1. 1.1 Development Cycle		5
2. Creating and Deleting Indexes	5	5
3. Django and MongoDB		5
4. Older Downloads		-
5. PyMongo and mod_wsgi		,
5. PyMongo and mod_wsgl)
6. Python Tutorial		5
7. Recommended Production Architectures		
8. Shard v0.7		
9. v0.8 Details	5	5
10. v0.9 Details		
11. v1.0 Details		
12. v1.5 Details		
13. v2.0 Details		
14. Building SpiderMonkey		
15. Documentation	6	ŝ
16. Dot Notation	6	3
17. Dot Notation		
18. Getting the Software		7
19. Language Support		
20. Mongo Administration Guide	!	7
20. Wiongo Administration Guide		_
21. Working with Mongo Objects and Classes in Ruby	!	1
22. MongoĎB Language Support	7	7
23. Community Info	7	7
24. Internals	7	7
25. TreeNavigation		
26. Old Pages		
26.1 Storing Data		
26.2 Indexes in Mongo		
26.3 HowTo		3
26.4 Searching and Retrieving	8	3
26.4.1 Locking	8	3
26.5 Mongo Developers' Guide	8	3
26.6 Locking in Mongo	8	3
26.7 Mongo Database Administration	8	3
26.8 Mongo Concepts and Terminology		á
26.9 MongoDB - A Developer's Tour		ر م
20.9 Miorigodos - A Developer's Tour		1
26.10 Updates	§	9
26.11 Structuring Data for Mongo	9	Э
26.12 Design Overview	9	Э
26.13 Document-Oriented Datastore		9
26.14 Why so many "Connection Accepted" messages logged?		
26.15 Why are my datafiles so large?	(ς.
26.16 Storing Files		á
26.17 Introduction - How Mongo Works		ر م
20.17 Introduction - How Worldy Works		2
26.18 Optimizing Mongo Performance	§	9
26.19 Mongo Usage Basics	9	Э
26.20 Server-Side Processing		
27. Home		10
27.1 Quickstart		10
27.2 Downloads		
27.2.1 1.0 Changelist		
27.2.2 1.2.x Release Notes		
27.2.3 1.4 Release Notes		
27.2.4 Version Numbers		
27.2.5 Getting Started	1	15
27.3 Drivers	1	16
27.3.1 Driver Syntax Table	1	17
27.3.2 Javascript Language Center		17
27.3.3 JVM Languages		
27.3.4 Python Language Center		
27.3.5 PHP Language Center		
27.3.5.1 Installing the PHP Driver		
27.3.5.2 PHP - Storing Files and Big Data		
27.3.5.3 Troubleshooting the PHP Driver	2	25
27.3.6 Ruby Language Center	2	26
27.3.6.1 Ruby Tutorial		27
27.3.6.1.1 Replica Pairs in Ruby		
27.3.6.2 Rails - Getting Started		
27.3.6.3 MongoDB Data Modeling and Rails		oc oc
27.3.6.4 Object Mappers for Ruby and MongoDB		
27.3.6.5 Ruby External Resources		
27.3.7 Java Language Center		
27.3.7.1 Java Driver Concurrency		41
27.3.7.2 Java - Saving Objects Úsing DBObject		41
27.3.7.3 Java Tutorial		

27.3.7.4 Java Types 27.3.8 C++ Language Center	
27 3.8 C++ Language Center	
27.0.0 011 Eariquage Contor	 47
27.3.8.1 C++ Tutorial	
27.3.8.2 Connecting	
27.3.9 Perl Language Center	 51
27.3.9.1 Contributing to the Perl Driver	 52
27.3.9.2 Perl Tutorial	53
27.0.0.2 of 10 Tutorial	 50
27.3.10 Online API Documentation	
27.4 Developer Zone	 53
27.4.1 Tutorial	54
27.4.2 Manual	
27.4.2.1 Connections	
27.4.2.2 Databases	 60
27.4.2.2.1 Commands	
27.4.2.2.2 Mongo Metadata	
27.4.2.3 Collections	 69
27.4.2.3.1 Capped Collections	
27.4.2.3.2 Using a Large Number of Collections	 7.1
27.4.2.4 Data Types and Conventions	 71
27.4.2.4.1 Internationalized Strings	 71
27.4.2.4.2 Object IDs	 72
27.4.2.4.2 Object IDS	 12
27.4.2.4.3 DB Ref	 72
27.4.2.5 GridFS	 74
27.4.2.6 Indexes	
27 A S A Indoving on a Bookground Operation	 77
27.4.2.6.1 Indexing as a Background Operation	 11
27.4.2.6.2 Multikeys	 78
27.4.2.7 Inserting	 79
27.4.2.7.1 Legal Key Names	90
27.4.2.7.2 Schema Design	 81
27.4.2.7.3 Trees in MongoDB	 82
27.4.2.8 Optimization	
27.4.2.8.1 Optimizing Storage of Small Objects	 06
27.4.z.o.i Optimizing Storage of Small Objects	 00
27.4.2.8.2 Query Optimizer	 86
