation

This example outlines the basic implementation of the `BridgeMembershipListener`.

Example 11.10 Implementation of BridgeMembershipListener

```
public class MyMembershipListenerImpl
extends BridgeMembershipListenerAdapter {
   public void memberJoined(BridgeMembershipEvent event) {
      // customer code
   }
   public void memberLeft(BridgeMembershipEvent event) {
      // customer code
   }
   public void memberCrashed(BridgeMembershipEvent event) {
      // customer code
   }
}
```

A membership listener must be registered before a `BridgeLoader` is initialized for the listener to receive the `memberJoined` event generated by the `BridgeLoader` initialization. In this example, because the cache is created after the listener registration, any loader declarations in `cache.xml` result in `memberJoined` events to the listener.

Example 11.11 BridgeMembershipListener Registration

```
DistributedSystem ds = DistributedSystem.connect(properties);

MyMembershipListenerImpl myListener = new MyMembershipListenerImpl();
BridgeMembership.registerBridgeMembershipListener(myListener);

Cache cache = CacheFactory.create(ds);
```

CqListener

The `CqListener` allows clients to receive notification of changes to the server cache that satisfy a query that the client has registered to run on the server using a `CqQuery`. The query and the `CqListener` are both associated with a single `CqQuery`. The query is run on the server while the `CqListener` is run on the client.

You can define one or more `CqListener`s for a single query. A `CqListener` is set for the `CqQuery` through `CqQuery` attributes factory and mutator, similar to how the *CacheListener* on page 305 is set for the region. Events sent to the `CqListener` do not update the client cache.

Each listener registered with a `CqQuery` receives all of its continuous query events. Applications can subclass the `CqListenerAdapter` class and override the methods for the events you need.

For additional information on continuous queries, including an overview of the API, see Chapter 14, *Continuous Querying*, on page 397.

GatewayEventListener

The `GatewayEventListener` gives you write behind cache listener capability. Events for this listener are queued and delivered in batched `List`s to this listener's `processEvents` method.

To implement:

1. Follow the general steps in *Implementing Cache Event Handlers* on page 300
2. Install your implementation of the `GatewayEventListener` in a gateway with no endpoints, inside a gateway hub with no incoming port specification. Example:

Example 11.12 GatewayEventListener Registration

```
<cache>
   <gateway-hub id="DB">
      <gateway id="DB">
         <gateway-listener>
            <class-name>MyGatewayListener</class-name>
         </gateway-listener>
         <gateway-queue ...
      </gateway>
   </gateway-hub>
</cache>
```

3. Enable gateway queueing for the regions whose events you want to handle:

Example 11.13 Enabling WBCL Event Queueing

```
<region name="data">
   <region-attributes scope="distributed-ack" data-policy="replicate"
enable-gateway="true"/>
</region>
```

Cache Writers

A cache writer is an event handler that synchronously handles changes to a region's contents before the changes take place. The main use of a cache writer is to keep external data sources synchronized with a cache region.

CacheWriter

The `CacheWriter` is GemFire's single-operation event handler. To provide write-through caching with your external data source, you implement the `CacheWriter` methods with application-specific behavior.

Install your writers where you need them:

- For partitioned regions, only the writer installed in the primary data host is called.
- For distributed regions, the closest available writer in the distributed system is called.
- For local regions, the cache writer in the local cache is called.

You can only install one cache writer in a region for each member. Across the distributed system, multiple writers can be defined for a distributed or partitioned region, but only one writer is invoked for any event. When an operation is going to be performed on the region, GemFire selects a `CacheWriter`, if there is one, and informs it of the pending operation. The writer can choose to disallow the operation by throwing the `CacheWriterException` in its notification methods.

The `CacheWriter` interface has callback methods to handle region destruction and clear, as well as entry creation, update, and destruction. The writer receives callbacks before region events occur and has the ability to abort the operations. If a `CacheWriterException` or any runtime exception is thrown during a cache writer callback method, the operation does not complete and a `CacheWriterException` propagates back to the caller.

The callbacks are called synchronously and will block the operation if the callback blocks. This is particularly likely if you perform cache operations from a callback. To avoid this, use the `Executor` interface discussed in *Writing Callback Methods* on page 304.

Applications can subclass the `CacheWriterAdapter` class and override only the methods for the events you need.

TransactionWriter

The GemFire transaction writer is used for synchronizing with a transaction before it commits. This is usually used to synchronize with a backend data source, but can do anything your application requires. This gives you write-through caching for your transactions and allows you to abort the commit if you need to. The writer installed in the member where the data is first modified is the only one called. In partitioned regions, this may be a different member than the calling application. In transactions that don't use partitioned regions, this is the calling application.

Install your writers where you need them:

- For partitioned regions, only the writer installed in the primary data host is called.
- For distributed regions, the closest available writer in the distributed system is called.
- For local regions, the cache writer in the local cache is called.

When the `TransactionWriter` method executes, the transaction that caused the writer to be called is waiting for the commit operation to return. Multiple transactions on the same cache can cause concurrent invocation of the `TransactionWriter` `beforeCommit` method. You can explicitly abort the commit by throwing a `TransactionWriterException` in the `beforeCommit` callback. Additionally, if any runtime exception is thrown from `beforeCommit`, the commit does not complete and a `TransactionWriterException` propagates back to the caller.

The callback is called synchronously and will block the commit if the callback blocks. This is particularly likely if you perform cache operations from the callback. To avoid this, use the `Executor` interface discussed in *Writing Callback Methods* on page 304.

Use the `CacheTransactionManager` interface to install a transaction writer in every cache where you host data that is modified inside transactions.

The transaction writer has just the one callback method, beforeCommit, which is invoked before a transaction is committed.

CacheCallback

The `CacheCallback` interface is the superinterface of most cache event handlers. `CacheCallback` has a single method, `close`, which is invoked when the region is closed or destroyed, when the cache is closed, or when the callback is removed from a region using an `AttributesMutator`.

You can implement `close` to clean up any external resources, such as database connections. For event handlers that inherit from `CacheCallback`, `close` is the only required method. Every `CacheWriter`, `CacheListener`, `RegionMembershipListener`, and `RegionRoleListener` must implement `close`.

Cache Event Handling in a Distributed System

This section explains the sequence of a distributed operation and the notification of the events arising directly or indirectly from that operation. A cache listener must be installed for the region in each system member that needs to receive notification of the resulting changes.

Specifying Which Entry Events the Region Receives

In the delivery of entry events from the distributed region, you have two options: The local cache can receive events for all entries in this region anywhere in the distributed system or only for entries that already exist locally.

Which events are received from a remote operation depends on the interest policy set for the region. To minimize message traffic, members with the default interest policy of `cache-content` receive only the

events for the entries stored in their local cache. Those with an interest policy of `all` receive all the events for operations anywhere in the region. For details, see `subscription-attributes` on page 118.

Events in a Partitioned Region

A distributed operation follows this sequence in a partitioned region:

- Apply the operation to the cache with the primary data entry, if appropriate.
- Do the distribution based on the `interest-policy` of the other members.
- Invoke the listener in the cache of each member that receives the distribution.
- Invoke the listener in the cache with the primary data entry.

This figure shows an example of entry creation in a four-member partitioned region with redundancy.

Figure 11.1 Cache Listeners in a Partitioned Region With Redundant Data

1. An application on member M_1 makes an API call to create an entry.
2. The partitioned region creates the new entry in the cache in M_2.

   M_2, where the primary copy resides, drives the rest of the procedure.
3. These two operations happen at the same time:
   - The partitioned region creates a secondary copy of the entry in the cache in M_3. Notice that creating the secondary copy does not invoke the listener on M_3.
   - M_2 distributes the event to M_4. This distribution to other members is based on interest policies. M_4 has an `interest-policy` of `all`, so it receives notification of all events anywhere in the region. Since M_1 and M_3 have an `interest-policy` of `cache-content`, and this event does not affect any pre-existing entry in their local caches, they do not receive the event.
4. The cache listener on M_4 handles the notification of the remote event on M_2.

5. Now that everything on the other members has completed successfully, the original create operation on M_2 succeeds and invokes the cache listener on M_2.

Events in a Distributed Region

A distributed operation follows this sequence in a distributed region:

- Apply the operation to the local cache, if appropriate.
- Invoke the local listener.
- Do the distribution.

Then each member that receives the distribution carries out its own operation in response, which invokes its local listener.

This figure shows an example of entry creation in a three-member distributed region.

Figure 11.2 Cache Listeners in a Distributed Region

These are the steps:

1. An entry is created through a direct API call on member M_1.
2. The create invokes the cache listener on M_1.
3. M_1 distributes the event to the other members.
4. M_2 and M_3 apply the remote change through their own local operations.

   M_3 does a create, but M_2 does an update, because the entry already existed in its cache.
5. The cache listener on M_2 receives callbacks for the local update.

   Since there is no cache listener on M_3, the callbacks from the create on M_3 are not handled.

Contents of the Entry Events

The `Event` object describes the operation performed in the local cache. If the event originated remotely, it describes the local application of the remote entry operation, such as the update in member M_2 or the create in M_3 in Figure 11.2. The only exception is when the local region has an `empty` data policy; then the event carries the information for the remote cache operation. See *Configuring an Empty Region* on page 318.

The `Event` object says whether it originated in the local cache or a remote cache. On M_1, for example, the event specifies that the it originated locally, while the events on the other members specify that they originated remotely. The `Event` object also stores the old value of the entry, which might be different in different members.

Occasionally the operation itself can be different—when the originator does a create, the remote region can turn that into an update (or vice-versa), as happened in Figure 11.2. If so, the `Operation` object remains stable; in both cases it says `CREATE`. But the callback method that is kicked off is different, `afterCreate` versus `afterUpdate`.

Ordered Operation Delivery

When a series of operations performed by a single thread is distributed, GemFire guarantees that the application of the operations in the remote members happens in the same order as the original operations in the local member. All the distributions for a thread are put into an ordered queue, and $event_2$ is not distributed to a given member until $event_1$ completes on that member. Thus, the order is preserved, but the relative time is not.

Slow Event Receivers

If one member is slower than the others, GemFire's response to the situation depends on the kind of region:

- **Partitioned region**–The local member waits until all the remote members activate their event handlers, complete their callbacks, and respond before distributing the next event to all the members.
- **Distributed region with distributed-ack scope**—The local member waits until all the remote members activate their event handlers, complete their callbacks, and respond before distributing the next event to all the members.
- **Distributed region with distributed-no-ack scope** —The local member puts the events in the queue, and the next event goes to each remote member as soon as it completes its callbacks from the previous event.

Multiple Listeners

If there are multiple listeners in the region, they are also invoked in the order they were installed, so you can write listeners that depend on work done by other listeners.

Multi-Threaded Applications

For partitioned regions, GemFire guarantees ordering of events across threads, but for distributed regions it doesn't. For multi-threaded applications that create distributed regions, you need to use your application synchronization to make sure that one operation completes before the next one begins. Distribution through the `distributed-no-ack` queue can work with multiple threads if you set the `conserve-sockets` attribute to true. Then the threads share one queue, preserving the order of the events in distributed regions. For details on `conserve-sockets`, see *Controlling Socket Use* on page 471.

Different threads can invoke the same listener, so if you allow different threads to send events, it can result in concurrent invocations of the listener. This is an issue only if the threads have some shared

state—if they are incrementing a serial number, for example, or adding their events to a log queue. Then you need to make your code thread safe.

Loosely Coupled Multi-Site Installations

As long as the events are generated by a single thread, the gateways preserve the ordering of events between sites. Gateways queue and forward most entry operations, but not all of them. For details, see *Operations That are Propagated Between Sites* on page 274.

Events in Transactions

If the event is part of a transaction, it works a little differently, both in the delivery of events to listeners and in triggering the callbacks.

Event Order

Transaction events are delivered to cache and transaction listeners after the commit. Some events may not be preserved for delivery in a remote member.

GemFire invokes the cache listeners for all the operations in the transaction event, in the order in which they are listed. The transaction listeners are called once, and passed the full transaction even.

An event access method `TransactionEvent.getEvents` returns an ordered collection of the events that comprise the transaction. So, if there is a transaction like this:

```
begin
invalidate(a)
invalidate(b)
invalidate(c)
put(d)
destroy(e)
commit
```

then the `TransactionEvent.getEvents` method returns a collection whose iterator would return `CacheEvent`s for `a`, `b`, `c`, `d`, and `e` in the order they were originally executed.

If a single data point is modified more than once inside the transaction, only the final event is provided in the event object list. For a transaction like this:

```
begin
invalidate(a)
invalidate(b)
put(a)
commit
```

the `TransactionEvent.getEvents` method returns a collection whose iterator returns `CacheEvent`s for `b` and `a` - in that order, with only the `put` on `a` preserved.

Callbacks

In transactions, `CacheWriter` callbacks are invoked as the cache operations are performed, with the most convenient `CacheWriter` used, as for non-transactional operations. The `TransactionWriter` is called before the commit, and can abort the commit if needed. All other callbacks, including the `TransactionListener` and `CacheListener`, are triggered after the commit is performed.

Applications Without Local Caching

An application can receive all events on a region without storing the data that comes with the events. In this way, the application can pass data through to other receivers without the overhead of caching. This configuration lets you separate event processing from data management.

For example, in the figure below, App_1 receives data from an outside source, but does not cache the entries in its local heap. App_1 processes the events and distributes the result to the other members of this distributed system: a cacheserver and App_2.

The cache server maintains the data entries in its cache, and therefore applies changes through local cache operations. The cache server here behaves the same way as members M_2 and M_3 in Figure 11.2 on page 315.

App_2, on the other hand, is an event consumer that is configured to receive events without storing the entries. App_2 uses the events received from App_1 to drive updates to an outside datastore.

Figure 11.3 Empty Producer With a Cache Server and an Empty Event Consumer

Configuring an Empty Region

To create a region without local caching (like the regions in the caches of App_1 and App_2 in the figure):

- For distributed regions or regions with `local` scope, disable caching by setting a `data-policy` (page 104) of `empty`. To receive events, set `subscription-attributes` (page 118) `interest-policy` to `all`.
- For partitioned regions, disable caching by setting the `local-max-memory` property to 0. For details, see *Accessing a Region* on page 94.

Event Behavior in Empty Regions

A region with `data-policy` set to `empty` behaves in the following ways.

Events that Originate Locally

Local events are delivered to each installed cache listener and, for distributed operations, the event is distributed to all interested members in the system. Events don't affect the local state, because there is no local caching. The `getOldValue` method returns `null`. This configuration can be used to create a very thin producer in the distributed system.

Events that Originate Remotely

If the subscription attribute `interest-policy` is set to `cache-content`, no remote events are delivered. If the `interest-policy` is set to `all`, all events for distributed operations that originate remotely are delivered to the cache listeners on the region - the very thin consumer model. The local state is not modified.

Remote entry events carry all of the information for the remote cache operation. The state of the event is the same as it would be in the originating cache, except:

- `isOriginRemote` returns `true`.
- `getOldValue` returns `null`.

Operation Behavior in Empty Regions

In a region without local caching, some methods behave differently.

Local Operations

`Region` methods that are based on local region state work normally. `Region` methods that are based on local entry state behave as shown in this table. Any local operations not mentioned here work normally.

Table 11.3 Behavior Changes to Local Methods in Empty Regions.

Local Method	Behavior in Empty Region
`getEntry`	Returns null.
`keys`	Returns an empty `Set`.
`values`	Returns an empty `Collection`.
`entries(false)`	Returns an empty `Set`.
`entrySet(false)`	Returns an empty `Set`.
`containsValueForKey`	Returns false.
`containsKey`	Returns false.
`writetoDisk`	Does nothing.
`existsValue`	Returns false.
`query`	Returns an empty `SelectResults`.
`selectValue`	Returns null.
`localInvalidate`	Throws `EntryNotFoundException`.
`localDestroy`	Throws `EntryNotFoundException`.
`localClear`	Does nothing.
`isEmpty`	Returns true.
`containsValue`	Returns false.
`size`	Returns 0.

Distributed Operations

Distributed operations continue to distribute even when the region does not have local caching. The methods behave as shown in this table. The distributed operations not mentioned here work normally.

Table 11.4 Behavior Changes of Distributed Methods in Empty Regions

Distributed Method	Behavior in Empty Region
`get`	Always does a `netSearch` (if distributed scope). Uses the normal `CacheLoader` rules.
`create`	Never throws `EntryExistsException`. Always distributes the create. Uses the normal `CacheWriter` rules.
`put`	Same as `create`, which means the remote members act as if the originator did a create.
`putAll`	Works normally.
`invalidate`	Always does a distributed entry invalidate. Never throws `EntryNotFoundException`.
`destroy`	Always does a distributed entry destroy (unless the cache writer aborts the operation). Never throws `EntryNotFoundException`.
`remove`	Like destroy, but also always returns `null`.

Region Features Disabled in Empty Regions

The region features in this table have no effect when local caching is turned off, or they are not allowed at configuration time. The other region features continue to work normally.

Table 11.5 Disabled Region Features in Empty Regions

Disabled Feature	Behavior
`local scope`	Listeners, loaders, and writers are called.
`entry-time-to-live`	`IllegalStateException` is thrown.
`entry-idle-time`	`IllegalStateException` is thrown.
`Eviction`	`IllegalStateException` is thrown.

Map Attributes:
`initial-capacity`
`load-factor`
`concurrency-level`
	All the map attributes become no-ops, since no entry map exists.

`IndexMaintenance Synchronous`	No index to maintain.
Queries	Normal behavior when no matching data is found.

In addition, the following changes in behavior are caused by disabling local storage:

- **Transaction conflict detection**—The only cause of transaction conflicts on transactions committed on a region with no local storage are other transactions committing in the same commit window. The commit window starts when commit is called and completes after reservations are made.
- **Concurrent invocations of callbacks for the same key**—Since no local entry exists to allow synchronization, cache callbacks (listener, writer, or loader) can receive concurrent calls for keys that are equal to each other.

11.3 Event Handling in the Administration API

The GemFire administration API provides programmatic access for managing a single distributed system. This section introduces the classes related to events and event handling in the administration API. For more information, see Chapter 19, *Developing System Administration Tools*, and the on-line Java API documentation for the classes mentioned in this section.

Admin Listeners

The listeners in this table can be installed in an administrative application to track the coming and going of members, caches, and regions.

Table 11.6 Hierarchy of After-Event Handlers for Administrative Events

Handler	Events Received	Description
`SystemMembershipListener` (page 324)	`SystemMembershipEvent` (page 323)	Installed in any admin application, not in cache. Tracks members as they join and leave the distributed system.
`SystemMemberCacheListener` (page 324)	`SystemMemberCacheEvent` (page 323) `SystemMemberRegionEvent` (page 323)	Installed in any admin application, not in cache. Tracks members, caches, and regions as they come and go.

Administrative Events

The administration API provides three event classes: `SystemMembershipEvent`, `SystemMemberCacheEvent`, and `SystemMemberRegionEvent`. Every event has a member ID that identifies the member where the event originated.

SystemMembershipEvent

The `SystemMembershipEvent` interface reports when a member has joined or left the distributed system and provides the member ID of the process that joined or left.

SystemMemberCacheEvent

The `SystemMemberCacheEvent` interface extends `SystemMembershipEvent` for cache events. The administrative application does not have a cache—this event reports when a cache is created or closed.

SystemMemberRegionEvent

The `SystemMemberRegionEvent` interface extends `SystemMembershipEvent` for region events. The administrative application does not have a cache—this event reports when a region is created or destroyed in the distributed system.

Administrative Event Listeners

The listeners in the administration API provide methods to be called after various events.

SystemMembershipListener

The `SystemMembershipListener` is an event handler interface in the administration API whose callback methods are invoked when members join or leave the GemFire distributed system. This listener receives `SystemMembershipEvent`s.

A `SystemMembershipListener` is set for the member, and you can call this listener from anywhere in your program.

The `SystemMembershipListener` has callback methods to handle member joins, crashes, and normal departures.

SystemMemberCacheListener

The `SystemMemberCacheListener` is an event handler interface whose callback methods can be used to track the life cycle of caches and regions in the GemFire distributed system. This listener receives `SystemMemberCacheEvent`s and `SystemMemberRegionEvent`s.

The `SystemMemberCacheListener` has callback methods for cache and region creation and removal.

Universal Membership Listener Adapter

The `UniversalMembershipListenerAdapter` is a convenience wrapper for `BridgeMembershipListener` that allows you to handle client/server membership events as if they were standard system membership events. The adapter also wraps `SystemMembershipListener`, allowing you to use one interface for both types of events.

GemFire Enterprise does not maintain ordering between client/server events and administrative events, however, even when you use the `UniversalMembershipListenerAdapter`.

Universal Membership Listener in com.gemstone.gemfire.cache.util

- **UniversalMembershipListenerAdapter**—A wrapper for `BridgeMembershipListener` and `com.gemstone.gemfire.admin.SystemMembershipListener`. The wrapper adapts `BridgeMembershipListener` events so they appear as `SystemMembershipListener` events.

  The `UniversalMembershipListenerAdapter` provides the same callback methods as the `SystemMembershipListener` and the `BridgeMembershipListener` for members joining and leaving the system.

Because the adapter presents both types of events as `SystemMembershipListenerEvent`s, you can adapt your `SystemMembershipListener` event code to handle `BridgeMembershipListener` events by modifying your listener implementation to extend this class and then making any necessary client/server-specific modifications.

Implementation

To use the universal adapter, have your listener extend it, as in the example below. The universal adapter can handle both client/server and system events. When you construct a listener that extends the universal adapter, the construction code automatically registers the underlying `BridgeMembershipListener`. This example shows a typical implementation for a client with no distributed system peers. Such a client is interested only in client/server events, and thus only needs to construct the listener.

Example 11.14 Implementing the Universal Adapter

```
public class MyMembershipListenerImpl
extends UniversalMembershipListenerAdapter {
   public void memberCrashed(SystemMembershipEvent event) {
      // customer code
   }
   public void memberLeft(SystemMembershipEvent event) {
      // customer code
   }
   public void memberJoined(SystemMembershipEvent event) {
      // customer code
   }
}
new MyMembershipListenerImpl();
```

If you also want to receive system events, you must construct the listener and then register your `AdminDistributedSystem` with it. This example shows the `admin` and listener code required to use the listener for client/server and system membership events. This is a typical use of the universal adapter for servers.

Example 11.15 Using the Universal Adapter for System and Client Membership Events

```
DistributedSystemConfig config =
   AdminDistributedSystemFactory.defineDistributedSystem(myDS, null);
AdminDistributedSystem adminDS =
   AdminDistributedSystemFactory.getDistributedSystem(config);
adminDS.connect();
MyMembershipListenerImpl myListener = new MyMembershipListenerImpl();
myListener.registerMembershipListener(adminDS);
```

For the listener to manage both client/server and system events, register the `AdminDistributedSystem` with the listener, as shown here. Do not register the listener with the `AdminDistributedSystem`.

Chapter 12

Delta Propagation

In most distributed data management systems, the data stored in the system tends to be created once and then updated frequently. These updates are sent to other members for event propagation, redundancy management, and cache consistency in general. Tracking only the changes in an updated object and sending only the updates, or deltas, mean lower network transmission costs and lower object serialization/deserialization costs. Performance improvements can be significant, especially when changes to an object instance are relatively small compared to the overall size of the instance.

In this chapter:

- How Delta Propagation Works (page 328)
- Supported Topologies and Limitations (page 329)
- Delta Propagation API (page 331)
- Delta Propagation Properties (page 332)
- Implementing Delta Propagation (page 334)
- Errors in Delta Propagation (page 336)
- Performance Considerations and Limitations (page 337)
- Example of Delta Propagation (page 338)

12.1 How Delta Propagation Works

GemFire propagates object deltas using methods that you program. The methods are in the `Delta` interface, which you implement in your cached objects' classes. If any of your classes are plain old Java objects, you need to wrap them for this implementation. This figure shows delta propagation for a change to an entry with key, `k`, and value object, `v`.

Figure 12.1 Delta Propagation

1. **`get` operation**. The `get` works as usual: the cache returns the full entry object from the local cache or, if it isn't available there, from a remote cache or from a loader.
2. **update methods**.You need to add code to the object's update methods so that they save delta information for object updates, in addition to the work they were already doing.
3. **`put` operation**.The `put` works as usual in the local cache, using the full value, then calls `hasDelta` to see if there are deltas and `toDelta` to serialize the information. Distribution is the same as for full values, according to member and region configuration. Gateways only send full values.
4. **receipt of delta at remote member**. `fromDelta` extracts the delta information that was serialized by `toDelta` and applies it to the object in the local cache. The delta is applied directly to the existing value or to a clone, depending on how you configure it for the region.
5. **additional distributions**. As with full distributions, receiving members forward the delta according to their configurations and connections to other members. For example, if VM1 is a client and VM2 is a server, VM2 forwards the delta to its peers and its other clients as needed. Receiving members do not recreate the delta; `toDelta` is only called in the originating member.

12.2 Supported Topologies and Limitations

This section describes circumstances in which GemFire uses delta propagation and those in which it sends the full value. Unless otherwise noted, all configuration is done in the `cache.xml` or through the `Cache` or `Region` APIs.

General Characteristics of Delta Propagation

To use the delta propagation feature, all updates on a key in a region must have value types that implement the `Delta` interface. You cannot mix object types for an entry key where some of the types implement delta and some do not. This is because, when a type implementing the delta interface is received for an update, the existing value for the key is cast to a `Delta` type to apply the received delta. If the existing type does not also implement the `Delta` interface, the operation throws a `ClassCastException`.

Sometimes `fromDelta` cannot be invoked because there is no object to apply the delta to in the receiving cache. When this happens, the system sends the full value. There are two possible scenarios:

- If the system can determine beforehand that the receiver does not have a local copy, it sends the initial message with the full value. This is possible when regions are configured with no local data storage, such as the partitioned region with `local-max-memory` set to 0 and the distributed region with `empty data-policy` and `subscription-attributes interest-policy` set to `all`. These configurations are used to accomplish things like provide data update information to listeners and to pass updates forward to clients.
- In less obvious cases, such as when an entry has been locally deleted, first the delta is sent, then the receiver requests a full value and that is sent. Whenever the full value is received, any further distributions to the receiver's peers or clients uses the full value.

GemFire does not propagate deltas for:

- Transactional commit
- The `putAll` operation
- VMs running GemFire versions 6.0 and earlier

Peer-to-Peer

GemFire system members can distribute and receive entry changes using delta propagation, with these requirements and caveats:

- Regions must be partitioned or have their `scope` set to `distributed-ack` or `global`. A `distributed-no-ack` scope would not allow the receiver to recover if an exception occurred while applying the delta.
- For partitioned regions, if a receiving peer does not hold the primary or a secondary copy of the entry, but still requires a value, the system automatically sends the full value.
- To receive deltas, a region must be non-`empty`. The system automatically sends the full value to empty regions. Empty regions can send deltas.

Client/Server

GemFire clients can always send deltas to the servers, and servers can usually sent deltas to clients. These configurations require the servers to send full values to the clients, instead of deltas:

- When the client's `gemfire.properties` setting `conflate-events` is set to `true`, the servers send full values for all regions.
- When the server region attribute `enable-subscription-conflation` is set to `true` and the client `gemfire.properties` setting `conflate-events` is set to `server`, the servers send full values for the region.
- Servers send full values to empty client regions.

Multisite (WAN)

Deltas are not sent over gateways. The full value is always sent.

12.3 Delta Propagation API

This section describes the interfaces, methods and classes for the delta propagation API. Is com.gemstone.gemfire a package?

com.gemstone.gemfire

Delta

This interface defines a contract between the application object and GemFire. Your application objects that implement `Delta` participate in delta propagation. `Delta` has the following methods, which you program to fit your application needs:

- `boolean hasDelta`—This method returns a boolean indicating whether the object has delta changes. This is called when the application puts the entry into the cache.
- `void toDelta(DataOutput out)`—If `hasDelta` returns true, GemFire calls `toDelta` to extract the delta from the application object into `java.io.DataOutput`. GemFire distributes the output from `toDelta` instead of the full object
- `void fromDelta(DataInput in)`—When a delta is received, GemFire calls `fromDelta` to extract the delta from `java.io.DataInput` and apply it to the local copy of the object.

InvalidDeltaException

This exception indicates that an error occurred while processing a delta on the receiving end. When this is thrown, GemFire requests a full value from the member that originated the operation. You can have GemFire send the full object for other exceptions in `fromDelta` by wrapping them in `InvalidDeltaException` in your implementation.

12.4 Delta Propagation Properties

Delta propagation properties can be configured through the API and through the `gemfire.properties` and `cache.xml` files.

delta-propagation

A distributed system boolean, configured in the `gemfire.properties`. When `true`, delta propagation is used. When `false`, full entry values are sent for every update. The default setting is `true`.

Example 12.1 Disabling Delta Propagation in gemfire.properties

```
delta-propagation=false
```

Example 12.2 Disabling Delta Propagation Through the API

```
private DistributedSystem system = null;
Properties props = new Properties();
props.setProperty("name", "cacheRunner");
props.setProperty("delta-propagation", "false");
system = DistributedSystem.connect(props);
```

cloning-enabled

A region attributes boolean, configured in the `cache.xml`, that affects how `fromDelta` applies deltas to the local cache. When `true`, the updates are applied to a clone of the value and then the clone is saved to the cache. When false, the value is modified in place in the cache. The default value is false.

Exceptions:

- If the `Cache` attribute `copy-on-read` is true, cloning is enabled, regardless of what this attribute is set to.
- Servers running continuous queries (CQs) must always clone the existing value and apply the received delta on it, in order to have old and new values for the CQ event. Servers not running CQs use this attribute.

Cloning can be expensive, but it ensures that the new object is fully initialized with the delta before any application code sees it.

When cloning is enabled, by default GemFire does a deep copy of the object, using serialization. You may be able to improve performance by implementing `java.lang.Cloneable` and then implementing the `clone` method, making a deep copy of anything to which a delta may be applied. The goal is to significantly reduce the overhead of copying the object while still retaining the isolation needed for your deltas.

Without cloning:

- It is possible for application code to read the entry value as it is being modified, possibly seeing the value in an intermediate, inconsistent state, with just part of the delta applied. You may choose to resolve this issue by having your application code synchronize on reads and writes.
- GemFire loses any reference to the old value because the old value is transformed in place into the new value. Because of this, your `CacheListener` sees the same new value returned for `EntryEvent.getOldValue` and `EntryEvent.getNewValue`.
- Exceptions thrown from `fromDelta` may leave your cache in an inconsistent state. Without cloning, any interruption of the delta application could leave you with some of the fields in your cached object changed and others unchanged. If you do not use cloning, keep this in mind when you program your error handling in your `fromDelta` implementation.

With cloning:

- The `fromDelta` method generates more garbage in the VM and performance is reduced.

Example 12.3 Enabling Cloning in cache.xml

```
<region name="region_with_cloning">
   <region-attributes cloning-enabled="true" scope="distributed-ack">
      <subscription-attributes interest-policy="all"/>
   </region-attributes>
</region>
```

Example 12.4 Enabling Cloning Through the API

```
AttributesFactory factory = new AttributesFactory();
factory.setScope(Scope.DISTRIBUTED_ACK);
factory.setCloningEnabled(true);
RegionAttributes attrs = factory.create();
cache.createRegion(REGION_NAME1, attrs);
```

12.5 Implementing Delta Propagation

By default, delta propagation is enabled in your distributed system. When enabled, delta propagation is used for objects that implement `com.gemstone.gemfire.Delta`. You will program the methods to extract delta information for your entries and to apply received delta information.

1. Choose whether to enable cloning. Cloning is disabled by default. See `cloning-enabled` (page 332).
2. Study your object types and expected application behavior to determine which objects should use delta propagation. Delta propagation is not beneficial for all data and data modification scenarios. For guidance on your decision, see *Performance Considerations and Limitations* on page 337.
3. If you are not enabling cloning, review all associated listener code for dependencies on `EntryEvent.getOldValue`. Without cloning, GemFire modifies the entry in place and so loses its reference to the old value. For delta events, the `EntryEvent` methods `getOldValue` and `getNewValue` both return the new value.
4. For every class where you want delta propagation, implement `com.gemstone.gemfire.Delta` and update your methods to support delta propagation. Exactly how you do this is depends on your application and object needs, but these steps describe the basic approach:

   4.1 If the class is a plain old Java object (POJO), wrap it for this implementation and update your code to work with the wrapper class.

   4.2 Define as transient any extra object fields that you use to manage delta state. This can help performance when the full object is distributed. Whenever standard Java serialization is used, the transient keyword indicates to Java to not serialize the field.

   4.3 Study the object contents to decide how to handle delta changes. Delta propagation has the same issues of distributed concurrency control as the distribution of full objects, but on a more detailed level. Some parts of your objects may be able to change independent of one another while others may always need to change together. Send deltas large enough to keep your data logically consistent. If, for example, field A and field B depend on each other, then your delta distributions should either update both fields or neither. As with regular updates, the fewer producers you have on a data region, the lower your likelihood of concurrency issues.

   4.4 In the application code that puts entries, put the fully populated object into the local cache. Even though you are planning to send only deltas, errors on the receiving end could cause GemFire to request the full object, so you must provide it to the originating `put` method. Do this even in empty producers, with regions configured for no local data storage. This usually means doing a `get` on the entry, unless you are sure it does not already exist anywhere in the distributed region.

   4.5 Change each field's update method to record information about the update. The information must be sufficient for `toDelta` to encode the delta and any additional required delta information when it is invoked.

   4.6 Write `hasDelta` to report on whether a delta is available.

   4.7 Write `toDelta` to create a byte stream with the changes to the object and any other information `fromDelta` will need to apply the changes. Before returning from `toDelta`, reset your delta state to indicate that there are no delta changes waiting to be sent.

   4.8 Write `fromDelta` to decode the byte stream that `toDelta` creates and update the object.

   4.9 Make sure you provide adequate synchronization to your object to maintain a consistent object state. If you do not use cloning, you will probably need to synchronize on reads and writes to avoid reading partially written updates from the cache.This might involve `toDelta`, `fromDelta`, `toData`, `fromData`, and other object access and update methods. Additionally,

your implementation should take into account the possibility of concurrent invocations of `fromDelta` and one or more of the object's update methods.

12.6 Errors in Delta Propagation

Errors in delta propagation fall into two categories based on how they are handled:

- Problems applying the delta that can be remedied by having the originating member send the full value in place of the delta. Your `put` operation does not see errors or exceptions related to the failed delta propagation. The system automatically does a full value distribution to the receiver where the problem occurs. This type of error includes:
  - Unavailable entry value in the receiving cache, either because the entry is missing or its value is null. In both cases, there is nothing to apply the delta to and the full value must be sent. This is most likely to occur if you destroy or invalidate your entries locally, either through application calls or through configured actions like eviction or entry expiration.
  - `InvalidDeltaException` thrown by `fromDelta` method, programmed by you. This exception enables you to avoid applying deltas that would violate data consistency checks or other application requirements.
  - Any error applying the delta in a client in server-to-client propagation. The client logs a warning in addition to retrieving the full value from the server.
- Problems creating or distributing the delta that cannot be fixed by distributing the full value. In these cases, your `put` operation fails with an exception. This type of error includes:
  - Error or exception in `hasDelta` or `toDelta`.
  - Error or exception in a server or peer receiver that fall outside of the error situations described above in the first category.

12.7 Performance Considerations and Limitations

The main purpose of delta propagation is to reduce the overhead of sending large amounts of data over the network. If you do not use cloning when applying the deltas, you can also expect to generate less garbage in your receiving VMs.

Generally, the larger your objects and the smaller the deltas, the greater the benefits of using delta propagation. Partitioned regions generally benefit more with higher redundancy levels.

Delta propagation does not show any significant benefits in some application scenarios. On occasion it results in degradation. The main areas of consideration are:

- Added costs of deserializing your objects to apply deltas. Applying a delta requires the entry value to be deserialized. Once this is done, the object is stored back in the cache in deserialized form. This aspect of delta propagation only negatively impacts your system if your objects are not already being deserialized for other reasons, such as for indexing and querying or for listener operations. Once stored in deserialized form, there are reserialization costs for operations that send the object outside of the VM, like distribution across gateways, values sent in response to `netSearch` or client requests, and storage to disk. The more operations that require reserialization, the higher the overhead of deserializing the object. As with all serialization efforts, you can improve performance in serialization and deserialization by providing custom implementations of `DataSerializable` for your objects.
- Cloning. Using cloning can affect performance and generates extra garbage. Not using cloning is risky, however, as you are modifying cached values in place. Make sure you synchronize your entry access to keep your cache from becoming inconsistent.
- Problems applying the delta that cause the system to go back to the originator for the full entry value. When this happens, the overall operation costs more than sending the full entry value in the first place. This can be additionally aggravated if your delta is sent to a number of recipients, all or most of them request a full value, and the full value send requires the object to be serialized.
- Disk I/O costs associated with overflow regions. If you use eviction with overflow to disk, on-disk values must be brought into memory in order to apply the delta. This is much more costly than removing the reference to the disk copy, as you would do with a full value distribution into the cache.

12.8 Example of Delta Propagation

In this example, the feeder client is connected to the first server, and the receiver client is connected to the second. The servers are peers to each other.

Figure 12.2 Example of Delta Propagation in the Client/Server

These are the main operations performed in the example:

1. In the Feeder client, the application updates the entry object and puts the entry. In response to the `put`, GemFire calls `hasDelta`, which returns true, so GemFire calls `toDelta` and forwards the extracted delta to the server. If `hasDelta` returned false, GemFire would distribute the full entry value.
2. In Server1, GemFire applies the delta to the cache, distributes the received delta to the server's peers, and forwards it to any other clients with interest in the entry (there are no other clients to Server1 in this example)
3. In Server2, GemFire applies the delta to the cache and forwards it to its interested clients, which in this case is just Receiver client.

This example shows the basic approach to programming a `Delta` implementation.

Example 12.5 Simple Delta Example Implementation

```
package delta;

import com.gemstone.gemfire.Delta;
import com.gemstone.gemfire.InvalidDeltaException;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.io.Serializable;

/**
 * Sample implementation of Delta
 *
 * @author GemStone Systems, Inc.
 * @since 6.1
 */
public class SimpleDelta implements Delta, Serializable {

// Original object fields
   private int intVal;
   private double doubleVal;

   // Added for delta - one boolean per field to track changed status
   private transient boolean intFldChd = false;
   private transient boolean dblFldChd = false;

   public SimpleDelta(){}

   public SimpleDelta(int i, double d){
      this.intVal = i;
      this.doubleVal = d;
   }

   public boolean hasDelta() {
      return this.intFldChd || this.dblFldChd;
   }

   public void toDelta(DataOutput out) throws IOException {
      System.out.println("Extracting delta from " + this.toString());
   // Write information on what has changed to the
   // data stream, so fromDelta knows what it's getting
      out.writeBoolean(intFldChd);
      if (intFldChd) {
   // Write just the changes into the data stream
         out.writeInt(this.intVal);
   // Once the delta information is written, reset the delta status field
         this.intFldChd = false;
         System.out.println(" Extracted delta from field 'intVal' = "
                                    + this.intVal);
      }
      out.writeBoolean(dblFldChd);
      if (dblFldChd) {
         out.writeDouble(this.doubleVal);
         this.dblFldChd = false;
         System.out.println(" Extracted delta from field 'doubleVal' = "
                                    + this.doubleVal);
```

```
            }
        }

        public void fromDelta(DataInput in) throws IOException,
    InvalidDeltaException {
            System.out.println("Applying delta to " + this.toString());
            // For each field, read whether there is a change
            if (in.readBoolean()) {
                // Read the change and apply it to the object
                this.intVal = in.readInt();
                System.out.println(" Applied delta to field 'intVal' = "
                          + this.intVal);
            }
            if (in.readBoolean()) {
                this.doubleVal = in.readDouble();
                System.out.println(" Applied delta to field 'doubleVal' = "
                          + this.doubleVal);
            }
        }
    // In the setter methods, add setting of delta-related
    // fields indicating what has changed
        public void setIntVal(int anIntVal) {
            this.intFldChd = true;
            this.intVal = anIntVal;
        }

        public void setDoubleVal(double aDoubleVal) {
            this.dblFldChd = true;
            this.doubleVal = aDoubleVal;
        }

        public String toString() {
            return "SimpleDelta [ hasDelta = " + hasDelta() + ", intVal = " +
                   this.intVal + ", doubleVal = {" + this.doubleVal + "} ]";
        }
    }
```

Chapter 13

Querying and Indexing

You can use the GemFire Enterprise® Caching API for Java to query your cached data. The query language supported in GemFire is essentially a subset of OQL (ODMG 3.0 Object Data Management Group, http://www.odmg.org). OQL is an SQL-like language with extended functionality for querying complex objects, object attributes and methods.

The GemFire Enterprise query language supports the complete ASCII character set and the full Unicode character set. Queries can also be optimized by defining indexes. The API is provided in `com.gemstone.gemfire.cache.query`.

The querying capabilities described here are used both for normal querying, covered in this chapter in detail, and as a part of the continuous querying functionality. In normal querying, each query is run once against all data currently available in the local cache. Continuous querying (CQ) is used in client server installations, and is run by the client against the server's cache. The CQ includes a query which is run by the GemFire Enterprise querying engine. Unlike the normal query, however, the CQ runs continuously against new server events. Events that satisfy the CQ are passed on to the client along with information about the query operation. For more information on continuous querying, see Chapter 14, *Continuous Querying*, on page 397.

In this chapter:

- Executing a Query (page 342)
- Example Data and Example Queries (page 343)
- Querying Object Data (page 347)
- How Querying Interacts with the Distributed Cache (page 350)
- The Query Language (page 352)
- Indexes (page 375)
- Querying Partitioned Regions (page 384)
- Performance Considerations (page 386)
- The API Interfaces and Classes (page 387)
- Programming Examples (page 389)

This chapter assumes general familiarity with SQL querying and indexing and with the information on the GemFire Enterprise cache provided in the previous chapters of this guide.

13.1 Executing a Query

You can execute a query in your member's distributed system or, for a client, in the server system.

1. Get the `QueryService` using the appropriate method for your cache type and the tier you want to query.

   Examples:

   ```
   // Query this peer/server cache and its peers
      QueryService qryService = cacheInstance.getQueryService();

   // Query this client cache
      QueryService qryService = clientCacheInstance.getLocalQueryService();

   // Query this client's server caches - assumes only one Pool is defined
      QueryService qryService = clientCacheInstance.getQueryService();

   // Query this client's server caches for the specified Pool
      QueryService qryService =
         clientCacheInstance.getQueryService(poolName);
   ```

2. Create a `Query` that is compatible with the OQL specification. Example:

   ```
   String queryString =
      "SELECT DISTINCT * FROM /portfolios "
      + "WHERE status = 'active' AND \"type\" = 'XYZ'";
   Query query = qryService.newQuery(queryString);
   ```

3. Use the `execute` method for the `Query` interface to submit the query string to the cache server.

   ```
   SelectResults results = (SelectResults)query.execute();
   ```

   The system evaluates the query string and returns the results.

4. If appropriate for your query, you can iterate through the returned objects as part of the query process. Example:

   ```
   SelectResults results = (SelectResults)query.execute();
   for (Iterator iter = results.iterator(); iter.hasNext(); )
   {
      Portfolio activeXYZPortfolio = (Portfolio) iter.next();
      ...
   }
   ```

The examples in this chapter all assume that you have already obtained the appropriate `QueryService`.

13.2 Example Data and Example Queries

This section introduces the data that is used in most of the querying examples in this guide and lists some example queries and their results.

The Example Data Used in the Queries

Most of the querying examples use the region of stock portfolios, `/portfolios`, whose keys are the portfolio ID and whose values contain the following data:

Example 13.1 Entry Value Structure for Region portfolios

```
Class Portfolio {
   String id; /* This id is used as the entry key and is stored here */
   String type;
   String status;
   Map positions; /* The value is the Position, with secId used as the key*/
}
Class Position {
   private String secId;
   private double mktValue;
   private double qty;
}
```

All queries that use the `/portfolios` *region example data assume that the* `/portfolios` *region has the* `value-constraint` *attribute set to* `cacheRunner.Portfolio`*. For information, see* *Specifying the Object Types of FROM Clause Collections* on page 361.

This is the sample data in the `/portfolios` region:

Table 13.1 Entry Values in Region portfolios

id (same as entry key)	type	status	positions		
			secId	mktValue	qty
XYZ-1	xyz	active	XXX	27.34	1000.00
			XXY	26.31	1200.00
			XXZ	24.30	1500.00
XYZ-2	xyz	active	YYY	18.29	5000.00
ABC-1	abc	active	AAA	24.30	10.00
			AAB	23.10	15.00
ABC-2	abc	inactive	BBB	50.41	100.00
			BBC	55.00	90.00

Querying the portfolios Region

This is a sampling of the queries that you could run against `/portfolios`, and the query results for the data in the previous table. For the first several, the coding examples are included as well to show how you would run the queries through the API.

Example 13.2 Retrieve All Portfolios That Are Active

Query string

```
SELECT DISTINCT * FROM /portfolios
WHERE status = 'active'
```

Results: Collection of `Portfolio` instances for IDs: `XYZ-1`, `XYZ-2`, `ABC-1`

Code

```
Region region = ...
SelectResults results = region.query("status = 'active'");
for (Iterator iter = results.iterator(); iter.hasNext(); )
{
    Portfolio activePortfolio = (Portfolio) iter.next();
    ...
}
```

Notes: This example code uses a `Region` query shortcut method. Its query string retrieves all active portfolios.

Example 13.3 Retrieve All Portfolios That Are Active and Have Type XYZ

Query string

```
SELECT DISTINCT * FROM /portfolios
WHERE status = 'active' AND "type" = 'XYZ'
```

Results: Collection of `Portfolio` instances for ids: `XYZ-1`, `XYZ-2`

Code

```
String queryString =
    "SELECT DISTINCT * FROM /portfolios "
    + "WHERE status = 'active' AND \"type\" = 'XYZ'";
Query query = qryService.newQuery(queryString);
SelectResults results = (SelectResults)query.execute();
for (Iterator iter = results.iterator(); iter.hasNext(); )
{
    Portfolio activeXYZPortfolio = (Portfolio) iter.next();
    ...
}
```

Notes: This code uses the GemFire query service, but could use a `Region.query` shortcut instead, as shown in Example 13.2. The `type` attribute is passed to the query engine in double quotes to distinguish it from the query keyword of the same name. For information on reserved keywords, see Appendix C, *Query Language Grammar and Reserved Words*, on page 523.

Example 13.4 Get the ID and Status of All Portfolios with Positions in secId 'YYY'

Query string

```
IMPORT cacheRunner.Position;
SELECT DISTINCT id, status FROM /portfolios
WHERE NOT (SELECT DISTINCT * FROM positions.values posnVal TYPE
Position WHERE posnVal.secId='YYY').isEmpty
```

Results `Collection` of `Struct` instances each containing an `id` field and a `status` field. For this data, the `Collection` length is 1, and the `Struct` contains data from the entry with id `XYZ-2`, like this: `[struct(id:XYZ-2,status:active)]`

Code

```
String queryString =
   "SELECT DISTINCT id, status FROM /portfolios "
   + "WHERE NOT "
   + "(SELECT DISTINCT * FROM positions.values posnVal TYPE
Position"
         + "WHERE posnVal.secId='YYY').isEmpty ";
Query query = qryService.newQuery(queryString);
SelectResults results = (SelectResults)query.execute();
for (Iterator iter = results.iterator(); iter.hasNext(); )
{
   Struct struct = (Struct) iter.next();
   int id = ((Integer) struct.get("id")).intValue();
   String status = (String) struct.get("status");
   ...
}
```

Notes This query uses the `TYPE` keyword to specify the class associated with `positions.values`. For more information on the use of `TYPE`, see *Specifying the Object Types of FROM Clause Collections* on page 361.