27.4.2.9 Querying	 87
27.4.2.9.1 Advanced Queries	87
27.4.2.0.1 Payallocal guerros	 01
27.4.2.9.2 Dot Notation (Reaching into Objects)	 91
27.4.2.9.3 Full Text Search in Mongo	 92
27.4.2.9.4 min and max Query Specifiers	93
27.4.2.9.5 OR operations in query expressions	
27.4.2.9.5 ON operations in query expressions	 94
	~ 4
27.4.2.9.6 Queries and Cursors	 94
27.4.2.9.7 Server-side Code Execution	 94 96
27.4.2.9.7 Server-side Code Execution	 94 96
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order	 94 96 99
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation	 94 96 99 99
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing	 94 96 99 99 102
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing	 94 96 99 99 102
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating	 94 96 99 99 102 102
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations	 94 96 99 99 102 102
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command	 94 96 99 99 102 105 106
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo	94 96 99 99 102 102 105 106 107
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo	94 96 99 99 102 102 105 106 107
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce	 94 96 99 102 102 105 106 107
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual	94 96 99 102 105 106 107 109 112
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell	94 96 99 99 102 105 106 107 109 112
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell	94 96 99 99 102 105 106 107 109 112 113 113
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11 MapReduce 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference	94 96 99 102 105 106 107 109 112 113 113
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11 MapReduce 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference	94 96 99 102 105 106 107 109 112 113 113
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ	94 96 99 99 102 105 106 107 109 113 113 115 116
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11 MapReduce 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types	94 96 99 99 102 105 106 107 109 112 113 115 116 118
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ	94 96 99 99 102 105 106 107 109 112 113 115 116 118
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11 MapReduce 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types	94 96 99 99 102 105 106 107 109 113 113 115 116 118 119
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center	94 96 99 99 102 105 106 107 109 112 113 115 116 118 119 120
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2	94 96 99 99 102 105 106 107 113 113 115 116 118 119 120
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent	 94 96 99 99 102 105 106 107 112 113 115 116 118 119 120 121
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2	 94 96 99 99 102 105 106 107 112 113 115 116 118 119 120 121
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11 Updating 27.4.2.11.3 Ipdating Data in Mongo 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2 Administration Basics	94 96 99 99 102 105 106 107 109 112 113 115 116 118 119 120 121 121
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2 Administration Basics 27.5.2.1 Starting and Stopping Mongo	94 96 99 99 102 105 106 107 109 112 113 115 116 118 119 120 121 121
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Command Line Parameters	94 96 99 99 102 105 106 107 109 113 113 115 116 118 119 120 121 121 121 121
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.8 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Updating 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration	94 96 99 99 102 105 106 107 109 112 113 115 116 118 120 121 121 121 121 123 124
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Command Line Parameters	94 96 99 99 102 105 106 107 109 112 113 115 116 118 120 121 121 121 121 123 124
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 Ifindandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2 Administration Basics 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools	94 96 99 99 102 105 106 107 112 113 115 116 118 120 121 121 121 121 121 123 124 124
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.8 Ogtring and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2 Administration Basics 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.4 Import Export Tools 27.5.2.5 GridFS Tools	94 96 99 99 102 105 106 107 112 113 113 115 116 118 120 121 121 121 121 121 122 124 124 124 124
27. 4.2.9.7 Server-side Code Execution 27. 4.2.9.8 Sorting and Natural Order 27. 4.2.9.9 Aggregation 27. 4.2.10 Removing 27. 4.2.11 Updating 27. 4.2.11.1 Atomic Operations 27. 4.2.11.2 findandmodify Command 27. 4.2.11.3 Updating Data in Mongo 27. 4.2.12 MapReduce 27. 4.2.13 Data Processing Manual 27. 4.3 mongo - The Interactive Shell 27. 4.3.1 Overview - The MongoDB Interactive Shell 27. 4.3.2 dbshell Reference 27. 4.4 Developer FAQ 27. 4.1 Compare Order for Types 27. 5.4 dmin Zone 27. 5.1 Hosting Center 27. 5.1.1 Amazon EC2 27. 5.1.2 Joyent 27. 5.2.1 Starting and Stopping Mongo 27. 5.2.2 Command Line Parameters 27. 5.2.3 File Based Configuration 27. 5.2.4 Import Export Tools 27. 5.2.5 GridFS Tools 27. 5.2.6 Http Interface	94 96 99 102 102 105 106 107 119 113 115 116 120 121 121 121 123 124 124 125 126 126 126 126
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2 Administration Basics 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.6 Ofthp Interface 27.5.2.6 Department of the Shell	94 96 99 102 105 106 107 109 112 113 115 116 120 121 121 121 121 124 124 126 126 128
27. 4.2.9.7 Server-side Code Execution 27. 4.2.9.8 Sorting and Natural Order 27. 4.2.9.9 Aggregation 27. 4.2.10 Removing 27. 4.2.11 Updating 27. 4.2.11.1 Atomic Operations 27. 4.2.11.2 findandmodify Command 27. 4.2.11.3 Updating Data in Mongo 27. 4.2.12 MapReduce 27. 4.2.13 Data Processing Manual 27. 4.3 mongo - The Interactive Shell 27. 4.3.1 Overview - The MongoDB Interactive Shell 27. 4.3.2 dbshell Reference 27. 4.4 Developer FAQ 27. 4.1 Compare Order for Types 27. 5.4 dmin Zone 27. 5.1 Hosting Center 27. 5.1.1 Amazon EC2 27. 5.1.2 Joyent 27. 5.2.1 Starting and Stopping Mongo 27. 5.2.2 Command Line Parameters 27. 5.2.3 File Based Configuration 27. 5.2.4 Import Export Tools 27. 5.2.5 GridFS Tools 27. 5.2.6 Http Interface	94 96 99 102 105 106 107 109 112 113 115 116 120 121 121 121 121 124 124 126 126 128
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.3 Kadministration Basics 27.5.2.1 Starting and Stopping Mongo 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 Import Export Tools 27.5.2.5 DBA Operations from the Shell 27.5.4 Sharding	94 96 99 102 105 106 107 109 112 113 115 116 118 120 121 121 121 122 124 124 126 128 130
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3.1 Overview - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5 Admin Zone 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 Griffs Tools 27.5.2.5 Griffs Tools 27.5.2.5 Griffs Tools 27.5.2.6 Http Interface 27.5.3 DBA Operations from the Shell 27.5.4 Sharding 27.5.4 Sharding 27.5.4 Sharding	94 96 99 99 102 105 106 107 109 112 113 115 116 118 120 121 121 121 122 124 124 126 128 130 130