The inner `SELECT` statement (on `positions`) is performed in its own context, or scope, with access to the outer scope owned by the outer `SELECT` statement (on `/portfolios`). The `FROM` in the outer `SELECT` creates a collection of `/portfolios` entry values. For each iteration of this outer collection, the `FROM` clause of inner `SELECT`, creates a collection of `Map` values. Thus for every outer `SELECT` iteration, the inner `SELECT` iterates over the complete `Collection` created.

Example 13.5 Get Distinct Positions From All Active Portfolios with at Least a $25.00 Market Value

Query string

```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal
FROM /portfolios qryP, qryP.positions.values posnVal TYPE Position
WHERE qryP.status = 'active' AND posnVal.mktValue >= 25.00
```

Results Collection of `Position` instances with `secId`: `XXX`, `XXY`

Notes This query assigns iterator variable names to the collections in the `FROM` clause. For example, the variable `qryP` is the iterator for the entry values in the `/portfolios` region. This variable is used in the second part of the `FROM` clause to access the values of the `positions` `Map` for each entry value.

Example 13.6 Get Distinct Entry Keys and Positions from Active Portfolios with at Least a $25.00 Market Value

Query string

```
IMPORT cacheRunner.Position;
SELECT DISTINCT key, posnVal
FROM /portfolios.entrySet, value.positions.values posnVal TYPE
Position
WHERE posnVal.mktValue >= 25.00
```

Results

```
A SelectResults of Struct instances containing key, Position pairs
[struct(key:XYZ-1,posnVal:Position [secId=XXX mktValue=27.34 qty=
1000.00]),
struct(key:XYZ-1,posnVal:Position [secId=XXY mktValue=26.31 qty=
1200.00])]
```

Notes

Retrieving the entry keys allows you to access the cached region entries if you need to update them. For more information, see *Modifying Your Data* on page 351.

13.3 Querying Object Data

This section compares object data querying with querying classic relational data and provides a graphic view of the changes in data visibility as a GemFire cache query is run. For information about how querying interacts with your cache configuration and your application data, see *How Querying Interacts with the Distributed Cache* on page 350.

Querying Object Data Versus Relational Data

Object data is generally nested and, to access the nested data, you drill down through the outer data layers. Querying nested data also requires a drill-down approach. Data that is not "on the surface" must be brought into scope inside the query before it can be accessed. This requires a special approach to querying that is slightly different from classic SQL querying.

The query package allows you to access cached regions and region data with SQL-style querying. If you have ever queried database tables using SQL, many of the query package concepts will be familiar to you. There are differences, however, between the relational database table storage model and GemFire's object storage model that translate into subtle but important differences in querying techniques. This section provides a comparison of the two storage models and their querying techniques.

The comparison uses the portfolios example from the prior section. Each portfolio has an ID, a status, a type, and a map of associated positions. Each position represents a secId, a market value, and a quantity. In a database you would store this information in a portfolio table and a position table, connecting the two tables through primary and foreign keys. In a cache region, the position data would be stored as a map inside the portfolio data. These are the data layouts followed by the models with some of the data from Table 13.1:

Figure 13.1 Storage Model Comparison: Database and Cache

Figure 13.2 Database and Cache Storage Models With Data

As you can see, the classic database storage model is flatter, with each table standing on its own. You can access position data independent of the portfolio data, or you can join the two tables if you need to by using multiple expressions in your `FROM` clause. On the other hand, with the object model you use multiple expressions in the `FROM` clause to drill down into the portfolio table and access the position data. This can be helpful, as the position-to-portfolio relationship is implicit and does not require restating in your queries. It may also hinder you when you want to ask general questions about the positions data, independent of the portfolio information. The two queries below illustrate these differences.

To list the market values for all positions of active portfolios, you query the two data models with these statements:

Database Table Query 1	Cache Query 1

`SELECT DISTINCT mktValue`
`  FROM portfolio, position`
`  WHERE status = "active"`
`  AND portfolio.id =`
`position.id`
	`IMPORT cacheRunner.Position;`
	`SELECT DISTINCT posnVal.mktValue`
	`  FROM /portfolios,`
	`  positions.values posnVal TYPE Position`
	`  WHERE status='active'`

The difference between the queries reflects the difference in data storage models. The database query must explicitly join the portfolio and position data in the `WHERE` clause, by matching the `position` table's foreign key, `id`, with the `portfolio` table's primary key. In the object model, this one-to-many relationship is implicitly specified by defining the `positions Map` as a field of the `portfolio` class.

This difference is reflected again when you query the full list of positions market values.

Database Table Query 2	Cache Query 2

`SELECT DISTINCT mktValue`
`  FROM position`
	`IMPORT cacheRunner.Position;`
	`SELECT DISTINCT posnVal.mktValue`
	`  FROM /portfolios,`
	`  positions.values posnVal TYPE Position`

The `position` table is independent of the `portfolio` table, so your database query runs through the single table. The cached position data, however, is only accessible via the `portfolio` objects. The cache query aggregates data from each `portfolios`'s individual `positions Map` to create the result set.

For the cache query engine, the `positions` *data only becomes visible when the first expression in the* `FROM` *clause,* `/portfolios`*, is evaluated.*

Data Accessibility in a GemFire Cache Query

Before a query is run, the only cached data that is directly accessible—in scope—are regions and region objects and methods. When a region or one of its objects or methods is referenced, the data returned by the reference comes into scope and can itself be referenced. For example, a region's keys and entries can be brought into scope by referencing `/portfolios.keySet` and `/portfolios.entrySet` in the `FROM` clause. These evaluate, respectively, to the collection of region keys and the collection of region `Entry` objects. The expression, `/portfolios`, evaluates by default to `/portfolios.values`, the collection of region entry values. This is used in the example query below.

This section shows the `/portfolios` data that is accessible as each part of a cache query is evaluated (the query used is the first one from the prior examples). For each figure, the part of the query that has been evaluated is in **bold**. The first figure shows the data that is in scope before the query is run. Note that the region fields and methods are visible.

Figure 13.3 Data in Scope Before the Query

```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal.mktValue
   FROM /portfolios,
   positions.values posnVal TYPE Position
   WHERE status='active'
```

Figure 13.4 Data in Scope After First FROM Expression is Evaluated

```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal.mktValue
   FROM /portfolios,
   positions.values posnVal TYPE Position
   WHERE status='active'
```

Figure 13.5 Data in Scope After Second FROM Expression is Evaluated

```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal.mktValue
   FROM /portfolios,
   positions.values posnVal TYPE Position
   WHERE status='active'
```

In cache querying, the `FROM` expression brings new data into the query scope. These concepts are discussed further in *Attribute Visibility* on page 354.

13.4 How Querying Interacts with the Distributed Cache

This section describes requirements and caveats in querying your data in the GemFire Enterprise cache.

Creating Queryable Objects

Any cached object that you intend to query must implement the `equals` and `hashCode` methods, and the methods must conform to the properties and behavior documented for `equals` and `hashCode` in the online Java API documentation for `java.lang.Object`. Inconsistent query results may occur if these methods are absent.

Maintaining Consistent, Queryable Data

The objects you query in your regions must all be of the same type. You will get exceptions if you run queries or keep indexes on regions that do not have homogeneous data. You can, however, use different subtypes.

To ensure that your data remains consistent for your queries, use the `key-constraint` (page 111) and `value-constraint` (page 119) region attributes to restrict the types of keys and values you allow in the region. Do this everywhere you work with the region. For example, in a client/server installation, if you are populating server regions from the client side, make sure to define the same constraints for the regions in the client caches as you do in the server caches.

Finding Your Cached Data

The querying engine does not run `netSearch` (page 136) to retrieve its results. Depending on your region configuration and how your applications put data into the region, this may affect how much of your data is visible to the query.

Partitioned Regions

All data is a partitioned region is available to be queried from any member where the partitioned region is defined. Partitioned region data is available to the local cache without the use of `netSearch`, so the `netSearch` limitation has no effect.

Other Regions

For all region types except partitioned, querying and indexing only work with the data that is available in the local cache. What data is available depends on the region's configuration. In particular, see the `refid` (page 116), `scope` (page 116), and `data-policy` (page 104) attribute settings.

This table summarizes the effects of region scope and data policy settings on the data available to your querying and indexing operations.

Table 13.2 The Effects of Region Scope and Data Policy on the Data Available for Querying and Indexing

Scope	Data Policy	
	Not `replicate`	`replicate` or `persistent-replicate`
`local`	FULL data set	FULL data set
`distributed-ack` or `distributed-no-ack`	Data set may not be complete	FULL data set (if no race conditions)
`global`	Data set may not be complete	FULL data set

A region with `local scope` is completely contained in the local cache, so any access to the local region accesses the entire region data set.

A distributed, replicated region reflects the state of the entire distributed region. Without replication, there may be entries in remote caches that are not in the local cache.

> *To ensure a complete data set for your queries and indexes, your region must be local and/or replicate.*

If you use a replicate `data-policy`, the non-global distributed scopes `distributed-ack` and `distributed-no-ack` can encounter race conditions during entry distribution that cause the local data set to be out of sync with the distributed region. A `global` scope guarantees data consistency across the distributed system, but at the cost of reduced performance. For a discussion, see *Choosing the Level of Distribution* on page 204.

Modifying Your Data

The query service is a data access tool. It does not provide any functionality for updating the cache. If you need to modify the cache based on information retrieved through querying, you need to retrieve the entry keys and use them in the standard entry update methods. For an example of entry key retrieval, see Example 13.6 on page 346.

Querying does not add data to your cache—it does not perform any loading or distribution activities.

Additional Information for Partitioned Region Queries

There are some restrictions that apply to querying in partitioned regions. You cannot do joins or cross-region queries between a partitioned region and any other region, partitioned or not. Additionally, there are limitations on query construction. See *Querying Partitioned Regions* on page 384 for more details about partitioned region queries.

Indexes in Overflow Regions

Indexes are not allowed on overflow regions. If you try to create an index on an overflow region, the operation throws an `UnsupportedOperationException`. It is also not possible to have your indexes overflow to disk.

13.5 The Query Language

The query language supported in GemFire is essentially a subset of OQL (ODMG 3.0 Object Data Management Group, http://www.odmg.org), which in turn is based on SQL-92. The query language provides the basic set of clauses, expressions, and operators used to retrieve elements from objects stored in cache regions. This section describes the operators, keywords, and operations that the query language supports.

> *Query language keywords such as* `SELECT`, `DATE`, *and* `NULL`, *are case-insensitive. Identifiers in query strings, such as attribute names, method names, and path expressions, are case-sensitive. So the attribute* `empId` *will only get mapped to the field,* `empId`, *or the methods,* `getEmpId()` *or* `empId()`. *It will not get mapped to the method* `EMPID()`. *Query language keywords are capitalized in the documentation for easy identification.*

This section covers the following:

- Accessing Cached Data (page 352)
- What is a Query String? (page 356)
- SELECT Statement Semantics (page 357)
- The FROM Clause (page 358)
- Helping the Query Engine Resolve Your Query's Attributes (page 360)
- The WHERE Clause (page 363)
- Joins (page 364)
- The SELECT Projection List (page 364)
- SELECT Statement Query Results (page 365)
- Additional Query Language Elements (page 367)

Accessing Cached Data

Accessing your cached data through the querying service is similar to how you would access database contents through SQL queries, but with some differences. How you specify your regions and region contents is particular to GemFire. The query language in GemFire supports drilling down into nested object structures. Regions can contain nested data collections that are unavailable until referenced in the `FROM` clause. For a general discussion of this, see *Querying Object Data* on page 347. This section discusses how to navigate to your cached data through the GemFire query service.

> *Querying and indexing only operate on local cache contents. For a discussion of this, see* *How Querying Interacts with the Distributed Cache* on page 350.

Basic Region Access

In the context of a query, the name of a region is specified by its name, preceded by a forward slash (/).

> *To be used for querying, region names in a region path are restricted to alphanumeric and underscore characters only. For all information on region naming requirements, see* *Region Naming Conventions* on page 92.

Region Querying Attributes

From a region path, you can access the `Region` object's public fields and methods, referred to as the region's attributes. Thus, `/portfolios.name` returns "`portfolios`" and `/portfolios.name.length` returns `10`. An attribute is mapped to a Java class member in three possible ways with the following priority until a match is found. If the attribute is named `x`, then:

```
public method getX()
public method x()
public field x
```

> *The term* attribute *in this context is not the same as a region attribute.*

Region Data

You can also access entry keys and entry data through the region path:

- `/portfolios.keySet` returns the `Set` of entry keys in the region
- `/portfolios.entrySet` returns the `Set` of `Region.Entry` objects
- `/portfolios.values` returns the `Collection` of entry values
- `/portfolios` returns the `Collection` of entry values

Note the last two bullets. In the FROM clause, `/portfolios.values` and `/portfolios` return the same thing.

> *These* `Collection`*s are immutable. Invoking modifier methods on them, such as* `add` *and* `remove`*, result in an* `UnsupportedOperationException`*.*

Region Object Representation

In GemFire, when a region path evaluates to a `Region` in a query, the type of object it refers to has the following class:

```
class QRegion
extends com.gemstone.gemfire.cache.Region, java.util.Collection {
}
```

`QRegion` contains a reference to the `Region` object and implements all the methods of the `Region` interface. The `QRegion` delegates the invocation on these methods to the underlying `Region` object. Some of the operations on `QRegion` may throw `UnsupportedOperationException` if they modify data.

Thus, the class inherits all the attributes and methods from both `Region` and `Collection`. When used as a `Collection`, the elements are the values stored in the region. This enables a query to use a region either as a collection of its values or as the actual region. The specific use in any instance depends on context. For example, in this query the `/portfolios` path is used as the `Collection` of portfolios in the region when you refer to `positions`.

```
SELECT DISTINCT positions FROM /portfolios
```

The same path specification refers to the `Region` and therefore can be used to access attributes like `keySet` that a `Region` understands, as in the following query.

```
SELECT DISTINCT * FROM /portfolios.keySet
```

Drilling Down: Modifying Query Scope

The query engine resolves names and path expressions according to the name space that is currently in scope in the query. (This is not the region `scope` attribute, but the scope of the query statement.) The initial name space for any query is composed of the region paths of the local cache and the attributes of

those paths. New name spaces are brought into scope based on the FROM clause in the SELECT statement. In this query, for example, the FROM expression evaluates to the collection of entry values in /portfolios. The attributes of these values are added to the initial scope of the query and status is resolved within the new scope.

```
SELECT DISTINCT *
   FROM /portfolios
   WHERE status = 'active'
```

Each FROM clause expression must resolve to a collection of objects available for iteration in the query expressions that follow. In the example above, /portfolios resolves to the Collection of entry values in the region. The entry value collection is iterated by the WHERE clause, comparing the status field to the string 'active'. When a match is found, the value object is added to the return set.

In the following distinct query, the collection specified in the first FROM clause expression is used by the rest of the SELECT statement.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
   FROM /portfolios, positions.values posnVal TYPE Position
   WHERE posnVal.qty > 1000.00
```

You could not change the order of the expressions in this FROM *clause; the second expression depends on the scope created by the first expression.*

Attribute Visibility

Within the current scope of a query, you can access any available object or object attribute. In querying, an object's attribute is any identifier that can be mapped to a public field or method in the object. In the FROM specification, any object that is in scope is valid, so at the beginning of a query, all locally cached regions and their attributes are in scope.

This query is valid because name resolves to the Region method getName.

```
/portfolios.name
```

This query is valid because toArray resolves to the Collection method with the same name.

```
SELECT DISTINCT * FROM /portfolios.toArray
```

You cannot, however, refer to the attribute of a collection object in the region path expression where the collection itself is specified. The following statement is invalid because neither Collection nor Region contain an attribute named positions. The entry values collection (specified by /portfolios) that *does* contain an attribute named positions is not yet part of the query name space.

```
/* INCORRECT: positions is not a querying attribute of Region or of
Collection */
   SELECT DISTINCT * FROM /portfolios.positions
```

This following SELECT statement is valid because positions is an element of the entry value collection that is specified by /portfolios. The entry value collection is in scope as soon as the specification in the FROM expression is complete (before WHERE or SELECT are evaluated).

```
SELECT DISTINCT positions FROM /portfolios
```

You can also refer to positions inside the FROM clause after the /portfolios entry value collection is created. In this example, positions is an element of the /portfolios entry value collection and values is an attribute of positions:

```
IMPORT cacheRunner.Position;
SELECT DISTINCT posnVal
   FROM /portfolios, positions.values posnVal TYPE Position
   WHERE posnVal.mktValue >= 25.00
```

After the comma in the `FROM` clause, `/portfolios` is in scope, so its value collection can be iterated. In this case, this is done with the second `FROM` clause specification, `positions.values`. This is discussed further in *SELECT Statement Semantics* on page 357.

Nested Query Scopes

Scopes can be nested using nested `SELECT` statements, and names in an inner scope hide identical names in an outer scope. In the query below, the inner `SELECT` creates a new scope, the `positions` of the current portfolio, inside the outer `SELECT`'s scope, `/portfolios`. This inner scope (the collection of entry values from the `/portfolios` region) is first searched for the `secId` element. The outer scope is searched only if the `secId` element is not found in the inner scope.

Example 13.7 Nested Query

```
IMPORT cacheRunner.Position;
SELECT DISTINCT * FROM /portfolios
   WHERE NOT
   (SELECT DISTINCT * FROM positions.values TYPE Position
      WHERE secId='YYY').isEmpty
```

This statement shows the outer scope in bold. The outer scope has all the attributes of a `Portfolio` in it.

Example 13.8 Outer Scope of a Nested Query

```
IMPORT cacheRunner.Position;
SELECT DISTINCT * FROM /portfolios
   WHERE NOT
   (SELECT DISTINCT * FROM positions.values TYPE Position
      WHERE secId='YYY').isEmpty
```

Now the statement with the inner scope in bold. The inner scope has all the attributes of a `Portfolio` in it (inherited from the outer scope) and all the attributes of a `Position` as well.

Example 13.9 Inner Scope of a Nested Query

```
IMPORT cacheRunner.Position;
SELECT DISTINCT * FROM /portfolios
   WHERE NOT
   (SELECT DISTINCT * FROM positions.values TYPE Position
      WHERE secId='YYY').isEmpty
```

When Names Cannot Be Resolved

When a query is executed, if a name or path expression resolves to more than one object in the scope, an `NameResolutionException` is thrown. If it cannot be resolved at all, a `NameNotFoundException` is thrown.

What is a Query String?

A query string is a fully-formed OQL statement that can be passed to a query engine and executed against a data set. When you use one of the GemFire methods to create queries, you pass in a query string as an argument. To build a query string, you combine supported keywords, expressions, and operators to create an expression that returns the information that you require.

A query string expression follows the rules specified by the query language grammar. A query expression can contain:

- Path expressions
- Attribute names
- Method invocations
- Operators
- Literals
- Query parameters
- The functions `IS_DEFINED`, `IS_UNDEFINED`, `ELEMENT`, `NVL` and `TO_DATE`
- `SELECT` statements

None of these elements, however, is required in a query string. For example, a `SELECT` statement is generally thought of as the starting point for a query, but it is not essential. In the example region, `/portfolios`, the following expressions are valid queries:

`/portfolios.size > 100`	returns `true` if the region `portfolios` has more than 100 values in it
`/portfolios[98].id = 12`	returns `true` if the portfolio with key 98 in region `/portfolios` has id 12, `false` if it doesn't, and `UNDEFINED` if the portfolio for that key doesn't exist

Further, queries do not need to access any data source. These expressions are all valid queries that do not require any region in the cache:

`NULL`	returns `null`
`42`	returns `42`
`42=59`	returns `false`

Query Strings in GemFire

To use a query string in a GemFire Java program, specify the string as a parameter in a `QueryService.newQuery` method, then execute the query using `Query.execute`, passing in the required parameters. Alternatively, if an expression evaluates to a `boolean` value, you can specify it using the `Region` shortcut methods, `existsValue`, `selectValue`, and `query`.

> *If your query requires any* `IMPORT` *statements, these must be included before the* `SELECT` *statement in the query string that is passed to the query engine. For more information on* `IMPORT`*, see* *Importing Object Classes and Typing or Typecasting Your Collections* on page 361.

The sections that follow discuss query expression elements in greater detail.

SELECT Statement Semantics

The basic format of a SELECT statement is

```
SELECT DISTINCT projectionList FROM collection1, [collection2, ...]
[WHERE clause]
```

The `DISTINCT` *keyword is required in this version of GemFire.*

In the `SELECT` statement, the `FROM` clause data is input to the `WHERE` clause, and the `WHERE` clause filters the data before passing it to the `SELECT` projection list. The `SELECT` statement output is the output from the projection list operations. This is similar to SQL for database access, where the `FROM` clause specifies the data tables, the `WHERE` clause specifies the rows, and the `SELECT` clause specifies the columns.

These are the semantics of the GemFire query `SELECT` statement:

- The `FROM` clause specifies the cached data to be accessed. Each expression in this clause must evaluate to a collection, which is a group of distinct objects of homogeneous type. There must be at least one collection specified. The collections specified in the `FROM` clause are iterated over by the rest of the query—including the rest of the `FROM` clause.
- The `WHERE` clause specifies which of the elements in the `FROM` clause collections are to be passed on to the `SELECT` clause—the rest are dropped. The `WHERE` clause specifications are called "`WHERE` search criteria." The `WHERE` clause is optional. Without it, all elements of the collections are added to the interim result.
- The projection list in the `SELECT` clause specifies the attributes to retrieve from the result elements and may also perform transformations. The `SELECT` clause projection list can be one or more expressions, or it can be `*` to put all elements in the return set without modification.

In the following query, the `FROM` clause accesses the collection of all entry values in the `/portfolios` region. This is passed to the `WHERE` clause, which searches for entry values whose `status` field is `'active'`. For each of the values found by the `WHERE` clause, the `SELECT` projection retrieves the value of the `type` attribute into the result set.

```
SELECT DISTINCT "type" FROM /portfolios WHERE status = 'active'
```

Query Language Reserved Keywords

In the query above, the `type` attribute is in double quotes to distinguish it from the reserved words of the query language. To access any method, attribute, or named object that has the same name as a query language reserved word, enclose the name within double quotation marks. Here is another example:

```
SELECT DISTINCT * FROM /region1 WHERE emps."select"() < 100000
```

A complete list of the query language reserved words is provided in Appendix C, *Query Language Grammar and Reserved Words*, on page 523.

Select Statement Results

A `SELECT` statement returns a result, so it can be composed with other expressions, like this:

```
(SELECT DISTINCT * FROM /portfolios WHERE status = 'active').size
```

`SELECT` statement results are discussed further in *SELECT Statement Query Results* on page 365.

Because select statements can be embedded in larger expressions, technically, a select statement is more appropriately named a `SELECT` expression.

The FROM Clause

The FROM clause establishes collections of objects that are iterated over by the remainder of the query. The attributes of the objects in these collections are added to the name space scope for the remainder of the FROM clause as well as for the WHERE clause and the SELECT projection list. For a discussion of the effects of the FROM expressions on query scope, see *Drilling Down: Modifying Query Scope* on page 353.

Each FROM clause expression must evaluate to a collection. The expression /portfolios.keySet is valid because it evaluates to a Collection, but /portfolios.name, which evaluates to a String, causes an exception to be thrown.

In the context of a query, a collection includes not just implementations of the java.util.Collection interface, but also:

- region path (evaluates to a Collection of entry values)
- java.Util.Map (evaluates to a Collection of entries)
- array (Object or primitive arrays)

Like the SQL query, which iterates over the tables named in its FROM *clause, the OQL query iterates over the collections established in its* FROM *clause.*

In the following query, positions.values evaluates to a Collection because positions is a Map, and the method values on Map returns a Collection.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
   FROM /portfolios, positions.values posnVal TYPE Position
   WHERE posnVal.qty > 1000.00
```

Every expression in the FROM *clause must evaluate to a* Collection*. For a* Map*, the* values *method returns a* Collection*.*

If positions were a List instead of a Map, this query could be used to retrieve the data:

```
IMPORT cacheRunner.Position;
SELECT DISTINCT "type"
   FROM /portfolios, positions posnVal TYPE Position
   WHERE posnVal.qty >= 1000.00
```

A List *is a* Collection*, so it can be accessed directly or through its* toArray *method.*

For each object type accessed in your FROM *clause, use the method that returns a* Collection *for that object.*

Each expression in the FROM clause can be any expression that evaluates to a Collection. For example, an expression in the FROM clause is typically a path expression that resolves to a region in the cache so that the values in the region become the collection of objects to filter. Other possibilities include a query parameter that is bound to a collection, or a nested SELECT statement.

For example, this is a simple SELECT statement that evaluates to a set of all the entry value objects of the region /portfolios with active status. The collection of entry values provided by the FROM clause is traversed by the WHERE clause, which accesses each element's status attribute for comparison.

```
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```

If the FROM clause has just one expression in it, the result of the clause is the single collection that the expression evaluates to. If the clause has more than one expression in it, the result is a collection of structs that contain a member for each of those collection expressions. For example, if the FROM clause contains

three expressions that evaluate to collections `C1`, `C2`, and `C3`, then the `FROM` clause generates a set of `struct(x1, x2, x3)` where `x1`, `x2`, and `x3` represent nested iterations over the collections specified. If the collections are independent of each other then this struct represents their cartesian product, which contains all the possible combinations of data from the tables.

In this query, the `FROM` clause produces a struct of portfolio, position pairs to be iterated. Each element in the struct contains the portfolio and one of its contained positions.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT * FROM /portfolios, positions TYPE Position
   WHERE qty > 1000.00
```

Helping the Query Engine Resolve Your Query's Attributes

To run successfully, the query engine must be able to access all of the attributes specified in your query. To do this, the engine must be able to do two things:

- Determine the object type of the elements in each collection specified in the `FROM` clause
- Associate each querying attribute used anywhere in the query with one of the collections in the `FROM` clause

The querying engine can do some object type discovery on its own and it can scan for the querying attributes within all of the object types in the `FROM` clause collections. Both of these activities use cycles, however, and the attributes search can fail if a same-named attribute is contained in more than one of the collections.

For the querying engine to process your queries with successfully and at optimal speed, you must:

- Use iterator variables to explicitly associate each querying attribute to a single collection in the `FROM` clause.
- Explicitly specify the object type for each collection in the `FROM` clause.

Using Iterator Variables

For each collection expressed in the `FROM` clause, you should associate an explicit variable and use the variable to access the collection's querying attributes. The variable is added to the current scope and becomes the iterator variable bound to the elements of the collection as they are iterated over. Iterator variables are required if the querying attributes you access in the query could belong to more than one collection in the FROM clause.

> *It is best practice to use iterator variables to avoid ambiguity in your queries. Even if there is no ambiguity at the time you write a query, a change to your cache could introduce ambiguity at a later point and cause the query without iterator variables to to fail.*

In this example, `pflo` and `posnVal` are both explicit iterator variables.

Example 13.10 Query Using Explicit Iterator Variables

```
IMPORT cacheRunner.Position;
SELECT DISTINCT pflo."type", posnVal.qty
   FROM /portfolios pflo, positions.values posnVal TYPE Position
   WHERE pflo.status = 'active' and posnVal.mktValue > 25.00
```

Specifying the Object Types of FROM Clause Collections

To provide object typing information to the query engine, you can specify the types for your `FROM` clause collection elements through region configuration or by providing the object type specifications in the query string. Usually, the best approach is a mix of the two.

When you use the region object constraints and when you type or typecast inside your queries, you tell the query engine the object types of the elements in the collections provided by the `FROM` clause. The specific types of the collections themselves— map or array or some other type—is not relevant to the querying engine. The only thing that matters is the types of the elements contained in the collections.

Using Region Configuration to Specify Object Types

The region configuration settings constrain the key and value types that are stored in the cache and automatically provide typing information to the query engine. This type of object specification works for all of your top-level cached objects and has the added advantage of protecting the homogeneity of the data for querying (see *Maintaining Consistent, Queryable Data* on page 350).

Set the types with the `value-constraint` (page 119) and `key-constraint` (page 111) region attributes for the regions you query. The querying engine refers to these attributes when accessing region contents.

In this example, `/portfolios` resolves to a collection of entry values. This collection is defined by the region's `value-constraint` to have elements of type `cacheRunner.Portfolio`. This query relies on the `value-constraint` to resolve `id`.

Example 13.11 Query Using value-constraint for Object Typing

```
SELECT DISTINCT * FROM /portfolios \
WHERE (id = 'XYZ-1' OR id = 'ABC-1')
```

Importing Object Classes and Typing or Typecasting Your Collections

You must provide explicit typing in your queries for the elements in collections that are nested in your region entries and for entries and keys in regions where you are not able to specify a value constraint or key constraint.

To provide explicit typing, import the objects' class definitions immediately before the `SELECT` statement in your query string and then type or typecast your collections in the `FROM` clause where you first reference them. Typing is a GemFire ease-of-use extension of the standard OQL typecasting syntax.

The `IMPORT` statement allows you to provide Java classes to the query engine. `IMPORT` statements are passed to the engine in the same string as the query that uses them. The `IMPORT` is only required in a query if the class name is used explicitly in the query string. The only place you can have a package name in a query is in an `IMPORT` statement.

These are valid `IMPORT` statements:

```
IMPORT com.myFolder.Portfolio;
IMPORT com.myFolder.Portfolio AS MyPortfolio;
```

The first form of the import statement allows `Portfolio` to be used as the name of the class `com.myFolder.Portfolio`. The second form provides an alternative class name to be used. This is useful when a class name is not unique across packages and classes in a single query. For example:

Example 13.12 Using Imported Classes

```
IMPORT com.commonFolder.Portfolio;
IMPORT com.myFolder.Portfolio AS MyPortfolio;
SELECT DISTINCT mpflo.status FROM /portfolios pflo TYPE Portfolio,
          /myPortfolio mpflo TYPE MyPortfolio,
WHERE pflo.status = 'active' and mpflo.id = pflo.id
```

Any `IMPORT` statements required by a query must be included in the string that you pass to the query engine. Thus, the string to run the previous example is this:

```
"IMPORT com.commonFolder.Portfolio; IMPORT com.myFolder.Portfolio AS
MyPortfolio; SELECT DISTINCT mpflo.status FROM /portfolios pflo TYPE
Portfolio, /mybranch/portfolios mpflo TYPE MyPortfolio, WHERE pflo.status =
'active' and mpflo.id = pflo.id"
```

Common type names do not require an `IMPORT` statement. This table lists the common types defined by the system and the Java types that they represent.

Table 13.3 Predefined Class Types

Type	Java Representation
`short`	`short` or `java.lang.Short`
`long`	`long` or `java.lang.Long`
`int`	`int` or `java.lang.Integer`
`float`	`float` or `java.lang.Float`
`double`	`double` or `java.lang.Double`
`char`	`char` or `java.lang.Character`
`string`	`java.lang.String`
`boolean`	`boolean` or `java.lang.Boolean`
`byte` or `octet`	`byte` or `java.lang.Byte`
`date`	`java.sql.Date`
`time`	`java.sql.Time`
`timestamp`	`java.sql.Timestamp`
`array<`*`type`*`>`	`java.lang.Object[]`
`collection<`*`type`*`>`	`java.util.Collection`
`list<`*`type`*`>`	`java.util.List`
`map<`*`type`*`,`*`type`*`>` or `dictionary<`*`type`*`,`*`type`*`>`	`java.lang.Map`
`set<`*`type`*`>`	`java.util.Set`

The querying engine treats primitives as equivalent to their respective wrapper types.

A query's `TYPE` specification can be an imported type or any of these pre-defined types.

With all object types available to the query engine, connect each `FROM` clause collection to a type by using either the `TYPE` keyword or with typecasting.

This query uses the imported `cacheRunner.Position` class to `TYPE positions.values` as a collection of values:

Example 13.13 Query Using IMPORT and TYPE for Object Typing

```
IMPORT cacheRunner.Position;
SELECT DISTINCT mktValue
   FROM /portfolios, positions.values TYPE Position
   WHERE mktValue > 25.00
```

Everything in these examples, including the `IMPORT` *statement, is passed to the query engine to run the query.*

This query is equivalent to the query in the previous example.

Example 13.14 Query Using IMPORT and Typecasting for Object Typing

```
IMPORT cacheRunner.Position;
SELECT DISTINCT mktValue
   FROM /portfolios, (collection<Position>)positions.values
   WHERE mktValue > 25.00
```

In the next example, the `FROM` clause collections are a collection of `Portfolio` instances and a collection of `Map.Entry`s whose values are instances of `Positions`. When `Map`s are used as collections, they are treated as collections of entries, that is, instances of `java.util.Map.Entry`.

Example 13.15 Query Using IMPORT and Typecasting for Object Typing

```
IMPORT cacheRunner.Position;
SELECT DISTINCT posEntry.value.mktValue
   FROM /portfolios, (map<string,Position>)positions posEntry
   WHERE posEntry.value.mktValue > 25.00
```

The `IMPORT` statements in these examples assume that path to the imported class is in the `CLASSPATH`. The class's package name cannot be used in the `FROM` clause. It must be specified in an `IMPORT` statement.

The WHERE Clause

The optional `WHERE` clause defines the search criteria for the selection. The `WHERE` clause refines the search to filter the set of elements specified by the `FROM` clause. Without a `WHERE` clause, the `SELECT` projection list receives the entire collection or set of collections, as specified in the `FROM` clause.

The query processor searches the collection for elements that match the conditions specified in the `WHERE` clause conditions. If there is an index on an expression matched by the `WHERE` clause, then the query processor may use the index to optimize the search, and thus avoid iterating over the entire collection. For information, see *Indexes* on page 375. `WHERE` clause expressions are boolean conditions that are evaluated for each element in the collection. If the expression evaluates to `true` for an element, the query processor passes that element on to the `SELECT` projection list. This example uses the `WHERE` clause to return the portfolio objects in the region that have a type `xyz`.

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz'
```

The next query returns the set of all portfolios with a type of `xyz` and active status.

```
SELECT DISTINCT * FROM /portfolios WHERE "type" = 'xyz' AND status =
'active'
```

Joins

If collections in the `FROM` clause are not related to each other, the `WHERE` clause can be used to join them. The statement below returns all the `Persons` from the `/Persons` region with the same name as a `Flower` in the `/Flowers` region.

```
SELECT DISTINCT p FROM /Persons p, /Flowers f WHERE p.name = f.name
```

Indexes are supported for region joins. To create indexes for region joins, you create single-region indexes for both sides of the join condition. These are used during query execution for the join condition.

Partitioned regions do not support region joins.

The SELECT Projection List

The projections in the `SELECT` project list are used to transform the results of the `WHERE` search operation. The projection list is either specified as * or as a comma delimited list of expressions. For *, the interim results of the `WHERE` clause are returned from the query. Otherwise, the set of objects in the interim results are iterated and the projections applied to each of the objects. During the application of the projection list, the attributes of the objects being traversed are in scope for name resolution.

You can also specify retrieval of the entry keys in your projection list. This allows you to access the associated cached entries for modification and other purposes. The following example shows how the `Region` entry key can be obtained by using the region entries in the `FROM` clause and using appropriate projections. This query runs on the `/portfolios` region, returning a set of `struct<key:string, id:string, secId:string>` where `key` is the key of the region entry, `id` is an entry ID, and `secId` is a secId of a positions `Map` for the entry:

```
SELECT DISTINCT key, entry.value.id, posnVal.secId
   FROM /portfolios.entrySet entry, entry.value.positions.values posnVal
   WHERE entry.value."type" = 'xyz' AND posnVal.secId = 'XXX'
```

You can assign arbitrary names to the return values by using the `fieldname:expression` syntax in the projection list. This modification to the query results in a set of `struct<newKey:string, newId:string, newSecId:string>`:

```
SELECT DISTINCT newKey: entry.key, newId: entry.value.id,
       newSecId: posnVal.secId
   FROM /portfolios.entrySet entry,
       entry.value.positions.values posnVal
   WHERE entry.value."type" = 'xyz' AND posnVal.secId = 'XXX'
```

SELECT Statement Query Results

The result of a `SELECT` statement is either `UNDEFINED` or is a collection that implements the `SelectResults` interface. For details on `SelectResults`, see the online Java documentation for `com.gemstone.gemfire.cache.query`.

A `SELECT` statement returns a result, so it can be composed with other expressions, as in the following example:

```
(SELECT DISTINCT * FROM /portfolios WHERE status = 'active').size
```

Query results are discussed further in *SELECT Statement Query Results* on page 365.

While this describes the semantics of the `SELECT` statement, in practice, some or all of the iterations may be avoided through the use of indexes (see *Indexes* on page 375) and optimizations performed by the query processor.

The `SelectResults` returned from the `SELECT` statement is either a collection of objects or a collection of `Structs` (containing the objects).

A `Collection` of objects is returned in two cases:

- When only one expression is specified by the projection list and that expression is not explicitly specified using the `fieldname:expression` syntax
- When the `SELECT` list is `*` and a single collection is specified in the `FROM` clause

In all other cases, a struct is generated with a field for each expression in the `SELECT` projection list. This table describes the contents of the `SelectResults` based on the `SELECT` projection list and the expressions in the `FROM` clause.

Table 13.4 Matrix of SelectResults Contents Based on SELECT and FROM Clause Specifications

FROM	SELECT		
	*	Single Expression	Multiple Expressions
single expression	Objects	Objects. (`Structs` if the projection specifies a fieldname.)	`Structs`
multiple expressions	`Structs`	Objects. (`Structs` if the projection specifies a fieldname.)	`Structs`

When a struct is returned, the name of each field in the struct is determined as follows:

- If a field is specified explicitly using the `fieldname:expression` syntax, the `fieldname` is used.
- If the `SELECT` projection list is * and an explicit iterator expression is used in the `FROM` clause, the iterator variable name is used as the field name.
- If the field is associated with a region or attribute path, the last attribute name in the path is used.

If names can not be decided based on these rules, arbitrary unique names are generated by the query processor.

These examples show how the projections and `FROM` clause expressions are applied:

`SELECT DISTINCT *`
`  FROM /portfolios`
`  WHERE status = 'active'`
	Returns the `Collection` of active portfolios.

`IMPORT cacheRunner.Position;`
`SELECT DISTINCT secId`
`  FROM /portfolios, positions.values TYPE Position`
`  WHERE status = 'active'`
	Returns the `Collection` of `secId`s from the positions of active portfolios.

`IMPORT cacheRunner.Position;`
`SELECT DISTINCT secIdFieldName:secId`
`  FROM /portfolios, positions.values TYPE Position`
`  WHERE status = 'active'`
	Returns the `Collection` of `struct<secIdField: String>` for the active portfolios. (Compare to the results for the prior query.)

`SELECT DISTINCT "type", positions`
`  FROM /portfolios`
`  WHERE status = 'active'`
	Returns a `Collection` of `struct<type: String, positions: map>` for the active portfolios. The second field of the struct is a `Map` (`java.utils.Map`) object, which contains the `positions` map as the value.

`IMPORT cacheRunner.Position;`
`SELECT DISTINCT *`
`FROM /portfolios, positions.values TYPE Position`
`WHERE status = 'active'`
	Returns a `Collection` of `struct<portfolios: Portfolio, values: Position>` for the active portfolios.

`IMPORT cacheRunner.Position;`
`SELECT DISTINCT *`
`  FROM /portfolios pflo, positions posn TYPE Position`
`  WHERE pflo.status = 'active'`
	Returns a `Collection` of `struct<pflo: Portfolio, posn: Position>` for the active portfolios.

The Limit Keyword

The limit keyword can be optionally placed at the end of the query string to limit the number of rows returned. For example, this query returns at most 10 rows:

```
select * from /portfolios limit 10
```

If you use the limit keyword, you cannot also run operations on the query result set that do any kind of summary activities, as these would be meaningless. For example, trying to run `add` or `addAll` on a `SelectResults` returned from a query with a limit clause throws an exception.

Additional Query Language Elements

This section discusses various aspects and tools of the GemFire query engine. The subsections are:

- Method Invocation (page 367)
- Operators (page 368)
- Rules for UNDEFINED (page 370)
- Functions (page 370)
- Construction Expressions (page 371)
- The IN Expression (page 371)
- The Like Predicate (page 371)
- Literals (page 372)
- Query Parameter References (page 373)
- Type Conversions (page 374)
- Comments (page 374)

Method Invocation

The query language supports method invocation inside query expressions. The query processor maps attributes in query strings using the attribute rules described in *Region Querying Attributes* on page 353.

> *Methods declared to return* `void` *evaluate to* `null` *when invoked through the query processor.*

Invoking Methods Without Parameters

If you know that the attribute name maps to a public method that takes no parameters, you can simply include the method name in the query string as an attribute. For example, `emps.isEmpty` is equivalent to `emps.isEmpty()`. In the following example, the query invokes `isEmpty` on `positions`, and returns the set of all portfolios with no positions.

```
SELECT DISTINCT * FROM /portfolios WHERE positions.isEmpty
```

Invoking Methods With Parameters

GemFire also supports the invocation of public methods with parameters. To invoke methods with parameters, include the method name in the query string as an attribute and provide the method arguments between parentheses. The following example passes the argument `Bo` to the public method to return all employees whose names start with the characters `Bo`.

```
SELECT DISTINCT * FROM /Employees WHERE name.startsWith('Bo')
```

For overloaded methods, the query processor decides which method to call by matching the runtime argument types with the parameter types required by the method. If only one method's signature matches the parameters provided, it is invoked. The query processor uses runtime types to match method signatures.

If more than one method can be invoked, the query processor chooses the method whose parameter types are the most specific for the given arguments. For example, if an overloaded method includes versions with the same number of arguments, but one takes a `Person` type as an argument and the other takes an `Employee` type (derived from `Person`), `Employee` is the more specific object type. If the argument passed to the method is compatible with both of these types, the query processor chooses the method with the `Employee` parameter type.

The query processor uses the runtime types of the parameters and the receiver to determine the proper method to invoke. Because runtime types are used, an argument with a `null` value has no typing information, and so can be matched with any object type parameter. When a `null` argument is used, if the query processor cannot determine the proper method to invoke based on the non-null arguments, it throws an `AmbiguousNameException`.

Operators

GemFire supports the following operator types in expressions:

- Comparison Operators (page 368)
- Logical Operators (page 368)
- Unary Operators (page 369)
- Map and Index Operators (page 369)
- Dot Operator (page 369)

Comparison Operators

Comparison operators compare two values and return the results, either true or false. GemFire query language supports the following comparison operators:

=	equal to	<	less than
<>	not equal to	<=	less than or equal to
!=	not equal to	>	greater than
		>=	greater than or equal to

The equal (=) and not equal (<>, `!=`) operators have lower precedence than the other comparison operators.

The equal (=) and not equal (<>, `!=`) operators can be used with `NULL`. To perform equality or inequality comparisons with `UNDEFINED`, use the `IS_DEFINED` and `IS_UNDEFINED` operators. For more information, refer to *Rules for UNDEFINED* on page 370.

The following example selects all the portfolios with more than one position.

```
SELECT DISTINCT * FROM /portfolios WHERE positions.size >= 2
```

This example selects all the portfolios whose status is active:

```
SELECT DISTINCT * FROM /portfolios WHERE status = 'active'
```

This example selects all the portfolios whose id is not ‘`XYZ-1`’:

```
SELECT DISTINCT * FROM /portfolios WHERE id <> 'XYZ-1'
```

Logical Operators

The operators `AND` and `OR` allow you to create more complex expressions by combining expressions to produce a boolean result (`TRUE` or `FALSE`). When you combine two conditional expressions using the `AND` operator, both conditions must evaluate to true for the entire expression to be true. When you combine two conditional expressions using the `OR` operator, the expression evaluates to true if either one or both of the conditions are true. You can create complex expressions by combining multiple simple conditional expressions with `AND` and `OR` operators. When expressions are combined using multiple `AND` and `OR` operators, the `AND` operator has higher precedence than the `OR` operator.

The following query string selects only those portfolios with a type of ‘`xyz`’ that have active status. If either condition is false for an element in the queried collection portfolios, that element is not included in the result.

```
SELECT DISTINCT * FROM /portfolios
   WHERE "type" = 'xyz' AND status = 'active'
```

The following query string selects only those portfolios whose ID is `XYZ-1` or whose ID is `ABC-1`. If either one of these conditions is true for an element of portfolios, that element is included in the result.

```
SELECT DISTINCT * FROM /portfolios
   WHERE id = 'XYZ-1' OR id = 'ABC-1'
```

In the following example, the query selects only those portfolios with positions that have a market value over 30.00 and that have an ID of `XYZ-1` or `ABC-1`.

```
IMPORT cacheRunner.Position;
SELECT DISTINCT * FROM /portfolios, positions.values TYPE Position
   WHERE mktValue > 30.00
   AND (id = 'XYZ-1' OR id = 'ABC-1')
```

Unary Operators

Unary operators operate on a single value or expression, and have lower precedence than comparison operators in expressions. GemFire supports the unary operator `NOT`. `NOT` is the negation operator, which changes the value of the operand to its opposite. That is, if the expression evaluates to `TRUE`, `NOT` changes this value to `FALSE`. The operand must be a boolean. The following query returns the set of portfolios that have positions.

```
SELECT DISTINCT * FROM /portfolios WHERE NOT positions.isEmpty
```

Map and Index Operators

Map and index operators access elements in key/value collections (such as maps and regions) and ordered collections (such as arrays, lists, and `String`s). The operator is represented by a set of square brackets “`[ ]`” immediately following the name of the collection. The mapping or indexing specification is provided inside these brackets.

- Array, list, and `String` elements are accessed using an index value. Indexing starts from zero, with `0` representing the first element, `1` the second element and so on. If `myList` is an array, list, or `String` and `index` is an expression that evaluates to a non-negative integer, then `myList[index]` represents the (`index+1`)th element of `myList`. The elements of a `String` are the list of characters that make up the string.
- Map and region values are accessed by key using the same syntax. The key can be any `Object`. In the case of a `Region`, the map operator performs a non-distributed `get` that does not cause a `netSearch` if the value is not present in the local cache. Thus, `myRegion[keyExpression]` is the equivalent of `myRegion.getEntry(keyExpression).getValue`.

Dot Operator

The dot operator separates attribute names in a path expression, and specifies the navigation through object attributes. An alternative equivalent to the dot is the right arrow, “`->`”.

Rules for UNDEFINED

The special value UNDEFINED behaves according to the following rules:

- Accessing an attribute of a NULL value results in UNDEFINED.
- IS_UNDEFINED(UNDEFINED) returns TRUE.
- IS_DEFINED(UNDEFINED) returns FALSE.
- When the predicate defined by a WHERE clause returns UNDEFINED, this is handled as if the predicate returns FALSE.
- UNDEFINED is a valid value of explicit construction expressions (see *Construction Expressions* on page 371) and can be in the results of a SELECT statement if the value of a projection expression is UNDEFINED.
- For AND and OR expressions:

Expression	Return Value
UNDEFINED AND FALSE	FALSE
UNDEFINED AND TRUE	UNDEFINED
UNDEFINED AND UNDEFINED	UNDEFINED
UNDEFINED OR FALSE	UNDEFINED
UNDEFINED OR TRUE	TRUE
UNDEFINED OR UNDEFINED	UNDEFINED

Any other operation with any UNDEFINED operands results in UNDEFINED. This includes the dot operator, method invocations with UNDEFINED arguments, and comparison operations. For example, the operation X=UNDEFINED evaluates to UNDEFINED, but IS_UNDEFINED(UNDEFINED) evaluates to TRUE.

Functions

The query language supports these functions:

ELEMENT(expr)—Extracts a single element from a collection or array. This function throws a FunctionDomainException if the argument is not a collection or array with exactly one element.

IS_DEFINED(expr)—Returns TRUE if the expression does not evaluate to UNDEFINED.

IS_UNDEFINED(expr)—Returns TRUE if the expression evaluates to UNDEFINED.

NVL(expr1, expr2)—Returns expr2 if expr1 is null. The expressions can be bind arguments, path expressions, or literals.

TO_DATE (date_str, format_str)—Returns a Java Date class object. The arguments must be Strings with date_str representing the date and format_str representing the format used by date_str. The format_str you provide is parsed using java.text.SimpleDateFormat.

For information on defined and undefined values, see *Rules for UNDEFINED* on page 370.

The following example returns a boolean indicating whether the single instance of the portfolio with ID `XYZ-1` is active.

```
ELEMENT(SELECT DISTINCT * FROM /portfolios
   WHERE id = 'XYZ-1').status = 'active'
```

The next example returns all portfolios with an undefined value for status. For example, if `p` is `null`, then `p.status` is undefined.

```
SELECT DISTINCT * FROM /portfolios pflo
   WHERE IS_UNDEFINED(pflo.status)
```

In most queries, undefined values are not included in the query results. The `IS_UNDEFINED` function allows undefined values to be included in a result set, so you can identify elements with undefined values.

Construction Expressions

The construction expression implemented in GemFire is the set constructor. A set can be constructed using `SET(e1, e2, ..., en)` where `e1, e2, ..., en` are expressions. This constructor creates and returns the set containing the elements `e1, e2, ..., en`. For example, `SET(1, 2, 3)` returns a `Set` containing the three elements `1`, `2`, `3`.

The IN Expression

The `IN` expression is a boolean indicating if one expression is present inside a collection of expressions of compatible type based on their `equals` semantics.

If `e1` and `e2` are expressions, `e2` is a collection, and `e1` is an object or a literal whose type is a subtype or the same type as the elements of `e2`, then

```
e1 IN e2
```

is an expression of type `boolean`. The expression returns:

- `TRUE`, if `e1` is not `UNDEFINED` and is contained in collection `e2`
- `FALSE`, if `e1` is not `UNDEFINED` and is not contained in collection `e2`
- `UNDEFINED`, if `e1` is `UNDEFINED`

A very simple example of this is the expression, `2 IN SET(1, 2, 3)`, which returns `TRUE`.

For a more interesting example, look at the case where the collection you are querying into is defined by a subquery. Suppose that, in addition to the `/portfolios` region, there is another region named `/company` with attributes `id`, `name`, and `address`. You can run the following query to retrieve the names and addresses of all companies for whom you have an active portfolio on file.

```
SELECT name, address FROM /company
   WHERE id IN (SELECT id FROM /portfolios WHERE status = 'active')
```

The interior `SELECT` statement returns a collection of `id`s for all `/portfolios` entries whose status is active. The exterior `SELECT` iterates over `/company`, comparing each entry's `id` with this collection. For each entry, if the `IN` expression returns `TRUE`, the associated `name` and `address` are added to the outer `SELECT`'s collection.

The Like Predicate

GemFire offers limited support for the `like` predicate, such as in

```
where x like "<STRING>"
```

This query

```
where x like 'abc'
```

is the same as

```
where x = 'abc'
```

The string can be either '%' terminated or '*' terminated, both of which act as a wildcard. So this query

```
where x like 'abc%'
```

is the same as

```
where x >= 'abc' and x < 'abd'
```

The wild card can only be used at the end. If it is found any where else an exception is thrown. Escaped wild card characters can exist any where in the string to represent the characters themselves.

If an index is present the `like` predicate can use it.

Literals

Query language expressions can contain literals as well as operators and attribute names. These are the literal types that GemFire supports.

- **`boolean`**–A boolean value, either `TRUE` or `FALSE`
- **`integer`** and **`long`**—An integer literal is of type `long` if it is suffixed with the ASCII letter `L`. Otherwise it is of type `int`.
- **`floating point`**—A floating-point literal is of type `float` if it is suffixed with an ASCII letter `F`. Otherwise its type is `double` and it can optionally be suffixed with an ASCII letter `D`. A double or floating point literal can optionally include an exponent suffix of `E` or `e`, followed by a signed or unsigned number.
- **`string`**—String literals are delimited by single quotation marks. Embedded single-quotation marks are doubled. For example, the character string `'Hello'` evaluates to the value `Hello`, while the character string `'He said, ''Hello'''` evaluates to `He said, 'Hello'`. Embedded newlines are kept as part of the string literal.
- **`char`**—A literal is of type `char` if it is a string literal prefixed by the keyword `CHAR`, otherwise it is of type `string`. The `CHAR` literal for the single-quotation mark character is `CHAR ''''` (four single quotation marks).
- **`date`**—A `java.sql.Date` object that uses the JDBC format prefixed with the `DATE` keyword: `DATE yyyy-mm-dd` In the `Date`, `yyyy` represents the year, `mm` represents the month, and `dd` represents the day. The year must be represented by four digits; a two-digit shorthand for the year is not allowed.
- **time**—A `java.sql.Time` object that uses the JDBC format (based on a 24-hour clock) prefixed with the `TIME` keyword: `TIME hh:mm:ss`. In the `Time`, `hh` represents the hours, `mm` represents the minutes, and `ss` represents the seconds.
- **`timestamp`**—A `java.sql.Timestamp` object that uses the JDBC format with a `TIMESTAMP` prefix: `TIMESTAMP yyyy-mm-dd hh:mm:ss.fffffffff` In the `Timestamp`, `yyyy-mm-dd` represents the date, `hh:mm:ss` represents the time, and `fffffffff` represents the fractional seconds (up to nine digits).
- **`NIL`**—Equivalent alternative of `NULL`.
- **`NULL`**—The same as `null` in Java.
- **`UNDEFINED`**—A special literal that is a valid value for any data type. An `UNDEFINED` value is the result of accessing an attribute of a null-valued attribute. Note that if you access an attribute that has an explicit value of `null`, then it is not undefined. For example if a query accesses the attribute

`address.city` and `address` is `null`, then the result is undefined. If the query accesses `address`, then the result is not undefined, it is null. For more information, see Rules for `UNDEFINED` and `Null`.

Comparing Values with java.util.Date

You can compare the temporal literal values `DATE`, `TIME`, and `TIMESTAMP` with `java.util.Date` values. However, there is no literal for `java.util.Date` in the query language.

Query Parameter References

You can substitute a query parameter for any expression. Query parameters are identified by a dollar sign (`$`) followed by a digit that represents the parameter's position in the parameter array passed in the `execute` method. Counting begins at `1`. That is, `$1` references the first bound attribute, `$2` references the second bound attribute, and so on.

The `Query` interface provides an overloaded `execute` method that accepts parameters inside an `Object` array. These are the two implementations of the method:

- `execute()` for queries that do not take parameters
- `execute(Object[])` for queries requiring parameters

The 0th element of the `Object` array is used for the first query parameter, and so on. If the parameter count or parameter types do not match the query specification, the execute method throws an exception. A `QueryParameterCountInvalidException` is thrown if you pass in fewer parameters than the query requires. If the parameter object type is not compatible with the type that the query expects, then a `TypeMismatchException` is thrown.

In this example, the first parameter, the integer `2`, is bound to the first element in the `Object` array. The second parameter, `active`, is bound to the second element.

Example 13.16 Using Query Parameters

```
// Selects all portfolios where positions.size > $1 and status = $2
Query query = this.cache.getQueryService().newQuery(
   "SELECT DISTINCT * FROM /portfolios
   WHERE positions.size > $1 AND status = $2");
Object[] params = new Object[2];
params[0] = new Integer(2);
params[1] = "active";
try {
   Object results = query.execute(params);
} catch {
   // Handle exceptions
}
   // Do something with the results
```

This example shows a query string with the region path replaced by a query parameter. To use a parameter in the `FROM` clause, the parameter reference (`$1` in this example) must be bound to a collection. This query could be used on any collection of `Portfolios` by passing it in as a query parameter.

```
IMPORT cacheRunner.Portfolio;
SELECT DISTINCT * FROM $1 TYPE Portfolio WHERE status = 'active'
```

Type Conversions

The GemFire query processor performs implicit conversions and promotions under certain cases in order to evaluate expressions that contain different types. The query processor performs binary numeric promotion, method invocation conversion, and temporal type conversion.

Binary Numeric Promotion

Binary numeric promotion widens all operands in a numeric expression to the widest representation used by any of the operands. In each expression, the query processor applies the following rules in order:

- If either operand is of type `double`, the other is converted to `double`.
- If either operand is of type `float`, the other is converted to `float`.
- If either operand is of type `long`, the other is converted to `long`.
- Both operands are converted to type `int`.

The query processor performs binary numeric promotion on the operands of the following operators:

- Comparison operators <, <=, >, and >=
- Equality operators = and <>

This is essentially the same behavior as in Java, except that chars are not considered to be numeric in the GemFire query language.

Method Invocation Conversion

Method invocation conversion in the query language follows the same rules as Java method invocation conversion, except that the query language uses runtime types instead of compile time types, and handles null arguments differently than in Java. One aspect of using runtime types is that an argument with a null value has no typing information, and so can be matched with any type parameter. When a `null` argument is used, if the query processor cannot determine the proper method to invoke based on the non-null arguments, it throws an `AmbiguousNameException`. For more information on method invocation in query strings, see *Method Invocation* on page 367.

Temporal Type Conversion

The temporal types that the query language supports include the Java types `java.util.Date`, `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`, which are all treated the same and can be freely compared and used in indexes. When compared with each other, these types are all treated as nanosecond quantities.

Comments

You can include comments in the query string. To insert a one-line comment, begin the line with two dashes `(--)`. To insert a multiple-line comment, or to embed a comment in a line of code, begin the comment with `/*` and end it with `*/`. The following query strings include comments:

```
SELECT DISTINCT * FROM /portfolios /* here is a comment */
   WHERE status = 'active'

SELECT DISTINCT * FROM /portfolios
   WHERE status = 'active'-- here is another comment
```

13.6 Indexes

An index can provide significant performance gains for query execution. A query run without the aid of an index iterates through every object in the collection. If an index is available that matches part or all of the query specification, the query iterates only over the indexed set, and query processing time can be reduced. When you are creating your indexes, remember that indexes incur maintenance costs as they must be updated when the indexed data changes. An index that requires many updates and is not used very often may require more system resources than no index at all. Indexes also consume memory. For information on the amount of memory used for indexes, see *Memory Overhead Introduced by the Cache API* on page 135 of the *GemFire Enterprise System Administrator's Guide*. *Indexes are not allowed on overflow regions. If you try to create an index on an overflow region, the operation will throw an* `UnsupportedOperationException`. *It is also not possible to have your indexes overflow to disk.*

Index Types

Functional Index

GemFire provides functional sorted indexes that allow the comparison of an attribute to a constant. The comparison can be performed using any of the relational operators, and does not have to be restricted to attributes or attributes paths.

A comparison can refer to a complex function with parameters, as long as the attribute, attribute path, or function evaluates to an object which supports the `Comparable` interface. Consequently, an index expression which evaluates to a `Comparable` object is valid. See the Sun Java API for information about the `Comparable` interface.

The following are examples of valid comparables:

```
java.util.Date      object
java.lang.Integer   Float
java.lang.String    String
java.lang.Boolean   Boolean
```

Primary Key Index

In addition to functional sorted indexes, GemFire supports primary key indexes. A primary key index uses the keys in the region itself. By creating a primary key index, you make the query service aware of the relationship between the values in the region and the keys in the region and enable the relationship to be used to optimize the execution of queries.

The query service is not automatically aware of the relationship between the region values and keys. For this, you must create the primary key index.

The `FROM` clause for a primary key index must be just a region path. The indexed expression is an expression that, when applied to an entry value, produces the key. For example, if a region has `Portfolios` as the values and the keys are the `id` field of the `Portfolios`, their indexed expression would simply be "`id`".

The primary key index is not sorted. Without sorting, you can only do equality tests. Other comparisons are not possible. To obtain a sorted index on your primary keys, create a functional index on the attribute used as the primary key.

Creating an Index

Indexes can be created through the API or through the declarative cache configuration file.

> *Index creation requires establishing a write lock on the region, so* `put`*s to the region are blocked while the index is created.*

This example shows how to create two indexes on the `/portfolios` region using the `cache.xml`:

Example 13.17 Creating an Index in the cache.xml

```
<region name="portfolios">
   <region-attributes . . . >
      <value-constraint>cacheRunner.Portfolio</value-constraint>
   </region-attributes>
   <index name="myFuncIndex">
      <functional from-clause="/portfolios" expression="status"/>
   </index>
   <index name="myPrimIndex">
      <primary-key field="id"/>
   </index>
   <entry> . . .
```

This creates the same two indexes using the `QueryService` API:

Example 13.18 Creating an Index Through the API

```
qryService.createIndex("myFuncIndex", IndexType.FUNCTIONAL,
   "status", "/portfolios");
qryService.createIndex("myPrimIndex", IndexType.PRIMARY_KEY,
   "id", "/portfolios");
```

For more examples of index creation see *Indexing Guidelines* on page 378.

These are the `com.gemstone.gemfire.cache.query.QueryService` methods for index creation followed by parameter descriptions:

```
Index createIndex(String name, IndexType indexType,
      String indexedExpression, String fromClause);

Index createIndex(String name, IndexType indexType,
      String indexedExpression, String fromClause,
      String imports);
```

- The *`indexType`* must be either `IndexType.FUNCTIONAL` or `IndexType.PRIMARY_KEY`.
- The *`fromClause`* resolves to a collection or list of collections which will correspond to the `FROM` clause or part of a `FROM` clause in a `SELECT` statement. The `FROM` clause must include exactly one region, but may include multiple `FROM` expressions as required to drill down into nested region contents. The collections that the `FROM` expressions evaluate to must be dependent on one and only one entry in the referenced region (otherwise the index could not be maintained on single entry updates). References to query parameters are not allowed.
- The *`indexedExpression`* refers to the elements of the collection (or collection of structs) that are referenced in the *`fromClause`*. This expression is used to optimize the comparison of the same path found in a query's `WHERE` clause when used to compare against a constant expression. For example, an index with *`indexedExpression`* "`mktValue`" might be used for a query with a `WHERE` clause of "`mktValue > 25.00`". The exact use and specification of the `indexedExpression` varies

depending on the *indexType*. For details, see the online API documentation for `com.gemstone.gemfire.cache.query.IndexType`. Query parameters and region paths are not allowed in the *indexedExpression*.

- The *imports* provides packages and classes used in variable typing in the `FROM` expressions. The use is the same as for the `FROM` clause in querying.

The ordering of `FROM` clause expressions is independent of whether an index is applied.

Configuring Index Maintenance

Indexes are automatically kept current with the region data they reference. The region attribute `IndexMaintenanceSynchronous` specifies whether the region indexes are updated synchronously when a region is modified or asynchronously in a background thread. The default mode is synchronous, since this provides the greatest consistency with region data. This declarative index creation sets the maintenance mode to asynchronous:

```
<region-attributes index-update-type="asynchronous">
</region-attributes>
```

Asynchronous index maintenance batches up multiple updates to the same region key.

Indexing Guidelines

Optimizing your queries with indexes involves a cycle of careful planning, testing, and tuning. Ill-defined indexes can degrade the performance of your queries instead of improving it. This section describes index usage in the query service and provides tips for creating indexes that improve performance. It also describes how to use indexes for cross-region queries. This section contains the following subsections:

- Regions Used in the Examples (page 378)
- General Indexing Tips and Traps (page 378)
- Indexing Tips and Traps by Query Type (page 379)

Regions Used in the Examples

- Region 1 `/securities`

  Security objects uniquely identifying each security.

```
class Security {
   public String secID;
   public String secName;
   public String status;
   public List investors; //List of investors holding this security
}
```

- Region 2 `/investors`

  People who own securities. Each investor object contains the `List` of securities held and the orders placed.

```
public class Investor {
   public String name;
   public int age;
   public String investorID;
   public List heldSecurities; //List of all securities held
   public List ordersPlaced; //List of all the orders placed
}
```

- Region 3 `/orders`

  All the orders placed by various investors. An order is in one of these states: executed, pending, or allocated

```
public class Order {
   public int orderID;
   public String investorID;
   public String status;
   public List securities;
}
```

General Indexing Tips and Traps

- In general an index will improve query performance if the `FROM` clauses of the query and index match exactly. If there is not an exact match, performance may degrade.
- The query evaluation engine does not have a cost-based optimizer. It tends to use as many indexes as available, so take care to create only the indexes you want to have used. There is more information about this in the next section.

Indexing Tips and Traps by Query Type

This section lists a variety of queries and for each, discusses related new capabilities in the product and good indexing practices. In many cases, you will arrive at the best indexing scheme by trying several possibilities and comparing performance. The following are covered:

- Single Region Queries (page 379)
- Multi-Region Disjoint Query (page 380)
- Indexing for Equi-Join Queries (page 380)
- Equi-join Queries With Two Regions (page 381)
- Equi-Join Query With More Than Two Regions (page 382)

Single Region Queries

Example 13.19 Single Region Example 1

```
SELECT DISTINCT * FROM /securities WHERE secID = "123"
```

In this case, if `secID` is also the key in the `/securities` region, creating a primary key index on `secID` is the best bet as a primary key index does not have maintenance overhead. If `secID` is not the key, a functional index on `secID` should improve performance.

Example 13.20 Single Region Example 2

```
SELECT DISTINCT * FROM /investors WHERE name = "xyz" AND age > 45
```

In queries such as this, you have the option of creating a functional index on `name`, `age`, or on both. As the query engine currently does not have an optimizer (see *General Indexing Tips and Traps*, above), you should compare the performance of the query for two scenarios. For the first scenario, create a single index on the condition you expect to have the smallest result set size. For the second scenario, keep the first index and add an index on the second condition. Quite possibly, adding the second index will degrade performance. If this is the case, remove it and keep only the first index.

The order of the two comparisons in the query can also impact performance. Generally speaking, in OQL queries, as in SQL queries, you should order your comparisons so the earlier ones give you the fewest results on which to run subsequent comparisons.

Example 13.21 Single Region Example 3

```
SELECT DISTINCT *
FROM /investors inv,
   inv.heldSecurities inv_hs,
   inv.ordersPlaced inv_op
WHERE inv.age > 40
```

Suppose an index is created with this `FROM` clause and indexed expression:

Index FROM Clause	Indexed Expression

`/investors inv_indx,`
`inv_indx.ordersPlaced inv_op_indx,`
`inv_indx.heldSecurities inv_hs_indx`
	`inv_indx.age`

In this case, the index FROM clause matches the query FROM clause exactly in terms of number and type of iterators, so the index is used even though the relative positions differ. (The iterator names used in the FROM clauses are irrelevant to the matching.)

Multi-Region Disjoint Query

Example 13.22 Multi-region Disjoint Example 1

```
SELECT DISTINCT *
   FROM /investors inv, /securities sc, inv.heldSecurities inv_hs
   WHERE sc.secID = "123"
      AND sc.status = "active"
      AND inv.name = "xyz"
      AND inv.age > 75
      AND inv_hs.secName = "YHOO"
```

This is a two-region query with no condition joining the regions. There are five conditions, all of which can be indexed.

For your indexes, you should view the query as consisting of two independent groups, the first grouped on /investors and the second on /securities:

- The first group of iterators is: /investors inv, inv.heldSecurities inv_hs
- The first group of dependent conditions is: inv.name, inv.age, inv_hs.secName

While creating indexes on the conditions, use the criteria mentioned in *Single Region Queries* on page 379. An additional consideration is that as far as possible, the index FROM clause should match the query FROM clause (for that group). Thus, suppose you identify that inv.name has a sufficiently small result set. You should create an index on the /investors region with:

Index FROM Clause	Indexed Expression
/investors inv, inv.heldSecurities	inv.name

If the index FROM clause just contained /investors inv, the query engine would need to expand the index result to the level of inv.heldSecurities, which may degrade the performance. Using both of the collections brings the index result closest to the query result.

Similarly for the second independent group,

- The second group of iterators is: /securities sc
- The second group of dependent conditions is: sc.secID, sc.status

In general for AND conditions, if you have an N-region disjoint query, it is advisable to create N indexes if possible, one for each region.

Indexing for Equi-Join Queries

> *Indexes intended for use in equi-join queries must be created as functional indexes. Primary key indexes are not applied to equi-join queries.*

Equi-join queries are queries in which two regions are joined through an equality condition in the WHERE clause. The following query joins /investors with /orders through the equi-join condition, inv.investorID = ord.investorID.

Example 13.23 Equi-join Example 1

```
SELECT DISTINCT inv.name, ord.orderID, ord.status
   FROM /investors inv, /orders ord
   WHERE inv.investorID = ord.investorID
```

You can greatly improve the performance of this query by creating indexes on the left and right sides of the equi-join condition. The query engine can quickly evaluate the equi-join condition by iterating over the keys of the left- and right-side indexes for an equality match.

These are the functional index `FROM` clauses and expressions you would create for the query:

Index FROM Clause	Indexed Expression
`/investors inv`	`inv.investorID`
`/orders ord`	`ord.investorID`

Equi-join Queries With Two Regions

Example 13.24 Equi-join Example 2

```
SELECT DISTINCT *
   FROM /investors inv, /securities sc, inv.heldSecurities inv_hs
   WHERE sc.status = "active"
      AND inv.name = "xyz"
      AND inv.age > 75
      AND inv_hs.secName = sc.secName
```

This query has an equi-join condition across two regions along with a number of single-region conditions. For performance, the single most important condition in this query `FROM` clause is the condition `inv_hs.secName = sc.secName`, because it joins the two regions.

The first step is to create two indexes for the equi-join condition:

Index FROM Clause	Indexed Expression
`/investors inv, inv.heldSecurities inv_hs`	`inv_hs.secName`
`/securities sc`	`sc.secName`

You might be able to further improve performance by adding indexes for some or all of the single-region conditions. To optimize, you must compare performance with just the first two indexes with performance after adding indexes for the single region conditions (`sc.status = "active"`, `inv.name = "xyz"`, `inv.age > 75`).

> *For queries containing equi-join conditions, additional indexes for the single-region conditions should be created only if you are able to create at least one such index for each region in the query. Any indexing on a subset of the regions in the query will degrade performance.*

If there aren't indexable conditions available for all regions in the query, therefore, you should just create the equi-join indexes. In the above query, if an index is created only on `sc.status` and there is no indexing on a condition dependent on `inv`, the performance will be worse than without the `sc.status` index. You should have an index on `sc.status` along with an index on `inv.age` or `inv.name` or both.

Equi-Join Query With More Than Two Regions

In the case of equi-join queries of three or more regions, first, identify all equi-join conditions. Create functional indexes for as many of the left- and right-side expressions as are required to join all regions. Do not create indexes for all left- and right-side expressions that you find in the equi-join conditions. If there are equi-join conditions that redundantly join two regions—in order to more-finely filter the data, for example—providing redundant indexes for these joins will negatively impact performance. Create indexes only on one equi-join condition for each region pair. This is illustrated in the following two query examples. The first example is a simple, multi-region equi-join query without redundant joins. The second uses multiple joins and must be studied carefully to provide the proper level of indexing help. First, the simple query:

Example 13.25 Equi-join Example 3

```
SELECT DISTINCT *
   FROM /investors inv, /securities sc,
      /orders or, inv.ordersPlaced inv_op, or.securities or_sec
   WHERE inv_op.orderID = or.orderID AND or_sec.secID = sc.secID
```

There are three regions in this query, joined together through two equi-join conditions in the `WHERE` clause. There are no conditions joining regions that have already been joined. In this case you create four indexes:

- An index for each of the two sides of `inv_op.orderID = or.orderID`. This covers the `/investors` and `/orders` region equi-join.
- A second pair of indexes for `or_sec.secID = sc.secID`. This covers the `/orders` and `/securities` equi-join.

This table lists the four indexes:

Index FROM Clause	Indexed Expression
`/investors inv, inv.ordersPlaced inv_op`	`inv_op.orderID`
`/orders or, or.securities or_sec`	`or.orderID`
`/orders or, or.securities or_sec`	`or_sec.secID`
`/securities sc`	`sc.secID`

Example 13.26 Equi-join Example 4

```
SELECT DISTINCT *
   FROM /investors inv, /securities sc,
      /orders or, inv.ordersPlaced inv_op, or.securities or_sec,
      sc.investors sc_invs
   WHERE inv_op.orderID = or.orderID
      AND or_sec.secID = sc.secID
      AND inv.investorID = sc_invs.investorID
```

This is the first query modified with an extra equi-join condition across two regions.

In this query, you have three equi-join conditions, but only two of the three are required to establish the relationship across the three regions. These are the join conditions in the query along with their relative index pairs:

- The first condition joins `/investors` with `/orders`:

Index FROM Clause	Indexed Expression
`/investors inv, inv.ordersPlaced inv_op`	`inv_op.orderID`
`orders or, or.securities or_sec`	`or.orderID`

- The second condition joins `/orders` with `/securities`:

Index FROM Clause	Indexed Expression
`/investors inv, inv.ordersPlaced inv_op`	`inv.investorID`
`/securities sc, sc.investors sc_invs`	`sc_invs.investorID`

- The third condition joins `/investors` with `/securities`:

Index FROM Clause	Indexed Expression
`/orders or, or.securities or_sec`	`or_sec.secID`
`/securities sc, sc.investors sc_invs`	`sc.secID`

Any two of these three relationships is sufficient to join the three regions. The query engine can only efficiently use indexes for two of the conditions. Any additional indexes for the equi-joins would introduce needless overhead into the system and may slow your query.

> *Only use the minimum indexing required to cover the equi-join conditions in your queries. Using additional indexing adds overhead and slows the querying evaluation.*

You can optimize your indexing in this case, however, by using the most performant indexing set. The most performant set will be that which narrows the data to the smallest result set possible. You should examine your data and experiment with the three index pairs to see which provides the best performance.

13.7 Querying Partitioned Regions

The basic unit of storage for a partitioned region is a bucket, which resides on a GemFire node and contains all the entries that map to a single hashcode. Querying a partitioned region involves distributing the query to all buckets across all nodes, then merging the result sets and sending the query results back to the requesting client. If a partitioned region is configured for high availability, then redundant copies of its data exist across multiple nodes in the form of primary and secondary buckets. In this case, the query is distributed to all primary buckets across all nodes.

Partitioned Region Query Guidelines

Partitioned region queries function the same as non-partitioned region queries, except for the restrictions listed in this section. Partitioned region queries that do not follow these guidelines generate an `UnsupportedOperationException` with a related error message.

- Queries for partitioned regions cannot do joins or cross-region queries between separate partitioned regions, or between a partitioned region and any other region.
- No regions other than the partitioned region can be referenced in the query.
- The query must be just a `SELECT` expression (as opposed to arbitrary OQL expressions), which can be preceded by one or more `IMPORT` statements.

  For example, this query is not allowed because it is not just a `SELECT` expression:

  ```
  (SELECT DISTINCT *FROM /prRgn WHERE attribute > 10).size
  ```

  However, this query is allowed:

  ```
  SELECT DISTINCT *FROM /prRgn WHERE attribute > 10
  ```

- The `SELECT` expression itself can be arbitrarily complex, including nested `SELECT` expressions, as long as only one partitioned region is referenced.
- The partitioned region reference can only be in the first `FROM` clause iterator. Additional `FROM` clause iterators are allowed if they do not reference any regions (such as drilling down into the values in the partitioned region).

  For example, this query is not allowed because the second `FROM` clause iterator references another region (`/portfolios`):

  ```
  SELECT DISTINCT *FROM /prRgn, /portfolios WHERE attribute > 10
  ```

- The first `FROM` clause iterator must contain only one reference to the partitioned region (the reference can be a parameter, such as `$1`).
- The first `FROM` clause iterator cannot contain a subquery, but subqueries are allowed for additional `FROM` clause iterators.

See *The Query Language* on page 352 for more information about constructing queries.

Indexing a Partitioned Region

A partitioned region can contain large amounts of data, so querying is optimized by using indexes. The commands for creating and removing indexes for a partitioned region are the same as a distributed region. See *Indexes* on page 375 for more information.

Index creation or removal for a partitioned region is a distributed operation, which means the index is created or removed on all nodes that host the partitioned region.

The Querying Process

This is a typical sequence of events when a query request is submitted to a partitioned region:

5. A query message is sent to each remote datastore. The message contains the query string, along with a list of buckets to query.
6. Each datastore executes the query over the buckets specified in the bucket list.
7. Each datastore merges all the result sets from the individual buckets.
8. The results from the datastores are merged and returned as the query results.

Queries Producing Large Result Sets

The size of a query result set depends on the restrictiveness of the query and the size of the total data set. A partitioned region can hold much more data than other types of regions, so there is more potential for larger result sets on partitioned region queries. This could cause the member receiving the results to run out of memory if the result set is very large. The following query could potentially return a very large result set:

```
SELECT DISTINCT * FROM /portfolios
```

Handling Query Failures

If a partitioned region data host fails during query execution, there is no automatic retry and the query fails with an exception. A partitioned region can recover from a failure, so the application can retry query execution after the failure occurs.

13.8 Performance Considerations

As with query processors that run against relational databases, the way a query is written can greatly effect execution performance. Among other things, whether indexes are used depends on how each query is stated. These are some of the things to consider when optimizing your GemFire queries for performance.

- Indexes are not used in expressions that contain `NOT`, so in a `WHERE` clause of a query, `qty >= 10` could have an index on `qty` applied for efficiency. However, `NOT( qty < 10)` could not have the index applied.
- For `AND` operators, you may get better results if the conditions that use indexes and/or are more selective come before other conditions in the query. The following example shows the `AND` operator with two conditions:

  ```
  SELECT DISTINCT * FROM /investors WHERE name = "xyz" AND age > 45
  ```

  You have the option of creating a functional index on `name`, `age`, or on both. As the query engine currently does not have an optimizer, you should compare the performance of the query for two scenarios. For the first scenario, create a single index on the condition you expect to have the smallest result set size. For the second scenario, keep the first index and add an index on the second condition. Quite possibly, adding the second index will degrade performance. If this is the case, remove it and keep only the first index.

13.9 The API Interfaces and Classes

The GemFire querying API for Java allows you to access all the querying functionality discussed in the previous sections. This section gives a general overview of the interfaces and classes that are provided by the API package, `com.gemstone.gemfire.cache.query`, and the shortcut methods provided in the `com.gemstone.gemfire.cache.Region` interface. Complete, current information on the classes and interfaces listed here is available in the online Java API documentation.

com.gemstone.gemfire.cache.query

Creating and Managing Queries and Indexes

- **QueryService**—This interface provides methods for creating queries and creating and managing indexes. It is the entry point to the query package. You can query your application's cache or, if the application is a client, you can query the server tier:
  - `Cache.getQueryService`–Queries the process cache.
  - `Pool.getQueryService` or `ClientCache.getQueryService`–Queries the servers associated with the pool.
  - `ClientCache.getLocalQueryService`–Queries the client's cache.

  This interface also provides methods for client/server continuous queries, or CQs, which are described in Chapter 14, *Continuous Querying*, on page 397.
- **Query**—The `Query` interface provides methods for managing the compilation and execution of a single query and for retrieving its query string. This interface is accompanied by the `QueryStatistics` interface, which provides information on query execution.
- **Index**—The `Index` interface provides information about an index including the name and type of the index and the region it indexes. There are several index-related interfaces. `IndexStatistics` provides lists of index keys and values and information about updates to the index and index usage. The `IndexType` class enumerates the indexes supported by the system.

Query Result Sets

- **SelectResults**—This interface is used to manipulate and retrieve the results of a `SELECT` expression.
- **Struct**—This interface is used when a `SELECT` statement returns more than one projection. For information on when this could happen, see *The SELECT Projection List* on page 364. This is accompanied by a `StructType`, which provides the `Struct`'s definition.

com.gemstone.gemfire.cache.Region

Region Shortcut Query Methods

The `Region` interface has several query shortcut methods. All take a query predicate, which is what you specify in the `WHERE` clause of a regular query (see *The WHERE Clause* on page 363). In the shortcut methods, the DISTINCT keyword is implicitly used.

- The `query` method retrieves a collection of values satisfying the query predicate. This call retrieves active portfolios, which in the sample data are the portfolios with keys `XYZ-1, XYZ-2,` and `ABC-1:`

  ```
  Collection activePflo = portfolioRegion.query("status = 'active' ");
  ```

- The `selectValue` method is used to retrieve a singleton result. If more than one result object is returned in the query result, this method throws a `FunctionDomainException`. This example requests the portfolio with `id ABC-1`:

  ```
  Portfolio a1 = (Portfolio) portfolioRegion.selectValue("id = 'ABC-1'");
  ```

- The `existsValue` method returns a boolean indicating if any entry exists that satisfies the predicate. This call returns `false`, because there is no entry with the indicated `type`:

  ```
  boolean entryQQQExists = portfolioRegion.existsValue("type = 'QQQ' ");
  ```

13.10 Programming Examples

This section includes the following examples:

- Index Management Examples (page 389)
  - Index Creation (page 389)
  - Retrieving Index Information (page 391)
  - Index Removal (page 392)
- Query Management Examples (page 393)
  - Query Creation and Execution (page 393)
  - Examining Query Results (page 394)

Index Management Examples

Index management through the API consists of creating indexes, updating and retrieving index information and statistics, and deleting indexes. This section provides examples of these operations.

Index Creation

The cacheRunner `mkidx` function creates an index based on user input. The parameters are index name, index type, indexed expression, and `FROM` clause. For the sample `/portfolios` data, valid calls to `mkidx` include:

Parameters for a functional sorted index

```
mkidx mktVal func posn.mktValue "/portfolios, positions.values posn"
```

Parameters for a primary key index

```
mkidx pfloKey pkey pflo.id "/portfolios pflo"
```

Parameters for a functional sorted index on the region keys

```
mkidx fnlKey func id "/portfolios"
```

This is the code that creates the indexes.

Example 13.27 Index Creation

```
/** Creates an index.
* Arguments are type, name, idxExpr, fromClause
*/
void mkidx(String command) throws QueryException {
   List args = new ArrayList();
   parseCommand(command, args);
   if (args.size() < 5) {
      System.out.println("mkidx requires 4 args: name type idxExpr
fromClause");
      return;
   }
   String name = (String)args.get(1);
   String type = (String)args.get(2);
   String idxExpr = (String)args.get(3);
   String fromClause = (String)args.get(4);

   IndexType indexType;
   if (type.toLowerCase().startsWith("func")) {
      indexType = IndexType.FUNCTIONAL;
   } else if (type.toLowerCase().startsWith("pkey")) {
      indexType = IndexType.PRIMARY_KEY;
   } else {
      throw new UnsupportedOperationException("Currently only support " +
      "functional and primary key indexes. No support for " + type + "." );
   }
   QueryService qs = this.cache.getQueryService();
   qs.createIndex(name, indexType, idxExpr, fromClause);
}
```

Retrieving Index Information

The `indexes` command lists all indexes in the cache and their statistics.

Example 13.28 Retrieving Index Information

```
void indexes(String command) {
   QueryService qs = this.cache.getQueryService();
   Collection indexes = qs.getIndexes();
   StringBuffer sb = new StringBuffer();
   sb.append("There are ");
   sb.append(indexes.size());
   sb.append(" indexes in this cache\n");

   for (Iterator iter = indexes.iterator(); iter.hasNext(); ) {
      Index index = (Index) iter.next();
      sb.append(index.getType());
      sb.append(" index \"");
      sb.append(index.getName());
      sb.append("\" on region ");
      sb.append(index.getRegion().getFullPath());
      sb.append("\n");
      String expr = index.getIndexedExpression();
      if (expr != null) {
         sb.append(" Indexed expression: ");
         sb.append(expr);
         sb.append("\n");
      }
      String from = index.getFromClause();
      if (from != null) {
         sb.append(" From: ");
         sb.append(from);
         sb.append("\n");
      }
      String projection = index.getProjectionAttributes();
      if (projection != null) {
         sb.append(" Projection: ");
         sb.append(projection);
         sb.append("\n");
      }
      sb.append(" Statistics\n");
      IndexStatistics stats = index.getStatistics();
      sb.append(stats.getTotalUses());
      sb.append(" uses\n");
      sb.append(stats.getNumberOfKeys());
      sb.append(" keys, ");
      sb.append(stats.getNumberOfValues());
      sb.append(" values\n");
      sb.append(stats.getNumUpdates());
      sb.append(" updates totaling ");
      sb.append(stats.getTotalUpdateTime() / 1e6);
      sb.append(" ms.\n");
   }
   System.out.println(sb.toString());
}
```

Index Removal

This is the code that removes an index from the cache. As with index creation, the query service is used to remove the index.

Example 13.29 Index Removal

```
void rmidx(String command) throws QueryException {
    List args = new ArrayList();
    parseCommand(command, args);
    if (args.size() < 1) {
        System.out.println("rmidx requires 1 arg: name");
        return;
    }
    String name = (String)args.get(1);

    QueryService qs = this.cache.getQueryService();
    Index index = qs.getIndex(this.currRegion, name);
    if (index == null) {
        String s = "There is no index named \"" + name +
            "\" in region " + this.currRegion.getFullPath();
        System.err.println(s);
    } else {
        qs.removeIndex(index);
    }
}
```

Query Management Examples

Queries are created through the `QueryService` and then managed through the resulting `Query` object.

Query Creation and Execution

The `CacheRunner.java exec` command runs a query string provided in input and prints the results. The query string can be as simple as `"42"` (which returns `42`) and as complex as a nested `SELECT` statement that accesses multiple regions and drills down into multiple levels of data within each region.

This is the part of the command that runs the query.

Example 13.30 Query Execution

```
void exec(String command) throws QueryException {
// remove the "exec" command from the string
// (any end of line '\' characters have already been removed)
String execCmd = new StringTokenizer(command).nextToken();
String queryString = command.substring(execCmd.length());

try {
   Query query = this.cache.getQueryService().newQuery(queryString);
   long startNanos = Utils.nanoTime();
   Object results = query.execute();
   long execNanos = Utils.nanoTime() - startNanos;
   System.out.println("Query Executed in " + (execNanos / 1e6) + " ms.");
```

Examining Query Results

For queries that are not `SELECT` statements, the return value is an object. For `SELECT` statements, the results are either `UNDEFINED` or are a `SelectResults` object. This code handles any type of return value. This is the rest of the `exec` command that you started listing in *Query Creation and Execution* on page 393.

Example 13.31 Managing Query Results

```
if (results instanceof SelectResults) {
   StringBuffer sb = new StringBuffer();

   SelectResults sr = (SelectResults) results;
   CollectionType type = sr.getCollectionType();
   sb.append(sr.size());
   sb.append(" results in a collection of type ");
   sb.append(type);
   sb.append("\n");

   sb.append(" [");
   if (sr.isModifiable()) {
      sb.append("modifiable, ");
   } else {
      sb.append("unmodifiable, ");
   }
   if (type.isOrdered()) {
      sb.append("ordered, ");
   } else {
      sb.append("unordered, ");
   }
   if (type.allowsDuplicates()) {
      sb.append("duplicates");
   } else {
      sb.append("no duplicates");
   }
   sb.append("]\n");
   ObjectType elementType = type.getElementType();
   for (Iterator iter = sr.iterator(); iter.hasNext(); ) {
      Object element = iter.next();
      if (elementType.isStructType()) {
         StructType structType = (StructType) elementType;
         Struct struct = (Struct) element;
         ObjectType[] fieldTypes = structType.getFieldTypes();
         String[] fieldNames = structType.getFieldNames();
         Object[] fieldValues = struct.getFieldValues();

         sb.append(" Struct with ");
         sb.append(fieldTypes.length);
         sb.append(" fields\n");

         for (int i = 0; i < fieldTypes.length; i++) {
            ObjectType fieldType = fieldTypes[i];
            String fieldName = fieldNames[i];
            Object fieldValue = fieldValues[i];
            sb.append(" ");
            sb.append(fieldType);
            sb.append(" ");
            sb.append(fieldName);
            sb.append(" = ");
```

```
                sb.append(fieldValue);
                sb.append("\n");
            }
        } else {
            sb.append(" ");
            sb.append(element);
        }
        sb.append("\n");
    }
    System.out.println(sb);
} else { // Just a regular object
    System.out.println(results);
}
```

This is the output from the execution of a very simple query, `/portfolios.isDestroyed`.

Example 13.32 Output for Simple Query

```
/portfolios> exec /portfolios.isDestroyed
Query Executed in 3.57215 ms.
results are not SelectResults
false
```

This is the output from the execution of a query that produces `SelectResults`:

Example 13.33 Output with SelectResults

```
/portfolios> exec \
IMPORT cacheRunner.Position; \
SELECT posnVal \
FROM /portfolios, positions.values posnVal TYPE Position \
WHERE posnVal.mktValue >= 25.00

Query Executed in 7.323534 ms.
8 results in a collection of type java.util.Set<cacheRunner.Position>
   [modifiable, unordered, no duplicates]
   Position [secId=RHAT qty=90.0 mktValue=67.356572]
   Position [secId=SAP qty=90.0 mktValue=67.356572]
   Position [secId=DELL qty=376.0 mktValue=101.34]
   Position [secId=APPL qty=90.0 mktValue=67.356572]
   Position [secId=IBM qty=8765.0 mktValue=34.29]
   Position [secId=AOL qty=978.0 mktValue=40.373]
   Position [secId=NOVL qty=376.0 mktValue=101.34]
   Position [secId=ORCL qty=376.0 mktValue=101.34]
/portfolios>
```

Chapter 14

Continuous Querying

Continuous querying in GemFire Enterprise® gives your clients a way to run queries against events in the server region. The clients register interest in events using simple query expressions. Events are sent to client listeners that you can program to do whatever your application requires.

Continuous queries (CQs) provide these main features:

- **Standard GemFire Enterprise query syntax and semantics**—CQ queries are expressed in the same language used for other GemFire Enterprise queries (see Chapter 13, *Querying and Indexing*, on page 341).
- **Standard GemFire Enterprise events-based management of CQ events**—The event handling used to process CQ events is based on the standard GemFire Enterprise event handling framework (see Chapter 11, *Handling Events*, on page 293).
- **Complete integration with the client/server architecture**— CQ functionality uses existing server-to-client messaging mechanisms to send events. All tuning of your server-to-client messaging also tunes the messaging of your CQ events. If your system is configured for high availability, your CQs are highly available, with seamless failover provided in case of server failure (see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201). If your clients are durable, you can define any of your CQs as durable as well (see *Durable Subscription Queues* on page 246).
- **Interest criteria based on data values**—CQ queries are run against the region's entry values. Compare this to register interest at *Client Interest Registration* on page 220.
- **Active query execution**—Once initialized, the queries only operate on new events instead of on the entire region data set. Events that change the query result are sent to the client immediately.

In this chapter:

- How Continuous Querying Works (page 398)
- The GemFire CQ API (page 400)
- State and Life Cycle of a CQ (page 401)
- Implementing Continuous Queries (page 402)

14.1 How Continuous Querying Works

Continuous querying allows you to subscribe to server-side events using SQL-type query filtering. With CQ, the client sends a query to the server side for execution and receives the events that satisfy the criteria. For example, in a region storing stock market trade orders, you can retrieve all orders over a certain price by running a CQ with a query like this:

```
SELECT * FROM /tradeOrder t WHERE t.price > 100.00
```

When the CQ is running, the server sends the client all new events that affect the results of the query. On the client side, listeners programmed by you receive and process incoming events. For this example query on `/tradeOrder`, you might program a listener to push events to a GUI where higher-priced orders are displayed. CQ event delivery uses the client/server subscription framework described in *Subscribing to Server Events* on page 238.

Your clients can execute any number of CQs, with each CQ given any number of listeners. This figure shows the logical architecture of continuous querying.

Figure 14.1 Logical Architecture of Continuous Querying

CQs do not update the client region. This is in contrast to other server-to-client messaging like the updates sent to satisfy interest registration and responses to `get` requests from the client's `Pool`. CQs serve as notification tools for the CQ listeners, which can be programmed in any way your application requires.

When a CQ is running against a server region, each server entry event is evaluated against the CQ query by the thread that updates the cache. If either the old or the new entry value satisfies the query, the thread puts a `CqEvent` in the client's queue. The `CqEvent` contains information from the original cache event plus information specific to the CQ's execution. Once received by the client, the `CqEvent` is passed to the `onEvent` method of all `CqListeners` defined for the CQ.

> *The region operations do not translate directly to specific query operations and the query operations do not specifically describe the region events. Instead, each query operation describes how its corresponding region event affects the query results. For more information on this, see* *The CqEvent Object* on page 405.

This figure shows the typical CQ data flow when entries are updated in the server cache. The steps are:

1. Entry events come to the server's cache from any source: the server or its peers, distribution from remote sites, or updates from a client.
2. For each event, the server's CQ executor framework checks for a match with the CQs it has running.
3. If the old or new entry value satisfies a CQ query, a CQ event is sent to the CQ's listeners on the client side. Each listener for the CQ gets the event. In this figure:
   - Both the new and old prices for entry X satisfy the CQ query, so that event is sent indicating an update to the query results.
   - The old price for entry Y satisfied the query, so it was part of the query results. The invalidation of entry Y makes it not satisfy the query. Because of this, the event is sent indicating that it is destroyed in the query results.
   - The price for the newly created entry Z does not satisfy the query, so no event is sent.

Figure 14.2 Data Flow of a Continuous Query

14.2 The GemFire CQ API

The GemFire Java CQ API allows your clients to create and manage CQs. The server side does not have an API. This section lists the primary GemFire Enterprise API for CQ management. For complete information on the classes and interfaces described here, see the online Java API documentation.

com.gemstone.gemfire.cache.query

- **QueryService**—This interface provides methods to:
  - create a new CQ and specify whether it is durable (available for durable clients)
  - execute a CQ, with or without an initial result set
  - list all the CQs registered by the client
  - close and stop CQs at the cache and region level
  - get a handle on `CqStatistics` for the client

  The `QueryService` *CQ methods are run against the server cache. The other* `QueryService` *methods for index creation and one-time queries are run against the client's cache as documented in* Chapter 13, *Querying and Indexing*, on page 341.

- **CqQuery**—This interface provides methods for managing a continuous query once it is created through the `QueryService`. This is the interface that is normally used to begin and end CQ execution and to retrieve other CQ-related objects such as the CQ attributes, CQ statistics, and the CQ state.
- **CqListener**—This application plug-in interface is programmed by you to handle continuous query events after they occur. This listener is usually implemented using the `CqListenerAdapter` interface in `com.gemstone.gemfire.cache.util`, which also inherits the `close` method from `CacheCallback` in `com.gemstone.gemfire.cache`. If the listener is declared in the `cache.xml`, it must also implement `Declarable`.
- **CqEvent**—This interface provides all the information sent from the server about the CQ event, which is passed to the CQ's `CqListener` methods.
- **CqAttributes, CqAttributesFactory, CqAttributesMutator**—These interfaces allow you to manage CQ attributes. The attributes are composed of `CqListener` plug-in specifications.
- **CqStatistics, CqServiceStatistics**—These interfaces allow you to access statistics information for a single CQ and for the query service's management of CQs as a whole. For a discussion of GemFire Enterprise statistics as they apply to the developer, see *Statistics* on page 464. For details on statistics, see *System Statistics* on page 235 of the *GemFire Enterprise System Administrator's Guide*.

14.3 State and Life Cycle of a CQ

A CQ has three possible states. These are maintained on the server and can be checked by the client with `CqQuery.getState`.

- **`STOPPED`**—The CQ has been created but not yet executed or it has been explicitly stopped from executing. A stopped CQ uses system resources. Stopping a CQ only stops the CQ event messaging from server to client. All server-side CQ processing continues, but new CQ events are not placed into the server's client queue. Stopping a CQ does not change anything on the client side (but, of course, the client stops receiving events for the CQ that is stopped). The CQ can be started or restarted by calling the `execute` method on `CqQuery`.
- **`RUNNING`**—The CQ is being executed on the server for all events in the region referenced by the query. Results are sent to all client listeners associated with the `CqQuery`.
- **`CLOSED`**—The CQ is stopped and is not using system resources. Invoking an execute or stop method on closed `CqQuery` throws an exception.

A CQ life cycle usually flows like this:

1. The client creates the CQ. This sets up everything for running the query and provides the client with a `CqQuery` object, but does not execute the CQ. At this point, the query is in a STOPPED state, ready to be closed or run.
2. The client runs the CQ with an API call to one of the `CqQuery` `execute*` methods. This returns initial results, if requested, and puts the query into a `RUNNING` state on the client and on the server.
3. The CQ is closed by a client call to `CqQuery.close`. This deallocates all resources in use for the CQ on the client and server. At this point, the cycle could begin again with the creation of a new `CqQuery` instance. You cannot rerun a closed CQ.

14.4 Implementing Continuous Queries

The continuous query (CQ) functionality requires standard client/server distributed system and cache configuration settings:

You can specify CQs for any of the server regions. To implement a CQ, follow these basic steps:

1. Configure the client `Pool` with `subscription-enabled` (page 212) set to `true`.

   If you have dependencies between CQs and interest subscriptions, so that you want the two types of subscription events to arrive as closely together on the client, use a single `Pool` *for everything. Using different pools can lead to time differences in the delivery of events because the* `Pool`*s might use different servers to process and deliver the event messages.*

2. Decide what data to track on the server and write your queries.

   Use your criteria for tracking data changes to write your CQ queries. For information on this, see *Writing the Query* on page 403.

3. Determine how to handle the CQ events on the client and write your listeners.

   Each of your CQs can have any number of listeners. In addition to your core CQ listeners, you might have listeners that you use for all CQs to track statistics or other general information. For information on writing your listeners, see *Writing the CQ Listener* on page 404.

4. Write the client code to put the queries and their listeners into named CQ queries and execute the queries. Make sure to close the queries if your client no longer needs them and when the client exits. For information and examples, see *Running the Continuous Query Code* on page 406.

5. Write any code you need for monitoring and managing your `CqQuery`. For information on the tools available, see *Managing Your CQs* on page 407.

6. If you want your CQs to be highly available, configure your servers for high availability. See *Highly Available Subscription Delivery* on page 241. When your servers are highly available, CQs are registered on primary and secondary servers and server failover is performed without any interruption to CQ messaging. CQ events messaging uses the same queues used for server-to-client messaging, described in *Subscribing to Server Events* on page 238.

7. If you want any of your CQs to be durable, configure your clients for durable messaging and indicate the durable CQs in their `QueryService.newCq` calls. See *Durable Subscription Queues* on page 246. When your clients are durable, you can create durable CQs whose events are maintained during client disconnects and replayed for the client when it reconnects. The process and data flow particular to durable CQs is described in the client/server section on durability.

8. Run your code, monitoring and tuning as needed. You can monitor CQ activities using `CqStatistics` and `CqServiceStatistics`.

Writing the Continuous Query

Each CQ uses a query and any number of listeners. The query filters the events on the server and the listener handles the events that make it through the query filter. With the query and the listener in hand, you can create and execute your query through the API.

Avoiding In-Place Modifications

Your application must be careful while modifying objects that have been put into the cache or retrieved from the server cache. The application should not do an in-place modification of the fields or objects that are specified in the `WHERE` clause attribute of the CQ query. An in-place modification can cause the CQ to receive no events, or receive inconsistent events. Any modifications **must** be made on a copy of the object.

Writing the Query

This is the basic syntax for the CQ query:

```
SELECT * FROM /fullRegionPath [iterator] [WHERE clause]
```

The CQ query must satisfy the standard GemFire Enterprise querying specifications described in *The Query Language* on page 352. It also must satisfy these restrictions:

- The `FROM` clause must contain only a single region path specification, with optional iterator variable.
- The query must be a SELECT expression only, preceded by zero or more IMPORT statements. This means the query *cannot* be a statement like "`/tradeOrder.name`" or "`(SELECT * from /tradeOrder).size`".
- The CQ query can not use:
  - Cross region joins
  - Drill-downs into nested collections
  - DISTINCT
  - Projections
  - Bind parameters

Queries not meeting these constraints generate an `UnsupportedOperationException` from the `QueryService` `newCq` method. Future versions of the product will address most of these constraints.

Writing the CQ Listener

CQ listener management is based on the GemFire Enterprise event management framework described in *The GemFire Event Framework* on page 294. The `CqListener` you program to handle the CQ events extends the `CacheCallback` interface that is used by all other cache event handlers. You must program the listener to satisfy the requirements for cache event listeners.

This example shows how you might program a simple `CqListener` to update a display screen based on the CQ events it receives. The listener retrieves the `queryOperation` and entry key and value from the `CqEvent` and then updates the screen according to the operation type provided in `queryOperation`.

Example 14.1 CqListener Implementation

```
// CqListener class
public class TradeEventListener implements CqListener {
   public void onEvent(CqEvent cqEvent) {
      // com.gemstone.gemfire.cache Operation associated with the query op
      Operation queryOperation = cqEvent.getQueryOperation();
      // key and new value from the event
      Object key = cqEvent.getKey();
      TradeOrder tradeOrder = (TradeOrder)cqEvent.getNewValue();
      if (queryOperation.isUpdate()) {
         // update data on the screen for the trade order
         . . .
      }
      else if (queryOperation.isCreate()) {
         // add the trade order to the screen
         . . .
      }
      else if (queryOperation.isDestroy()) {
         // remove the trade order from the screen
         . . .
      }
   }
   public void onError(CqEvent cqEvent) {
      // handle the error
   }
   // From CacheCallback
   public void close() {
      // close the output screen for the trades
      . . .
   }
}
```

CQ events do not change your client cache. They are provided as an event service only. This allows you to have any collection of CQs without storing large amounts of data in your regions. If you need to persist information from CQ events, program your listener to store the information where it makes the most sense for your application.

> *Be very careful if you choose to update your cache from your* `CqListener`*. If your listener updates the region that is queried in its own CQ and that region has a* `Pool` *named, the update will be forwarded to the server. If the update on the server satisfies the same CQ, it may be returned to the same listener that did the update, which could put your application into an infinite loop. This same scenario could be played out with multiple regions and multiple CQs, if the listeners are programmed to update each other's regions.*

The CqEvent Object

The `CqEvent` contains this information:

- Entry key and new value.
- Base operation that triggered the cache event in the server. This is the standard `Operation` class instance used for cache events in GemFire Enterprise.
- `CqQuery` object associated with this CQ event.
- Query operation associated with this CQ event. This operation describes the change affected to the query results by the cache event. Possible values are
  - `CREATE`, which corresponds to the standard database `INSERT` operation
  - `UPDATE`
  - `DESTROY`, which corresponds to the standard database `DELETE` operation
- `Throwable` object, returned if an error occurred running the `CqQuery` for the cache event. This is non-null only for `CqListener onError` calls.

This table describes the query operation based on whether the old and new entry values in the region entry event satisfy the query criteria.

Table 14.1 Query Operation Based on Old and New Entry Values

Old Entry Value	New Entry Value	Query Operation
No value or value does not satisfy the query criteria	No value (operation is invalidate or destroy) or value does not satisfy the query	N/A - no event
	Value satisfies the query	create
Value satisfies the query	No value (operation is invalidate or destroy) or value does not satisfy the query	destroy
	Value satisfies the query	update

You can use the query operation to decide what to do with the `CqEvent` in your listeners. For example, a `CqListener` that displays query results on screen might stop displaying the entry, start displaying the entry, or update the entry display depending on the query operation.

When an Error Occurs in a Running CQ

When an error occurs in CQ execution on the server, specific information on the error itself is stored in the server's log file. The server passes the event information to the client and the client's `CqListener onError` method is invoked. From the `onError` method, a call to the `CqEvent.getThrowable` method returns an error string of this type:

```
Exception while processing the Event. Please see server log for more
detail.
```

The server log will contain an error with "`Error while processing CQ`" like this.

```
[error 2007/12/18 12:03:18.903 PST gemfire1 <RMI TCP Connection(2)-
10.80.10.91> tid=0x18] Error while processing CQ on the event, key :
key-1, CqName :testCQEvents_0, ClientId :
identity(carlos(3249):52623/35391,connection=1,durableAttributes=null)
Error :No public attribute named 'ID' was found in class java.lang.Object
```

Usually, errors in CQ execution are caused by data errors, like invalid object types that are stored in the server region. In this case, the query is trying to read into an `Object` of type `Portfolio` for an entry where an empty object has been stored. You can avoid these types of errors by placing constraints on the region entries or by otherwise controlling the types of objects stored in your server regions.

Running the Continuous Query Code

Create your CQ from an instance of the `QueryService`. Once created, the CQ is maintained primarily through the `CqQuery` interface. This example shows the basic calls in the CQ life cycle.

Example 14.2 CQ Creation, Execution, and Close

```
// Get cache and queryService - refs to local cache and QueryService
// Create client /tradeOrder region configured to talk to the server

// Create CqAttribute using CqAttributeFactory
CqAttributesFactory cqf = new CqAttributesFactory();

// Create a listener and add it to the CQ attributes
//callback defined below
CqListener tradeEventListener = new TradeEventListener();
cqf.addCqListener(tradeEventListener);
CqAttributes cqa = cqf.create();

// Name of the CQ and its query
String cqName = "priceTracker";
String queryStr = "SELECT * FROM /tradeOrder t where t.price > 100.00";

// Create the CqQuery
CqQuery priceTracker = queryService.newCq(cqName, queryStr, cqa);

try {
   // Execute CQ
   priceTracker.execute();
   } catch (Exception ex){
      ex.printStackTrace();
   }
   // Now the CQ is running on the server, sending CqEvents to the listener
   . . .
}
// End of life for the CQ - clear up resources by closing
priceTracker.close();
```

Executing with Initial Results

CQ execution can be done with or without an initial result set, by calling `CqQuery.execute` or `CqQuery.executeWithInitialResults`. The initial `CqResults` returned from `executeWithInitialResults` is the same as the one you would get if you ran the query ad hoc, by calling `QueryService.newQuery.execute` on the server cache, but with the key added.

Example 14.3 Handling the CqResults

```
CqQuery EOTracker = cqService.newCq(cqName1, queryStr1, cqa1);
CqResults results = EOTracker.executeWithInitialResults();
for (Object o : results.asList()) {
   Struct s = (Struct)o;             // Struct with Key, value pair.
   System.out.println("key: " + s.get("key") + " value: " + s.get("value"));
}
```

If you are managing a data set from the CQ results, you can initialize the set by iterating over the initial result set and then updating it from your listeners as events arrive. For example, you might populate a new screen with initial results and then update the screen from a listener.

Just as with the standalone query, the initial `CqResults` represents a possibly moving snapshot of the cache. To prevent loss of events, the server begins managing CQ events when it begins creating the result set. After the execute call returns, the server starts sending events. Due to possible overlap between the initial results processing and event enqueueing, the client may receive events that are already represented in the result set.

Managing Your CQs

This section discusses generally how to access and manage your CQs from your client. The calls discussed here are all executed specifically for the calling client. A client cannot access or modify the CQs belonging to another client. For detailed method usage, see the online Java documentation.

Accessing Your CQs

You can access a single named CQ, an array of all CQs registered for a particular region, and an array of all CQs registered in the client using the `QueryService` `getCq*` methods. You can also access the CQ used to produce a `CqEvent` through the `CqEvent.getCq` method.

Accessing CQ Statistics

CQ runtime statistics are available for the client through the `CqServiceStatistics` and `CqStatistics` interfaces described under *The GemFire CQ API* on page 400. You can get information on the events generated by a specific CQ from the `CqStatistics` object returned by `CqQuery.getStatistics`. You can get higher-level information about the CQs the client has registered, running, and so on, from the `CqServiceStatistics` object returned by `QueryService.getCqStatistics`.

For both the client and server, you can access these statistics by loading the statistics archive file into VSD. The client's statistics are for the single client only. The server's pertain to all clients with CQs on this server.

Modifying CQ Attributes

You can modify the attributes for an existing CQ using the methods provided by `CqQuery.getCqAttributesMutator`. The attributes consist of a list of listeners.

Executing CQs

Individual CQs are executed using `CqQuery execute*` methods. You can also execute all CQs for the client or for a region through the client `QueryService`. CQs that are running can be stopped or closed.

Stopping CQs

Individual CQs are stopped using the `CqQuery` stop method. You can also stop all CQs for the client through the `QueryService`. Stopped CQs are in the same state as new CQs that have not yet been executed. You can close or execute a stopped CQ.

Closing CQs

Individual CQs are closed using the `CqQuery` method `close`. You can also close all CQs for the client through the `QueryService`. Closed CQs cannot be executed. CQs are also closed in the following cases:

- **The client closes its cache before closing all of its CQs**—Closing the client cache closes the `QueryService` and all associated CQs on the client and server.
- **The client disconnects from its server**—This might be because of a network outage or some other failure. When a client disconnects all the CQs created by the client are removed from the server and put into a `CLOSED` state on the client.

The server region is destroyed—When a server region is destroyed, all associated CQs are also cleaned up on the server and the region destroy event is sent to the client. On the client, the `CqListener.close` method is called for all CQs on the region.

Chapter 15

GemFire Transactions

This chapter describes the GemFire transactions feature. A transaction is a sequential group of data manipulation operations, performed as if it were one operation. A transaction defines a logical unit of work that either completely succeeds or produces no result at all. A distributed transaction is a transaction that accesses and updates data on two or more networked resources, and so must be coordinated among those resources. Transactions have been supported in relational database systems for ages, and even though the nature of data operations in GemFire Enterprise® is somewhat different, the underlying principles and use patterns are the same.

GemFire Enterprise® supports transactions that provide atomicity, consistency, and isolation.

In addition to explicit API support for committing and rolling back transactional operations, GemFire implements the JTA interface, so it can be used in transactions involving external data sources and application servers.

GemFire can be used as a JTA transaction coordinator or it can enlist in transactions controlled by an external JTA coordinator.

In this chapter:

- The Transactions API (page 410)
- GemFire Transaction Semantics (page 411)
- Cache Design for Transactions (page 413)
- Implementing GemFire Transactions (page 414)
- How GemFire Transactions Work (page 416)
- Using JTA transactions with GemFire (page 435)

15.1 The Transactions API

The GemFire API for distributed transactions has the familiar relational database methods, begin, commit, and rollback.

This example begins a transaction, updates two replicated regions, `cash` and `trades`, and commits. If the `commit` fails, it throws a `CommitConflictException` and the transaction is rolled back.

Example 15.1 Running a Transaction

```
Cache c = CacheFactory.create(DistributedSystem.connect(null));
AttributesFactory af = new AttributesFactory();
af.setDataPolicy(DataPolicy.REPLICATE);
Region<String, Integer> cash = c.createRegion("cash", af.create());
Region<String, Integer> trades = c.createRegion("trades", af.create());
CacheTransactionManager txmgr = c.getCacheTransactionManager();
try {
   txmgr.begin();
   final String customer = "Customer1";
   final Integer purchase = Integer.valueOf(1000);
// Decrement cash
   Integer cashBalance = cash.get(customer);
   Integer newBalance =
      Integer.valueOf((cashBalance != null ? cashBalance : 0)
         - purchase);
   cash.put(customer, newBalance);
// Increment trades
   Integer tradeBalance = trades.get(customer);
   newBalance =
      Integer.valueOf((tradeBalance != null ? tradeBalance : 0)
         + purchase);

   trades.put(customer, newBalance);
   txmgr.commit();
}
catch (CommitConflictException conflict) {
   txmgr.rollback();
}
```

This is the core GemFire transactions API:

com.gemstone.gemfire.cache

- **CacheTransactionManager**—Manages transactions and transaction plug-ins for the cache.
- **TransactionEvent**—Work done in the transaction.
- **TransactionId**—Unique GemFire transaction identifier.
- **TransactionWriter—**Application plug-in called before the commit. Usually used to synchronize with a backend data source.
- **TransactionListener—**Application plug-in for handling transaction events after the commit.

This simple API masks a lot of behind-the-scenes details that are carried out to ensure the transaction completes with the familiar semantics of Atomicity, Consistency, and Isolation.

15.2 GemFire Transaction Semantics

This section compares GemFire transaction semantics to the semantics of traditional relational databases.

Atomicity

A common database technique to achieve atomicity is two-phase locking on rows. This provides coordination between the current transaction and other transactions with overlapping changes. Holding write locks prevents reads of transactional changes until all changes are complete, providing all or nothing behavior. If problems occur during a transaction, perhaps due to relational constraints or storage limitations, the locks protect other processes from reading until the problems can be resolved. The resolution often involves reverting data back to the original pre-transaction state. Rolling back a database transaction in this case would require releasing write locks.

A common distributed technique for all or nothing behavior is the Two Phase Commit Protocol, which requires all processes participating in the transaction to agree to commit or abort the transaction.

The all or nothing behavior of GemFire transactions is implemented using a per thread transaction state which collects all transactional changes. The set of `Entry` changes are atomically reserved by a reservation system. The reservation blocks other, intersecting transactions from proceeding, allowing the commit to check for conflicts and reserve resources in an all-or-nothing fashion prior to making changes to the data.

Since GemFire is distributed, the transaction state is distributed to the appropriate members and processed in a batch, to protect against VM failure in the middle of the commit. After all changes have been made, locally and remotely, the reservation is released.

Rolling back the transaction is as simple as discarding the transaction state.

Reads do not use the reservation system and so are very fast.

This system has traits similar to the Multiversion Concurrency Control algorithm:

- Each thread can write new versions of `Entry` data without impacting read performance
- Reads view only the most recent committed version of an entry

Consistency

Database consistency is often described in terms of referential integrity. References between tables are often described in terms of primary and foreign key constraints. A database transaction that updates multiple columns, potentially in different tables, must ensure that all updates are completed to maintain consistency in an atomic fashion.

For GemFire, the application controls the consistency between regions and their entries. GemFire transactions use the reservation system to ensure that changes to all entries in all regions are committed. Once a reservation is obtained, GemFire does a conflict check to ensure that the entries scheduled for change are in the same state as they were when the transaction first accessed them.

Isolation

GemFire isolates transactions at the process thread level. So while a transaction is in progress, its changes are only visible inside the thread that is running the transaction. Threads inside the same process and in other processes do not see the changes until after the commit finishes. Relational database transactions often isolate changes per JDBC connection.

GemFire reads, like `get` and `getEntry`, are volatile, meaning they do not use locks. This makes reads very fast with respect to other entry operations. It also allows transactional writes to proceed without being slowed down by in-progress reads.

Visibility, related to isolation, is the atomic nature in which committed writes are viewable by read operations. Reads during a GemFire transaction have *repeatable read* isolation, so once the committed value reference is read for a given key, it always returns the same reference. If a transaction write, like `put` or `invalidate`, deletes a value for a key that has already been read, subsequent reads return the transactional reference.

Durability

Relational databases provide durability by using disk storage for recovery and transaction logging. GemFire does not support on-disk or in-memory durability for transactions.

15.3 Cache Design for Transactions

You can run transactions on any type of cache region except client regions, persistent regions, and regions with global scope. Transaction with any of these throws `UnsupportedOperationExeption`.

> *This chapter uses replicated and partitioned regions for all discussions. If you use non-replicated distributed regions, follow the guidelines for replicated regions.*

Performance

The most common region configurations for use with transactions are distributed replicated and partitioned.

- Replicated regions are better suited for running transactions on small to mid-size data sets. To ensure all or nothing behavior, at commit time, distributed transactions use the global reservation system of the GemFire distributed lock service. This works well as long as the data set is reasonably small.
- Partitioned regions are the right choice for highly-performant, scalable operations. Transactions on partitioned regions use only local locking, and only send messages to the redundant data stores at commit time. Because of this, these transactions perform much better than distributed transactions. There are no global locks, so partitioned transactions are extremely scalable as well.

Data Location

Transactions must be run on a data set hosted entirely by one member.

- For replicated or other distributed regions, the transaction uses only the data set in the member where the transaction is run.
- For partitioned regions, you must colocate all your transactional data in a single member.
- For transactions run on partitioned and distributed region mixes, you must colocate the partitioned region data and make sure the distributed region data is available in any member hosting the partitioned region data.

For transactions involving partitioned regions, any member with the regions defined can run the transactional application, regardless of whether the member hosts data for the regions. If the transactional data resides on a remote member, the transaction is carried out by proxy in the member hosting the data.

Transactions and Functions

You can run a function inside a transaction and you can run a transaction inside a function, as long as your combination of functions and transactions does not result in nested transactions. You cannot call a function from inside a transaction if the function runs its own transaction.

Clients can run transactions in servers by calling functions that run transactions.

15.4 Implementing GemFire Transactions

You can run transactions on the application's distributed system or on a server distributed system. If you run your transaction inside a function in the server distributed system, the steps given here apply to the server system.

1. Choose whether to run using the GemFire cache transaction manager or using the JTA transaction manager. The JTA manager, obtained through the JTA `UserTransaction` interface, manages the GemFire transaction and any configured JDBC connection. The portions of this documentation pertaining to JTA assume you understand how to use JTA transactions
2. Configure the cache `copy-on-read` behavior in the members hosting the transactional data, or perform cache updates that avoid in-place changes. See *Safe Entry Modification* on page 140. This allows the transaction manager to control when your cache updates are visible outside the transaction.
3. In the members hosting the transactional data, configure your regions for your transactions:

   3.1 For replicated regions, use `distributed-ack scope`. This is particularly important if you have more than one data producer. With one data producer, you can safely use `distributed-no-ack`.

   3.2 For partitioned regions, custom partition and colocate data between regions so all the data for any single transaction is hosted by a single data store. See *Colocating Data in Partitioned Regions* on page 182. If you run the transaction from another member, it will run by proxy in the member hosting the data. The partitioned region must be defined for the application that runs the transaction, but the data can be hosted in a remote member.

   3.3 For mixed partition and replicated region transactions, make sure your replicated regions are hosted in every member that hosts the partitioned region data. All data for a single transaction must reside in a single host.

   3.4 If you use delta propagation, set `cloning-enabled` (page 332) to `true`. This lets GemFire do conflict checks at commit. Without this, when you run the transaction, you'll get this exception:

   ```
   UnsupportedOperationInTransactionException("Delta without
   cloning cannot be used in transaction");
   ```

   3.5 If you use JTA and want any regions to not participate in the transactions, set the region attribute `ignore-jta` to `true`. It is `false` by default.
4. Update your event handler code in your data hosts to handle your transactions. For all your event handlers, refer to the general steps at *Implementing Cache Event Handlers* on page 300.

   4.1 Cache event handlers are all used for transactions. Cache listeners are called after the commit, instead of after each cache operation, and they receive the conflated transaction events. Cache writers and loaders are called as usual, as the operations are done. follow these additional guidelines:

   - Make sure cache callbacks are transactionally aware. A transactional operation can launch callbacks that are not transactional.
   - Make sure cache listeners will operate properly with the entry event conflation done for transactional operations. See *How Transaction Events Are Managed* on page 417.

4.2 Transaction event handlers are cache-wide. You can install one transaction writer and any number of transaction listeners.

- Program with synchronization for thread-safety. Your listener and writer methods may be called simultaneously by different threads for different transactions.
- Keep your transactional callback implementations lightweight and avoid doing anything that might cause them to block.

5. Program to retrieve the transaction manager.

- GemFire cache transaction manager:

```
CacheTransactionManager txManager =
    custRegion.getCache().getCacheTransactionManager();
```

- JTA transaction manager:

```
Context ctx = cache.getJNDIContext();
UserTransaction txManager =
    (UserTransaction)ctx.lookup("java:/UserTransaction");
```

6. Program to run your transaction.

```
try {
    txManager.begin();
    // ... do work
    txManager.commit();
} catch (CommitConflictException conflict) {
    txManager.rollback();
}
```

6.1 Start each transaction with a `begin` operation.

6.2 Run the GemFire and other operations that you want included in the transaction.

- If your transaction runs on a mix of partitioned and replicated regions, perform your first transaction operation on a partitioned region. This sets the data host for the entire transaction.
- If you did not configure `copy-on-read` to `true`, be sure your cache updates avoid in-place changes. See *Safe Entry Modification* on page 140.
- Take into account the behavior of transactional and non-transactional operations. All transactional operations that are run after the `begin` and before the `commit` or `rollback` are included in the transaction. See *Transactional Versus Non-transactional Operations* on page 426.

6.3 End each transaction with a `commit` or a `rollback`.

Do not leave any transaction in an uncommitted and unrolled back state. Transactions do not time out, so will remain in the system for the life of your application.

7. Review all of your code for compatibility with transactions. When you commit a transaction, while the commit is taking place, the changes are visible in the distributed cache. This is also known as *in transition* commits. This provides better performance than locking everything to do the transaction updates, but it means that another process accessing data used in the transaction might get some data in the pre-transaction state and some in the post-transaction state.

For example say key `1` and `2` are written to in a transaction so both of their values change from `A` to `B`. In another thread, it is possible to read key `1` with value `B` and key `2` with value `A`, while the transaction is being committed. This is possible due to the nature of GemFire reads. This choice sacrifices atomic visibility in favor of performance; reads don't block writes, writes don't block reads.

15.5 How GemFire Transactions Work

All the regions in a member cache can participate in a transaction. A Java application can operate on the cache using multiple transactions. A transaction is associated with only one thread, and a thread can operate on only one transaction at a time. Child threads do not inherit existing transactions.

Figure 15.1 Transactions Operating on the GemFire Cache

The Transaction View

A transaction is isolated from changes made concurrently to the cache. Each transaction has its own private view of the cache, including the entries it has read and the changes it has made. The first time the transaction touches an entry in the cache, either to read or write, it produces a snapshot of that entry's state in the transaction's view. The transaction remembers the entry's original state and uses it at commit time to discover write conflicts. The transaction maintains its current view of the entry, which reflects only the changes made within the transaction.

Figure 15.2 Transactional View of the Cache

Committing

When a commit succeeds, the changes recorded in the transaction view are merged into the cache. If the commit fails or the transaction is rolled back, all of its changes are dropped.

When a transaction is committed, the transaction management system uses a two-phase commit protocol:

1. Reserves all the entries involved in the transaction from changes by any other transactional thread. For distributed regions, it reserves the entries in the entire distributed system. For partitioned regions, it reserves them on the data host, where the transaction is running.
2. Checks the cache for conflicts on affected keys, to make sure all entries are still in the same state they were in when this transaction first accessed them.

   If any conflict is detected, the manager rolls back the transaction.

   If no conflict is detected, the manager:

   2.1 Calls the `TransactionWriter` in the member where the transaction is running. This allows the system to write through transactional updates to an external data source.

   2.2 Updates the local cache and distributes the updates to the other members holding the data. Cache listeners are called for these updates, in each cache where the changes are made, the same as for non-transactional operations.

   2.3 Calls the `TransactionListener` in the member where the transaction is running.
3. Releases the transaction reservations on the entries.

The manager updates the local cache and distributes the updates to other members in a non-atomic way.

- If other threads read the keys the transaction is modifying, they may see some in their pre-transaction state and some in their post-transaction state.
- If other, non-transactional, sources update the keys the transaction is modifying, the changes may intermingle with this transaction's changes. The other sources can include distributions from remote members, loading activities, and other direct cache modification calls from the same member. When this happens, after your commit finishes, the cache state may not be what you expected.

If the transaction fails to complete any of the steps, a `CommitConflictException` is thrown to the calling application.

Once the members involved in the transaction have been asked to commit, the transaction completes even if one of the participating members were to leave the system during the commit. The transaction completes successfully so long as all remaining members are in agreement.

Each member participating in the transaction maintains a membership listener on the transaction coordinator. If the transaction coordinator goes away after issuing the final commit call, the transaction completes in the remaining members.

How Transaction Events Are Managed

Transactional cache operations are handled somewhat differently inside transactions than out.

During the Transaction

While the transaction is running, each transactional operation is passed to the cache writer local to the transactional view, if one is available. As with cache operations outside of transactions, the cache writer can abort the operation.

Each operation the cache writer allows is applied to the transactional view in the cache and appended to the `CacheEvent` list in the `TransactionEvent` object.

Event Conflation

The cache events are conflated, so if a key already has an event in the list, that event is removed and the current operation is added to the end of the list.

So this series of calls inside a transaction:

```
Region.create (A, W);
Region.put (A, valX);
Region.put (B, valQ);
Region.invalidate (A);
Region.put (A, valY);
```

results in these events stored in the `CacheEvent` list, with only the most recent operation left:

```
put (B, valQ)
put (A, valY)
```

At commit and After

When the transaction is committed, GemFire passes the `TransactionEvent` to the transaction writer local to the transactional view, if one is available.

After commit, GemFire:

- Passes the `TransactionEvent` to each installed transaction listener.
- Walks the `CacheEvent` list, calling all locally installed listeners for each operation listed.
- Distributes the `TransactionEvent` to all interested caches.

  For GemFire and global JTA transactions, the `EntryEvent`*s contain the GemFire transaction ID. JTA transaction events do not contain the JTA transaction ID.*

Application Plug-Ins With Transactions

All of the standard GemFire application plug-ins work with transactions, and the transaction interface offers other plug-ins that support transactional operation.

Transaction Plug-Ins

- TransactionWriter—When you commit a transaction, if a transaction writer is installed in the cache where the data updates were performed, it is called. The writer can do whatever work you need, including aborting the transaction.
- TransactionListener—When the transaction ends, its thread calls the transaction listener to perform the appropriate follow-up for successful commits, failed commits, or voluntary rollbacks. The transaction that caused the listener to be called no longer exists by the time the listener code executes.

  Transaction listeners have access to the transactional view and thus are not affected by non-transactional update operations. `TransactionListener` methods cannot make transactional changes or cause a rollback. They can, however, start a new transaction. Multiple transactions on the same cache can cause concurrent invocation of `TransactionListener` methods, so implement methods that do the appropriate synchronizing of the multiple threads for thread-safe operation.

  A transaction listener can preserve the result of a transaction, perhaps to compare with other transactions, or for reference in case of a failed commit. When a commit fails and the transaction ends, the application cannot just retry the transaction, but must build up the data again. For most applications, the most efficient action is just to start a new transaction and go back through the application logic again.

The rollback and failed commit operations are local to the member where the transactional operations are run. When a successful commit writes to a distributed or partitioned region, however, the transaction results are distributed to other members the same as other updates. The transaction listener on the receiving members reflect the changes the transaction makes in that member, not the originating member. Any exceptions thrown by the transaction listener are caught by GemFire and logged.

Cache Plug-Ins

- CacheLoader—When a cache loader is called by a transaction operation, values loaded by the cache loader may cause a write conflict when the transaction commits.
- CacheWriter—During a transaction, if a cache writer exists, its methods are invoked as usual for all operations, as the operations are called in the transactions. The `netWrite` operation is not used. The only cache writer used is the one in the member where the transactional data resides.
- CacheListener—The cache listener callbacks - local and remote - are triggered after the transaction commits. The system sends the conflated transaction events, in the order they were stored.

  The `EntryEvent` that the callbacks receive has a unique GemFire transaction ID, so the cache listener can associate each event, as it occurs, with a particular transaction. `EntryEvents` that are not part of a transaction return `null` instead of a transaction ID.

Transactions in the Distributed Cache

The data scope of a GemFire cache transactional view is the cache that hosts the transactional data. For partitioned regions, this may be a remote host to the host running the transaction application. Otherwise, the host is the same one running the application. A transaction is managed on a per-cache basis, so multiple regions in the cache can participate in a single transaction.

Transactions and Partitioned Regions

In partitioned regions, transaction operations are done first on the primary data host then distributed to other members from there, regardless of which member initializes the cache operation. This is the same as normal cache operations.

- The member running the transaction code is called the transaction initiator.
- The member that hosts the data—and the transaction—is called the transactional data host.

So the transactional data host may be local or remote to the transaction initiator. In either case, when the transaction commits, data distribution is done from the transactional data host in the same way.

In this figure, M_1 runs two transactions.

- The first, T1, works on data whose primary buckets are hosted by, M_1, so M_1 is both initiator and data host for the transaction.
- The second transaction, T2, works on data whose primary buckets are hosted by M_2, so M_1 is the transaction initiator and M_2 is the transactional data host.

Figure 15.3 Transactions on a Partitioned Region

The transaction is managed on the data host. This includes the transactional view, all operations, and all local cache event handling. In the figure, when T2 is committed, the cache on M_2 is updated and the transaction events distributed throughout the system, exactly as if the transaction had originated on M_2.

The first region operation in the transaction determines the transactional data host. All other operations must also work with that as their transactional data host:

- All partitioned region data managed inside the transaction must use the transactional data host as their primary data store. In the figure above, if transaction T1 tried to put entry W, it would get a `TransactionDataNotColocatedException`.
- All non-partitioned region data managed inside the transaction must be available on the transactional data host and must be distributed. Operations on regions with local scope are not allowed in transactions with partitioned regions.

The next figure shows a transaction that uses two partitioned regions and one replicated region. As with the single region example, all local event handling is done on the transactional data host.

For a transaction in these data keys to work, the first operation must be on one of the partitioned regions, to establish M_2 as the transactional data host. Running the first operation on a key in the replicated region would establish M_1 as the transactional data host, and subsequent operations on the partitioned region data would fail with a `TransactionDataNotColocated` exception.

Figure 15.4 Transactions on a Partitioned Region with Other Regions

Transactions and Replicated Regions

For replicated regions, the transaction and its operations are applied to the local member and the resulting transaction state is distributed to other members according to the attributes of each region.

> *If possible, use* `distributed-ack scope` *for your regions where you will run transactions.*

For distributed regions, the region's scope affects how data is distributed during the commit phase. Transactions are supported for these region scopes:

- **local**—No distribution, handles transactional conflicts locally. Not compatible with partitioned regions.
- **distributed-no-ack**—Handles transactional conflicts locally, less coordination between members.
- **distributed-ack**—Handles transactional conflicts both locally and between members.

Local Scope

Transactions on regions with `local` scope have no distribution, but they perform conflict checks in the local member. You can have conflict between two threads when their transactions change the same entry, like object `Y` in this figure. In this case, the first transaction to start the commit process "wins." The other transaction's commit fails with a conflict, and its changes are dropped.

Figure 15.5 Transactional Conflict with Local Scope

You can also have a conflict when a non-transactional operation and a transaction modify the same entry. In that case, the transaction's commit fails. For a definition of a non-transactional operation, see *Transactional Versus Non-transactional Operations* on page 426.

distributed-ack Scope

The `distributed-ack` scope is designed to protect data consistency. This scope provides the highest level of coordination among transactions in different members. When the `commit` call returns for a

transaction run on all `distributed-ack` regions, you can be sure that the transaction's changes have already been sent and processed. In addition, any callbacks in the remote member have been invoked.

In this figure, notice that even after the commit operation is launched, the transaction continues to exist during the data distribution (step 3). The commit does not complete until the changes are made in the remote caches and the application in M_1 receives the callbacks to verify that the tasks are complete.

Figure 15.6 Conflicting Transactions in distributed-ack Regions with Multiple Writers

1. Before commit. *Transactions* T1 *and* T2 *each change the same entries in Region B in their local cache.* T1 *also makes changes to Region A.*

2. Conflict detected and eliminated. *The distributed system recognizes the potential conflict from Transactions* T1 *and* T2 *using the same entry.* T1 *started to commit first, so it is allowed to continue.* T2's *commit failed with a conflict.*

3. Changes in transit. *Transaction* T1 *commits and its changes are merged into the local cache. The commit does not complete until GemFire also distributes the changes to the remote regions and acknowledgement is received.*

4. After commit. *Region A in* M_2 *and Region B in* M_3 *reflect the changes from transaction* T1*, and* M_1 *has received acknowledgement. Results may not be identical in different members if their region attribute settings, such as expiration, are different.*

distributed-no-ack Scope

If you need faster transactions with distributed regions, `distributed-no-ack` scope gives the best performance. This scope is appropriate for:

- applications with multiple writers that write to different keys
- applications with only one writer

> *Do not use* `distributed-no-ack` *scope with multiple writers that work on the same data sets.*

This figure shows how using the no-ack scope can produce unexpected results. These two transactions are operating on the same data point, in region B. Since they use no-ack scope, the conflicting changes cross paths and leave the data in an inconsistent state.

Figure 15.7 Conflicting Transactions in distributed-no-ack Regions with Multiple Writers

Transactional Versus Non-transactional Operations

Between the `begin` operation and the `commit` or `rollback` operation are a series of ordinary GemFire operations. When they are launched from a transaction, the GemFire operations can be classified into two types:

- Transactional operations that affect the transactional view
- Non-transactional operations that do not affect the transactional view

An operation that acts directly on the `Cache` does not usually act on the transactional view.

Transactional Operations

The `CacheTransactionManager` methods are the only ones used specifically for cache transactions. Otherwise, you use the same GemFire methods as usual. Most methods that run within a transaction affect the transactional view and do not change the cache until the transaction commits. Methods that behave this way are considered *transactional operations*. Transactional operations are classified in two ways: whether they modify the transactional view or the cache itself, and whether they create write conflicts with other transactions.

In general, methods that create, destroy, invalidate, update, or read region entries are considered transactional. Transactional operations are standard GemFire operations that can operate either in the context of a transaction or independently.

Transactional operations that can cause write conflicts are those that modify an entry, such as put, a load done to satisfy a get operation, create, delete, local delete, invalidate and local invalidate.

Transactional read operations do not cause conflicts directly, but they can modify the transactional view. Read operations look for the entry in the transaction view first and then, if necessary, go to the cache. If the entry is returned by a cache read, it is stored as part of the transactional view. At commit time, the transaction uses the initial snapshot of the entry in the view to discover write conflicts. Read operations on collections behave differently. For details, see *Collections and Region.Entry Instances* on page 429.

Non-transactional Operations

A few methods, when running in the context of a transaction, have no effect on the transactional view but an immediate effect on the cache. They are considered *non-transactional operations*. Often, non-transactional operations are administrative, such as `Region.destroy` and `Region.invalidate`. These operations are not supported within a transaction. If you call them, the system throws an exception of this type: `UnsupportedOperationInTransactionException(destroyRegion() is not supported while in a transaction)`.

Entry Operations

Transactional `Entry` *operations can be rolled back.*

Operation	Methods	Transactional	Write Conflict
create	`Region.create`, `put`, `putAll` `Map.put`, `putAll`	yes	yes

modify	`Region.put`, `putAll`
	`Map.put`, `putAll`
	`Region.Entry.setValue`
	`Map.Entry.setValue`
		yes	yes

load	`Region.get` `Map.get`	yes	yes
creation using a `netSearch`	`Region.get` `Map.get`	yes	**no**
update using a `netSearch`	`Region.get` `Map.get`	yes	**no**
local destroy	`Region.localDestroy`	yes	yes

distributed destroy	`Region.destroy`
	`Region.remove`
	`Map.remove`
		yes	yes

local invalidate	`Region.localInvalidate`	yes	yes
distributed invalidate	`Region.invalidate`	yes	yes
set user attribute	`Region.Entry.setUserAttribute`	yes	yes
read	`Region.get`, `getEntry`, `containsKey`, `containsValue`, `containsValueForKey`, `keySet*`, `entrySet*`, `values*`	yes	**no**

*These collection operations become transactional reads only when you access the keys or values within the set. For details, see *Collections and Region.Entry Instances* on page 429.

Some transactional write operations also do a read before they write, and these can complete a transactional read even when the write fails. The following table of entry operations notes the conditions under which this can happen.

These operations can add a snapshot of an entry to the transaction's view even when the write operation does not succeed.

Operation	Methods	Reads Without Writing
create	`Region.create`	when it throws an `EntryExistsException`
local destroy	`Region.localDestroy`	when it throws an `EntryNotFoundException`
distributed destroy	`Region.destroy`	when it throws an `EntryNotFoundException`
local invalidate	`Region.localInvalidate`	when it throws an `EntryNotFoundException` or the entry is already invalid
distributed invalidate	`Region.invalidate`	when it throws an `EntryNotFoundException` or the entry is already invalid

Region Operations

When you create a region in a transaction, any data from the `getInitialImage` operation goes directly into the cache, rather than waiting for the transaction to commit.

Operation	Method	Affected	Write Conflict
local destroy	`Region.localDestroyRegion`	cache	yes
distributed destroy	`Region.destroyRegion`	cache	yes
close	`Region.close`	cache	yes
mutate attributes		cache	**no**
set user attributes	`Region.setUserAttribute`	cache	**no**
local clear	`Region.localClear`	cache and transaction	**no**
distributed clear	`Region.clear` `Map.clear`	cache and transaction	**no**
local invalidate	`Region.localInvalidateRegion`	cache	yes
distributed invalidate	`Region.invalidateRegion`	cache	yes

Cache Operations

Operation	Method	Affected State	Write Conflict
create	`createRegion`	committed	**no**
close	`close`	committed	yes

No-Ops

Any operation that has no effect in a non-transactional context remains a no-op in a transactional context. For example, if you do two `localInvalidate` operations in a row on the same region, the second `localInvalidate` is a no-op. No-op operations do not:

- Cause a listener invocation
- Cause a distribution message to be sent to other members
- Cause a change to an entry
- Cause a conflict

A no-op can do a transactional read.

Queries and Indexes

Queries and indexes reflect the cache and ignore the changes made by ongoing transactions. If you do a query from inside a transaction, the query does not reflect the changes you made inside that transaction.

Collections and Region.Entry Instances

Collections and region entries used in a transaction must be created inside the transaction. After the transaction has completed, the application can no longer use any region entry or collection or associated iterator created within the transaction. If you try to use these, you get an `IllegalStateException`.

Region collection operations include `Region.keySet`, `Region.entrySet`, and `Region.values`. You can create instances of `Region.Entry` through the `Region.getEntry` operation or by looking at the contents of the result returned by a `Region.entrySet` operation.

Eviction and Expiration Operation With Transactions

Entry expiration and LRU eviction affect the committed state. They are not part of the transaction, and they cannot be rolled back.

Eviction

LRU eviction operations do not cause write conflicts with existing transactions, despite destroying or invalidating entries. LRU eviction is deferred on entries modified by the transaction until the commit completes. Since anything touched by the transaction has had its LRU clock reset, eviction of those entries is not likely to happen immediately after the commit.

When a transaction commits its changes in a region with distributed scope, the operation can invoke eviction controllers in the remote caches, as well as in the local cache.

Expiration

Local expiration actions do not cause write conflicts, but distributed expiration can cause conflicts and prevent transactions from committing in the members receiving the distributed operation.

Using Local Expiration with Transactions

When you are using transactions on a region, make expiration local, if possible. For every instance of that region, configure an expiration action of local invalidate or local destroy. In a `cache.xml` declaration, use a line similar to this:

```
<expiration-attributes timeout="60" action="local-invalidate"/>
```

For details, see *Keeping Your Data Current With Expiration* on page 176.

In regions modified by a transaction, local expiration is suspended. Expiration operations are batched and deferred per region until the transactions complete. Once cleanup starts, the manager processes pending expirations. Transaction that need to change the region wait until the expirations are complete.

Using Distributed Expiration with Transactions

With partitioned and replicated regions, you cannot use local expiration. When you are using distributed expiration, the expiration is not suspended during a transaction, and expiration operations distributed from another member can cause write conflicts.

In replicated regions, you can avoid conflicts by setting up your distributed system this way:

1. Choose an instance of the region to drive region-wide expiration. Use a replicated region, if there is one.
2. Configure distributed expiration only in that region instance. The expiration action must be either invalidate or destroy. In a `cache.xml` declaration, use a line similar to this:

   ```
   <expiration-attributes timeout="300" action="destroy"/>
   ```

3. Run your transactions in the member where you configured expiration.

Monitoring and Troubleshooting Transactions

Unlike database transactions, GemFire does not write a transaction log to disk. To get the full details about committed operations, use a transaction listener to monitor the `TransactionEvents` and their contained cache `Events` for each of your transactions.

Statistics on Cache Transactions

During the operation of GemFire cache transactions, if statistics are enabled, transaction-related statistics are calculated and accessible from the `CachePerfStats` statistic resource. Because the transaction's data scope is the cache, these statistics are collected on a per-cache basis. See Appendix B, *System Statistics*, on page 235.

Commit

In a failed commit, the exception lists the first conflict that caused the failure. Other conflicts can exist, but are not reported. For more on the commit process, see *Committing* on page 417.

Capacity Limits

A transaction can create data above and beyond the capacity limit set in the region's eviction attributes. The capacity limit does not take effect until commit time. Then, any required eviction action takes place as part of the commit.

Interaction With the Resource Manager

The GemFire resource manager, which controls overall heap use, either allows all transactional operations or blocks the entire transaction. If a cache reaches the critical threshold in the middle of a commit, the commit is allowed to finish before the manager starts blocking operations. For information on the resource manager, see *Controlling Heap Use With the Resource Manager* on page 186.

Coding Examples

Example 15.2 Transaction Run in Multiple Regions

```
/**
 * An example for using GemFire Transactions.
 * Creates a Customer and Order partitioned region, and then
 * updates the customer and one customer order in a transaction.
*/
public class TransactionalPeer {
/** The region for storing customer information*/
   private Region<CustomerId, String> custRegion;
/** The region for storing order information of a customer*/
   private Region<OrderId, String> orderRegion;

   private static final int MAX_KEYS_IN_REGION = 10;

   public TransactionalPeer(boolean isEmpty) {
      createCache();
      createRegions(isEmpty);
   }