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Administration Basics 27.5.2.1 Starting and Stopping Mongo 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.6 Http Interface 27.5.3 DBA Operations from the Shell 27.5.4 Sharding 27.5.4 Sharding 27.5.4.1 Sharding FAQ 27.5.4.1 Sharding FAQ 27.5.4.2 Sharding Introduction	94 96 99 99 102 105 106 107 109 112 113 113 115 116 120 121 121 121 124 126 126 128 130 131
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Oxfing and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.113 Data Processing Manual 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3 noveriwer - The MongoDB Interactive Shell 27.4.3 Cyreiver - The MongoDB Interactive Shell 27.4.4 Compare Order for Types 27.5 Admin Zone 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2 Administration Basics 27.5.2.1 Starting and Stopping Mongo 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.2.5 Sharding 27.5.4.1 Sharding FAQ 27.5.4.1 Sharding FAQ 27.5.4.3 Configuring Introduction 27.5.4.3 Configuring Introduction 27.5.4.3 Configuring Sharding	94 96 99 99 102 105 106 107 109 112 113 115 116 118 119 120 121 121 121 124 126 126 128 130 131 131
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.9 Aggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 dbshell Reference 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Administration Basics 27.5.2.1 Starting and Stopping Mongo 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.6 Http Interface 27.5.3 DBA Operations from the Shell 27.5.4 Sharding 27.5.4 Sharding 27.5.4.1 Sharding FAQ 27.5.4.1 Sharding FAQ 27.5.4.2 Sharding Introduction	94 96 99 99 102 105 106 107 109 112 113 115 116 118 119 120 121 121 121 124 126 126 128 130 131 131
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 doshell Reference 27.4.4.1 Compare Order for Types 27.4.4.1 Compare Order for Types 27.5.4 Admin Zone 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.2.5 Brading FAQ 27.5.4.1 Sharding FAQ 27.5.4.2 Sharding FAQ 27.5.4.2 Sharding FAQ 27.5.4.3 DBA Operations from the Shell 27.5.4.3 Configuring Sharding 27.5.4.3 Ta Sample Configuration Session	94 96 99 99 102 105 106 107 119 113 113 115 116 118 120 121 121 121 124 126 128 130 131 131 134 135
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.2 findandmodify Command 27.4.2.13 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.4.1 Compare Order for Types 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Administration Basics 27.5.2.3 File Based Configuration 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.4 Sharding 27.5.4.1 Sharding FAQ 27.5.4.2 Sharding Introduction 27.5.4.3 Sharding Administration	94 96 99 102 102 105 106 107 113 113 113 115 116 118 120 121 121 121 121 124 126 128 130 131 131 134 135 136 131 131 131 131 131 131 131 131 131
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.9.8 Oggregation 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.3 Updating Data in Mongo 27.4.2.12 MapReduce 27.4.2.13 Data Processing Manual 27.4.3 mongo - The Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.2 Obshell Reference 27.4.4.1 Compare Order for Types 27.5.4 Hosting Center 27.5.1 Hosting Center 27.5.1 Hosting Center 27.5.1.2 Joyent 27.5.2.1 Starting and Stopping Mongo 27.5.2.2 Command Line Parameters 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.4.1 Sharding FAQ 27.5.4.1 Sharding FAQ 27.5.4.2 Sharding Introduction 27.5.4.3 Sharding Introduction 27.5.4.3 Configuring Sharding 27.5.4.3 Sharding Introduction 27.5.4.3 Sharding Administration 27.5.4.4 Sharding Administration 27.5.4.5 Sharding and Failover	94 96 99 102 102 105 106 107 113 113 115 116 120 121 121 121 123 124 124 126 128 130 131 131 134 135 136 131 131 131 131 131 131 131 131 131