   /**
    * Connects to the distributed system and creates the cache
    */
   protected void createCache() {
      Properties props = new Properties();
      system = DistributedSystem.connect(props);
      cache = CacheFactory.create(system);
   }

   /**
    * Creates the customer and order regions
    * @param isEmpty true if these regions should not store data locally,
false otherwise
    */
   protected void createRegions(boolean isEmpty) {
      PartitionAttributesFactory paf =
         new PartitionAttributesFactory<CustomerId, String>();
      paf.setPartitionResolver(new CustomerOrderResolver());
      if (isEmpty) {
         paf.setLocalMaxMemory(0);
      }
      AttributesFactory af = new AttributesFactory<CustomerId, String>();
      af.setDataPolicy(DataPolicy.PARTITION);
      af.addCacheListener(new LoggingCacheListener());
      af.setPartitionAttributes(paf.create());
      custRegion = cache.createRegion("customer", af.create());

      paf = new PartitionAttributesFactory<OrderId, String>();
      paf.setColocatedWith("customer");
      paf.setPartitionResolver(new CustomerOrderResolver());
      if (isEmpty) {
         paf.setLocalMaxMemory(0);
      }
      af = new AttributesFactory<OrderId, String>();
      af.setDataPolicy(DataPolicy.PARTITION);
      af.addCacheListener(new LoggingCacheListener());
```

```
        af.setPartitionAttributes(paf.create());
        orderRegion = cache.createRegion("order", af.create());
    }

    /**
     * Runs the example by asking for user input for the type of transaction
     * to be run
     * @throws IOException if there is Exception while reading user input
     *
     */
    private void runExample() throws IOException {
        System.out.println("Please start the other VM and press enter to
populate regions");
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        reader.readLine();
        if (custRegion.size() == 0) {
            System.out.println("Populating region...");
            populateRegion();
            System.out.println("Complete");
        } else {
            System.out.println("Regions already populated");
        }
        Random random = new Random();
        while (true) {
            System.out.println("Press 1 to run a transaction, 2 to run a
transactional function");
            String input = reader.readLine();
            CustomerId custToUpdate = new
CustomerId(random.nextInt(MAX_KEYS_IN_REGION+1));
            OrderId orderToUpdate = new OrderId(random.nextInt(100),
custToUpdate);
            if ("1".equals(input.trim())) {
                CacheTransactionManager mgr =
custRegion.getCache().getCacheTransactionManager();
                System.out.println("Starting a transaction...");
                mgr.begin();
                int randomInt = random.nextInt(1000);
                System.out.println("for customer region updating
"+custToUpdate);
                custRegion.put(custToUpdate, "updatedCustomer_"+randomInt);
                System.out.println("for order region updating "+orderToUpdate);
                orderRegion.put(orderToUpdate, "newOrder_"+randomInt);
                mgr.commit();
                System.out.println("transaction completed");
            } else if ("2".equals(input.trim())) {
                System.out.println("Executing Function");
                Set filter = new HashSet();
                filter.add(custToUpdate);
                System.out.println("Invoking Function");
                //please refer to the function service documentation for more
information

FunctionService.onRegion(custRegion).withFilter(filter).withArgs(
                        orderToUpdate).execute(new
TransactionalFunction()).getResult();
                System.out.println("Function invocation completed");
            } else {
                continue;
```

```
            }
        }
    }

    /**
     * gets us started by putting some data in the customer
     * and order regions
     */
    private void populateRegion() {
        for (int i=0; i<MAX_KEYS_IN_REGION/2; i++) {
            CustomerId custId = new CustomerId(i);
            OrderId orderId = new OrderId(i, custId);
            custRegion.put(custId, "customer_"+i);
            orderRegion.put(orderId, "order_"+i);
        }
    }
    public static void main(String[] args) throws IOException {
        boolean isEmpty = false;
        if (args.length > 1) {
            showUsage();
            System.exit(1);
        } else if (args.length == 1) {
            if ("empty".equalsIgnoreCase(args[0])) {
                isEmpty = true;
            } else {
                showUsage();
                System.exit(1);
            }
        }
        TransactionalPeer peer = new TransactionalPeer(isEmpty);
        peer.runExample();
    }

    /**
     * prints the usage
     */
    private static void showUsage() {
        System.out.println("java TransactionalPeer [empty]");
    }
}
```

Example 15.3 Transactional Function

```
/**
 * This function does transactional updates to customer and order regions
 */
public class TransactionalFunction extends FunctionAdapter {

  private Random random = new Random();
/* (non-Javadoc)
   * @see
com.gemstone.gemfire.cache.execute.FunctionAdapter#execute(com.gemstone.gemf
ire.cache.execute.FunctionContext)
   */
  @Override
  public void execute(FunctionContext context) {
    RegionFunctionContext rfc = (RegionFunctionContext)context;
    Region<CustomerId, String> custRegion = rfc.getDataSet();
    Region<OrderId, String>
       orderRegion = custRegion.getCache().getRegion("order");
    CacheTransactionManager
       mgr = custRegion.getCache().getCacheTransactionManager();
    CustomerId custToUpdate = (CustomerId)rfc.getFilter().iterator().next();
    OrderId orderToUpdate = (OrderId)rfc.getArguments();
    System.out.println("Starting a transaction...");
    mgr.begin();
    int randomInt = random.nextInt(1000);
    System.out.println("for customer region updating "+custToUpdate);
    custRegion.put(custToUpdate,
       "updatedCustomer_"+custToUpdate.getCustId()+"_"+randomInt);
    System.out.println("for order region updating "+orderToUpdate);
    orderRegion.put(orderToUpdate,
       "newOrder_"+orderToUpdate.getOrderId()+"_"+randomInt);
    mgr.commit();
    System.out.println("transaction completed");
    context.getResultSender().lastResult(Boolean.TRUE);
  }

  /* (non-Javadoc)
   * @see com.gemstone.gemfire.cache.execute.FunctionAdapter#getId()
   */
  @Override
  public String getId() {
    return "TxFuntion";
  }

}
```

15.6 Using JTA transactions with GemFire

GemFire Enterprise provides an implementation of the Java Transaction API (JTA). JTA is an industry standard for global transactions controlling multiple resources. With JTA, you can write your transactions using the same API, whether they run in J2EE or not. Using GemFire is just like using a database, with the same begin and commit. All the application knows is the JTA level, and the transaction manager handles the rest.

The parties involved in a JTA transaction include:

- The Java application, responsible for starting the global transaction
- The JTA transaction manager, responsible for opening, committing, and rolling back transactions
- The transaction resource managers, including the GemFire cache transaction manager and the JDBC resource manager, responsible for managing operations in the cache and database

> *See the Sun documentation for more information on topics such as JTA,* `javax.transaction`*, committing and rolling back global transactions, and the related exceptions.*

The GemFire JTA implementation also supports the J2EE Connector Architecture (J2CA) `ManagedConnectionFactory`.

While a global transaction is running, each cache operation that would participate in a GemFire transaction tries to join through JTA synchronization. If the running global transaction has been marked for rollback, however, the GemFire operation is not allowed to join and the operation call throws a `FailedSynchronizationException`.

JTA provides direct coordination between the GemFire cache and another transactional resource, such as a database. Using JTA, your application controls all transactions in the same standard way, whether the transactions act on the GemFire cache, a JDBC resource, or both together. When a JTA global transaction is done, the GemFire transaction and the database transaction are both complete.

The GemFire JTA is initialized when the cache is initialized. If a global transaction exists when you use the cache, the cache operations automatically enlist themselves with the transaction, where appropriate. Operations on a region automatically detect and become associated with the existing global transaction through JTA synchronization. If the global transaction has been marked for rollback, any cache operation that attempts to enlist throws a `FailedSynchronizationException`.

The GemFire cache transaction's commit or rollback is triggered when the global transaction commits or rolls back. When the global transaction is committed using the `UserTransaction` interface, the transactions of any registered JTA resources are committed, including the GemFire cache transaction. If the cache or database transaction fails to commit, `UserTransaction` throws a `TransactionRolledBackException`. If a commit or rollback is attempted directly on a GemFire transaction that is registered with JTA, that action throws an `IllegalStateException`.

If there are cache writer and cache loader with transactional data sources, the writer and loader participate in the transaction. If the JTA rolls back its transaction, the changes made by the cache loader and the cache writer are rolled back.

You can disable JTA for any region by setting the region attribute `ignore-jta` to `true`.

Chapter 16

Function Execution

Using the GemFire Enterprise® function execution service, you can execute application functions on a single member, in parallel on a subset of members, or in parallel on all members of a distributed system. You achieve linear scalability by partitioning horizontally the application data such that concurrent operations by distributed applications can execute independently across host members. For example, if you restrict application requirements for transactions to a single member, and all required transaction data can be colocated to a single member or a small subset of members, you can achieve parallel execution by vectoring the concurrent accessors to the ever-growing number of partitioned region buckets.

Most scalable enterprise applications grow in data volume, where the number of data items managed rather than the size of each item grows over time. Assuming the preceding logic (especially true for OLTP class applications), you derive significant benefits by routing the data-dependent application code to the fabric member hosting the data. Routing of application code to the data fabric member of interest is *data-aware function routing*, or *behavior routing*.

In this chapter:

- Solutions and Use Cases (page 438)
- How Function Execution Works (page 439)
- Executing Functions in GemFire (page 446)

16.1 Solutions and Use Cases

The function execution service provides solutions for the following application use cases:

- An application that executes a server-side transaction or carries out data updates using the GemFire distributed locking service.
- An application that needs to initialize some of its components once on each server, which might be used later by executed functions.
- Initialization and startup of a third-party service, such as a messaging service.
- Any arbitrary aggregation operation that requires iteration over local data sets that can be done more efficiently through a single call to the cache server.
- Any kind of external resource provisioning that can be done by executing a function on a server.

16.2 How Function Execution Works

Where Functions are Executed

You specify the members that run the function or the data set over which to run the function.

- **Member**. Execute the function in your own distributed system or a server system, if you are connected to another system as a client. Execute a function in a single member of the system, in a subset of the members, or on all members. Specify data sets for this type of function through the arguments you pass in to the function.
- **Data set**. Specify a region and possibly a set of keys on which to run the function.

How Functions are Executed

1. When you call the `execute` method on the `Function Execution` object, the execution is sent to all members where it needs to run. The locations are determined by the `FunctionService on*` method calls, region configuration, and any filters. With `onRegion`, for a client region, the function is sent to the server system. For a non-client region, the function is handled in the calling application's distributed system.
2. If the function has results, the result is returned to the `execute` method call in a `ResultCollector` object.
3. The originating member collects results using `ResultCollector.getResult`.

Highly Available Functions

Generally, function execution errors are returned to the calling application. You can code for high availability for `onRegion` functions that return a result, so `GemFire` automatically retries a function if it does not execute successfully. To be highly available, the function must be coded and configured for it, and the calling application must invoke it using the results collector `getResult` method.

When a failure occurs (such as an execution error or member crash while executing), the system responds as follows:

- Waits for all calls to return
- Sets a boolean indicating a re-execution is being done
- Calls the result collector's `clearResults` method
- Executes the function

For client regions, the system retries the execution according to `com.gemstone.gemfire.cache.client.Pool retryAttempts`. If the function fails to run every time, the final exception is returned to the `getResult` method.

For member calls, the system retries until it succeeds or no data remains in the system for the function to operate on.

Data-Independent Function Execution

The figure shows the sequence of events for a data-independent function executed against all available servers.

Figure 16.1 Data-Independent Function Invoked from a Client

The next figure shows the sequence of events for a data-independent function executed against all available members in the calling application's distributed system:

Figure 16.2 Data-Independent Function Invoked on all Members

Data-Dependent Function Execution

The next figure shows a data-dependent function run by a client. The client region is connected to the server system, where the region is partitioned, so the function automatically goes to the server system to run against all servers holding data for the region.

Figure 16.3 Data-Dependent Function Invoked from a Client

The next figure shows the same data-dependent function with the added specification of a set of keys on which to run. Servers that do not hold any keys are left out of the function execution.

Figure 16.4 Data-Dependent Function with Filter Invoked from a Client

This data-dependent call is on a region that is not configured as a client region, so it runs in the caller's distributed system.

Figure 16.5 Data-Dependent Function with Filter Invoked from a Non-Client Region

This scenario demonstrates the steps in a call to a highly available function. The call fails the first time on one of the participating members and is successfully run a second time on all members.

Figure 16.6 Highly Available Data Dependent Function with Failure on First Execution

16.3 Executing Functions in GemFire

This section describes how to execute functions in GemFire. In the procedures that follow it is assumed that you have defined the members and regions where you will run functions.

Main Tasks

1. Code the `Function` implementation. If you need high availability, code and configure for that. For partitioned regions, if you need special handling of custom partitioned data and data colocated between regions, code for that.
2. If your function returns results and you need special results handling, code a custom `ResultsCollector` implementation.
3. Write the XML or application code to register the function, if needed (generally recommended).
4. Write the application code to run the function and, if the function returns results, to get the results from the collector.

See the Javadocs in `com.gemstone.gemfire.cache.execute`.

The following subsections describe how to execute these tasks:

- Writing the Function Code (page 447)
- Registering the Function (page 449)
- Running the Function From the Calling Application (page 450)
- Running the Function in Custom-Partitioned and Colocated Partitioned Regions (page 451)
- Storing and Retrieving Function Results (page 452)

Writing the Function Code

You can implement the `Function` interface or extend the `FunctionAdapter` class. Both are in the `com.gemstone.gemfire.cache.execute` package. The adapter provides some default implementations for methods, which you can override.

Code the methods you need for the function. These steps do not have to be done in this order.

1. Code `getId` to return a unique name for your function. You can use this name to access the function through the `FunctionService` API.
2. For high availability, code `isHa` to return `true` to indicate to GemFire that it can re-execute your function after one or more members fails. If you set `isHa` to `true`, your function must return a result and `hasResult` must also return `true`.
3. Code `hasResult` to return `true` if your function returns results to be processed and `false` if your function does not return any data - the fire and forget function. `FunctionAdapter hasResult` returns `true` by default.
4. If the function will be executed on a region, code `optimizeForWrite` to return `false` if your function only reads from the cache, and `true` if your function updates the cache. The method only works if, when you are running the function, the `Execution` object is obtained through a `FunctionService onRegion` call. `FunctionAdapter optimizeForWrite` returns `false` by default.
5. Code the `execute` method to perform the work of the function:

   5.1 Make `execute` thread-safe to accommodate simultaneous invocations.

   5.2 For high availability, code `execute` to accommodate multiple identical calls to the function. Use the `RegionFunctionContext isPossibleDuplicate` to determine whether the call may be a high-availability re-execution. This boolean is set to true on execution failure and is false otherwise.

   This boolean can be set following a failure from another member's execution of the function, so it only indicates that the execution might be a repeat run in the current member.

Example 16.1 Coding the Function execute Method

```
package quickstart;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import com.gemstone.gemfire.cache.execute.FunctionAdapter;
import com.gemstone.gemfire.cache.execute.FunctionContext;
import com.gemstone.gemfire.cache.execute.RegionFunctionContext;
import com.gemstone.gemfire.cache.partition.PartitionRegionHelper;

public class MultiGetFunction extends FunctionAdapter {

  public void execute(FunctionContext fc) {
    RegionFunctionContext context = (RegionFunctionContext)fc;
    Set keys = context.getFilter();
    Set keysTillSecondLast = new HashSet();
    int setSize = keys.size();
    Iterator keysIterator = keys.iterator();
    for(int i = 0; i < (setSize -1); i++)
    {
      keysTillSecondLast.add(keysIterator.next());
    }
    for (Object k : keysTillSecondLast) {
      context.getResultSender().sendResult(

(Serializable)PartitionRegionHelper.getLocalDataForContext(context)
              .get(k));
    }
    Object lastResult = keysIterator.next();
    context.getResultSender().lastResult(

(Serializable)PartitionRegionHelper.getLocalDataForContext(context)
            .get(lastResult));
  }

  public String getId() {
    return getClass().getName();
  }
}
```

In the preceding example:

The context provided to the function is the `FunctionContext`, which is automatically extended to `RegionFunctionContext` if you get the `Execution` object through a `FunctionService onRegion` call.

The `FunctionContext` holds the function ID, the `ResultSender` object for passing results back to the originator, and function arguments provided by the originator VM.

For data dependent functions, the `RegionFunctionContext` holds the `Region` object, the `Set` of key filters, and a boolean indicating multiple identical calls to the function, for high availability implementations.

For partitioned regions, use the `PartitionRegionHelper` to access additional information and data for the region. The additional information includes a view into the region and into the region and its colocated regions based on the function's region context. See the Javadocs for details.

Registering the Function

This section applies to functions that are invoked using the `Execution.execute(String functionId)` signature. When this method is invoked, the calling application sends the function ID to all members where the `Function.execute` is to be run. Receiving members use the ID to look up the function in the local `FunctionService`. In order to do the lookup, all of the receiving member must have previously registered the function with the function service.

The alternative to this method is `Execution.execute(Function function)`. When this method is invoked, the calling application serializes the instance of `Function` and sends it to all members where the `Function.execute` is to be run. Receiving members deserialize the `Function` instance, create a new local instance of it, and run execute from that. This is not available for non-Java client invocation of functions on servers.

Your Java servers must register functions that are invoked by non-Java clients. You may want to use registration in other cases to avoid the overhead of sending `Function` instances between members.

> *Modifying a function instance after registration has no effect on the registered function. If you want to execute a new function, you must register it with a different identifier.*

Register your functions using one of these methods:

Example 16.2 XML Function Registration

```
<cache>
    ...
    </region>
    <function-service>
       <function>"com.bigCompany.tradeService.cache.func.TradeCalc1"
       </function>
    </function-service/>
</cache>
```

Example 16.3 Java Function Registration

```
myFunction myFun = new myFunction();
FunctionService.registerFunction(myFun);
```

Running the Function From the Calling Application

Before you begin, follow the preceding steps for coding and registering the function.

In every member where you want to explicitly execute the function and process the results:

1. Use one of the `FunctionService on*` methods to create an `Execute` object.

   The `on*` methods, `onRegion`, `onMembers`, and so on, define the highest level at which the function is run. If you use `onRegion` you can further narrow your run scope by setting key filters. The function run using `onRegion` is referred to as a data-dependent function. Functions run with the other on* methods are data-independent functions.

2. Use the `Execution` object as needed for additional function configuration. You can:
   - Provide a key `Set` to `withFilters` to narrow the execution scope. This works only for `onRegion Execution` objects. You can retrieve these in your `Function execute` method through `RegionFunctionContext.getFilter`.
   - Provide function arguments to `withArgs`. You can retrieve these in your `Function` execute method through `FunctionContext.getArguments`.
   - Define a custom `ResultCollector`
3. Call the `Execution` object `execute` method to run the function.
4. To run a function with high availability, call `getResult` from the results collector returned from `execute`. Calling a highly available function without using `getResult` disables the high availability functionality.

Example 16.4 Running the Function - For Executing Members

```
MultiGetFunction function = new MultiGetFunction();
FunctionService.registerFunction(function);

writeToStdout("Press Enter to continue.");
stdinReader.readLine();

Set keysForGet = new HashSet();
keysForGet.add("KEY_4");
keysForGet.add("KEY_9");
keysForGet.add("KEY_7");

Execution execution = FunctionService.onRegion(exampleRegion)
    .withFilter(keysForGet)
    .withArgs(Boolean.TRUE)
    .withCollector(new MyArrayListResultCollector());

ResultCollector rc = execution.execute(function);
```

Running the Function in Custom-Partitioned and Colocated Partitioned Regions

This section pertains to partitioned regions where you have logically grouped your data in one or both of these ways:

- Colocation of data across multiple partitioned regions; the same buckets are stored together in host members. See *Colocating Data in Partitioned Regions* on page 182.
- Custom partitioning of data within a region; like data is stored in the same bucket. (Also see following Note.)

You can run your functions on particular key sets, either in one region or across colocated regions.

> *This process does not work with custom partitioning that is implemented using the callback argument. You must have implemented partitioning through entry keys or by assigning a custom partition resolver class to the region as a whole.*

Follow the steps in *Running the Function From the Calling Application* on page 450, adding these steps to access your colocated data.

1. In the application where you execute your function:
   1.1 Use `FunctionService onRegion` to specify the single region or any one of the colocated regions.
   1.2 Use `Execution.withFilters` to narrow the scope to the keys you are interested in
2. Inside your `Function.execute` method, use the `PartitionRegionHelper` to get the data you need:
   - For single regions, use `getLocalDataForContext`.
   - For colocated regions, use `getLocalColocatedRegions`.

Example 16.5 Example Using the PartitionRegionHelper in the Function Execute Method

```
public Serializable execute(FunctionContext context) {
    if (context instanceof RegionFunctionContext) {
        RegionFunctionContext rc = (RegionFunctionContext) context;
        if (PartitionRegionHelper.isPartitionedRegion(rc.getDataSet())) {
            Region efficientReader =
                  PartitionRegionHelper.getLocalDataForContext(rc);
            efficientReader.get("someKey");
            ...
        }
    }
...
```

Storing and Retrieving Function Results

This section applies only to functions that return results.

When you execute a function that returns results, the function stores the results into a `ResultCollector` and returns the `ResultCollector` object. The calling application can then retrieve the results through the `ResultCollector` `getResult` method:

```
ResultCollector rc = execution.execute(function);
List result = (List)rc.getResult();
```

GemFire's default `ResultCollector` collects all results into an `ArrayList`. Its `getResult` methods block until all results are received. Then they return the full result set.

If you do not wish to use the default collector, you can program your own `ResultCollector` code to collect and retrieve results:

1. Write a class that extends `ResultCollector` and code the methods to store and retrieve the results as you need. Note that the methods are of two types:
   - `addResult` and `endResults` are called by GemFire when results arrive from the `Function` instance `SendResults` methods.
   - `getResult` is available to your executing application (the one that calls `Execution.execute`) to retrieve the results.
2. Use high availability for `onRegion` functions that have been coded for it (see *Writing the Function Code* on page 447):
   - 2.1 Code the `ResultCollector` `clearResults` method to remove any partial results data. This readies the instance for a clean function re-execution.
   - 2.2 When you invoke the function, call the result collector `getResult` method. This enables the high availability functionality.
3. Use the `Execution` object in your executing member to call the `withCollector` method, passing it your custom collector. Example:

```
Execution execution = FunctionService.onRegion(exampleRegion)
    .withFilter(keysForGet)
    .withArgs(Boolean.TRUE)
    .withCollector(new MyArrayListResultCollector());
```

See the `com.gemstone.gemfire.cache.execute` Javadocs for `ResultCollector` and `ResultSender`.

Chapter 17

Managing Member Relationships

GemFire Enterprise® membership roles give you the ability to decide under what circumstances a distributed system can continue reliable operation after a disruption such as a network failure. A membership role describes how a system member relates to other members, or what purpose it fills. Members can establish interdependent relationships among themselves by playing one or more membership roles. Membership roles are optional, so they are not required for every member in the system.

You configure membership roles at the member level and the region level. First you identify all the roles your system members play, such as Producer, Consumer, or Backup. Then you decide what roles a given member fills, and add those roles to the member's system-level configuration. When the member connects to the distributed system, it declares those roles.

Then for each region you decide which of those membership roles must be present somewhere in the distributed system to enable reliable access to that region. In the cache configuration file you configure the region's role requirements and appropriate actions to take when the distributed system loses or regains one of those required roles.

For example, you could prevent a producer from putting data into the cache if there isn't an associated consumer or set of consumers. The producer needs to be dependent on a consumer being present, so the producer should not be allowed to modify the cache if the consumer is terminated or has not been started.

> *The member relationships management described in this chapter applies to peers in a distributed system. It does not apply to relationships between clients and servers in client/server installations, to relationships between distributed systems in multi-site installations, or to peers hosting partitioned regions.*

In this chapter:

- Setting Membership Roles (page 454)
- Setting Membership Requirements on a Region (page 455)
- Role Loss Scenarios (page 460)

17.1 Setting Membership Roles

With membership roles, you can configure a cache so that one or more of its regions requires another system member to be present in order for the region to be used reliably. You can set up inter-member dependencies so that processing can be halted or altered when dependencies are not satisfied. A member can play multiple roles, and different members can declare that they play the same role. Typical role names are `Producer`, `Consumer`, and `Backup`.

You can specify membership roles to help you identify processes and what domain functions a process performs. The `admin` API, cache events, and other system activities expose the `com.gemstone.gemfire.distributed.DistributedMember` interface, which indicates the member generating the cache event. `DistributedMember` then uses the `getRoles` method to expose all roles assigned to that system member.

Roles are declared in the `gemfire.properties` file, as shown in the following example:

```
roles=RoleName,RoleName,...
```

Which could be specified as:

```
roles=Producer,Backup,...
```

17.2 Setting Membership Requirements on a Region

Membership attributes configure a region to require one or more roles for reliable access to the region. The members with the required roles defined in their `gemfire.properties` file must also have the same region defined in their cache. See *Membership Attributes* on page 455 for additional details.

The `RegionRoleListener` handles region reliability membership events by receiving notification of the loss or gain of a role. The notification includes the member that left or joined that caused the loss or gain of that role. Any number of region role listeners may be registered or removed. Notification of role loss will only occur after all instances of that role have left the system membership. Notification of role gain will only occur when a member playing that role joins if no other member is currently playing that role. For more information, see `RegionRoleListener` on page 306.

If a role required by a region is not present, then the region's specified loss action is put into effect. See `loss-action` on page 456 for more information about setting member access conditions.

When one or more missing roles return to the distributed membership, the `resumption-action` (page 458) specifies how the region is affected by the resumption of reliability. The region becomes available for use after its resumption action is executed. Notification is sent to the `RegionRoleListener` that the region is available for full use and that the region has performed its resumption action. If there was no actual loss of roles, because the roles were not present when the region was created, then the region is available immediately.

> *By default, membership attributes are not used. For them to take effect, you must also define membership roles.*

Membership Attributes

Membership attributes provide options for configuring a region to require one or more membership roles. To satisfy a required role, a member must have the role defined in its `gemfire.properties` file and must also define the same region in its cache as the region that requires the role. If another member has the role defined, but does not create the same region, the role for the region is not satisfied. That is, if member Producer has region `stockMarketQuotes` with a required role of `consumer`, and member Consumer specifies `consumer` as one of its roles in its `gemfire.properties` file, then Consumer satisfies the required role for Producer's region only if Consumer also has the region `stockMarketQuotes` defined.

If a required role is not present, then the member's `loss-action` setting specifies how region access is affected when a role is missing. When all of the required roles are again present, the `resumption-action` specifies how the region is affected. Each role is a user-defined string, such as `ReportUpdater`.

Membership attributes have no effect unless one or more roles are specified. A region's `membership-attributes` are defined by setting its two attributes, `loss-action` (what happens when a role is missing) and `resumption-action` (what happens when all the roles are present again), and then specifying the roles whose loss or resumption will trigger these actions. For example:

Example 17.1 Setting Membership Attributes

```
<membership-attributes loss-action="limited-access"
          resumption-action="none">
   <required-role name="TradeProcessor"/>
   <required-role name="ReportUpdater"/>
</membership-attributes>
```

These settings cannot be changed after the region is created.

loss-action

The `loss-action` setting specifies how access to the region is affected when one or more of the roles specified in the region's membership attributes are offline and no longer present in the system membership. This table defines the options for `loss-action`.

Specification (default in **bold**)	**Definition**
`full-access`	Access to the region is unaffected when required roles are missing.
`limited-access`	Only local access to the region is allowed when required roles are missing. All distributed write operations on the region will throw `RegionAccessException` while any required roles are absent. Reads which result in a `netSearch` behave normally, while any attempt to `load` is not allowed.
`no-access`	The region is unavailable when required roles are missing. All operations including read and write access are denied. All read and write operations on the region will result in `RegionAccessException` while any required roles are absent. Basic administration of the region is allowed, including `close` and `localDestroyRegion`. *no-access is the default loss-action only if you define membership roles. Without roles, there is always full access to the region.*
`reconnect`	Loss of required roles causes the entire cache to be closed. In addition, this process will disconnect from the distributed system and then reconnect. Attempting to use any existing references to the regions or cache will throw a `CacheClosedException`. *This setting requires a full cache configuration in the cache.xml file. All programmatically defined cache and region attributes are lost if they are not also in the cache.xml.*

limited-access

With a region specified as `limited-access`, only local access to the region is allowed when required roles are missing. All distributed write operations on the region will throw `RegionAccessException` while any required roles are absent. Reads that result in a `netSearch` behave normally, while any attempt to `load` is not allowed. Distributed operations that were originated from another process will bypass checking required roles and update the local cache.

When a loss action of `limited-access` is in effect, the following region operations throw a `RegionAccessException`:

`becomeLockGrantor`	`getDistributedLock`
`clear`	`getRegionDistributedLock`
`create`	`put`
`destroy`	`putAll`
`destroyRegion`	`remove`
`get` (that does search or load)	

In addition, distributed expiration is deferred until reliability is resumed. All other operations are unaffected by `limited-access`.

no-access

When a region's `loss-action` specifies `no-access` due to unsatisfied required roles, all region operations, except for administrative operations including `close` and `localDestroyRegion`, throw a `RegionAccessException`. The exception contains a list of the roles that are unsatisfied and notes the loss action of the region. The `RegionRoleListener` for the region receives an `afterRoleLoss` notification that required roles are now missing, and `loss-action` for the region is activated.

When a loss action of `no-access` is in effect, the following region operations throw a `RegionAccessException`:

`becomeLockGrantor`	`isEmpty`
`clear`	`keySet`
`containsKey`	`localClear`
`containsValue`	`localDestroy`
`containsValueForKey`	`localInvalidate`
`create`	`localInvalidateRegion`
`destroy`	`put`
`destroyRegion`	`putAll`
`entrySet`	`query`
`get`	`querying`
`getDistributedLock`	`remove`
`getEntry`	`selectValue`
`getRegionDistributedLock`	`size`
`invalidate`	`values`
`invalidateRegion`	

In addition, local expiration and distributed expiration are deferred until reliability is resumed. All other operations are unaffected by `no-access`.

reconnect

Specifying `reconnect` causes a member to disconnect from the distributed system as soon as its required roles are missing and then reconnect to attempt to restore missing roles. If network segmentation occurs due to a failed network link, it can partition the network into multiple sets of disconnected computers. Segmentation can cause a member to lose some of its required roles to other network segments. Immediately after the required roles are lost, the member's `reconnect` policy attempts to reconnect the member to the correct side of the distributed system that has the missing required roles. If

members on both sides of the segmented network have all of their required roles satisfied, then the `reconnect` policy is not triggered and both sides continue to function properly.

You can set the property `max-wait-time-reconnect` in the `gemfire.properties` file to change the maximum number of milliseconds to wait for the distributed system to reconnect in case of required role loss. The system will attempt to reconnect `max-num-reconnect-tries`, and the wait timeout applies to each reconnection attempt.

This setting requires a full cache configuration in the `cache.xml` file.

> *At reconnect, the cache is reinitialized using the* `cache.xml` *file. All programmatically defined cache and region attributes are lost if they are not also in the* `cache.xml`.

resumption-action

The `resumption-action` setting specifies how the region is affected by resumption of reliability when one or more missing required roles return to the distributed membership. This table defines the options for `resumption-action`.

Specification (default in **bold**)	**Definition**
`none`	No special action takes place when reliability resumes.
`reinitialize`	Resumption of reliability causes the region to be locally cleared of all data and replicated regions to do a new `getInitialImage` operation to repopulate the region. Any existing references to this region become unusable because any subsequent methods invoked on those references throw a `RegionReinitializedException`.

> *Selecting* `reinitialize` *could cause a region to be reinitialized immediately after it is created.*

required-role

The nested `required-role` elements in `membership-attributes` specify which membership roles must be present in the distributed system for reliable access to this region.

To query the status of a region's `required-roles`, use these methods:

- `MembershipAttributes.getRequiredRoles` returns the `Set` of roles currently required for the reliability of a region.
- `RequiredRoles.checkForRequiredRoles` returns a `Set` of any roles specified in the region's `membership-attributes` that are currently missing.
- `RequiredRoles.waitForRequiredRoles` waits for a set timeout period specified in milliseconds for any missing roles to become present, and then returns a `Set` of any roles that are missing.

You can set the `member-timeout` property in the `gemfire.properties` file to specify how many milliseconds to wait for a role to return to the system before canceling an operation. For details, see *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

Configuration Examples

The following `cache.xml` snippet establishes the membership attributes for a region by setting its `loss-action` to `no-access` and `resumption-action` to `none`.

Example 17.2 Setting Required Roles in the cache.xml File

```
<region-attributes scope="distributed-ack" statistics-enabled="true">
   <membership-attributes loss-action="no-access" resumption-action="none">
      <required-role name="consumer"/>
   </membership-attributes>
</region-attributes>
```

This code sample sets the `LossAction` to `RECONNECT` and the `ResumptionAction` to `NONE`:

Example 17.3 Setting Required Roles Through the API

```
MembershipAttributes ra = new MembershipAttributes
   (roleRequiredRegionA, LossAction.RECONNECT, ResumptionAction.NONE);
```

17.3 Role Loss Scenarios

The following scenarios describe situations where a membership role is either missing, lost, or becomes partitioned and show how GemFire Enterprise handles the absence of the role. Disruptions can happen if the system is unable to start, if it is incorrectly configured, if a process terminates, or if a network failure segments the network into separate disconnected groups of computers.

If some members are lost, but there are members remaining that satisfy the region's membership attributes, then the region continues to operate normally. If any role goes missing, then the membership attributes go into effect for the region.

> *Loss of a required role does not generate an alert or produce any default logging. To find out about this event, you must register a* `RoleMembershipListener` *on the* `Region`*. Then you can log the event or do something else in reaction to it.*

Missing Role When a Region is Created

A process has a region defined, and the region is configured to require one process with the role of `Producer` to be available. The region has a `loss-action` of `no-access` and a `resumption-action` of `reinitialize`. Its distributed system has the `member-timeout` property set to `15000` (15 seconds).

When the process creates the region, the cache may not see a `Producer` role member available. When the process attempts to access the region, the cache waits up to the `member-timeout` period for the `Producer` member to become available. If a `Producer` becomes available before the `member-timeout` period elapses, the process continues normally. Otherwise, its `loss-action` is enforced and a `RegionAccessException` is thrown.

Since the `loss-action` is `no-access`, the application receives a `RegionAccessException` each time it tries to perform a distributed operation on the region (including `put` and `get`, but not `localDestroy`). When a `Producer` becomes available, the region stops throwing `RegionAccessException` and any waiting process threads proceed.

Lost Role When the Cache is Inactive

When the cache is not being used because the application is either between operations or is a listener-style application that just waits for events, the cache still monitors membership information and notices the loss of a required role for its regions. For listener-style applications, GemFire notifies the `RegionRoleListener` when the region becomes unavailable and then provides additional notification when the region performs its resumption action and is fully available. The notification triggers the listener's `afterRoleLoss` and `afterRoleGain` callback methods. For details, see `RegionRoleListener` on page 306.

Lost Role During distributed-no-ack Update

GemFire's membership management behavior depends on the region's scope and the types of communications used to deliver messages. If TCP/IP sockets are used, GemFire can detect role loss immediately when a transmission to a member fails, unless messages are being queued. If UDP sockets are used for multicast communications, GemFire can take 15 seconds or more to determine that a member is unreachable, depending on the `member-timeout` setting. For UDP, or TCP with slow receiver, many `distributed-no-ack` updates might be sent to many members before the loss of a required role is detected.

With `distributed-no-ack` scope, operations check to ensure that required roles are present before beginning distribution, and will throw a `RegionAccessException` if the operation conflicts with the `loss-action` of the region.

Lost Role During distributed-ack Update

During an update, GemFire first sends the update to all members known to have the affected region and then waits for acknowledgement. While waiting for ACKs, the cache watches membership events and notes whether any of the recipients of the update have left the system. During this period, if GemFire detects the loss of a required role for the region, then the cache waits for ACKs from other members and throws a `RegionDistributionException`. The exception provides a reference to the region and a collection of the lost roles.

Lost Role During a Global Update

If the region's scope is `global`, the loss of a required role during an update has the same behavior as the loss of a required role during an operation where the scope is `distributed-ack`.

Lost Role During a Transaction

Losing a required role is during the processing of a transaction can prevent the transaction from committing. If the required role is missing before the commit begins, then the commit fails. If the role is lost during the during the transmission of commit messages to other members, the cache will complete the commit operation and throw a `CommitDistributionException`. If the transaction commit started before losing the role, the commit completes for all active members and then a `RegionAccessException` is thrown. The exception lists any membership roles that departed during the distribution of the transaction.

Network Failure

In the case of network segmentation, regions on either side of the segmentation may continue to function.

This figure describes two network failure scenarios that might occur and shows how the reliability of the network members is handled by required roles.

Figure 17.1 Required Roles Behavior With a Segmented Network

The three members in this initial distributed system have all required roles satisfied, so system operation continues.

Scenario 1

After network segmentation, members M_1 and M_2 can satisfy their required roles so they continue. M_3 in Segment 2 cannot satisfy its required role provided by the inaccessible M_1, so its loss action activates.

Scenario 2

M_1 has no required role dependencies, so it continues. M_2 cannot satisfy its required role provided by the inaccessible M_1, so its loss action activates.

Chapter 18

Monitoring and Tuning Your Applications

GemFire Enterprise® provides a number of tools for monitoring and tuning your GemFire system. System monitoring is available through the administration and JMX APIs (located under `com.gemstone.gemfire.admin`), and through the study of archived logging and statistics information. System managing and monitoring is available through the GemFire Enterprise Monitor version 2 (GFMon) program, for which information is available in the GemFire Enterprise Monitor User's Manual. You can also use standard Java tools for tracking member performance and resource consumption.

System tuning can be accomplished through declarative configuration files and through the programming APIs. Information on the GemFire configuration files, GemFire statistics, and the JMX Agent is provided in the *GemFire Enterprise System Administrator's Guide*. Information on the administration API is available in Chapter 19, *Developing System Administration Tools*, on page 479.

In addition, this chapter points out specific areas that deserve particular attention because they can affect data consistency between caches.

In this chapter:

- Statistics (page 464)
- Performance Controls (page 470)
- Maintaining Cache Consistency (page 476)

18.1 Statistics

When Java applications and cache servers join a distributed system, they indicate whether to enable statistics sampling and whether to archive the statistics that are gathered. For detailed information about setting the configuration attributes that affect statistics collection for the cache, see *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

The GemFire Enterprise installation includes standard statistics for caching and distribution activities and provides an API for creating application-defined statistics.

This section discusses statistics as they apply to the developer in the following subsections:

- Application-Defined Statistics (page 464)
- Distributed System and Caching Statistics (page 466)
- Partitioned Region Statistics (page 468)
- Transient Region and Entry Statistics (page 469)

Application-Defined Statistics

The GemFire package, `com.gemstone.gemfire`, includes interfaces for defining and maintaining your own statistics. This section gives an overview of the primary statistics API. For complete information on the classes and interfaces described here, see the online Java API documentation.

com.gemstone.gemfire

The interfaces are:

- **StatisticDescriptor**—Describes an individual statistic. Each statistic has a name and information on the statistic it holds, such as its class type (long, int, etc.) and whether it is a counter that always increments, or a gauge that can vary in any manner.
- **StatisticsType**—A logical type that holds a list of `StatisticDescriptor`s and provides access methods to them. The `StatisticDescriptor`s contained by a `StatisticsType` are each assigned a unique ID within the list. `StatisticsType` is used to create a `Statistics` instance.
- **Statistics**—An instantiation of an existing `StatisticsType` object with methods for setting, incrementing, and getting individual `StatisticDescriptor` values.
- **StatisticsFactory**—This interface creates instances of `Statistics`. You can also use it to create instances of `StatisticDescriptor` and `StatisticsType`, because it implements `StatisticsTypeFactory. DistributedSystem` is an instance of `StatisticsFactory.`
- **StatisticsTypeFactory**—This interface creates instances of `StatisticDescriptor` and `StatisticsType`.

The statistics interfaces are instantiated using statistics factory methods that are included in the package. For coding examples using statistics factory methods to instantiate the statistics interfaces, see the online Java API documentation for `StatisticsFactory` and `StatisticsTypeFactory`.

As an example, an application server might collect statistics on each client session in order to gauge whether client requests are being processed in a satisfactory manner. Long request queues or long server response times could prompt some capacity-management action such as starting additional application servers. To set this up, each session-state data point is identified and defined in a `StatisticDescriptor` instance. One instance might be a `RequestsInQueue` gauge, a non-negative integer that increments and decrements. Another could be a `RequestCount` counter, an integer that always increments. A list of these descriptors is used to instantiate a `SessionStateStats StatisticsType`. When a client connects, the application server uses the `StatisticsType` object to

create a session-specific `Statistics` object. The server then uses the `Statistics` methods to modify and retrieve the client's statistics. This figure illustrates the relationships between the statistics interfaces and shows the implementation of this use case.

Figure 18.1 com.gemstone.gemfire Statistics API

Distributed System and Caching Statistics

GemFire Enterprise uses the API described above to provide standard statistics for your system and cache. When sampling and archiving are enabled, these statistics can be studied in archive files through the `gemfire stats` command or VSD. For information on these tools, see *System Statistics* on page 235 of the *GemFire Enterprise System Administrator's Guide*. For information on enabling sampling and archiving, see *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

For each member that connects to a distributed system and creates a cache, an instance of each `StatisticsType` is created. For pure Java applications, all statistics are collected except ProcessStats and SystemStats.

- **CacheClientNotifierStatistics**—Statistics regarding cache server operations sent to clients. For a list of statistics, see *Cache Client Notifier Statistics* on page 239 of the *GemFire Enterprise System Administrator's Guide*. For the statistics that are most useful in detecting slow receivers, see *VM Statistics* on page 278 of the *GemFire Enterprise System Administrator's Guide*.
- **CacheClientProxyStatistics**—Statistics regarding cache server operations sent to a single client. For a list of statistics, see *Cache Client Proxy Statistics* on page 239 of the *GemFire Enterprise System Administrator's Guide*.
- **CacheClientUpdaterStatistics**—Statistics in a client and pertain to server-to-client data pushed from the server over a queue to the client (they are the client side of the server's CacheClientNotifierStatistics). For a list of statistics, see *Cache Client Updater Statistics* on page 240 of the *GemFire Enterprise System Administrator's Guide*.
- **CachePerfStats** (available if the member creates a cache)—Statistics on the GemFire Enterprise cache. Can be used to determine the type and number of cache operations being performed and how much time they consume. For a list of statistics, see *Cache Performance Statistics* on page 240 of the *GemFire Enterprise System Administrator's Guide*. If you are running transactions in your distributed system, additional statistics are available; see *Partitioned Region Statistics* on page 283 of the *GemFire Enterprise System Administrator's Guide*.
- **CacheServerStatistics**—Statistics used for cache servers and for gateway receivers. For a list of statistics, see *Cache Server Statistics* on page 242 of the *GemFire Enterprise System Administrator's Guide*.
- **ClientStatistics**—Statistics in a client that describe all the messages sent from the client to a specific server. For a list of statistics, see *Client Statistics* on page 246 of the *GemFire Enterprise System Administrator's Guide*.
- **CQStatistics**—Statistics regarding continuous querying information for a single CQ and for the query service's management of CQs as a whole. For a list of statistics, see *CQ Statistics* on page 253 of the *GemFire Enterprise System Administrator's Guide*.
- **DiskDirStatistics**—Statistics regarding the disk usage for a region's disk directory.
- **DiskRegionStatistics**—Statistics regarding operations performed on a disk region. For a list of statistics, see *Disk Region Statistics* on page 254 of the *GemFire Enterprise System Administrator's Guide*.
- **DistributionStats**—Statistics on the GemFire distribution layer. These can be used to tell how much message traffic there is between this member and other distributed system members. For the

statistics that are most useful in detecting slow receivers, see *VM Statistics* on page 278 of the *GemFire Enterprise System Administrator's Guide*.

- **DLockStats**—Statistics on distributed locks. For a list of statistics, see *DLock Statistics* on page 263 of the *GemFire Enterprise System Administrator's Guide*.
- **GatewayStatistics**—Statistics on an outgoing gateway queue and connection. For a list of statistics, see *Gateway Statistics* on page 267 of the *GemFire Enterprise System Administrator's Guide*.
- **GatewayHubStatistics**—Statistics on a gateway hub. For a list of statistics, see *Gateway Hub Statistics* on page 267 of the *GemFire Enterprise System Administrator's Guide*.
- **LocatorStatistics**—Statistics on locators. For a list of statistics, see *Locator Statistics* on page 268 of the *GemFire Enterprise System Administrator's Guide*.
- **LRUStatistics**—Statistics on LRU activity for a single region. For a list of statistics, see *LRU Statistics – Count-based* on page 268 of the *GemFire Enterprise System Administrator's Guide* and *LRU Statistics – Size-based* on page 268 of the *GemFire Enterprise System Administrator's Guide*.
- **PoolStatistics**—Statistics in a client in a client that describe one of the client's connection pools. For a list of statistics, see *Pool Statistics* on page 269 of the *GemFire Enterprise System Administrator's Guide*.
- **ProcessStats**—Operating system statistics on the VM's process. These can be used to determine the member's CPU, memory, and disk usage. Operating system statistics are not available in *pure Java* mode, where GemFire Enterprise runs without the use of the GemFire native library. For a list of statistics, see *Process Statistics – Linux* on page 269 of the *GemFire Enterprise System Administrator's Guide*, *Process Statistics – Solaris* on page 269 of the *GemFire Enterprise System Administrator's Guide*, and *Process Statistics – Windows* on page 271 of the *GemFire Enterprise System Administrator's Guide*.
- **StatSampler**—Shows how much time is spent collecting statistics.
- **SystemStats**—Operating system statistics on the member's machine. These can be used to determine total cpu, memory, and disk usage on the machine. Operating system statistics are not available in pure Java mode. For a list of statistics, see *System Statistics – Linux* on page 273 of the *GemFire Enterprise System Administrator's Guide* and *System Statistics – Solaris* on page 275 of the *GemFire Enterprise System Administrator's Guide*.
- **VMStats**—Show the VM's Java usage. These can be used to detect possible problems with memory consumption. For a list of statistics, see *VM Statistics* on page 278 of the *GemFire Enterprise System Administrator's Guide*.
- **VMGCStats**—Show the time used for VM garbage collection. This statistic is available on JDK 1.5 and later VMs. For a list of statistics, see *VMGC Statistics* on page 279 of the *GemFire Enterprise System Administrator's Guide*.
- **VMMemory Stats**—Show details on how the Java heap memory is being used. This statistic is available on JDK 1.5 and later VMs. For a list of statistics, see *VM Memory Usage Statistics* on page 279 of the *GemFire Enterprise System Administrator's Guide*.
- **VMMemoryPool Stats**—These statistics describe memory usage in difference garbage collector memory pools. For a list of statistics, see *VM Memory Pool Statistics* on page 279 of the *GemFire Enterprise System Administrator's Guide*.

Partitioned Region Statistics

Whenever you run members that define partitioned regions, GemFire gathers statistics specific to the operation of partitioned regions. Statistics are gathered for each member. Partitioned region statistics provide run-time and historical data on these areas:

- Region operations executed on the current member's partitioned region data
- Message traffic to and from this member requesting operations on the partitioned region
- Data entry distribution among the buckets hosted by this member

Partitioned region statistics are gathered for each member. Time-based partitioned region statistics, like all other time-based GemFire statistics, are disabled by default. For details on turning on time-based statistics, see *Distributed System and Caching Statistics* on page 466.

Unlike the transient GemFire region statistics, partitioned region statistics can be archived and charted. You can view them through the Visual Statistics Display (VSD).

For a list of partitioned region statistics and instructions for viewing them, see *Partitioned Region Statistics* on page 283 of the *GemFire Enterprise System Administrator's Guide*.

Transient Region and Entry Statistics

GemFire Enterprise provides a standard set of statistics for cached regions and region entries. You can enable and disable statistics gathering at the region level with the `statistics-enabled` (page 117) region attribute. Enabling these statistics requires extra memory per entry.

> *Unlike other GemFire statistics, these region and entry statistics are not archived and cannot be charted.*

The hit and miss counts collected in these statistics can be useful for fine-tuning your system's caches. If you have a region's entry expiration enabled, for example, and see a high ratio of misses to hits on the entries, you might choose to increase the expiration times.

> *This information does not apply to partitioned region statistics; see Partitioned Region Statistics on page 468 for details on those statistics.*

The standard GemFire statistics for regions and region entries provide this information:

- **Hit and miss counts**—For the entry, the hit count is the number of times the cached entry was accessed through the `Region.get` method and the miss count is the number of times these hits did not find a valid value. For the region these counts are the totals for all entries in the region. The API provides `get` methods for the hit and miss counts, a convenience method that returns the hit-to-miss ratio, and a method for zeroing the counts.
- **Last accessed time**—For the entry, this is the last time a valid value was retrieved from the locally cached entry. For the region, this is the most recent "last accessed time" for all entries contained in the region. This statistic is used for idle timeout expiration activities that are documented in *Keeping Your Data Current With Expiration* on page 176.
- **Last modified time**—For the entry, this is the last time the entry value was updated (directly or through distribution) due to a load, create, or put operation. For the region, this is the most recent "last modified time" for all entries contained in the region. This statistic is used for idle timeout expiration activities that are documented in *Keeping Your Data Current With Expiration* on page 176.

You can retrieve region and entry statistics through the `getStatistics` methods of the `Region` and `Region.Entry` objects.

This example prints the current statistics values for a distributed region or a region with `local` scope:

Example 18.1 Retrieving Region Statistics

```
PrintStream out = System.out;
out.println("Current Region:\n\t" + this.currRegion.getName());
RegionAttributes attrs = this.currRegion.getAttributes();
if (attrs.getStatisticsEnabled()) {
   CacheStatistics stats = this.currRegion.getStatistics();
   out.println("Stats:\n\tHitCount is " + stats.getHitCount() +
      "\n\tMissCount is " + stats.getMissCount() +
      "\n\tLastAccessedTime is " + stats.getLastAccessedTime() +
      "\n\tLastModifiedTime is " + stats.getLastModifiedTime());
}
```

18.2 Performance Controls

This section provides tuning suggestions of particular interest to developers, primarily programming techniques and cache configuration. For more information on tuning the GemFire distributed system, hosts, and networks, see *Monitoring and Tuning the Distributed System* on page 139 of the *GemFire Enterprise System Administrator's Guide*.

Configuring the System on page 41 of the *GemFire Enterprise System Administrator's Guide* describes all of the system configuration parameters and how to set them. For cache configuration, see Chapter 4, *Data Regions*, on page 85. Some of these controls relate to the GemFire statistics described in *Statistics* on page 464.

Region `scope` can affect system performance. See *Choosing the Level of Distribution* on page 204 for more information.

Data Serialization

GemFire provides the interface `com.gemstone.gemfire.DataSerializable`, which you can use in place of Java's standard object serialization. The interface allows faster and more compact data serialization than the standard serialization. You can further speed serialization by registering the instantiator for your `DataSerializable` class through `com.gemstone.gemfire.Instantiator`, eliminating the need for reflection to find the right serializer. The `DataSerializable` interface is not supported with complex object graphs. You can provide your own serialization through the API or in the `cache.xml` through the `serialization-registration` element. For more information, see the online Java documentation in the `com.gemstone.gemfire` package for `DataSerializable` and `DataSerializer`.

This example registers the `byteholder` class instantiator with the data serialization framework:

Example 18.2 Registering a DataSerializable Class Instantiator

```
Instantiator.register(new Instantiator(byteholder.class, 17) {
   public DataSerializable newInstance() {
      return new byteholder();
   }
});
```

> *Using the* `DataSerializable` *and* `Serializable` *properties on the same data results in increased memory usage and lower throughput than using* `Serializable` *on the entire data, especially if the* `Serializable` *entries are in collections. The bigger the data collection, the lesser throughput as the metadata for the collection entries is not shared when using the* `DataSerializable` *property.*

In addition to speeding standard object serialization, you can use this interface to serialize any custom objects you store in the cache (see *Data Serialization Requirements* on page 57).

Setting Cache Timeouts

The cache initialization properties listed in this section are declared in the `cache.xml` file; for details, see *Configuring the Cache* on page 74. These timeouts can also be set through methods of the `com.gemstone.gemfire.cache.Cache` interface.

- **search-timeout**—This specifies how long a `netSearch` operation can wait for data before timing out. The default is 5 minutes. You may want to change this based on your knowledge of the network load or other factors. The `netSearch` operation is described in *Accessing an Entry* on page 135.

The next two attributes describe timeout settings for locking in regions with `global` scope. Locking operations can time out in two places: when waiting to obtain a lock (lock time out); and when holding a lock (lock lease time). Operations that modify objects in a global region use automatic locking. In addition, you can manually lock a global region and its entries through the `com.gemstone.gemfire.cache.Region` API. For more information on implementing manual distributed locking using the API, see the online API documentation for `Region`. The explicit `lock` methods provided by the APIs allow you to specify a lock timeout parameter. The lock time out for implicit operations and the lock lease time for implicit and explicit operations are governed by these cache-wide parameter settings:

- **lock-timeout**—This specifies the timeout for object lock requests. The setting affects automatic locking only, and does not apply to manual locking. The default is 1 minute. If a lock request does not return before the specified timeout period, it is cancelled and returns with a failure.
- **lock-lease**—This specifies the timeout for object lock leases. The setting affects both automatic locking and manual locking. The default is 2 minutes. Once a lock is obtained, it may remain in force for the lock lease time period before being automatically cleared by the system.

Locking is discussed generally in *Locking* on page 205.

Controlling Socket Use

For peer-to-peer communication, you can manage socket use at the system member level and at the thread level. The `conserve-sockets` setting indicates whether application threads share sockets with other threads or use their own sockets for distributed system member communication. This setting has no effect on communication between a server and its clients or between gateways in multisite installations, but it does control the server's and gateway's communication with their peers. The servers and gateway hubs connect to their peers to service requests from clients and remote sites. In client/server settings in particular, where there can be a large number of clients for each server, controlling peer-to-peer socket use is an important part of tuning server performance.

You configure `conserve-sockets` for the member as a whole in `gemfire.properties`. For details, see *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*. Additionally, you can change the sockets conservation policy for the individual thread through the API.

When `conserve-sockets` is set to `false`, each application thread uses a dedicated thread to send to each of its peers and a dedicated thread to receive from each peer. Disabling socket conservation requires more system resources, but can potentially improve performance by removing socket contention between threads and optimizing distributed ACK operations. For distributed regions, the `put` operation, and destroy and invalidate for regions and entries, can all be optimized with `conserve-sockets` set to `false`. For partitioned regions, setting `conserve-sockets` to `false` can improve general throughput.

You can override the `conserve-sockets` setting for individual threads. The methods are provided in `com.gemstone.gemfire.distributed.DistributedSystem`:

- **setThreadsSocketPolicy**—Sets the calling thread's individual socket policy, overriding the policy set for the application as a whole. If set to `true`, the calling thread shares socket connections with other threads. If `false`, the calling thread has its own sockets.
- **releaseThreadsSockets**—Frees any sockets held by the calling thread. Threads hold their own sockets only when `conserve-sockets` is `true`. Threads holding their own sockets can call this method to avoid holding the sockets until the `socket-lease-time` has expired.

A typical implementation might set `conserve-sockets` to true at the application level and then override the setting for the specific application threads that perform the bulk of the distributed operations. The example below shows an implementation of the two API calls in a thread that performs benchmark tests. The example assumes the class implements `Runnable`. Note that the invocation, `setThreadsSocketPolicy(false)`, is only meaningful if `conserve-sockets` is set to `true` at the application level.

Example 18.3 Managing Socket Use at the Thread Level

```
public void run() {
   DistributedSystem.setThreadsSocketPolicy(false);
   try {
      // do your benchmark work
   } finally {
      DistributedSystem.releaseThreadsSockets();
   }
}
```

Management of Slow Receivers

This section discusses options for handling slow members that receive data distribution.

> *The slow receiver options control only to peer-to-peer communication between distributed regions using TCP/IP. This discussion does not apply to client/server or multi-site communication, or to communication using the UDP unicast or IP multicast protocols.*

Most of the options for handling slow members are related to on-site configuration during system integration and tuning. For this information, see *Slow Receivers with TCP/IP* on page 143 of the *GemFire Enterprise System Administrator's Guide*.

For details on selecting a communication protocol, see *Peer-to-Peer Messaging and Distribution* on page 65 of the *GemFire Enterprise System Administrator's Guide*.

Slowing is more likely to occur when applications run many threads, send large messages (due to large entry values), or have a mix of region configurations.

> *If you are experiencing slow performance and are sending large objects (multiple megabytes), before implementing these slow receiver options make sure your socket buffer sizes are large enough for the objects you distribute. The socket buffer size is set using* `gemfire.socket-buffer-size`*. For more information, see System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide.*

By default, distribution between system members is performed synchronously. With synchronous communication, when one member is slow to receive, it can cause its producer members to slow down as well. This can lead to general performance problems in the distributed system.

The specifications for handling slow receipt primarily affect how your members manage distribution for regions with `distributed-no-ack` scope, but it can affect other distributed scopes as well. If no regions have `distributed-no-ack` scope, this mechanism is unlikely to kick in at all. When slow receipt handling does kick in, however, it affects all distribution between the producer and consumer, regardless of scope. Partitioned regions ignore the `scope` attribute, but for the purposes of this discussion you should think of them as having an implicit `distributed-ack` scope.

Configuration Options

The slow receiver options are set in the producer member's region attribute, `enable-async-conflation` (page 106), and in the consumer member's `async*` `gemfire.properties` settings (see *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*).

Delivery Retries

If the receiver fails to receive a message, the sender continues to attempt to deliver the message as long as the receiving member is still in the distributed system. During the retry cycle, GemFire throws warnings that include this string:

```
will reattempt
```

The warnings are followed by an info message when the delivery finally succeeds.

Asynchronous Queueing For Slow Receivers

Your consumer members can be configured so that their producers switch to asynchronous messaging if the consumers are slow to respond to cache message distribution.

When a producer switches, it creates a queue to hold and manage that consumer's cache messages. When the queue empties, the producer switches back to synchronous messaging for the consumer. The settings that cause the producers to switch are specified on the consumer side in `gemfire.properties` file settings. For more information, see the `async*` properties listed in *System Properties in the gemfire.properties File* on page 48 of the *GemFire Enterprise System Administrator's Guide*.

If you configure your consumers for slow receipt queueing, and your region scope is `distributed-no-ack`, you can also configure the producer to conflate entry update messages in its queues. This configuration option is set as the region attribute `enable-async-conflation` (page 106). By default `distributed-no-ack` entry update messages are not conflated.

Depending on the application, conflation can greatly reduce the number of messages the producer needs to send to the consumer. With conflation, when an entry update is added to the queue, if the last operation queued for that key is also an update operation, the previously enqueued update is removed, leaving only the latest update to be sent to the consumer. Only entry update messages originating in a region with

`distributed-no-ack` scope are conflated. Region operations and entry operations other than updates are not conflated.

Figure 18.2 Async Distributed System Queue Conflation

Some conflation may not occur because entry updates are sent to the consumer before they can be conflated. For this example, assume no messages are sent while the update for Key A is added.

> *This method of conflation is different from that used for gateway-to-gateway conflation in multi-site installations. Compare to* *Site-to-Site Queue Conflation* on page 277*. It behaves the same as server-to-client conflation, described at* *Conflating the Subscription Queue* on page 255*.*

You can enable queue conflation on a region-by-region basis. You should always enable it unless it is incompatible with your application needs. Conflation reduces the amount of data queued and distributed.

These are reasons why conflation might not work for your application:

- With conflation, earlier entry updates are removed from the queue and replaced by updates sent later in the queue. This is problematic for applications that depend on a specific ordering of entry modifications. For example, if your receiver has a `CacheListener` that needs to know about every state change, you should disable conflation.
- If your queue remains in use for a significant period and you have entries that are updated frequently, you could have a series of update message replacements resulting in a notable delay in the arrival of *any* update for some entries. Imagine that update 1, before it is sent, is removed in favor of a later update 2. Then, before update 2 can be sent, it is removed in favor of update 3, and so on. This could result in unacceptably stale data on the receiver.

Increasing the Ratio of Cache Hits

The more often a `get` fails to find a valid value in the first cache and has to try a second cache, the more the overall performance is affected. A common cause of misses is expiration or eviction of the entry. If you have a region's entry expiration or eviction enabled, monitor the region and entry statistics. If you see a high ratio of misses to hits on the entries, consider increasing the expiration times or the maximum values for eviction, if possible. For configuration details, see *Keeping Your Data Current With Expiration* on page 176 and `eviction-attributes` on page 108. For more on hit and miss statistics, see *Transient Region and Entry Statistics* on page 469.

18.3 Maintaining Cache Consistency

Maintaining data consistency between caches in a distributed GemFire system is vital for ensuring its functional integrity and preventing data loss. This section describes configurations, settings and actions that can produce reliable cache consistency.

Take Care With Revoked Disk Stores

If you need to restart a region with a disk store, consider the state of the entire distributed region or partitioned region.

> *If you revoke a member's disk store, do not restart that member with its disk stores—in isolation—at a later time.*

GemFire stores information about your persisted data and prevents you from starting a member with a revoked disk store in the running system. But GemFire cannot stop you from starting a revoked member in isolation, and running with its revoked data. This is an unlikely situation, but it is possible to do:

1. Members A and B are running, both storing Region data to disk
2. Member A goes down
3. Member B goes down
4. At this point, Member B has the most recent disk data
5. Member B is not usable for some reason. Perhaps its host machine is down or cut off temporarily.
6. To get the system up and running, you start Member A, and use the command line tool to revoke Member B's status as member with the most recent data. The system loads Member A's data and you run forward with that.
7. Member A is stopped.
8. At this point, both Member A and Member B have information in their disk files indicating they are the gold copy members.
9. If you start Member B, it will load its data from disk.
10. When you start Member A, the system will recognize the incompatible state and report an exception, but by this point, you good data in both files, with no way to combine them.

Prevent Primary and Secondary Gateway Hubs from Going Offline

In a multi-site installation, if the primary gateway hub goes offline, a secondary gateway hub must take over primary responsibilities as the failover system. The existing secondary gateway hubs detect that the primary gateway hub has gone offline, and a secondary one becomes the new primary gateway hub. Since the queue is distributed, its contents are available to all gateways. So, when a secondary gateway becomes primary, it is able to start processing the queue where the previous primary left off with no loss of data.

If both the primary gateway hub and all its secondary hubs go offline and messages are in their queues, data loss could occur, because there is no failover system.

Avoid Using the Admin Interface to Start a Server

Avoid using the GemFire admin interface to start a server and specify the ping interval. When you specify the ping interval, it is not sent to the new server, so this could cause the server to adopt a zero-millisecond interval. Consequently, the server terminates all client connections. The symptoms of this behavior are clients that continually move their interest lists between servers, clients that do not receive cache update notifications, or only receive them sporadically.

Verify That isOriginRemote is Set to False

The `isOriginRemote` flag for a server or a multi-site gateway is set to `false` by default, which ensures that updates are distributed to other members. Setting its value to `true` in the server or the receiving gateway member applies updates to that member only, so updates are not distributed to peer members.

Optimize socket-buffer-size

In a multi-site installation using gateways, if the link between sites is not tuned for optimum throughput, it could cause messages to back up in the cache queues. If a queue overflows because of inadequate buffer sizes, it will become out of sync with the sender and the receiver will be unaware of the condition. You can configure the send-receive buffer sizes of the TCP/IP connections used for data transmissions by changing the `socket-buffer-size` attribute of the `gateway-hub` element in the `cache.xml` file. Set the buffer size by determining the link bandwidth and then using ping to measure the round-trip time.

See Socket Buffer Size (page 265) for more information about this setting. For more information about system tuning options, see *Monitoring and Tuning the Distributed System* on page 139 of the *GemFire Enterprise System Administrator's Guide*.

Understand Cache Transactions

This is a list of transaction behaviors and how they can affect cache consistency. Understanding the operation of GemFire transactions can help you minimize situations where the cache could get out of sync.

- Transactions only operate within one member, and they do not support partitioned regions or distributed regions with global scope. Transactions provide consistency within one cache, but the distribution of results to other members is not as consistent.
- Multiple transactions in a cache can create inconsistencies because of read committed isolation. Since multiple threads cannot participate in a transaction, most applications will be running multiple transactions.
- An in-place change to directly alter a key's value without doing a put can result in cache inconsistencies. With transactions, it creates additional difficulties because it breaks read committed isolation. If at all possible, use copy-on-read instead. For more information, see *Safe Entry Modification* on page 140.
- In `distributed-no-ack` scope, two conflicting transactions in different members can commit simultaneously, overwriting each other as the changes are distributed.
- If a cache writer exists during a transaction, then each transaction write operation triggers a cache writer's related call. Regardless of the region's scope, a transaction commit can invoke a cache writer only in the local cache and not in the remote caches.
- A region in a cache with transactions may not stay in sync with a region of the same name in another cache without transactions.
- Two applications running the same sequence of operations in their transactions may get different results. This could occur because operations happening outside a transaction in one of the members can overwrite the transaction, even in the process of committing. This could also occur if the results of a large transaction exceed the machine's memory or the capacity of GemFire Enterprise. Those limits can vary by machine, so the two members may not be in sync.

Chapter 19

Developing System Administration Tools

The GemFire Enterprise® Administration API is provided in the `com.gemstone.gemfire.admin` package. The online Java API documentation provides extensive usage details for the interfaces, classes, and their methods. This chapter provides an overview of the primary interfaces and classes of the API followed by a programming examples section.

In this chapter:

- The API Interfaces and Classes (page 480)
- Programming Examples (page 485)

For general information on administering a distributed system, refer to *Overview of GemFire System Administration* on page 29 of the *GemFire Enterprise System Administrator's Guide*. The GemFire Administration API provides programmatic access to GemFire functionality.

19.1 The API Interfaces and Classes

This section gives an overview of the primary interfaces and classes that are provided by the GemFire Administration API package `com.gemstone.gemfire.admin`. Complete information on the package is available in the online Java API documentation, which you can navigate to through the `docs/index.html` file.

com.gemstone.gemfire.admin

The `com.gemstone.gemfire.admin` package includes the administration API and the health monitoring API.

System Administration

The administration API allows you to configure, start, and stop a distributed system and many of its components. The API is made up of distributed system administration, component administration, and cache administration. In addition to the core components listed here, the administration API provides interfaces to issue and handle system member alerts and to monitor statistics. The interfaces for the distributed system and its components have accompanying configuration interfaces, such as `ManagedEntityConfig` for `ManagedEntity`.

Distributed System Administration

- **AdminDistributedSystemFactory**—Used to define and retrieve an `AdminDistributedSystem` instance.
- **AdminDistributedSystem**—This is the administrative interface for managing a GemFire Enterprise system. Use the `AdminDistributedSystemFactory` class to define and create an instance of `AdminDistributedSystem`. With the `AdminDistributedSystem` instance, you can perform tasks on the distributed system such as defining, starting, and stopping GemFire cache servers, merging system logs, and obtaining administrative interfaces to applications that host GemFire Enterprise caches.
- **DistributedSystemConfig**—Describes a distributed system to administer by specifying its configuration. The `DistributedSystemConfig.ConfigListener` interface handles changes to the distributed system configuration.
- **CacheVm**—Describes a GemFire cache server system member, which is a long-lived caching application started with the *`productDir`*`/bin/cacheserver` script. A cache server is often used as the server process in a client/server architecture. This can be configured through the `CacheVmConfig` interface. For more on cache servers, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.

Component Administration

- **ManagedEntity**—An entity that can be managed, including being started and stopped, through the admin API. Applications cannot be managed in this way, but the other system members can.
- **DistributionLocator**—A single locator process, of which a distributed system may use zero or more. Extends the `ManagedEntity` interface. Locators are system members that can be configured, started and stopped through this interface. For information on locators, see *Using Locators For Peer and Client/Server Discovery* on page 62 of the *GemFire Enterprise System Administrator's Guide*.
- **CacheVm**—Manages a cache server in a distributed system. Extends `SystemMember` and `ManagedEntity`.

- **SystemMember**—Monitors a member of a GemFire Enterprise system. Provides access to the member's cache.
- **SystemMembershipListener**—Application plug-in interface whose callback methods are invoked when members join or leave the GemFire distributed system. Receives `SystemMembershipEvent`s.
- **SystemMembershipEvent**—An event interface that reports when a member has joined or left the distributed system and provides the member ID of the process that joined or left.

Cache Administration

- **SystemMemberCache**—Represents a `SystemMember`'s cache. Through this interface, like through the local application's `Cache` interface, you can create regions in a remote member's cache. You can also enable and disable server processes and modify cache attributes.
- **SystemMemberCacheServer**—Represents a server for a cache, which is instantiated through the `SystemMemberCache` interface. Through this interface, the server can be configured, started, and stopped. For information on servers, see Chapter 8, *Client/Server Architecture and Configuration Basics*, on page 201.
- **SystemMemberRegion**—Represents a `SystemMember`'s view of one of its cache regions. You cannot access a region's contents through this API.
- **SystemMemberCacheListener**—Application plug-in interface whose callback methods can be used to track the life cycle of caches and regions in the GemFire distributed system. Receives `SystemMemberCacheEvent`s and `SystemMemberRegionEvent`s.
- **SystemMemberCacheEvent**—An event interface that reports when a cache is created or closed. Provides the member ID and the operation. Extends `SystemMembershipEvent`.
- **SystemMemberRegionEvent—**An event interface that reports when a region has joined or left the distributed system. Provides the member ID, the operation, and the region name. Extends `SystemMemberCacheEvent`.
- **AlertListener**—A log listener whose callback methods are invoked when it receives notice of serious log messages.

This figure shows the relationships between the primary interfaces of the administration API.

Figure 19.1 GemFire Core Admin Interfaces

The other area of functionality in the administration API is the health monitoring API, discussed in the next section.

System Health Monitoring

The health monitoring API allows you to configure and monitor system health indicators for GemFire Enterprise distributed systems and their components. There are three levels of health: good health that indicates that all GemFire components are behaving reasonably, okay health that indicates that one or more GemFire components are slightly unhealthy and may need some attention, and poor health that indicates that a GemFire component is unhealthy and needs immediate attention.

Because each GemFire application has its own definition of what it means to be healthy, the metrics that are used to determine health are configurable. `GemFireHealthConfig` provides methods for configuring the health of the distributed system and members that host `Cache` instances. Health can be configured on both a global and per-machine basis. `GemFireHealthConfig` also allows you to configure how often GemFire's health is evaluated.

The health administration APIs allow you to configure performance thresholds for each component type in the distributed system (including the distributed system itself). These threshold settings are compared to system statistics to obtain a report on each component's health. A component is considered to be in `good` health if all of the user-specified criteria for that component are satisfied. The other possible health settings, `okay` and `poor`, are assigned to a component as fewer of the health criteria are met.

The interfaces in the `admin` package that relate to health monitoring are:

- **GemFireHealth—**Health configuration and monitoring for a distributed system. Includes `set` and `get` methods for distributed system health configuration (`DistributedSystemHealthConfig`) and for components residing on individual host machines in the distributed system (`GemFireHealthConfig`). Also provides methods for getting the overall health of the system, for retrieving diagnostic information in the case of ill health, and for resetting all health indicators in the system to optimal.
- **DistributedSystemHealthConfig—**Health configuration for the distributed system. Settings include the number of unexpected departures from the distributed system by system members.
- **GemFireHealthConfig—**Health configuration for components of a distributed system that reside on a given machine. This combines `MemberHealthConfig`, and `CacheHealthConfig`.
- **CacheHealthConfig—**Health configuration for cache instances. Settings include:
  - Durations for `netSearch` and load operations
  - Cache hit ratio
  - Event delivery queue size
- **MemberHealthConfig—**Health configuration for distributed system members. Settings include:
  - VM process size
  - Incoming and outgoing message queue sizes
  - Number of timeouts waiting for replies from other members
  - Multicast retransmission rates
- **GemFireHealth.Health**—An enumerated type for the health of GemFire.

Health monitoring is managed from the `GemFireHealth` interface. You use it to set and access configuration settings for the distributed system and for system components. You can set default configuration settings for distributed system components that you can override for any host machine. This figure shows the relationships between the interfaces.

Figure 19.2 Setting the GemFire Health Configuration

`GemFireHealth` oversees all configuration settings and provides monitoring and management tools. The host machines that do not have a `GemFireHealthConfig` associated with them are monitored using the default GemFire health configuration settings.

GemFire health configuration settings cover cache and member configurations. Member configuration settings apply to each member in the distributed system. Cache settings apply to every cache in the system.

19.2 Programming Examples

This section provides examples for the following:

- Distributed System Management (page 485)
- Accessing Distributed System Processes (page 488)
- Starting and Stopping Managed Entities (page 488)
- Accessing System Member Information (page 489)
- Cache Management (page 489)
- Region Management (page 491)
- Health Monitoring (page 493)

Distributed System Management

An instance of `AdminDistributedSystem` implements the administrative interface for controlling, monitoring, and managing a GemFire distributed system.

The following examples show how to use the `AdminDistributedSystemFactory` class to define and create an instance of `AdminDistributedSystem`. The first example shows how to create a dedicated admin interface and the second example illustrates the method to create a colocated admin interface. All attributes set on `DistributedSystemConfig` (in these examples, a multicast address and port number) are used to define an `AdminDistributedSystem`.

Invoking `com.gemstone.gemfire.admin.AdminDistributedSystem.connect` configures the connection using `DistributedSystemConfig` system properties (including its defaults), and any settings in the `gemfire.properties` file. For details on `gemfire.properties` and system properties, see Chapter 3, *Configuring the System*, on page 41 in the *GemFire Enterprise System Administrator's Guide*.

The `AdminDistributedSystemFactory.setEnableAdministrationOnly` method enables you to configure GemFire administration as either dedicated or colocated. A dedicated admin interface limits the VM only to administration APIs, while a colocated interface allows all GemFire APIs. To define an interface as dedicated, set the `AdminDistributedSystemFactory.setEnableAdministrationOnly` property to `true` as shown in the following example. The default value is `false`.

If `setEnableAdministrationOnly` is set to `true`, be careful to only use the GemFire APIs from the `com.gemstone.gemfire.admin` package. In particular, do not create a `Cache` or a normal `DistributedSystem`. For the caveats associated with using a colocated admin interface, see *Limitations for Using a Colocated Admin API* on page 487.

Example 19.1 Creating a Distributed System Administration Interface (dedicated to administration only)

```
AdminDistributedSystemFactory.setEnableAdministrationOnly(true);
DistributedSystemConfig config =
   AdminDistributedSystemFactory.defineDistributedSystem();
config.setMcastAddress("224.0.0.250");
config.setMcastPort(10333);
AdminDistributedSystem admin =
AdminDistributedSystemFactory.getDistributedSystem(config);
admin.connect();
// Wait for the connection to be made
long timeout = 30 * 1000;
try {
   if (!admin.waitToBeConnected(timeout)) {
      String s = "Could not connect after " + timeout + "ms";
      throw new Exception(s);
   }
}
catch (InterruptedException ex) {
   String s = "Interrupted while waiting to be connected";
   throw new Exception(s, ex);
}
```

The following example creates an `AdminDistributedSystem` colocated on a `DistributedSystem` connection, which allows the Admin API and Caching API to be used in the same member.

Example 19.2 Creating a Distributed System Administration Interface (colocated with DistributedSystem connection)

```
Properties props = new Properties();
props.setProperty("mcast-address", "224.0.0.250");
props.setProperty("mcast-port", "10333");
// Before connecting to either AdminDistributedSystem or DistributedSystem,
// call setEnableAdministrationOnly.
AdminDistributedSystemFactory.setEnableAdministrationOnly(false);
DistributedSystem connection = DistributedSystem.connect(props);
DistributedSystemConfig config =
AdminDistributedSystemFactory.defineDistributedSystem(connection, null);
AdminDistributedSystem admin =
AdminDistributedSystemFactory.getDistributedSystem(config);
admin.connect();
// Wait for the connection to be made
long timeout = 30 * 1000;
try {
   if (!admin.waitToBeConnected(timeout)) {
      String s = "Could not connect after " + timeout + "ms";
      throw new Exception(s);
   }
}
catch (InterruptedException ex) {
   String s = "Interrupted while waiting to be connected";
   throw new Exception(s, ex);
}
```

You can use the `AdminDistributedSystem admin` object to start and stop all managed members of the distributed system:

Example 19.3 Starting a Distributed System

```
this.system.start();
```

Example 19.4 Stopping a Distributed System

```
this.system.stop();
```

Limitations for Using a Colocated Admin API

While using the Admin API, the following limitations must be noted:

- The `AlertListener` listens only for remote alerts.
- The Admin API can manage only remote members.
- The connection does not display as an Admin connection.

Accessing Distributed System Processes

Once instantiated, the `AdminDistributedSystem` can be used to access the various distributed system entities. The following code iterates through the list of all member applications in the system and identifies a match by name. The interface also provides a list of locators, through `getDistributionLocators` and a list of cache servers, through `getCacheVms`.

Example 19.5 Locating a System Member by Name

```
SystemMember findMember(String name) throws AdminException {
   if (name == null) {
      return null;
   }
   SystemMember[] apps = this.system.getSystemMemberApplications();
   for (int i = 0; i < apps.length; i++) {
      if (name.equals(apps[i].getName())) {
         return apps[i];
      }
   }
   return null;
}
```

Starting and Stopping Managed Entities

Locators and cache servers are started and stopped when the distributed system is started and stopped. They can also be started and stopped individually. The `DistributionLocator` and `CacheVm` interfaces extend `ManagedEntity`, which provides `start` and `stop` methods. This example stops the selected cache server.

Example 19.6 Stopping a cacheserver

```
CacheVm csvr = (CacheVm) this.entity.getObject();
csvr.stop();
```

Accessing System Member Information

Applications and cache servers are system members. In the admin API, applications are represented by `SystemMember` objects, and the `CacheVm` interface extends the `SystemMember` interface. Through the `SystemMember` interface you can access configuration parameters, license information and the member's `AdminDistributedSystem` object. This code lists basic `SystemMember` information:

Example 19.7 Listing System Member Details

```
void listSystemMember() {
    SystemMember member = (SystemMember) this.entity.getObject();
    System.out.println("System Member Application Details:");
    System.out.println("\t" + "id: " + member.getId());
    System.out.println("\t" + "host: " + member.getHost());

    ConfigurationParameter[] configs = member.getConfiguration();
    System.out.println("Configuration Parameters:");
    for (int i = 0; i < configs.length; i++) {
        System.out.println("\t"
            + configs[i].getName() + "=" + configs[i].getValueAsString());
    }

    try {
        StatisticResource[] stats = member.getStats();
        System.out.println("Statistic Resources:");
        for (int i = 0; i < stats.length; i++) {
            System.out.println("\t" + stats[i].getName());
        }
    } catch (AdminException e) {
        e.printStackTrace();
    }
}
```

Cache Management

The `SystemMember` interface also provides the `SystemMemberCache` object, which allows you to manage cache attributes, cache statistics, and cache regions. This example retrieves the `SystemMemberCache` object from a `SystemMember`.

Example 19.8 Retrieving a Member's Cache

```
SystemMember member = (SystemMember) this.entity.getObject();
SystemMemberCache cache = member.getCache();
```

This example lists the cache contents after updating its information with the `refresh` method.

Example 19.9 Listing Cache Contents

```
void listSystemMemberCache() {
   SystemMemberCache cache = (SystemMemberCache) this.entity.getObject();
   cache.refresh();

   System.out.println("System Member Cache Details:");
   System.out.println("\t" + "id: " + cache.getId());
   System.out.println("\t" + "name: " + cache.getName());
   System.out.println("\t" + "isClosed: " + cache.isClosed());
   System.out.println("\t" + "lockTimeout: " + cache.getLockTimeout());
   System.out.println("\t" + "lockLease: " + cache.getLockLease());
   System.out.println("\t" + "searchTimeout: " + cache.getSearchTimeout());
   System.out.println("\t" + "upTime: " + cache.getUpTime());

   System.out.println("Region Names:");
   Set regionNames = cache.getRegionNames();
   for (Iterator it = regionNames.iterator(); it.hasNext();) {
      System.out.println("\t" + it.next());
   }

   System.out.println("Cache Statistics:");
   Statistic[] stats = cache.getStatistics();
   for (int i = 0; i < stats.length; i++) {
      System.out.println("\t" +
            stats[i].getName() + "=" + stats[i].getValue());
   }
}
```

In addition to the information it gives you, the `SystemMemberCache` allows you to configure what the cache does, including starting `CacheVms` in remote caches. For complete details on available functionality, see the online Java API documentation. The next section shows region management in the cache.

Region Management

The `SystemMemberCache` interface allows you to create regions in the cache. While a cache is usually managed by its own member, this capability is particularly useful for dynamically creating regions in a `cacheserver`. You can create a region in a `cacheserver` that outlives any of the applications that create or use it. For more information, see *Replication and Local Destroy and Invalidate Operations* on page 200. If you need to create regions across many members, you might be better served by the dynamic region functionality, described in *Dynamic Region Management* on page 122.

This example shows region creation using the `SystemMemberCache`.

Example 19.10 Creating a Region

```
void createRegion(String command) throws AdminException {
    String name = parseName(command);
    if (name.length() == 0) {
        System.out.println("Please provide a name for the region");
    }
    else {
        if (this.entity.getContext().isSystemMemberCache()) {
            SystemMemberCache cache =
                (SystemMemberCache) this.entity.getObject();
            AttributesFactory fac = new AttributesFactory();
            SystemMemberRegion rgn =
                cache.createRegion(name, fac.create());
        }
        else {
            System.out.println("Please choose a cache context before creating
a region.");
        }
    }
}
```

This region creation method returns a `SystemMemberRegion`, similar to how the `Cache` and `Region` interfaces operate in the `com.gemstone.gemfire.cache` package.

As with the other administrative entities, `Region` contents are available through the API. This example lists a region's attributes and number of entries.

Example 19.11 Retrieving Region Contents

```
void listSystemMemberRegion() {
   SystemMemberRegion region = (SystemMemberRegion) this.entity.getObject();
   region.refresh();

   System.out.println("System Member Region Details:");
   System.out.println("\t" + "name: " + region.getName());
   System.out.println("\t" + "entryCount: " + region.getEntryCount());

   System.out.println("Region Attributes:");
   System.out.println("\t" + "UserAttribute: " + region.getUserAttribute());
   System.out.println("\t" + "CacheLoader: " + region.getCacheLoader());
   System.out.println("\t" + "CacheWriter: " + region.getCacheWriter());
      . . .
   System.out.println("\t" + "DataPolicy: " + region.getDataPolicy());

   System.out.println("Region Statistics:");
   System.out.println("\t" + "LastModifiedTime: "
      + region.getLastModifiedTime());
   System.out.println("\t" + "LastAccessedTime: "
      + region.getLastAccessedTime());
   System.out.println("\t" + "HitCount: " + region.getHitCount());
   System.out.println("\t" + "MissCount: " + region.getMissCount());
   System.out.println("\t" + "HitRatio: " + region.getHitRatio());
}
```

Health Monitoring

This example registers and configures health MBeans for the distributed system, then invokes a `setupHealthForAllHosts` method to create health monitoring instances for all machines running distributed system components.

Example 19.12 Configuring Health Objects

```
private void setupHealthMonitors() throws Exception {
// register the GemFireHealth MBean for the system...
   this.healthName = (ObjectName) this.mbs.invoke(this.systemName,
             "monitorGemFireHealth",new Object[0], new String[0]);

   String systemId = (String) this.mbs.getAttribute(this.systemName, "id");

// get the names of the default health config MBeans
   this.systemHealthConfigName =
      new ObjectName("GemFire:type=DistributedSystemHealthConfig,id="
             + systemId);
   this.defaultHealthConfigName = new ObjectName
      ("GemFire:type=GemFireHealthConfig,id=" + systemId + ",host=default");
   . . .
   Integer seconds = (Integer)
      this.mbs.getAttribute(this.defaultHealthConfigName,
             "healthEvaluationInterval");

// Configure StringMonitor for the health attribute on GemFireHealth MBean
   this.healthMonitorName =
      new ObjectName("monitors:type=String,attr=health,system=" + systemId);
   this.mbs.createMBean("javax.management.monitor.StringMonitor",
             this.healthMonitorName);

   AttributeList al = new AttributeList();
   al.add(new Attribute("ObservedObject", this.healthName));
   al.add(new Attribute("ObservedAttribute", "healthStatus"));
   al.add(new Attribute("GranularityPeriod",
             new Integer(seconds.intValue() * 1000))); // in millis
   al.add(new Attribute("StringToCompare",
             GemFireHealth.GOOD_HEALTH.toString()));
   al.add(new Attribute("NotifyMatch", new Boolean(true)));
   al.add(new Attribute("NotifyDiffer", new Boolean(true)));
   this.mbs.setAttributes(this.healthMonitorName, al);

   this.mbs.addNotificationListener(this.healthMonitorName,
             this, null, this.mbs);
   this.mbs.invoke(this.healthMonitorName, "start",
             new Object[0], new String[0]);
   . . .

// next register health config MBeans for each host in the system...
   setupHealthForAllHosts();
}
```

The `setupHealthForAllHosts` method runs through all system members, calling another method, `checkHostFor`, that checks to see if the member's host has a health configuration registered. This is how the `setupHealthForAllHosts` method works:

Example 19.13 Iterate System Members, Registering Health Configuration for the Member's Machines

```
private void setupHealthForAllHosts() throws Exception {
// make sure health config MBeans are registered for application hosts...
   ObjectName[] memberNames = (ObjectName[])
      this.mbs.invoke(this.systemName, "manageSystemMemberApplications",
      new Object[0], new String[0]);
   for (int i = 0; i < memberNames.length; i++) {
      checkHostFor(memberNames[i]);
   }
}
```

This is the `checkHostFor` method:

Example 19.14 Register Health Configuration for Each Member's Machine

```
private void checkHostFor(ObjectName memberName) throws Exception {
   if (memberName == null) return;
   synchronized(this.healthConfigHostMap) {

// get the member's host...
      String host = (String) this.mbs.getAttribute(memberName, "host");

// is there a config MBean registered for the host yet?
      ObjectName gemfireHealthConfigName =
         (ObjectName) this.healthConfigHostMap.get(host);

      if (gemfireHealthConfigName == null
            || !this.mbs.isRegistered(memberName)) {
// if not registered, create a new config MBean for the host...
         gemfireHealthConfigName =
               (ObjectName)this.mbs.invoke(this.healthName,
                     "manageGemFireHealthConfig",
                     new Object[] { host },
                     new String[] { "java.lang.String" });
         this.healthConfigHostMap.put(host, gemfireHealthConfigName);
      }
   }
}
```

Once the health configuration is registered, when changes happen in the system the listener is notified and can handle the events. In this example case, if a new member has joined, the method calls `checkHostFor` to make sure the member's host has a health configuration registered. In all events, the method prints a message to standard output.

Example 19.15 Event Handler for Health-Related Notifications

```
public void handleNotification(Notification notification, Object handback) {
    String type = notification.getType();
    System.out.println("handleNotification: " + type);
    if ("gemfire.distributedsystem.member.joined".equals(type)) {
        String memberId = notification.getMessage();
        System.out.println("MEMBER JOINED: " + memberId);
// Make sure the new member's machine has a registered health configuration
        try {
            checkHostFor(findSystemMember(memberId));
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    else if ("gemfire.distributedsystem.member.left".equals(type)) {
        String memberId = notification.getMessage();
        System.out.println("MEMBER LEFT: " + memberId);
    }
    else if ("gemfire.distributedsystem.member.crashed".equals(type)) {
        String memberId = notification.getMessage();
        System.out.println("MEMBER CRASHED: " + memberId);
    }
    else if ("gemfire.distributedsystem.alert".equals(type)) {
        String alert = notification.getMessage();
        System.out.println("ALERT: " + alert);
    }
    else if
(MonitorNotification.STRING_TO_COMPARE_VALUE_DIFFERED.equals(type)) {
        String health = notification.getMessage();
        System.out.println("HEALTH UPDATED: " + health);
    }
}
```

Appendix

A gemfire.properties Example File

This appendix lists the contents of the sample `gemfire.properties` file that is included with the GemFire Enterprise® installation in the `defaultConfigs` directory. The default file provides a starting point for application programmers wishing to configure their distributed system connection using a `gemfire.properties` file. *Configuring the System* on page 41 of the *GemFire Enterprise System Administrator's Guide* describes the contents of the `gemfire.properties` file in detail.

Example A.1 $GEMFIRE/defaultConfigs/gemfire.properties

```
#Fri Apr 17 19:29:00 PDT 2009
ack-severe-alert-threshold=0
ack-wait-threshold=15
archive-disk-space-limit=0
archive-file-size-limit=0
async-distribution-timeout=0
async-max-queue-size=8
async-queue-timeout=60000
bind-address=
cache-xml-file=cache.xml
conflate-events=server
conserve-sockets=true
delta-propagation=true
departure-correlation-window=1800
disable-tcp=false
durable-client-id=
durable-client-timeout=300
enable-network-partition-detection=false
enable-time-statistics=false
gemfire.sqlfabric.bytearray-store=false
license-file=gemfireLicense.zip
license-type=evaluation
locators=
log-disk-space-limit=0
log-file=
log-file-size-limit=0
log-level=config
max-num-reconnect-tries=3
max-wait-time-reconnect=10000
mcast-address=239.192.81.1
mcast-flow-control=1048576, 0.25, 5000
mcast-port=10334
```

```
mcast-recv-buffer-size=1048576
mcast-send-buffer-size=65535
mcast-ttl=32
member-timeout=5000
name=
remove-unresponsive-client=false
roles=
security-=
security-client-accessor=
security-client-accessor-pp=
security-client-auth-init=
security-client-authenticator=
security-client-dhalgo=
security-log-file=
security-log-level=config
security-peer-auth-init=
security-peer-authenticator=
security-peer-verifymember-timeout=1000
server-bind-address=
socket-buffer-size=32768
socket-lease-time=60000
ssl-ciphers=any
ssl-enabled=false
ssl-protocols=any
ssl-require-authentication=true
start-locator=
statistic-archive-file=
statistic-sample-rate=1000
statistic-sampling-enabled=false
tcp-port=0
udp-fragment-size=60000
udp-recv-buffer-size=1048576
udp-send-buffer-size=65535
```

Appendix

B cache.xml File

To ease the task of managing the initialization of the GemFire Enterprise® Cache, you can define a cache structure in an XML file, commonly named `cache.xml`. A cache XML file helps standardize the cache definition and can be used by multiple programs to ensure consistent definitions across a distributed system. You can use the `cache.xml` file to initialize or update a cache structure.

This is a listing of a declarative file that creates a distributed data region with an event listener for handling data updates.

Example B.1 Simple XML File Listing

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<!--
  | PushConsumer.xml
| Configures a consumer region for a data replication example.
-->
<cache>
   <region name="exampleRegion">
      <region-attributes refid="REPLICATE">
         <cache-listener>
            <class-name>quickstart.SimpleCacheListener</class-name>
         </cache-listener>
      </region-attributes>
   </region>
</cache>
```

In this appendix:

- Cache XML Basics (page 500)
- Example Files (page 501)
- GemFire Enterprise Cache XML DTD (page 505)

B.1 Cache XML Basics

The contents of the cache XML file are used to populate or update a cache, either when a cache server starts up or when a Java application explicitly creates its cache or explicitly loads a new structure into an existing cache. The XML file may have any name, but is generally referred to as `cache.xml`.

File Contents

The contents of a declarative XML file correspond to APIs of the `com.gemstone.gemfire.cache` package. With the cache XML file, you can accomplish declaratively many of the cache management activities that you can program through the API.

- Each declarative cache file must include a `DOCTYPE` of one of the following forms.

  Server or peer cache:

  ```
  <!DOCTYPE cache PUBLIC
     "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
  "http://www.gemstone.com/dtd/cache6_5.dtd">
  ```

  Client cache:

  ```
  <!DOCTYPE client-cache PUBLIC
     "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
  "http://www.gemstone.com/dtd/cache6_5.dtd">
  ```

- The contents of the file must conform to the XML definition in `$GEMFIRE/dtd/cache6_5.dtd` (see *GemFire Enterprise Cache XML DTD* on page 505).

- If you specify a class name, it must have a public zero-argument constructor and must implement the `Declarable` interface. Parameters declared in the XML are passed to the class's `init` method. For details, see the GemStone Java API for the `com.gemstone.gemfire.cache.Declarable` interface.

B.2 Example Files

The examples provided here show cache and region initialization for an application. These examples show only a subset of data configurations that are possible. For detailed information about cache and region configuration, including the default attribute settings, see the online Java documentation for the `Cache` and `RegionAttributes` interfaces. Other example files are provided throughout this guide.

This example lists a cache initialization file that creates two regions:

Example B.2 Declarative Cache Initialization: Two Regions

```
<?xml version="1.0"?>

<!-- MultiuserSecurityServer.xml -->

<!DOCTYPE cache PUBLIC
  "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<cache>
   <cache-server port="40404" maximum-time-between-pings="20000"/>
   <region name="exampleRegion" refid="REPLICATE"/>
   <region name="functionServiceExampleRegion" refid="PARTITION"/>
</cache>
```

This example creates a region with a listener event handler and data expiration settings:

Example B.3 Declarative Cache Initialization: Custom Region Attribute Settings

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<!--
  | DataExpiration.xml
  |
  | Configures a region with entry idle time expiration.
 -->
<cache>
   <region name="exampleRegion">
      <region-attributes>
         <entry-idle-time>
            <expiration-attributes timeout="10" action="destroy" />
         </entry-idle-time>
         <cache-listener>
            <class-name>quickstart.SimpleCacheListener</class-name>
         </cache-listener>
      </region-attributes>
   </region>
</cache>
```

The example below shows the creation of a region named `customers`. Points of interest for region are:

- The region is distributed.
- Its entries have time-to-live and idle-time expiration actions set. The time-to-live action is set to destroy entries that remain unmodified for fifty minutes. The idle-time action performs the default action of invalidation on entries that remain unaccessed for ten minutes.
- Any region data that is written to disk is written asynchronously to the two user-specified directories. These directories must exist prior to loading this `cache.xml` file. The data is written every thousand bytes or thirty seconds, whichever comes first.
- The region has a `LRUMemoryAttributes` eviction controller that overflows LRU entry values to disk if the region's memory footprint reaches fifty megabytes.
- The region is initialized with one entry whose value is a declarable object.

Example B.4 Declarative Cache Initialization of a Region with a Data Entry

```
<?xml version="1.0"?>
<!-- A sample GemFire declarative caching XML File -->
<!DOCTYPE cache PUBLIC
   "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">
<cache search-timeout="60" lock-lease="300">
   <region name="customers">
      <region-attributes scope="distributed-no-ack"
            statistics-enabled="true">
         <entry-time-to-live>
            <expiration-attributes action="destroy" timeout="3000"/>
         </entry-time-to-live>

         <entry-idle-time>
            <expiration-attributes timeout="600"/>
         </entry-idle-time>

         <cache-loader>
            <class-name>com.company.app.DBLoader</class-name>
            <parameter name="URL">
               <string>jdbc://12.34.56.78/mydb</string>
            </parameter>
         </cache-loader>

      </region-attributes>

      <!-- Add an entry whose value is a Declarable object -->
      <entry>
         <key><string>Louise Jones</string></key>
         <value>
            <declarable>
               <class-name>com.company.app.Customer</class-name>
               <parameter name="name">
                  <string>Louise Jones</string>
               </parameter>
               <parameter name="address1">
                  <string>123 Main Street</string>
               </parameter>
               <parameter name="city">
                  <string>New York</string>
               </parameter>
```

```
                        <parameter name="state">
                            <string>NY</string>
                        </parameter>
                        <parameter name="zip">
                            <string>12345</string>
                        </parameter>
                    </declarable>
                </value>
            </entry>
        </region>
    </cache>
```

The next example shows the use of stored named region attributes in the XML file. The attributes are defined for the entire cache, outside of any `region` element, and used in a number of regions. In the `region` elements that reference the stored attributes, all explicit attribute declarations override the stored settings.

Example B.5 Declarative Cache Initialization Using the RegionAttributes ID Outside a region Element

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC "-//GemStone Systems, Inc.//GemFire Declarative
Caching 6.5//EN" "http://www.gemstone.com/dtd/cache6_5.dtd">
<cache lock-lease="120" lock-timeout="60" search-timeout="300">
   <region-attributes id="attTemplate" scope="local" data-policy="normal"
         initial-capacity="16" load-factor="0.75" concurrency-level="16"
         statistics-enabled="true">
      <key-constraint>java.lang.String</key-constraint>
   </region-attributes>
   <region name="rlocal">
      <region-attributes refid="attTemplate">
      </region-attributes>
   </region>
   <region name="rdistnoack">
      <region-attributes refid="attTemplate" scope="distributed-no-ack">
      </region-attributes>
   </region>
   <region name="rglobalreplication">
      <region-attributes refid="attTemplate" scope="global"
            data-policy="replicate">
      </region-attributes>
   </region>
</cache>
```

The identifier can also be specified inside a `region` element the first time the attributes are used in a region, as in this example:

Example B.6 Declarative Cache Initialization Using the RegionAttributes ID Within a region Element

```
<region name="rlocal">
   <region-attributes id="attTemplate" scope="local"
         data-policy="normal" initial-capacity="16" load-factor="0.75"
         concurrency-level="16" statistics-enabled="true">
      <key-constraint>java.lang.String</key-constraint>
   </region-attributes>
</region>
<region name="rdistnoack">
   <region-attributes refid="attTemplate" scope="distributed-no-ack">
   </region-attributes>
</region>
```

It is not possible to explicitly set an attribute to `null` inside the `cache.xml` file. For attributes that default to `null`, you cannot set the attribute to a non-`null` value in a template and then override back to the default in a `region-attributes` element that references the template. In the `rdistnoack` region declaration in the example, you could not reset the `key-constraint` to `null`.

B.3 GemFire Enterprise Cache XML DTD

This example presents the file `cache6_5.dtd`, which defines the XML that is used by GemFire Enterprise for declarative caching. The contents of the XML cache initialization file must conform to the data type definitions in `cache6_5.dtd`. You can find this file in the `dtd` subdirectory of your GemFire installation.

Example B.7 cache6_5.dtd

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ======================================================== -->
<!--

This is the XML DTD for the GemFire distributed cache declarative
caching XML file.  All declarative cache files must include a DOCTYPE
of the following form:

  <!DOCTYPE cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">

If you are declaring a client cache then use this DOCTYPE:

  <!DOCTYPE client-cache PUBLIC
    "-//GemStone Systems, Inc.//GemFire Declarative Caching 6.5//EN"
"http://www.gemstone.com/dtd/cache6_5.dtd">

The contents of a declarative XML file correspond to APIs found in the
com.gemstone.gemfire.cache and com.gemstone.gemfire.cache.client
packages.  A declarative caching XML file is used to populate a Cache
or a ClientCache when it is created.

The top-level element in this syntax is "cache" or "client-cache".
All elements are listed here in alphabetical order.

The following conventions apply to all GemFire distributed cache
declarative caching XML file elements unless indicated otherwise.

- In elements that contain PCDATA, leading and trailing whitespace in
  the data may be ignored.

- In elements whose value is an "enumerated type", the value is case
  sensitive.

-->

<!-- ======================================================== -->
<!--
An "asynchronous-writes" element specifies how entry data should be
written to disk asynchronous.
-->
<!ELEMENT asynchronous-writes EMPTY>
<!ATTLIST asynchronous-writes
  bytes-threshold  CDATA #REQUIRED
  time-interval    CDATA #REQUIRED
>
```

```
<!-- ========================================================== -->
<!--
The "bridge-server" element specifies a Cache Bridge Server that is
started when the Cache is declaratively initialized.
NOTE: as of 5.7 this element is deprecated;
      use the "cache-server" element instead.
-->
<!ELEMENT bridge-server (group*, client-subscription?, custom-load-probe?)>
<!ATTLIST bridge-server
   bind-address                CDATA #IMPLIED
   hostname-for-clients        CDATA #IMPLIED
   max-connections             CDATA #IMPLIED
   maximum-message-count       CDATA #IMPLIED
   maximum-time-between-pings  CDATA #IMPLIED
   max-threads                 CDATA #IMPLIED
   message-time-to-live        CDATA #IMPLIED
   notify-by-subscription      (false | true) #IMPLIED
   port                        CDATA #IMPLIED
   socket-buffer-size          CDATA #IMPLIED
   load-poll-interval          CDATA #IMPLIED
>

<!-- ========================================================== -->
<!--
The "cache-server" element specifies a Cache Server
that will accept requests from clients and
that is started when the Cache is declaratively initialized.
-->
<!ELEMENT cache-server (group*, client-subscription?, custom-load-probe?)>
<!ATTLIST cache-server
   bind-address                CDATA #IMPLIED
   hostname-for-clients        CDATA #IMPLIED
   max-connections             CDATA #IMPLIED
   maximum-message-count       CDATA #IMPLIED
   maximum-time-between-pings  CDATA #IMPLIED
   max-threads                 CDATA #IMPLIED
   message-time-to-live        CDATA #IMPLIED
   notify-by-subscription      (false | true) #IMPLIED
   port                        CDATA #IMPLIED
   socket-buffer-size          CDATA #IMPLIED
   load-poll-interval          CDATA #IMPLIED
>

<!--
 A "client-subscription" element describes the eviction policy,
 capacity and overflow-directory for client subscription config.
-->
<!ELEMENT client-subscription EMPTY>
<!ATTLIST client-subscription
   eviction-policy             (entry | mem) #REQUIRED
   capacity                    CDATA #REQUIRED
   disk-store-name             CDATA #IMPLIED
   overflow-directory          CDATA #IMPLIED
>

<!-- ========================================================== -->
<!--
A "custom-load-probe" element describes a callback that reports
        the load on a cache-server.
```

```
-->
<!ELEMENT custom-load-probe (
        class-name,
        parameter*
        )>

<!-- ========================================================== -->
<!--
The "cache" element is the root element of the declarative cache file
on a peer or server.
This element configures a GemFire Cache and describes the root regions it
contains, if any.
-->

<!ELEMENT cache (
    cache-transaction-manager?,
    dynamic-region-factory?,
    gateway-hub*,
    cache-server*,
    bridge-server*,
    pool*,
    disk-store*,
    region-attributes*,
    (jndi-bindings | region | vm-root-region)*,
    function-service?,
    resource-manager?,
    serialization-registration?,
    backup*
    )>
<!ATTLIST cache
  copy-on-read           (false | true) #IMPLIED
  is-server              (false | true) #IMPLIED
  lock-timeout           CDATA #IMPLIED
  lock-lease             CDATA #IMPLIED
  message-sync-interval  CDATA #IMPLIED
  search-timeout         CDATA #IMPLIED
>

<!--
The "client-cache" element is the root element of the declarative cache file
on a client.
This element configures a GemFire ClientCache and describes the root regions
it
contains, if any.
-->
<!ELEMENT client-cache (
    dynamic-region-factory?,
    pool*,
    disk-store*,
    region-attributes*,
    region*,
    function-service?,
    resource-manager?,
    serialization-registration?
    )>
<!ATTLIST client-cache
  copy-on-read           (false | true) #IMPLIED
>
```

```
<!-- ========================================================== -->
<!--
The  backup element defines files or directories that should
be backed up when the system wide backup command is invoked.
Disk stores with persistent data are automatically backed up,
they do not need to be listed with this element.
-->
<!ELEMENT backup (#PCDATA)>

<!-- ========================================================== -->
<!--
The "resource manager" element configures the behavior of the resource
manager.
The resource manager provides support for resource management of its
associated Cache -->
<!ELEMENT resource-manager EMPTY>
<!ATTLIST resource-manager
  critical-heap-percentage  CDATA #IMPLIED
  eviction-heap-percentage  CDATA #IMPLIED
>


<!-- ========================================================== -->
<!--
The "function service" element configures the behavior of the function
service.
The function service allows users to execute functions on data stored in
GemFire -->
<!ELEMENT function-service (
    function*
  )>

<!-- ========================================================== -->
 <!--
The "function" element defines a function for registration in the function
service
-->
<!ELEMENT function (class-name, parameter*)>


<!-- ========================================================== -->
<!--
A "cache-listener" element describes a region's CacheListener.
-->
<!ELEMENT cache-listener (
    class-name,
    parameter*
  )>

<!-- ========================================================== -->
<!--
A "cache-loader" element describes a region's CacheLoader.
-->
<!ELEMENT cache-loader (
class-name,
    parameter*
  )>
```

```
<!-- ========================================================= -->
<!--
A "cache-transaction-manager" element allows insertion of cache-level
transaction listeners.
-->
<!ELEMENT cache-transaction-manager (transaction-listener*,transaction-
writer?)>

<!-- ========================================================= -->
<!--
A "cache-writer" element describes a region's CacheWriter.
-->
<!ELEMENT cache-writer (
    class-name,
    parameter*
  )>

<!-- ========================================================= -->
<!--
A serialization-registration contains a set of serializer or
instantiator tags to register customer DataSerializer extensions
or DataSerializable implementations respectively.
-->
<!ELEMENT serialization-registration (
    serializer*,
    instantiator*
  )>

<!-- ========================================================= -->
<!--
A serializer registers a custom class which extends DataSerializer
to support custom serialization of non-modifiable object types inside
GemFire.
-->
<!ELEMENT serializer (
    class-name
  )>

<!-- ========================================================= -->
<!--
An instantiator registers a custom class which implements the
DataSerializable interface to support custom object serialization
inside GemFire.
-->
<!ELEMENT instantiator (
    class-name
  )>
<!--
The id that the instantiator should associate with the DataSerializable
type.
-->
<!ATTLIST instantiator
    id  CDATA #REQUIRED
>

<!-- ========================================================= -->
<!--
A "class-name" element specifies the name of a Declarable class, an
instance of which is created when a declarative cache XML file is
```

```
read.
-->
<!ELEMENT class-name (#PCDATA)>

<!-- ========================================================= -->
<!ELEMENT config-property (
    config-property-name,
    config-property-type,
    config-property-value
  )>

<!-- ========================================================= -->
<!ELEMENT config-property-name (#PCDATA)>

<!-- ========================================================= -->
<!ELEMENT config-property-type (#PCDATA)>

<!-- ========================================================= -->
<!ELEMENT config-property-value (#PCDATA)>

<!-- ========================================================= -->
<!--
A "disk-store" element specifies a DiskStore for persistence.
-->
<!ELEMENT disk-store (disk-dirs?)>
<!ATTLIST disk-store
  name                          CDATA #REQUIRED
  auto-compact                  (false | true) #IMPLIED
  compaction-threshold          CDATA #IMPLIED
  allow-force-compaction        (false | true) #IMPLIED
  max-oplog-size                CDATA #IMPLIED
  time-interval                 CDATA #IMPLIED
  write-buffer-size             CDATA #IMPLIED
  queue-size                    CDATA #IMPLIED
>

<!-- ========================================================= -->
<!--
A "pool" element specifies a client to server connection pool.
-->
<!ELEMENT pool (locator+|server+)>
<!ATTLIST pool
  free-connection-timeout                CDATA #IMPLIED
  load-conditioning-interval             CDATA #IMPLIED
  min-connections                        CDATA #IMPLIED
max-connections                        CDATA #IMPLIED
  retry-attempts                         CDATA #IMPLIED
  idle-timeout                           CDATA #IMPLIED
  ping-interval                          CDATA #IMPLIED
  name                                   CDATA #REQUIRED
  read-timeout                           CDATA #IMPLIED
  server-group                           CDATA #IMPLIED
  socket-buffer-size                     CDATA #IMPLIED
  subscription-enabled                   (false | true) #IMPLIED
  subscription-message-tracking-timeout  CDATA #IMPLIED
  subscription-ack-interval              CDATA #IMPLIED
  subscription-redundancy                CDATA #IMPLIED
  statistic-interval                     CDATA #IMPLIED
  thread-local-connections               (false | true) #IMPLIED
```

```
  pr-single-hop-enabled                  (false | true) #IMPLIED
  multiuser-authentication               (false | true) #IMPLIED
>

<!-- ========================================================== -->
<!--
The "custom-expiry" element specifies a callback that specifies
custom expiration attributes.  See the CustomExpiry class.
-->
<!ELEMENT custom-expiry (
    class-name,
    parameter*
  )>
<!ATTLIST custom-expiry
>

<!-- ========================================================== -->
<!--
A "declarable" element specifies a Declarable object to be placed in a
Region entry.
-->
<!ELEMENT declarable (
    class-name,
    parameter*
  )>

<!-- ========================================================== -->
<!--
A "disk-dir" element specifies one of a region or diskstore's disk
directories.
-->
<!ELEMENT disk-dir (#PCDATA)>
<!ATTLIST disk-dir
  dir-size              CDATA #IMPLIED
>

<!-- ========================================================== -->
<!--
A "disk-dirs" element specifies the region's disk directories.
-->
<!ELEMENT disk-dirs (disk-dir+)>
<!-- ========================================================== -->
<!--
A "disk-write-attributes" element describes how entry data is written
to disk. Deprecated as of 6.5 use disk-store instead.
-->
<!ELEMENT disk-write-attributes (asynchronous-writes | synchronous-writes)>
<!ATTLIST disk-write-attributes
  max-oplog-size  CDATA #IMPLIED
  roll-oplogs     CDATA #IMPLIED
>

<!-- ========================================================== -->
<!--
A "dynamic-region-factory" element configures a dynamic region factory for
this cache.  If this optional element is missing then the cache does not
support dynamic regions.

The optional "disk-dir" sub-element can be used to specify the directory to
```

```
store the persistent files that are used for dynamic region bookkeeping.
It defaults to the current directory.

The pool-name attribute can be used to set the name of the connection
        pool used by client applications in a client/server cache
configuration.
        It should not be specified in servers or peers.

The optional "cache-loader" and "cache-writer" are for legacy client client
         applications that doen't use a connection pool. They must be a
        bridge-loader and a bridge-writer. They should not be specified in
        servers or peers.
-->
<!ELEMENT dynamic-region-factory (
    disk-dir?,
    cache-loader?,
    cache-writer?
    )>
<!ATTLIST dynamic-region-factory
        disable-persist-backup     (true | false) #IMPLIED
        disable-register-interest  (true | false) #IMPLIED
        pool-name                  CDATA #IMPLIED
>

<!-- ======================================================== -->
<!--
An "entry" element desribes an entry to be added to a region.  Note
that if an entry with the given key already exists in the region, it
will be replaced.
-->
<!ELEMENT entry (
    key,
    value
  )>
<!-- ======================================================== -->
<!--
A "entry-idle-time" element specifies a Region's entries' idle time.
-->
<!ELEMENT entry-idle-time (expiration-attributes)>

<!-- ======================================================== -->
<!--
A "entry-time-to-live" element specifies a Region's entries' time to
live.
-->
<!ELEMENT entry-time-to-live (expiration-attributes)>

<!-- ======================================================== -->
<!--
Create eviction attributes that monitor and control the size of the Region.
-->
<!ELEMENT eviction-attributes
    (lru-entry-count | lru-heap-percentage | lru-memory-size)?
  >

<!-- ======================================================== -->
<!--
An "expiration-attributes" element describes expiration.
-->
```

```
<!ELEMENT expiration-attributes (
  custom-expiry?
  )>
<!ATTLIST expiration-attributes
  action   (destroy | invalidate | local-destroy | local-invalidate)
#IMPLIED
  timeout  CDATA #REQUIRED
>

<!-- ======================================================== -->
<!--
A functional type of index needs a from-clause, expression which are
mandatory.
The import string is used for specifying the type of Object in the region or
the type of Object which the indexed expression evaluates to.
-->
<!ELEMENT functional EMPTY>
<!ATTLIST functional
  expression   CDATA #REQUIRED
  from-clause  CDATA #REQUIRED
  imports      CDATA #IMPLIED
>

<!-- ======================================================== -->
<!--
An "index" element describes an index to be created on a region in
GemfireCache. The index node, if any, should all come immediately after the
"region-attributes" node. There can be any number of  functional types
of index on a Region, but only at max one index of type "primary-key". The
"name" attribute is a required field which identifies the name of the index.
-->
<!ELEMENT index (functional | primary-key)>
<!ATTLIST index
  name  CDATA #REQUIRED
>

<!-- ======================================================== -->
<!--
The "gateway" element specifies a Gateway that is
initialized when a GatewayHub is declaratively initialized.
-->
<!ELEMENT gateway (
    (gateway-endpoint+ | gateway-listener+),
    gateway-queue?
  )>
<!ATTLIST gateway
  early-ack            (false | true) #IMPLIED
  id                   CDATA #REQUIRED
  socket-buffer-size   CDATA #IMPLIED
  socket-read-timeout  CDATA #IMPLIED
>

<!-- ======================================================== -->
<!--
The "gateway-endpoint" element specifies an endpoint that is
initialized when a Gateway is declaratively initialized.
-->
<!ELEMENT gateway-endpoint EMPTY>
<!ATTLIST gateway-endpoint
```

```
  host  CDATA #REQUIRED
  id    CDATA #REQUIRED
  port  CDATA #REQUIRED
>

<!-- ========================================================== -->
<!--
The "gateway-hub" element specifies a Gateway Hub that is
initialized when the Cache is declaratively initialized.
-->
<!ELEMENT gateway-hub (gateway*)>
<!ATTLIST gateway-hub
  id                          CDATA #REQUIRED
  maximum-time-between-pings  CDATA #IMPLIED
  port                        CDATA #IMPLIED
  socket-buffer-size          CDATA #IMPLIED
  startup-policy              (primary | secondary | none) #IMPLIED
  distribution-policy         (all | parallel) #IMPLIED
  manual-start                (false | true) #IMPLIED
>

<!-- ========================================================== -->
<!--
The "gateway-listener" element specifies a listener that is
initialized when a Gateway is declaratively initialized.
-->
<!ELEMENT gateway-listener (
    class-name,
    parameter*
  )>
<!ATTLIST gateway-listener
>

<!-- ========================================================== -->
<!--
The "gateway-queue" element specifies a queue that is
initialized when a Gateway is declaratively initialized.
-->
<!ELEMENT gateway-queue EMPTY>
<!ATTLIST gateway-queue
  alert-threshold       CDATA #IMPLIED
  batch-conflation      (false | true) #IMPLIED
  batch-size            CDATA #IMPLIED
  batch-time-interval   CDATA #IMPLIED
  enable-persistence    (false | true) #IMPLIED
  disk-store-name       CDATA #IMPLIED
  roll-oplogs           (false | true) #IMPLIED
  maximum-queue-memory  CDATA #IMPLIED
  overflow-directory    CDATA #IMPLIED
>

<!-- ========================================================== -->
<!--
A "group" element specifies a single server group.
-->
<!ELEMENT group (#PCDATA)>

<!-- ========================================================== -->
<!--
```

```
For every datasource which is bound to the JNDI tree, there will exist one
jndi-binding tag. This tag will describe the property & configuration of the
datasource . The attributes of jndi-binding are used for configuration by
Gemfire, while the property element is used to configure the custom property
for the datasource. Of the attributes list the type & jndi-name are
required.

impleDataSource.xa-datasource-class is needed when type is
XAPooledDataSource,
while conn-pooled-datasource-class is needed when type is PooledDataSource.
jdbc-driver-class is needed when the type is SimpleDataSource.  It is
advisable
to set the user-name & password as attributes rather than as a property
element. Alternatively, they may be set both as attributes of jndi-binding
tag
as well as an attribute of property tag
-->
<!ELEMENT jndi-binding (config-property*)>
<!ATTLIST jndi-binding
  blocking-timeout-seconds      CDATA #IMPLIED
  conn-pooled-datasource-class  CDATA #IMPLIED
  connection-url                CDATA #IMPLIED
  idle-timeout-seconds          CDATA #IMPLIED
  init-pool-size                CDATA #IMPLIED
  jdbc-driver-class             CDATA #IMPLIED
jndi-name                      CDATA #REQUIRED
  login-timeout-seconds         CDATA #IMPLIED
  managed-conn-factory-class    CDATA #IMPLIED
  max-pool-size                 CDATA #IMPLIED
  password                      CDATA #IMPLIED
  transaction-type              CDATA #IMPLIED
  type                           (ManagedDataSource | SimpleDataSource |
                                 PooledDataSource | XAPooledDataSource)
#REQUIRED
  user-name                     CDATA #IMPLIED
  xa-datasource-class           CDATA #IMPLIED
>

<!-- ======================================================== -->
<!--
A jndi-bindings element will contain the jndi-binding for each of the
datasources which are to be bound with the JNDI Context.
-->
<!ELEMENT jndi-bindings (jndi-binding*)>

<!-- ======================================================== -->
<!--
A "key" element describes the key in a Region entry.
-->
<!ELEMENT key (string | declarable)>

<!-- ======================================================== -->
<!--
A "key-constraint" element specifies the name of a class to which the
keys of a region are constrained.
-->
<!ELEMENT key-constraint (#PCDATA)>

<!-- ======================================================== -->
```

```
<!--
A "locator" element specifies the host and port that a server locator is
listening on
-->
<!ELEMENT locator EMPTY>
<!ATTLIST locator
  host  CDATA #REQUIRED
  port  CDATA #REQUIRED
>

<!-- ========================================================= -->
<!--
Create an LRU eviction controller which performs the action if the
Region has more than the maximum number of Entries in the Region.
-->
<!ELEMENT lru-entry-count EMPTY>
<!ATTLIST lru-entry-count
    action   (local-destroy | overflow-to-disk) #IMPLIED
    maximum  CDATA #IMPLIED
  >
<!-- ========================================================= -->
<!--
Create an LRU which evicts entries using the specified action
when the heap is over the resource manager critical-heap-percentage.
When the threshold has been exceeded each create or put operation
is required to evict the same size of data which was added. The
optional class-name and parameters allow for the declaration
of an ObjectSizer, which will be used to determine data size.
The class must implement the ObjectSizer interface.
-->
<!ELEMENT lru-heap-percentage (
    class-name,
    parameter*
)?>
<!ATTLIST lru-heap-percentage
    action  (local-destroy | overflow-to-disk) #IMPLIED
  >

<!-- ========================================================= -->
<!--
Create an LRU that performs the action when the memory size of the Region is
over the maximum.  The optional class-name and parameters allow for the
declaration an ObjectSizer, which is used to measure the size of each Object
in the Region.  The class must implement the ObjectSizer interface.
-->
<!ELEMENT lru-memory-size (
    class-name,
    parameter*
  )?>
<!ATTLIST lru-memory-size
    action   (local-destroy | overflow-to-disk) #IMPLIED
    maximum  CDATA #IMPLIED
  >

<!-- ========================================================= -->
<!--
A "membership-attributes" element configures a region for membership
constraints based on required roles.
-->
```

```
<!ELEMENT membership-attributes (required-role*)>
<!ATTLIST membership-attributes
  loss-action        (full-access | limited-access | no-access | reconnect)
                     #IMPLIED
  resumption-action  (none | reinitialize) #IMPLIED
>

<!-- ========================================================== -->
<!--
A "parameter" element describes a parameter used to initialize a
Declarable object.
-->
<!ELEMENT parameter (string | declarable)>
<!ATTLIST parameter
  name  CDATA #REQUIRED
>

<!-- ========================================================== -->
<!--
A "partition-attributes" element configures a region for partitioned
data storage.

  cache-loader element removed from partition-attributes in release 5.1
  local-max-memory attribute added to partition-attributes in release 5.1
  total-max-memory attribute added to partition-attributes in release 5.1
  total-num-buckets attribute added to partition-attributes in release 5.1
  local-properties and global-properties attributes removed in release 5.1
  partition-resolver attribute added to partition-attributes in release 5.7
  partition-listener attribute added to partition-attributes in release 6.1
  colocated-with attribute added to partition-attributes in release 5.7
  redundancy-delay attribute added to partition-attributes in release 6.0
  startup-redundancy-delay attribute added to partition-attributes in
release 6.0
-->
<!ELEMENT partition-attributes (partition-resolver?, partition-listener*)>
<!ATTLIST partition-attributes
  local-max-memory        CDATA #IMPLIED
  recovery-delay          CDATA #IMPLIED
  redundant-copies        CDATA #IMPLIED
  startup-recovery-delay  CDATA #IMPLIED
  total-max-memory        CDATA #IMPLIED
  total-num-buckets       CDATA #IMPLIED
  colocated-with          CDATA #IMPLIED
>
<!--
        A "partition-resolver" element describes a partitioned region's
custom
        PartitionResolver.
-->
<!ELEMENT partition-resolver (class-name, parameter*)>
<!ATTLIST partition-resolver
  name CDATA #IMPLIED
>

<!--
        A "partition-listenerr" element describes a partitioned region's
custom
        PartitionListener.
-->
```

```
<!ELEMENT partition-listener (class-name, parameter*)>

<!-- Will put refid later.....refid   IDREF #IMPLIED -->
<!-- ========================================================= -->
<!--
A primary-key type of index needs a field attribute  which is mandatory.
There should be only one or zero primary-index defined for a region
-->
<!ELEMENT primary-key EMPTY>
<!ATTLIST primary-key
  field  CDATA #REQUIRED
>
<!-- ========================================================= -->
<!--
A "region" element describes a region (and its entries) in GemFire
distributed cache.  It may be used to create a new region or may be
used to add new entries to an existing region.  Note that the "name"
attribute specifies the simple name of the region; it cannot contain a
"/".
If "refid" is set then it defines the default region attributes to use for
this region.
A nested "region-attributes" element can override these defaults.
If the nested "region-attributes" element has its own "refid" then it will
cause the
"refid" on the region to be ignored.
-->
<!ELEMENT region (
    region-attributes*,
    index*,
    entry*,
    region*
)>
<!ATTLIST region
  name  CDATA #REQUIRED
  refid CDATA #IMPLIED
>

<!-- ========================================================= -->
<!--
A "region-attributes" element describes the attributes of a region to
be created.
-->
<!ELEMENT region-attributes (
    key-constraint?,
    value-constraint?,
    region-time-to-live?,
    region-idle-time?,
    entry-time-to-live?,
    entry-idle-time?,
    disk-write-attributes?,
    disk-dirs?,
    partition-attributes?,
    membership-attributes?,
    subscription-attributes?,
    cache-loader?,
    cache-writer?,
    cache-listener*,
    eviction-attributes?
  )>
```

```
<!ATTLIST region-attributes
  concurrency-level               CDATA #IMPLIED
  data-policy                     (empty | normal | partition
                                  | persistent-replicate | preloaded |
replicate
                                  | persistent-partition) #IMPLIED
  early-ack                       (false | true) #IMPLIED
enable-async-conflation         (false | true) #IMPLIED
  enable-gateway                  (false | true) #IMPLIED
  enable-subscription-conflation  (false | true) #IMPLIED
  hub-id                          CDATA #IMPLIED
  id                              CDATA #IMPLIED
  ignore-jta                      (false | true) #IMPLIED
  index-update-type               (asynchronous | synchronous) #IMPLIED
  initial-capacity                CDATA #IMPLIED
  is-lock-grantor                 (false | true) #IMPLIED
  load-factor                     CDATA #IMPLIED
  mirror-type                     (keys | keys-values | none) #IMPLIED
  multicast-enabled               (false | true) #IMPLIED
  persist-backup                  (false | true) #IMPLIED
  pool-name                       CDATA #IMPLIED
  disk-store-name                 CDATA #IMPLIED
  disk-synchronous                (false | true) #IMPLIED
  publisher                       (false | true) #IMPLIED
  refid                           CDATA #IMPLIED
  scope                           (distributed-ack | distributed-no-ack |
global
                                  | local) #IMPLIED
  statistics-enabled              (false | true) #IMPLIED
  cloning-enabled                 (false | true) #IMPLIED
>

<!-- ======================================================== -->
<!--
A "region-idle-time" element specifies a Region's idle time.
-->
<!ELEMENT region-idle-time (expiration-attributes)>

<!-- ======================================================== -->
<!--
A "region-time-to-live" element specifies a Region's time to live.
-->
<!ELEMENT region-time-to-live (expiration-attributes)>

<!-- ======================================================== -->
<!--
A "required-role" element specifies a role that is required for reliable
access to the region
-->

<!-- ======================================================== -->
<!ELEMENT required-role EMPTY>
<!ATTLIST required-role
  name  CDATA #REQUIRED
>

<!-- ======================================================== -->
<!--
```

```
A "server" element specifies the host and port that a cache server is
listening on
-->
<!ELEMENT server EMPTY>
<!ATTLIST server
  host  CDATA #REQUIRED
  port  CDATA #REQUIRED
>

<!-- ======================================================== -->
<!--
A "string" element specifies a String to be placed in a Region entry.
-->
<!ELEMENT string (#PCDATA)>

<!-- ======================================================== -->
<!--
A "subscription-attributes" element configures how a region subscribes
to remote distributed regions.
-->
<!ELEMENT subscription-attributes EMPTY>
<!ATTLIST subscription-attributes
  interest-policy  (all | cache-content) #IMPLIED
>

<!-- ======================================================== -->
<!--
A "synchronous-writes" element specifies that entry data should be
written to disk synchronously.
-->
<!ELEMENT synchronous-writes EMPTY>

<!-- ======================================================== -->
<!--
A "transaction-listener" describes a TransactionListener.
-->
<!ELEMENT transaction-listener (class-name, parameter*)>

<!-- ======================================================== -->
<!--
A "transaction-writer" describes a TransactionWriter.
-->

<!ELEMENT transaction-writer (class-name, parameter*)>



<!-- ======================================================== -->
<!--
A "value" element describes the value in a Region entry
-->
<!ELEMENT value (string | declarable)>

<!-- ======================================================== -->
<!--
A "value-constraint" element specifies the name of a class to which the
values of a region are constrained.
-->
<!ELEMENT value-constraint (#PCDATA)>
```

```
<!-- ========================================================= -->
<!--
A "vm-root-region" element is the same as "region".
This element exists for backwards compatiblity and should not be used
in new xml files.
-->

<!-- ========================================================= -->
<!ELEMENT vm-root-region (
    region-attributes,
    index*,
    entry*,
    region*
  )>
<!ATTLIST vm-root-region
  name  CDATA #REQUIRED
>
```

Appendix C

Query Language Grammar and Reserved Words

This appendix lists the reserved words and the query language grammar for GemFire Enterprise®.

Reserved Keywords

These words are reserved for the query language and may not be used as identifiers. The words with asterisk (*) after them are not currently used by GemFire Enterprise, but are reserved for future implementation.

```
abs*            dictionary      interval*       orelse*
all*            distinct        is_defined      query*
and             double          is_undefined    select
andthen*        element         last*           set
any*            enum*           like            short
array           except*         limit           some*
as              exists*         list*           string
asc*            false           listtoset*      struct*
avg*            first*          long            sum*
bag*            flatten*        map             time
boolean         float           max*            timestamp
by*             for*            min*            to_date
byte            from*           mod*            true
char            group*          nil             type
collection      having*         not             undefine*
count*          import          null            undefined
date            in              nvl             union*
declare*        int             octet           unique*
define*         intersect*      or              where
desc*                           order*
```

Language Grammar

Notation used in the grammar

n	A nonterminal symbol that has to appear at some place within the grammar on the left side of a rule. All nonterminal symbols have to be derived to be terminal symbols.
t	The terminal symbol **t** (shown in **courier new bold**)
x y	x followed by y
x | y	x or y
(x | y)	x or y
[x]	x or empty
{ x }	possibly empty sequence of x
comment	descriptive text

Grammar list

symbol ::= expression

query_program ::= [imports semicolon] query [semicolon]

imports ::= import { semicolon import }

import ::= **IMPORT** qualifiedName [AS identifier]

query ::= selectExpr | expr

selectExpr ::= **SELECT DISTINCT** projectionAttributes fromClause [whereClause]

projectionAttributes ::= ***** | projectionList

projectionList ::= projection { comma projection }

projection ::= field | expr [**AS** identifier]

field ::= identifier colon expr

fromClause ::= **FROM** iteratorDef { comma iteratorDef }

iteratorDef ::= expr [[**AS**] identifier] [**TYPE** identifier] | identifier **IN** expr [**TYPE** identifier]

whereClause ::= **WHERE** expr

expr ::= castExpr

castExpr ::= orExpr | left_paren identifier right_paren castExpr

orExpr ::= andExpr { **OR** andExpr }

andExpr ::= equalityExpr { **AND** equalityExpr }

equalityExpr ::= relationalExpr { (**=** | **<>** | **!=**) relationalExpr }

relationalExpr ::= inExpr { (**<** | **<=** | **>** | **>=**) inExpr }

inExpr ::= unaryExpr { **IN** unaryExpr }

unaryExpr ::= [**NOT**] unaryExpr

postfixExpr ::= primaryExpr { left_bracket expr right_bracket }
| primaryExpr { dot identifier [argList] }

argList ::= left_paren [valueList] right_paren

qualifiedName ::= identifier { dot identifier }

primaryExpr ::= functionExpr
| identifier [argList]
| undefinedExpr
| collectionConstruction
| queryParam
| literal
| (query)
| region_path

functionExpr ::= **ELEMENT** left_paren query right_paren
| **NVL** left_paren query comma query right_paren
| **TO_DATE** left_paren query right_paren

undefinedExpr ::= **IS_UNDEFINED** left_paren query right_paren
| **IS_DEFINED** left_paren query right_paren

collectionConstruction ::= **SET** left_paren [valueList] right_paren

valueList ::= expr { comma expr }

queryParam ::= **$** integerLiteral

region_path ::= forward_slash region_name { forward_slash region_name }

region_name ::= name_character { name_character }

identifier ::= letter { name_character }

literal ::= booleanLiteral
| integerLiteral
| longLiteral
| doubleLiteral
| floatLiteral
| charLiteral
| stringLiteral
| dateLiteral
| timeLiteral
| timestampLiteral
| **NULL**
| **UNDEFINED**

booleanLiteral ::= **TRUE** | **FALSE**

integerLiteral ::= [dash] digit { digit }

longLiteral ::= integerLiteral **L**

floatLiteral ::= [dash] digit { digit } dot digit { digit } [(**E** | **e**) [plus | dash] digit { digit }] **F**

doubleLiteral ::= [dash] digit { digit } dot digit { digit } [(**E** | **e**) [plus | dash] digit { digit }] [**D**]

charLiteral ::= **CHAR** single_quote character single_quote

stringLiteral ::= single_quote { character } single_quote

dateLiteral ::= **DATE** single_quote integerLiteral dash integerLiteral dash integerLiteral single_quote

timeLiteral ::= **TIME** single_quote integerLiteral colon
integerLiteral colon integerLiteral single_quote

timestampLiteral ::= **TIMESTAMP** single_quote
integerLiteral dash integerLiteral dash integerLiteral integerLiteral colon

integerLiteral colon
digit { digit } [dot digit { digit }] single_quote

letter ::= *any unicode letter*

character ::= *any unicode character except 0xFFFF*

name_character ::= letter | digit | underscore

digit ::= *any unicode digit*

The expressions in the following are all terminal characters.

dot ::= `.`

left_paren ::= `(`

right_paren ::= `)`

left_bracket ::= `[`

right_bracket ::= `]`

single_quote ::= `'`

underscore ::= `_`

forward_slash ::= `/`

comma ::= `,`

semicolon ::= `;`

colon ::= `:`

dash ::= `-`

plus ::= `+`

Language Notes

- Query language keywords such as `SELECT`, `NULL`, and `DATE` are case-insensitive. Identifiers such as attribute names, method names, and path expressions are case-sensitive.
- Comment lines begin with -- (double dash).
- Comment blocks begin with /* and end with */.
- String literals are delimited by single-quotes. Embedded single-quotes are doubled. Examples:

  ```
  'Hello' value = Hello
  'He said, ''Hello''' value = He said, 'Hello'
  ```

- Character literals begin with the `CHAR` keyword followed by the character in single quotation marks. The single-quotation mark character itself is represented as `CHAR''''` (with four single quotation marks).
- In the `TIMESTAMP` literal, there is a maximum of nine digits after the decimal point.

Glossary

ACK wait threshold A time-to-wait for message acknowledgement between system members.

administrative event See entry key.

API **A**pplication **P**rogramming **I**nterface. GemFire provides APIs to cached data for Java applications.

application program A program designed to perform a specific function directly for the user or, in some cases, for another application program. GemFire applications use the GemFire application programming interfaces (APIs) to modify cached data.

attribute Querying: A named member of a data object. The public fields and methods of an object may be accessed as attributes in the context of a query.

Region: See region attributes.

attribute path A sequence of attributes separated by a dot (.), applied to objects where the value of each attribute is used to apply the next attribute.

blocking A behavior associated with synchronization functions. Blocking behavior is exhibited as waiting for a signal to proceed, regardless of how long it takes. See also timeout.

cache In-memory GemFire data storage created by an application or cache server for data storage, distribution, and management. This is the point of access for Java applications for all caching features, and the only view of the cache that is available to the application. Cache creation creates a connection to the distributed system. See also local and remote.

cache-local Residing or occurring in the local cache.

cache.xml Common name for the XML file that declares the initial configuration of a cache. This file is used to customize the behavior of the GemFire cache server process and can be used by any Java application. Applications can also configure the cache through the GemFire Java APIs. You can give this file any name.

cache event See entry key.

cache listener User-implemented plug-in for receiving and handling region entry events. A region's cache listener is called after an entry in the local cache is modified. See also cache writer.

cache loader

User-implemented plug-in for loading data into a region. A region's cache loader is used to load data that is requested of the region but is not available in the distributed system. For a distributed region, the loader that is used can be in a different cache from the one where the data-request operation originated. See also netSearch and netLoad.cache misses, where a requested key is not present or has a null value in the local cache.

cache miss

The situation where an key's value is requested from a cache and the requested key is not present or has a null value. GemFire responds to cache misses in various ways, depending on the region and system configuration. For example, a client region goes to its servers to satisfy cache misses. A region with local scope uses its data loader to load the value from an outside data source, if a loader is installed on the region.

cache server

A long-lived, configurable GemFire distributed system member process.

cache transaction

A native GemFire transaction, managed by GemFire and not by JTA. This type of transaction operates only on data available from the GemFire cache in the local member. See also JTA and global transaction.

cache writer

User-implemented plug-in intended for synchronizing the cache with an outside data source. A region's cache writer is a synchronous listener to cache data events. The cache writer has the ability to abort a data modification. See also cache listener and netWrite.

client

A GemFire application that is configured as a standalone distributed system member, with regions configured as client regions. Client configuration uses the `<client-cache>` `cache.xml` element and the `ClientCache` API.

client region

A GemFire cache region that is configured to go to one or more GemFire servers, in a separate GemFire distributed system, for all data distribution activities. Among other things, client regions go to servers to satisfy cache misses, distribute data modifications, and to run single queries and continuous queries.

collection

Used in the context of a query for a group of distinct objects of homogeneous type, referred to as elements. Valid collections include the `java.util.Collection` as well as `Set`, `Map`, `List`, and arrays. The elements in a collection can be iterated over. Iteration over a `Map` traverses its entries as instances of `Map.Entry`. A region can also be treated as a collection of its values. See also QRegion.

commit

A transactional operation that merges a transaction's result into the cache. Changes are made in an "all or none" fashion. Other changes from outside the current transaction are kept separate from those being committed.

concurrency-level

Region attribute that specifies an estimate of the number of threads ever expected to concurrently modify values in the region. The actual concurrency may vary; this value is used to optimize the allocation of system resources.

conflation

Combining entries in a message queue for better performance. When an event is added to queue, if a similar event exists in the queue, there are two ways to conflate the events. One way is to remove the existing entry from wherever it resides in the queue, and add the new entry to the end of the queue. The other way is to replace the existing entry with the new entry, where it resides in the queue, and add nothing to the end of the queue. In GemFire, region entry update events, server events going to clients, and gateway events going to remote distributed systems can all be conflated.

connection

The connection used by an application to access a GemFire Enterprise system. A Java application connects to its GemFire distributed system when it creates its cache. The

application must connect to a distributed system to gain access to the GemFire functionalities. A client connects to a running GemFire server to distribute data and events between itself and the server tier. These client connections are managed by server connection pools within the client applications. Gateways connect to remote site GemFire gateway hubs to distribute data events between sites.

consumer

GemFire member process that receives data and/or events from other members. Peer consumers are often configured with replicated regions, so all changes in the distributed system arrive into the local cache. Client consumers can register subscriptions with their servers so that updates are automatically forwarded from the server tier. See producer.

coordinator

The member of the distributed system that sends out membership views. This is typically the locator in GemFire.

data region

A logical grouping of data within a cache. Regions usually contain data entries (see **entry**). Each region has a set of region attributes governing activities such as expiration, distribution, data loading, events, and capacity control. In addition, a region can have an application-defined user attribute.

data accessor

In the context of a region, a member configured to use a region, but not store any data for it in the member's local cache. Common use cases for data accessors are thin clients, and thin producer and consumer applications. Accessors can put data into the region and receive events for the region from remote members or servers, but they store no data in the application. See also data host.

data entry

See entry.

data host

In the context of a region, a member configured to store data for the region. Members that do not store data are commonly referred to as data accessors. This is used mostly for partitioned regions, where data is spread across the distributed system among the data hosts.

data region

See region.

data-policy

Region attribute used to determine what events the region receives from remote caches, whether data is stored in the local cache, and whether the data is persisted to disk. For disk persistence, writes are performed according to the cache disk-store configuration.

deadlock

A situation in which two or more processes are waiting indefinitely for events that will never occur.

destroy

Distributed**:** To remove a cached object across the distributed cache.

Local**:** To remove a cached object from the local cache only.

disk region

A persistent region.

disk-store

Cache element specifying location and write behavior for disk storage. Used for persistence and overflow of data. The cache can have multiple disk stores, which are specified by name for region attributes, client subscription queues (for servers), and wan gateway queues.

distributed cache

A collection of caches spread across multiple machines and multiple locations that functions as a single cache for the individual applications.

distributed system

One or more GemFire system members that have been configured to communicate with each other, forming a single, logical system.

distributed-ack scope

Data distribution setting that causes synchronous distribution operations, which wait for acknowledgement from other caches before continuing. Operations from multiple caches can arrive out of order. This scope is slower but more reliable than `distributed-no-ack`.

distributed-no-ack-scope

Data distribution setting that causes asynchronous distribution operations, which return without waiting for a response from other caches. This scope produces the best performance, but is prone to race conditions.

entry

A data object in a region consisting of a key and a value. The value is either null (invalid) or a Java object. A region entry knows what region it is in. An entry can have an application-defined user attribute. See also region data, entry key, and entry value.

entry key

The unique identifier for an entry in a region.

entry value

The data contained in an entry.

event

An action recognized by the GemFire system members, which can respond by executing callback methods. The GemFire API produces two types of events: cache events for detail-level management of applications with data caches and administrative events for higher-level management of the distributed system and its components. An operation can produce administrative events, cache events, or both.

eviction-attributes

Region attribute that causes the cache to limit the size of the region by removing old entries to make space for new ones.

expiration

A cached object expires when its time-to-live or idle timeout counters are exhausted. A region has one set of expiration attributes for itself and one set for all of its entries.

expiration action

The action to be taken when a cached object expires. The expiration action specifies whether the object is to be invalidated or destroyed and whether the action is to be performed only in the local cache or throughout the distributed system. A destroyed object is completely removed from the cache. A region is invalidated by invalidating all entries contained in the region. An entry is invalidated by having its value marked as invalid. `Region.getEntry.getValue` returns `null` for an invalid entry.

In GemFire, expiration attributes are set at the region level for the region and at the entry level for entries. See also idle timeout and time-to-live.

factory method

An interface for creating an object which at creation time can let its subclasses decide which class to instantiate. The factory method helps instantiate the appropriate subclass by creating the correct object from a group of related classes.

forced disconnect

Forcible removal of a member from membership without the member's consent.

gateway

Configured inside a gateway-hub, a gateway defines a single remote distributed system site in a multi-site installation and manages communication with the remote site. The gateway might have multiple endpoints assigned to a single remote site. The gateway also specifies queue management parameters for its endpoints.

gateway-hub

GemFire caching entity that represents its distributed system in a multi-site installation. The hub manages the gateways that send messages to other distributed system sites and listens on an incoming port for connections from remote gateways.

gemfire
Command-line utility that allows you to perform various GemFire management tasks from the command line, including locator start and stop, online disk store management, log management, and license information listing.

gemfire.properties
Common name for the file used for distributed system configuration, including system member connection and communication behavior, logging and statistics files and settings, and security settings. Applications can also configure the distributed system through the GemFire Java APIs. You can give this file any name.

global scope
Data distribution setting that provides locking across the distributed system for load, create, put, invalidate, and destroy operations on the region and its entries. This scope is the slowest, but it guarantees consistency across the distributed system.

global transaction
A JTA-controlled transaction in which multiple resources, such as the GemFire cache and a JDBC database connection, participate. JTA coordinates the completion of the transaction with each of the transaction's resources. See also JTA and cache transaction.

HTTP
World Wide Web's **H**yper**t**ext **T**ransfer **P**rotocol. A standard protocol used to request and transmit information over the Internet or other computer network.

idle timeout
The amount of time a region or region entry may remain in the cache without being accessed before being expired. Access to an entry includes any get operation and any operation that resets the entry's time-to-live counter. Region access includes any operation that resets an entry idle timeout and any operation that resets the region's time-to-live.

Idle timeout attributes are set at the region level for the region and at the entry level for entries. See also time-to-live and expiration action.

initial capacity
Region attribute. The initial capacity of the map used for storing region entries.

invalid
The state of an object when the cache holding it does not have the current value of the object.

invalidate
Distributed**:** To mark an object as being invalid across the distributed cache.

Local**:** To mark an object as being invalid in the local cache only.

JDBC
Java **D**ata**B**ase **C**onnectivity. A programming interface that lets Java applications access a database via the SQL language.

JMX
Java **M**anagement e**X**tensions. A set of specifications for dynamic application and network management in the J2EE development and application environment.

JNDI
Java **N**aming and **D**irectory **I**nterface. An interface to naming and directory services for Java applications. Applications can use JNDI to locate data sources, such as databases to use in global transactions. GemFire allows its JNDI to be configured in a `cache.xml` configuration file.

JTA
Java **T**ransaction **A**PI. The local Java interfaces between a transaction manager (JTS) and the parties involved in a global transaction. GemFire can be a member of a JTA global transaction. See also global transaction.

key constraint
Enforcing a specific entry key type. The `key-constraint` region attribute, when set, constrains the entries in the region to keys of the specified object type.

listener
An event handler. The listener registers its interest in one or more events, such as region entry updates, and is notified when the events occur.

load factor

Region attribute. The load factor of the map used for storing entries.

local

Local cache: The part of the distributed cache that is resident in the current member's memory. This term is used to differentiate the cache where a specific operation is being performed from other caches in the same distributed system or in another distributed system. See also remote.

Region with local scope: A region whose scope is set to local. This type of region does not distribute anything with other member's in the distributed system.

Region shortcuts: In the `RegionShortcut` and settings, `LOCAL` means the scope is set to local. All client regions have local scope. In the `ClientRegionShortcut` settings, `LOCAL` means the region does not connect to the client's servers.

local scope

Data distribution setting that keeps data private and visible only to threads running within the local member. A region with local scope is completely contained in the local cache. Client regions are automatically given local scope.

locator

GemFire process that tracks system members and provides current membership information to joining members so they can establish communication. For server systems, the locator also tracks servers and server load and, when a client requests a server connection, the locator sends the client to one of the least loaded servers. See also gemfire.

LRU

Least **R**ecently **U**sed. Used to refer to region entry or entries most eligible for eviction due to lack of interest by client applications. GemFire offers eviction controllers that use the LRU status of a region's entries to determine which to evict to free up space. Possible eviction actions are local destroy and overflow. See also resource manager.

member

A process that has established a connection to a distributed system. This can be a Java or Native Client application with a distributed system connection and a cache or a GemFire process such as a locator or cacheserver. The minimal GemFire process configuration is a member that is connected to a distributed system.

message queue

A first-in, first-out data structure in a GemFire Enterprise system member that stores messages for distribution in the same order that the original operations happened in the local member. Each thread has its own queue. Depending on the kind of queue, the messages could be going between two members of a distributed system, a client and server, or two members in different distributed systems. See also conflation.

mirroring

See replicate.

multicast

A form of UDP communications where a datagram is sent to multiple processes in one network operation.

named region attributes

Region attributes that are stored in the member memory and can be retrieved through their region attributes `refid` setting. GemFire provides standard predefined named region attributes, that are stored using region shortcut `refid`s. You can use any stored attributes that you wish, setting an `id` when you create them and using the id setting in the `refid` you want to use to retrieve them.

netLoad

The method used by GemFire Enterprise to load an entry value into a distributed region. The `netLoad` operation invokes all remote cache loaders defined for the region until either the entry value is successfully loaded or all loaders have been tried.

netSearch The method used by GemFire Enterprise to search remote caches for a data entry that is not found in the member's local cache region. This method operates only on distributed regions.

netWrite The method used by GemFire Enterprise to invoke a cache writer for region and region entry events. This method operates only on distributed regions. For each event, if any cache writer is defined for the region, the `netWrite` operation invokes exactly one of them.

network partitioning A situation that arises from a communications partition that causes processes to become unaware of one another.

OQL **O**bject **Q**uery **L**anguage, SQL-92 extended for querying object data. GemFire supports a subset of OQL.

overflow Eviction option for eviction controllers. This causes the values of LRU entries to be moved to disk when the region reaches capacity. Writes are performed according to the cache disk-store configuration.

oplog / operation log The files in a disk-store used for the cache operations.

partitioned region A region that manages large volumes of data by partitioning it into manageable chunks and distributing it across multiple machines. Defining partition attributes or setting the region attribute data-policy to `partition` makes the region a partitioned region.

peer A GemFire member application that is not configured as a client. Peer configuration uses the `<cache>` `cache.xml` element and the `Cache` API. Peers can also be configured as servers to client applications and as gateway-hubs to remote distributed systems.

persistent region A region with the attribute data-policy set to persistent-replicate.

persistent-partition A region attribute setting identifying a region as a partitioned region whose data is persisted to disk. With persistence, all region entry keys and values are stored in an operation log on disk as well as being stored in memory. Also referred to as disk region. Writes are performed according to the cache disk-store configuration.

persistent-replicate A region attribute setting identifying a region as a replicate whose data is persisted to disk. With persistence, all region entry keys and values are stored in an operation log on disk as well as being stored in memory. Also referred to as disk region. Writes are performed according to the cache disk-store configuration.

producer A GemFire member process that puts data into the cache for consumption by other members. Producers may be configured with empty regions, where the data they put into the cache is not stored locally, but causes cache update events to be sent to other members. This is a common configuration in peer members and for client processes. See consumer.

pull model Data distribution model where each process receives update only for the data in which the process has explicitly expressed interest. In a GemFire peer member, this is accomplished using a distributed, non-replicated region and creating the data entries that are of interest in the local region. When updates happen for the region in remote caches, the only updates that are forwarded to the local cache are those for entries that are already defined in the local cache. In a GemFire client, you get pull behavior by specifically subscribing to the entries of interest. See push model.

pure Java mode Running GemFire Enterprise without the use of the GemFire native library. GemFire Enterprise can run in this mode with limited capabilities.

push model Data distribution model where each process receives updates for everything in the data set. In a GemFire peer member, this is accomplished using a replicated region. All data modifications, creations, and deletes in remote caches are pushed to the replicated region. In a GemFire client, you get push behavior by registering interest in all keys in the region. See pull model.

QRegion The region object representation in a GemFire query. A `QRegion` extends `com.gemstone.gemfire.cache.Region` and `java.util.Collection` so that the single region specification can provide access both to region attributes and to region data collections such as keys and entry values. See also collection.

query string A fully-formed SQL statement that can be passed to a query engine and executed against a data set. A query string may or may not contain a `SELECT` statement.

race condition Anomalous behavior caused by the unexpected dependence on the relative timing of events. Race conditions often result from incorrect assumptions about possible ordering of events.

range-index An XPath index optimized for range-queries with the added index maintenance expense of sorting the set of values. A range index allows faster retrieval of the set of nodes with values in a certain range. See also structure-index and value-index.

region A logical grouping of data within a cache. Regions usually contain data entries (see **entry**). Each region has a set of region attributes governing activities such as expiration, distribution, data loading, events, and capacity control. In addition, a region can have an application-defined user attribute.

region attributes The class of attributes governing the creation, distribution, and management of a region and its entries.

region data All of the entries directly contained in the region.

region entry See entry.

region shortcut Enums `RegionShortcut` and `ClientRegionShortcut` defining the main region types in GemFire for peers/servers and clients, respectively. Region shortcuts are predefined named region attributes.

remote Resident or running in a cache other than the current member's cache, but connected to the current member's cache through `GemFire`. For example, if a member does not have a data entry in the region in its local cache, it can do a `netSearch` in an attempt to retrieve the entry from the region in a remote cache within the same distributed system. Or, if the member is a client, it can send a request to a server in an attempt to retrieve the entry from the region in a remote server cache in the server's distributed system. In multi-site installations, a gateway sends events from the local cache to remote caches in other distributed systems. See also local.

replicated region A region with data-policy set to replicate or persistent-replicate.

replicate Region data-policy specification indicating to copy all distributed region data into the local cache at region creation time and to keep the local cache consistent with the distributed region data.

resource manager GemFire process that works with your VM's tenured garbage collection (GC) to control heap use and protect your VM from hangs and crashes due to memory overload. The manager prevents the cache from consuming too much memory by evicting old data and,

if the collector is unable to keep up, by refusing additions to the cache until the collector has freed an adequate amount of memory. Eviction is done for regions configured for LRU eviction based on heap percentage. See also LRU and eviction-attributes.

role

The purpose a member fills in a distributed system, or how a member relates to other members. These optional membership roles specify the circumstances under which a member continues operation after incidents such as network failures. Members can fill one or more roles. Any number of members can be configured to satisfy the same role, and a member can be configured to play any number of roles.

rollback

A transactional operation that excludes a transaction's changes from the cache, leaving the cache undisturbed.

scope

Region attribute: In non-partitioned regions, a distribution property for data identifying whether it is distributed and, if so, whether distribution acknowledgements are required and whether distributed synchronization is required. A distributed region's cache loader and cache writer (defined in the local cache) can be invoked for operations originating in remote caches. A region that is not distributed has a local scope. See also replicate and data-policy.

Querying: The data context for the part of the query currently under evaluation. The expressions in a `SELECT` statement's `FROM` clause can add to the data that is in scope in the query.

SELECT statement

A statement of the form `SELECT` *`projection_list`* `FROM` *`expressions`* `WHERE` *`expressions`* that can be passed to the query engine, parsed, and executed against data in the local cache.

serialization

The process of converting an object or object graph to a stream of bytes.

server

A GemFire member application that is configured as a peer in its own system and as a server to connecting GemFire client applications.

server group

An optional logical grouping of servers in a server distributed system. There is always the default server group made up of all available server in the server distributed system. Clients can specify the server group in their server pool configuration. Then the pool only connects to those servers. If no group is specified, the default is used.

server connection pool

The cache entity that manages client connections to servers.

socket

The application interface for TCP/IP communications. UDP provides unicast and multicast datagram sockets, while TCP provides server and connection sockets. TCP server sockets are used by server processes to create connection sockets between the server and a client.

SQL

Structured **Q**uery **L**anguage.

SSL

Secure **S**ocket **L**ayer. A protocol for secure communication between Java VMs.

statistics enabled

Region attribute. Specifies whether to collect statistics for the region.

struct

A data type that has a fixed number of elements, each of which has a field name and can contain an object value.

structure-index

An XPath index that is basically a pre-computed query. Any legal XPath expression can be used. The index maintains lists of all nodes that match the expression used to create it. If a

query is performed that has the same expression as the index then the result is available without XPath evaluation. See also range-index and value-index.

system member See member.

TCP The **T**ransmission **C**ontrol **P**rotocol is a part of the internet protocol (IP) suite that provides unicast communications with guaranteed delivery. The TCP protocol is connection-based, meaning that a TCP socket can only be used to send messages between one pair of processes at a time. Compare to UDP.

timeout A behavior associated with synchronization functions. Timeout behavior is exhibited as refusal to wait longer than a specified time for a signal to proceed. See also blocking.

time-to-live The amount of time a region or region entry may remain in the cache without being modified before being expired. Entry modification includes creation, update, and removal. Region modification includes creation, update, or removal of the region or of any of its entries.

Time-to-live attributes are set at the region level for the region and at the entry level for entries. See also idle timeout and expiration action.

transaction See cache transaction and global transaction.

transaction listener User-implemented plug-in for receiving and handling transaction events. A transaction listener is called after a transaction commits. See also transaction writer.

transaction writer User-implemented plug-in intended for synchronizing the cache with an outside data source. A transaction writer is a synchronous listener to cache transactions. The transaction writer has the ability to veto a transaction. See also transaction listener.

transactional view The result of a history of transactional operations for a given open transaction.

transport layer The network used to connect the GemFire system members in a GemFire Enterprise system.

TTL See time-to-live.

UDP The **U**ser **D**atagram **P**rotocol is a part of the internet protocol (IP) suite that provides simple, unreliable transmission of datagram messages from one process to another. Reliability must be implemented by applications using UDP. The UDP protocol is connectionless, meaning that the same UDP socket can be used to send or receive messages to or from more than one process. Compare to TCP.

unicast A message sent from one process to another process (point-to-point communications). Both UDP and TCP provide unicast messaging.

URI **U**niform **R**esource **I**dentifier. A unique identifier for abstract or physical resources on the World Wide Web.

user attribute An optional object associated with a region or a data entry where an application can store data about the region or entry. The data is accessed by the application only. GemFire Enterprise does not use these attributes. Compare to region attributes, which are used by GemFire.

value constraint Enforcing a specific entry value type. The `value-constraint` region attribute, when set, constrains the entries in the region to values of the specified object type. Value constraints

can be used to provide object typing for region querying and indexing. The value-constraint is only checked in the cache that does the entry put or create operation. When the entry is distributed to other caches, the value constraint is not checked.

value-index An XPath index that operates much as a structure-index does, but that separates the nodes that match the XPath expression into sets mapped by each node's value. This allows further filtering of the nodes to be evaluated in a query by going directly to those with a specific value. See also structure-index and range-index.

view A collection of member identifiers that defines the membership group in JGroups.

VM **V**irtual **M**achine, also referred to as a Java VM.

XML E**X**tensible **M**arkup **L**anguage. An open standard for describing data, XML is a markup language similar to HTML. Both are designed to describe and transform data, but where HTML uses predefined tags, XML allows tags to be defined inside the XML document itself. Thus, virtually any data item can be identified. The XML programmer creates and implements data-appropriate tags whose syntax is defined in a DTD file or an XML schema definition.

XML schema The definition of the structure, content, and semantics used in an XML document. The XML schema is a superset of DTD. Unlike DTD, XML schemas are written in XML syntax, which, although more verbose than DTD, are more descriptive and can have stronger typing. Files containing XML schema definitions generally have the `xsd` extension.

XPath A language that describes a way to locate and process items in Extensible Markup Language (XML) documents by using an addressing syntax based on a path through the document's logical structure or hierarchy.

Index

A

ack timeout for `distributed-ack` region 165
`ack-wait-threshold` 165
administration API 479–495
    interfaces and classes 480
    managing cache server 480–491
API 387
    administration 479–480
    caching 91
    distributed system 454
    health monitoring 483
    logging 53
    membership 454
    overview 53
    primary packages 53
    queries and indexes 387
    region attributes 91
    transactions
        JTA 435
    See also `com.gemstone.gemfire`
application
    caching disabled 318
application plug-ins
    cache loader 199
    cache operations, effects on 136, 140, 143
    callbacks with transactions 253
    distributed region 162, 193, 418
    invocation of 193, 418
    local region 193
    transactions 418
    See also event handler
application server as GemFire client 82
architectural overview of distributed system 36

B

backlog for server socket 265
`batch-size` gateway queue attribute 273
`batch-time-interval` gateway queue attribute 273
`bind-address` server configuration parameter 207
bridge conflation. *See* messaging
`BridgeMembershipListener` 296
    callback methods 309
bucket units, storage in partitioned region 46

C

cache
    attributes
        `copy-on-read` 74, 140
        `is-server` 74
        `lock-lease` 75, 471
        `lock-timeout` 74, 471
        `message-sync-interval` 75
        region and entry lock timeouts 166, 471
        `search-timeout` 75, 471
        timeout for `netSearch` 75, 471
    caching disabled 318
    closing
        durable client 248
    configuration
        attributes 74
    configuration file. *See* `cache.xml`
    data type definition 505

elements
cache-server 75
cache-transaction-manager 75
disk-store 75
dynamic-region-factory 75
function-service 75
gateway-hub 75
jndi-bindings 75
pool 75
region 76
region-attributes 76
resource-manager 76
serialization-registration 76
inactive, membership role loss while 460
initialization file 499
scenarios 41
clustered servers 43
request/response queue management 44
session state management 44
server
configuring the cache 74
statistics 469
writer
configuring 103
cache event
cache listener 305–308
cache writer 312
handling in distributed system 313, 317
cache listener 39
adding multiple 307
BridgeMembershipListener 296
callbacks
client/server 253
class hierarchy 296
client/server
durable messaging caveats 253
CqListener 296–297
creating 300
events
partitioned region 314
receiving 296–297, 323
multiple in a region 306
order of execution 306
partitioned region 102
RegionMembershipListener 296, 305
RegionRoleListener 296, 306
removing 306
thread-safe use 316
TransactionListener 296
transactions 419
See also application plug-ins
cache loader 136, 199
configuring 102
distributed region 197
invocation in distributed regions 136
partitioned regions 102
region 39
region scope 197
region, plug-in 39
transactions 418–419, 435
See also application plug-ins
cache server 201, 227
adding 42
in client/server 45, 203
in-memory storage 40
managing through administration API 480–491
cache transaction
collections 429
indexing 428
operations 426
querying 428
repeatable-read isolation 416
cache writer 39, 312, 435
callbacks
client/server 253
class hierarchy 297
creating 300
entry
invalidation/destruction, effects on 143
management, effects on 140
retrieval, effects on 137
events 312–313
receiving 297
region scope 312
transactions 419
See also application plug-ins
cache.xml 55, 70
application plug-ins in
cache.xml declaration 300
cache attributes in 471
client example 226
DTD 500, 505
file specification 55
region creation example 94
server example 225
CacheCallback 296–297
CacheClientNotifierStatistics 466
CacheClientProxyStatistics 466
CacheClientUpdaterStatistics 466
CacheHealthConfig, health monitoring API 483
cache-loader region attribute 102
CachePerfStats 430, 466
cacheRunner example program 389
cache-server cache element 75
CacheServerStatistics 466
cache-transaction-manager cache element 75
cache-writer region attribute 103
caching API

functional overview 91–96
programming example 389
querying overview 387
callback 294
BridgeMembershipListener 309
CacheListener 305
methods, writing
durable clients, for 253
RegionMembershipListener 305
TransactionListener 308
CLASSPATH environment variable 59
client/server 57
custom classes 57, 119
pure Java mode 52
client configuration 209
client configuration parameters
free-connection-timeout 210
idle-timeout 210
load-conditioning-interval 210
locator 210
max-connections 210
min-connections 210
multiuser-authentication 210
name 211
ping-interval 211
pr-single-hop-enabled 211
read-timeout 211
retry-attempts 211
server 211
server-group 211
socket-buffer-size 211
statistic-interval 211
subscription-ack-interval 212
subscription-enabled 212
subscription-message-tracking-timeout 212
subscription-redundancy-level 212
thread-local-connections 212
client/server 201, 227
architecture 201, 203, 227
BridgeMembershipListener 206, 296, 309, 324
cache callbacks 253
CacheServer 206
caching 45
cacheReadyForEvents message 248
caching classes, customized 53
client connection pool 261
client destruction of server entry 218
client interest list registration 220
bulk load 221
client region configuration
register interest 220
client requests 228
client subscription 229
client wait for server response 266
ClientCache 206
client-to-client data flow 223
client-to-server updates 217
communication sockets
size 265
timeout 266
configuration
server parameters list 207–209
connection management 228–231, 234
slow 234–235
continuous query update notification
durable 248
CqListener 296–297, 310
durable client 246
cache listener callbacks 305
closing the cache 248
entry destruction 217, 219–220, 234
entry update notification 398
durable 247
exceptions, suppressing 264
failover client to new server 254
grouping servers 205
locator configuration 204–205
multi-site installation 268
multi-user client 82
performance 259–260
Pool 206
pool connection 228
preserving unified data set on servers 220
queue 229
queue conflation. *See* messaging
region destruction 217, 219–220, 234
RegionAttributes 206
registering interest in entry keys 220
registering interest in entry values 397
running SQL-style queries 40
server CLASSPATH 57
server garbage collection 264
server overloading 266
server peer connections 260
sockets
size 265
timeout 266
static server list 215
subscription 238
subscription connection 229
subscription management 239
high availability 241–245
subscription queue configuration 257
tuning 264
UniversalMembershipListener 325
ClientCache client configuration 209

client-cache client configuration 209
ClientRegionShortcut 97–99
ClientStatistics 466
collection, transactions handling of 429
com.gemstone.gemfire 53, 464
    DataSerializable 53, 57, 470
    LogWriter 53
    StatisticDescriptor 464
    StatisticFactory 464
    Statistics 53, 464
    StatisticsTypeFactory 464
    StatisticType 464
    SystemFailure 53
com.gemstone.gemfire.admin 53, 480
    AdminDistributedSystem 480
    AdminDistributedSystemFactory 480
    AlertListener 481
    CacheVM 480
    CacheVm 480
    DistributedSystemConfig 480
    DistributedSystemConfig.ConfigListener 480
    DistributionLocator 309, 480
    ManagedEntity 480
    SystemMember 481
    SystemMemberCache 481
    SystemMemberCacheEvent 323, 481
    SystemMemberCacheListener 481
    SystemMemberCacheServer 481
    SystemMemberListener 481
    SystemMemberRegion 481
    SystemMemberRegionEvent 323
    SystemMembershipEvent 323, 481
com.gemstone.gemfire.cache 53, 124, 130, 410
    BridgeMembershipEvent 299
    BridgeMembershipListener 309, 324
    Cache 53, 55
        attributes 471
    CacheEvent 130, 297
    CacheListener 419
    CacheLoader 136
    CacheStatistics 117, 130, 177
    CacheTransactionManager 410
    CacheWriter 312
    Declarable 300
    EntryEvent 130, 298
    EvictionController 429
    LoaderHelper 136
    Region 53, 130, 135, 471
        entry management methods 135, 137, 140
        methods for partitioned regions 130
        query shortcuts 387
        statistics retrieval 117
    Region.Entry 53, 129–130, 138
        statistics retrieval 117
    Region.existsValue 388
    Region.query 387
    Region.selectValue 388
    RegionAttributes 206
    RegionEvent 298
    RegionMembershipListener 305
    RegionRoleListener 306
    RoleEvent 298
    Scope 307
    SystemMemberRegionEvent 481
    TransactionEvent 298, 410
    TransactionId 410
    TransactionListener 308, 410
com.gemstone.gemfire.cache.client 53
    ClientCache 206
    Pool 206
com.gemstone.gemfire.cache.query 53, 387, 400
    CqAttributes 400
    CqAttributesFactory 400
    CqAttributesMutator 400
    CqEvent 299, 400, 405
    CqListener 296–297, 310, 400, 404
    CqQuery 400
        example 406
    CqServiceStatistics 400
    CqStatistics 400
    Index 387
    Query 387
    QueryService 387, 390–393, 400
    QueryService.createIndex 390
    QueryService.getIndex 392
    QueryService.getIndexes 391
    QueryService.newQuery 393
    QueryService.removeIndex 392
    SelectResults 387, 394
    SelectResults.getCollectionType 394
    Struct 387
com.gemstone.gemfire.cache.security 54
com.gemstone.gemfire.cache.server 53
    ServerLoadProbe
        client/server
            ServerLoadProbe 206
com.gemstone.gemfire.cache.util 53, 270
    BridgeMembershipListener 206
    BridgeMembershipListenerAdapter 309
    CacheListenerAdapter 305
    CacheServer 206
    CacheWriterAdapter 312
    CqListenerAdapter 310
    Gateway 270
    Gateway.Endpoint 270

GatewayEventListener 311
GatewayHub 270
GatewayQueueAttributes 270
RegionMembershipListenerAdapter 305
RegionRoleListenerAdapter 306
TransactionListenerAdapter 308
com.gemstone.gemfire.distributed 54
DistributedLockService 166
DistributedMember 454
commit conflict. See cache transaction repeatable-read
communication
socket configuration 471
concurrency level, configuring 103
concurrency-level region attribute 103
configuration
custom attributes 120
files 55
distributed system connection 70
java.lang.System properties, using 55
user attributes 120
configuration files 471
connections
controlling use
client/server 259–262
conserve-sockets 260, 471
containsKeyOnServer
unexpected results 207
continuous query 397–408
accessing 407
API 400
client/server subscription 229
close method 408
CLOSED 401
closing 408
destruction 408
durable 248
event handler 296, 299, 404
avoiding infinite loop 404
event object 405
executing 408
high availability 402
life cycle 401
management 402
modifying 407
newCq exception 403
query operation 405
query requirements 403
resource use 401
retrieving 407
RUNNING 401
state 401
statistics 407
STOPPED 401
stopping 408
writing the query 403
copy-on-read 140
copy-on-read cache attribute 74
CQStatistics 466
custom objects 57
custom-load-probe server configuration parameter 207

D

data
consistency 117
distribution 117
transactions 422
management 169
operations 129
policy, configuring 104
recovery
disk overflow, from 175
redundancy 152
replication 162
serialization 57, 470
transmission buffer 472
updating 129–180
client/server 217, 219–220, 234
queue conflation 256, 474
data colocation
partitioned region 157
data type definition (DTD), GemFire Enterprise cache 505
data-policy region attribute 104
interest-policy, interaction with 118
replicate 162–163
DataSerializable.
See com.gemstone.gemfire
destroy, expiration, action 179
destruction 95, 143
distributed 95
entry 143
local 95, 143
disk
overflow. *See* overflow of region entry data 40
storage, region name requirements 92
disk space
statistics archive file, managing 467
DiskDirStatistics 466
DiskRegionStatistics 466
disk-store cache element 75
disk-store-name
client queue selection 257
disk-store-name gateway queue attribute 273
DistribributedSystemHealthConfig, health monitoring API 483
distributed region 39, 116–117, 168, 316

pull distribution model 163
push distribution model 163
transactions 422
distributed system
architecture 36
configuration 69–70
event handling 313, 317
events 294
member 36
joining, notification 306
leavng, notification 306
performance controls 470
monitoring and tuning 463–477
multiple
client/server. *See* client/server
WAN. *See* multi-site
slow, using gateways 274
statistics 464, 466
`CacheClientNotifierStatistics` 466
`CacheClientProxyStatistics` 466
`CacheClientUpdaterStatistics` 466
`CachePerfStats` 466
`CacheServerStatistics` 466
`ClientStatistics` 466
`CQStatistics` 466
`DiskDirStatistics` 466
`DiskRegionStatistics` 466
`DistributionStats` 466
`DLockStats` 467
`GatewayHubStatistics` 467
`GatewayStatistics` 467
`LocatorStatistics` 467
`LRUStatistics` 467
`PoolStatistics` 467
`ProcessStats` 467
`StatSampler` 467
`SystemStats` 467
`VMGCStats` 467
`VMMemory*Stats` 467
`VMMemoryPoolStats` 467
`VMStats` 467
transactions in 420
`distributed-ack` scope. *See* `scope` region attribute
`DistributedMember`.
See `com.gemstone.gemfire.distributed`
`distributed-no-ack` scope. *See* `scope` region attribute
distribution of custom objects 57
`DistributionStats` 466
`DLockStats` 467
dot and forward slash operators, query string 369
durable messaging
client timeout, tuning 247
`durable-client-ID` system property 246
`durable-client-timeout` system property 246
`dynamic-region-factory` cache element 75

E

empty region
disabled features in 321
partitioned region, `local-max-memory` 149
`enable-async-conflation` region attribute, slow consumer 473
`enable-conflation` gateway queue attribute 273
`enable-gateway` region attribute 107
`enable-persistence` gateway queue attribute 273
`enable-subscription-conflation` region attribute 107
encrypting password 64
entry 129–144
access 143
attributes, user-defined 120
avoiding modifying in-place 140
creation 131
data redundancy 152
destruction 143
client/server 217–220, 234
event 298
expiration 176
configuring 108
invalidation 143
key 129
modification 140
distribution of 161
persistence to disk 171–172
retrieval 135, 137–138
copy-on-read 140
local cache 96, 138
`netLoad` 136
`netSearch` 136
`netWrite`, interaction with 137
region scope, effects on 136–137
statistics 469
update notification
durable 247
value 129
`entry-idle-time` region attribute 108
`entry-time-to-live` region attribute 108
environment
variable
`GF_JAVA` 50
`equals` method 350
event
administrative 295, 323
client/server 325
bridge membership 309, 324
cache 295–322

cache listener 305–308
cache writer 312
callback 294
client querying server 310, 398, 404
client/server membership 309, 324
continuous query 399
cycle 294
distributed operation 294
object 294
originator 323
remote
interest-policy 319
role, membership 298, 306
transaction 298
transaction ID in 298

event handler
adapter classes 300
after-event 300–301
before-event 300
cache listener 301
continuous query 404
creating 300
initializing 300

eviction controller 40, 172
configuring 108
entry management, effects on 140

eviction-attributes region attribute 108, 172

examples
programming 67

exception
client/server, suppressing 264
gateway 276
handling 65

expiration
action 179
and netSearch 180
region attributes 176
configuring 108, 116
transactions 429
interaction with netSearch 136

F

failover
gateway hub 285

fieldname:expression syntax in queries 364

file descriptor 260

file handle
see file descriptor

for clients 45, 203

free-connection-timeout client configuration parameter 210

funciton execution
high availability 439

function execution 450
data dependent 442–444
highly available 445
data independent 440–441
function registration 449
FunctionService.on methods 450
high availability 447, 450, 452
implementing Function 447
implementing FunctionAdapter 447
in partitioned region 451
on data set 439
on member 439
results 452

function-service cache element 75

G

garbage collection
client/server 264

gateway
configuration 270
endpoint 270
hub 107, 270
distribution, troubleshooting 477
queue
batches 275
configuration 270
conflation 277
processing 274–275
processing failures 276, 476
slow distributed system 274
temporary queue 274

gateway configuration parameters
id 273
socket-buffer-size 273
socket-read-timeout 273

gateway-endpoint configuration parameters
host 273
id 273
port 273

gateway-hub cache element 75

gateway-hub configuration parameters
id 272
port 272
socket-buffer-size 272
startup-policy 272

GatewayHubStatistics 467

GatewayStatistics 467

gemfire command 55

GemFire Region
Map interface 130

GemFire services 39

gemfire.asynchronous-gateway-distribution-enabled 274

gemfire.properties file 55
gemfire.socket-buffer-size 472
GemFireHealth, health monitoring API 483
GemFireHealth.Health, health monitoring API 483
GemFireHealthConfig, health monitoring API 483
gemfireLicense.zip file 55
GF_JAVA environment variable 50
global scope. *See* scope region attribute
global transaction. *See* JTA transaction
global-max-buckets partition attribute 115
group server configuration parameter 208
growth, managing 201, 227

H

HA queue for client/server
  configuration 244
hashCode method 350
hashing
  distributed regions
    initial capacity 103, 110
    load factory 112
health monitoring
  API 483
    CacheHealthConfig 483
    DistributedSystemHealthConfig 483
    GemFireHealth 483
    GemFireHealth.Health 483
    GemFireHealthConfig 483
    MemberHealthConfig 483
hierarchical cache. *See* client/server caching
high availability
  configuring client/server 402
  configuring partitioned region 153
  continuous query 402
  partitioned regions 152
  performance impact 152, 402
  region 171
  resource planning 153
hit count statistic 469
host gateway endpoint configuration parameter 273
hostname-for-clients server configuration parameter 208

I

id
  gateway
    configuration parameter 273
    endpoint configuration parameter 273
    hub configuration parameter 272
ID of region attributes 504
idle timeout 179, 469
idle-timeout client configuration parameter 210
ignore-jta region attribute 110
importing classes for querying 362
indexing 341, 375–395
  creation of 376
  functional 375
  guidelines 378–383
  maintenance 377
  overflow region 351
  partitioned region 384
  primary key 375
  transactions 428
index-update-type region attribute 110
initial-capacity region attribute 110
initialization file, cache.xml 499
in-memory storage, cache server 40
interest registration for entry updates
  client/server
    durable 247
interest-policy region attribute
  data-policy, interaction with 118
invalidate, expiration action 179
invalidation 95
  distributed 95, 143
  entry 143
  local 95, 143
is-lock-grantor 166
is-lock-grantor region attribute 111
is-server cache attribute 74

J

Java application
  logging and statistics 53
  running 50
    in pure Java mode 52
  scenarios
    multiple members 41
    single member 41
Java Transaction API (JTA), compatibility with transaction manager 40
java.lang.System 55
Javadocs 25
JDBC
  listener, connecting from 301
  loader, connecting from 199
  query, formats used by 372
  versions supported 59
JDK 50
JNDI data sources 59
jndi-bindings cache element 75
JRE 50
JTA transaction 435

disabling 110
pooled connections 60
simple connections 63
JVM failure 65

K

key-constraint region attribute 111
keySetOnServer
unexpected results 207
keyword in querying 344, 527
prefix for literals 372
keywords, querying 352

L

last accessed time statistic 469
last modified time statistic 469
limited-access membership attribute 456
listener, UniversalMembershipListener 325
load-conditioning-intervalt client configuration parameter 210
load-factor region attribute 112
load-poll-interval server configuration parameter 208
local region 116–117, 168
local scope. *See* scope region attribute
local-max-memory partition attribute 114
locator 45, 203
locator client configuration parameter 210
LocatorStatistics 467
lock grantor 166
locking
configuring 111
explicit
DistributedLockService 166
entry API 166
Region API 166
global scope 165–166
implicit 166
lock grantor 166
manual 166
non-region 166
region and entry lock timeouts 166, 471
lock-lease cache attribute 75, 166, 471
lock-timeout cache attribute 74, 166, 471
logging 66
client/server exceptions, suppressing 264
loss-action membership attribute 456
LRU eviction
configuring 108
transactions 429
LRU eviction controller 40
LRUStatistics 467

M

ManagedDataSource 59
manual-start
gateway hub configuration parameter 272
Map Interface, GemFire Region 130
max-connections client configuration parameter 210
max-connections server configuration parameter 208, 261
maximum-message-count server configuration parameter 209
maximum-queue-memory gateway queue attribute 273
maximum-time-between-pings
gateway hub configuration parameter 272
maximum-time-between-pings server configuration parameter 209
max-threads server configuration parameter 260–261
max-threadsserver configuration parameter 208
MemberHealthConfig, health monitoring API 483
membership
attributes
limited-access 456
loss-action 455–456
membership roles 455
no-access 457
reconnect 457
required-role 458
resumption-action 455, 458
configuring 112
listener 96, 296, 305
reliability 455
requirements, setting at the region level 40
role 454
MembershipAttributes 306
membership-attributes region attribute 112
memory limitation
client/server subscription queue configuration 257
empty partition 149
empty region 104
gateway
queue 267
temporary queue 274
local-destroy 109
LRU_MEMORY for eviction 108
overflow to disk 171
overflow-to-disk 109
partitioned region LOCAL_MAX_MEMORY 114
query index 375
statistics, enabling 469
message-sync-interval cache attribute 75
message-sync-interval server configuration

parameter 207
`message-time-to-live` server configuration parameter 209, 244
messaging 318
asynchronous queue conflation, configuring 106
client/server, configuring 246, 265
durable client 246
order of delivery 316
queue
conflation, configuring 107
conflation, multi-site 277
gateway 275
gateway, configuring 270, 273
gateway, failure 276
gateway, slow distributed system 274
multi-site, configuring 477
multi-site, conflation 277
multi-site, overflow to disk 267
peer-to-peer, conflation 473
peer-to-peer, slow receivers, for 473
role loss detection 460
statistics for tracking, defining 464
synchronization 75
scope 316
slow consumer
configuring 106
`distributed-no-ack` scope 473
queue conflation 473
methods
`equals` 350
`hashCode` 350
methods, writing 304
`min-connections` client configuration parameter 210
miss count statistic 469
modify in-place, avoiding 140
`multicast-enabled` region attribute 113
multi-hub configurations 279
multi-site
client/server 268
endpoint-hub relationship 269
example 286
failures 285
`Gateway` 270
`Gateway.Endpoint` 270
`GatewayHub` 270
`GatewayQueueAttributes` 270
messaging failures 276
performance 274, 277
propagated operations 274
topologies 289
multi-site installation. *See* gateway hub
`multiuser-authentication` client configuration parameter 210

N

`name` client configuration parameter 211
named region attributes 97, 99
`NameNotFoundException` 355
`NameResolutionException` 355
nesting 355
net operations 162
`netLoad` 136
automatic invocation from `netSearch` 136
`netSearch` 136
client, disabled for 224
expiration 136
invocation from cache loader helper 136
`netLoad`, automatic invocation of 136
timeout specified in cache attribute 75, 471
network failure
membership roles and network segmentation 460
multi-site parallel network topology 289
partitioned region redundancy 153–154
region scope, effect of 165
network, WAN. *See* multi-site
`netWrite`
entry retrieval, interaction with 137
`no-access` membership attribute 457

O

OQL query language 352
overflow of region entry data 40, 170–172, 182
configuring for
example 173
entry retrieval 175
indexing with 351

P

package 53
administration 53
caching 53
distributed system 54
queries and indexes 53
security authorization 54
statistics 53
utility classes 53
partition attributes
configuring 113
`global-max-buckets` 115
local-max-memory 114
`redundant-copies` 114
stored 100
`partition-attributes` region attribute 113
partitioned region 39, 46–47, 146
adding host members 153

bucket units 46
bucket, configuration 115
cache listener 102
cache loader 102
configuring partition attributes 113
creating region instances 46
data colocation 157
data management 156
data redundancy 152
`distributed-ack` scope 165
`distributed-no-ack` scope 165
entry
destruction 144
modification 142
event 295
`global` scope 165
high availability 152
configuration 153
planning 153
indexing 384
local data storage, without 149
rebalancing 150
redundant storage, fault tolerance 47
`Region` method behavior 130
region operations 46
statistics 468
password, encrypting 64
performance 470–473
client/server 259–260
controls 470–473
`copy-on-read`, effect of 140
disk storage 175
`distributed-no-ack` scope 425
high availability impact 152
indexes for querying 375
locking 167
querying 386
persist region data to disk 39
persistence of region entry data
eviction 185
persistence of region entry data to disk 171–172
configuring for
example 173, 185
entry retrieval 175
recovery from disk 175
`ping-interval` client configuration parameter 211
pool
client/server subscription management 229
`pool` cache element 75
`pool` region attribute 115
PooledDataSource 59
`PoolStatistics` 467
`port` gateway
endpoint configuration parameter 273
hub configuration parameter 272
`port` server configuration parameter 209
`portfolios` region, sample data 343
`ProcessStats` 467
productDir 49
programming
with GemFire 49
programming examples
caching API 389
`pr-single-hop-enabled` client configuration parameter 211
pull distribution model 163
pure Java mode 52
pure Java mode, system scenarios 42
push distribution model 163

Q

`QRegion` interface 353
`query` 387
query language
OQL 352
SQL-92 352
query string 356–383
comments in 374
comparison operators, comparison operators, query string 368
construction expression 371
conversions 374
binary numeric promotion 374
method invocation 374
temporal type 374
dot and forward slash operators 369
`fieldname:expression` usage 364
functions 370
`IMPORT`, use of 363
`IN` expression 371
`java.util.Date` 373
literals 372
logical operators, logical operators, query string 368
map and index operators, map and index operators, query string 369
method invocation in 367
object type 362
importing classes 361
specifying 361
operators 368–369
parameter references 373
`SELECT` 356–357
`FROM` clause 354, 357–358, 360
projection list 357, 364
results 365
`WHERE` clause 357, 363–364

unary operators 369
UNDEFINED 370
use of region attribute ValueConstraint 361
querying 341–395
accessing different object types 358
attribute visibility 354
attributes with keyword names 344
distributed cache 350
modifying the cache 351
overflow region 351
region scope 350
events in client/server 397
examples 344–346
keywords 352, 527
literal keywords 372
nested scopes 355
object data 347–349
bringing objects into scope 349, 353, 355
compared to relational data 347
object-type dependent 358
performance considerations 386
programmatic query creation 356
QRegion interface 353
region
name requirements 92
object representation 353
path specification 352
reserved words 357
SelectResults 365
transactions 428

R

read-through caching 136
readTimeout client configuration parameter 211
reconnect membership attribute 457
redundant entries, performance impact 152
redundant storage, on for fault tolerance in partitioned region 47
redundant-copies partition attribute 114
refid of region attributes 504
refid of region attributes 504
regex interest registration 220
region 86
access 94
API, functional overview 91–96
attributes
API, functional overview 91
cache-writer 103
concurrency-level 103
data-policy 104
enable-async-conflation 106
enable-gateway 107
enable-subscription-conflation 107
entry-idel-time 108
entry-time-to-live 108
ignore-JTA 110
index-update-type 110
initial-capacity 110
is-lock-generator 111
key-constraint 111
load-factor 112
LRU eviction 108
membership-attributes 112
multicast-enabled 113
partitioned-attributes 113
pool 115
region-idle-time 116
region-time-to-live 116
scope 116
statistics-enabled 117
subscription-attributes 118
template 504
user-defined 120
closing 96
creation 305
membership role loss during 460
destruction
client/server 217, 219–220, 234
high availability of 171
index, configuring 110
naming conventions 92
partitioned. *See* partitioned region
pathname specification 94
RegionMembershipListener 305
RegionRoleListener 306
role 306
shortcuts 97–99
statistics 469
region attributes
stored 97
storing and retrieving 97, 99
region cache element 76
region instances, creating for partitioned region 46
region shortcuts 97
Region. *See* com.gemstone.gemfire.cache
Region.Entry.
See com.gemstone.gemfire.cache
region-attributes cache element 76
region-idle-time region attribute 116

RegionMembershipListener 296, 306
RegionRoleListener 296, 455
regions
  cache loader 39
  distributed 39
  LRU, eviction controller 40
  memory consumption 40
  partitioned 39
  persist region data 39
  replicated 39
  setting membership requirements for 40
RegionShortcut 97
region-time-to-live region attribute 116
register interest 220
  client region configuration 220
regular expressions
  interest registration 220
reliability, member 455
repeatable-read. See cache transaction repeatable-read
replicate. *See* data
replicated region 39
required-role membership attribute 458
resource manager 186
  background eviction 187
  configuring 188
  thresholds 186
resource-manager cache element 76
resumption-action membership attribute 458
retry-attempts client configuration parameter 211
role, membership 453–454
  configuring 112
  event 298
  inactive cache 460
  listener 296, 455
  loss 460
  membership attributes 455
  region creation 460
  RegionMembershipListener 306
  transactions 461

S

scalability 201
scope of cache queries 347, 349, 353, 355, 358
scope region attribute 116, 168
  application plug-ins 116
  cache loader 197
  cache writer 312
  data consistency 351
  distributed 161
    application plug-ins 162
    net operations 162
  distributed scopes, incompatibility of 165
  distributed-ack 117, 316
    ack timeout 165
    data distribution 165
    indexing 351
    membership role loss 461
    querying 351
    transactions 422
  distributed-ack, entry retrieval 137
  distributed-no-ack 117, 316
    data distribution 165
    indexing 351
    membership role loss 460
    querying 351
    transactions 425
  distributed-no-ack scope
    slow consumer 473
  distributed-no-ack, entry retrieval 137
  entry retrieval 137
  global 117, 164, 166
    locking 130, 471
    data distribution 165
    entry retrieval 137
    indexing 351
    locking 166
    membership role loss 461
    querying 351
  local 117, 164, 168
    disabled in empty region 321
    entry retrieval 136–137
    indexing 351
    netSearch, disabling 224
    querying 351
    transactions 422
  locking, distributed 111, 130, 471
  message delivery ordering 316
  performance 473
  querying considerations 350
  slow receiver 473
  transactions 422–425
search-timeout cache attribute 75
search-timeout cache attribute 471
security
  client/server
    multi-user 82
SELECT statement 357
SelectResults, querying 365
serialization of data 57
serialization-registration cache element 76
server
  socket configuration 265
  troubleshooting 476–477
server balancing 232
server client configuration parameter 211
server conditioning 232

server configuration parameters
bind-address 207
custom-load-probe 207
group 208
hostname-for-clients 208
load-poll-interval 208
max-connections 208
maximum-message-count 209
maximum-time-between-pings 209
max-threads 208
message-sync-interval 207
message-time-to-live 209
port 209
socket-buffer-size 209
server grouping 233
server locator 45, 203–205, 230
server-group client configuration parameter 211
ServerLoadProbe 232
services, GemFire 39
SimpleDataSource 59
slow consumer. *See* messaging 106
socket
configuration 471
conserve-sockets 260, 471
releasing 472
socket-lease-time 472
thread use 260, 471
socket backlog 265
server configuration 265
socket-buffer-size
gateway configuration parameter 273
gateway hub configuration parameter 272
socket-buffer-size client configuration parameter 211
socket-buffer-size server configuration parameter 209, 265–266
socket-lease-time 472
socket-read-timeout gateway configuration parameter 273
SQL
ad-hoc querying 352
continuous query 403
SQL-style queries, running in client/server applications 40
startup
cache structure, defining 499
startup-policy gateway
hub configuration parameter 272
statistic-interval client configuration parameter 211
statistics 55, 66, 464
application 464
cache transaction 430
distributed system 466
enabling for a region 117
entry 469
hit count 469
last accessed time 469
last modified time 469
miss count 469
partitioned region 468
region 469
retrieving 117, 177, 469
viewing in VSD 56
Statistics API. *See* com.gemstone.gemfire
statistics-enabled region attribute 117, 464
StatSampler 467
subscription
client/server 220
subscription-ack-interval client configuration parameter 212
subscription-attributes region attribute 118
subscription-enabled client configuration parameter 212
subscription-message-tracking-timeout client configuration parameter 212
subscription-redundancy-level client configuration parameter 212
support, technical 29
system administration tools, developing 479
system failure 65
system property 274
SystemStats 467

T

TCP/IP connection buffer size 472
technical support 29
template, region attributes 504
thin client model 319
thin produer model 319
thread
asynchronous indexing 110
background for asynchronous indexing 377
controlling use
client/server 259–262
configuring maximum 103
configuring maximum in server 208
distributed-no-ack for data distribution 316
gateway processor 275
IDs, tracking on client 240
ordered operations in a single 316
slow receiver with many 472
socket configuration for 471

thread-safe
dynamic region 128
listener use 316
loader method 198
plug-ins 123
transaction listener 308, 418
`thread-local-connections` client configuration parameter 212
time-to-live 179
tools, system administration 479
transaction
application plug-ins 418
event 298
ID in entry event 298
membership role loss during 461
troubleshooting 477
unexpected operation sequence 425
unexpected results 425
transaction event
handling 317
transaction facility
API 435
transaction listener 296, 308
callback methods 308
callbacks 308
commit conflict investigation 430
creating 302
over-capacity recovery 430
transport layer 36
TTL. *See* time-to-live
tuning
durable client timeout 247
typographical conventions 26

U

unary operators, query string 369
`UNDEFINED query string` 370
`UniversalMembershipListener` 325
user-defined
objects 57

V

Visual Statistics Display (VSD) 56
VM
statistics 467
`VMGCStats` 467
`VMMemory*Stats` 467
`VMMemoryPoolStats` 467
`VMStats` 467

W

WAN. *See* multi-site
WBCL 299, 311
write behind cache listener 297
event 299
event handler 299
`GatewayEventListener` 311
write-ahead listener. *See* cache writer
write-behind listener. *See* cache listener
write-through caching 312–313

X

XAPooledDataSource 59
XML file. *See* `cache.xml`