27.4.2.9.7 Server-side Code Execution 27.4.2.9.8 Sorting and Natural Order 27.4.2.10 Removing 27.4.2.11 Updating 27.4.2.11 Updating 27.4.2.11.1 Atomic Operations 27.4.2.11.2 findandmodify Command 27.4.2.11.2 findandmodify Command 27.4.2.13 Updating Data in Mongo 27.4.2.13 Data Processing Manual 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.3.1 Overview - The MongoDB Interactive Shell 27.4.4.1 Compare Order for Types 27.4.4 Developer FAQ 27.4.4.1 Compare Order for Types 27.5.1 Hosting Center 27.5.1.1 Amazon EC2 27.5.1.2 Joyent 27.5.2.2 Administration Basics 27.5.2.3 File Based Configuration 27.5.2.3 File Based Configuration 27.5.2.4 Import Export Tools 27.5.2.5 GridFS Tools 27.5.2.5 GridFS Tools 27.5.4 Sharding 27.5.4.1 Sharding FAQ 27.5.4.2 Sharding Introduction 27.5.4.3 Sharding Administration	94 96 99 102 102 105 106 107 113 113 115 116 120 121 121 121 123 124 124 126 128 130 131 131 134 135 136 131 131 131 131 131 131 131 131 131

27.5.5 Replication	137
27.5.5.1 Master Slave	
27.5.5.2 Replica Pairs	
27.5.5.3 Master Master Replication	
27.5.6 Production Notes	 142
27.5.7 Security and Authentication	 143
27.5.8 Mongo Architecture	 144
27.5.9 How to do Snapshotted Queries in the Mongo Database	 144
27.5.10 Backups	
27.5.11 Database Profiler	 146
27.5.12 Diagnostics and Troubleshooting	 1/10
27.5.13 Durability and Repair	 140
27.6 Contributors	
27.6.1 C++ Coding Style	
27.6.2 Project Ideas	
27.6.3 UI	 151
27.6.4 Source Code	 152
27.6.5 Building	
27.6.5.1 Building for FreeBSD	
27.6.5.2 Building for Linux	 152
27.6.5.3 Building for OS X	 152
27.6.5.4 Building for Solaris	
27.6.5.5 Building for Windows	 158
27.6.5.5.1 Boost 1.41.0 Visual Studio 2010 Binary	 159
27.6.5.6 Building Spider Monkey	
27.6.6 Database Internals	 160
27.6.6.1 Caching	 160
27.6.6.2 Concurrency	 161
27.6.6.3 Cursors	
27.6.6.4 Sharding Internals	
27.6.6.4.1 Moving Chunks	 101
27.6.6.4.2 Sharding Commands	
27.6.6.4.3 Sharding Design	 162
27.6.6.4.4 Sharding Use Cases	 163
27.6.6.4.5 Shard Ownership	 163
27.6.6.4.6 Splitting Shards	 164
27.6.6.5 Error Codes	 164
27.6.6.6 Internal Commands	
27.6.6.7 Replication Internals	
27.6.6.8 Smoke Tests	 166
27.6.6.9 Pairing Internals	 166
27.6.7 Writing Drivers and Tools	
27.6.7.1 Overview - Writing Drivers and Tools	 100
27.6.7.1.1 Mongo Driver Requirements	 167
27.6.7.1.2 Spec, Notes and Suggestions for Mongo Drivers	
27.6.7.1.3 Feature Checklist for Mongo Drivers	 171
27.6.7.1.4 Conventions for Mongo Drivers	 172
27.6.7.2 Mongo Wire Protocol	 173
27.6.7.3 BSON	 177
27.6.7.4 Mongo Extended JSON	
27.6.7.5 GridFS Specification	
27.6.7.6 Implementing Authentication in a Driver	
27.6.7.7 Notes on Pooling for Mongo Drivers	
27.6.7.8 Error Handling in Mongo Drivers	
27.6.8 Contributing to the Documentation	
27.6.8.1 Mongo Documentation Style Guide	
27.7 Community	
27.7.1 Community Tools	 192
27.7.2 MongoDB Commercial Services Providers	 192
27.7.3 User Feedback	 193
27.8 About	
27.8.1 Roadmap	
27.8.2 Production Deployments	
27.8.3 Articles	
27.8.4 Benchmarks	
27.8.5 Use Cases	
27.8.5.1 Use Case - Session Objects	
27.8.6 FAQ	
27.8.7 Product Comparisons	
27.8.7.1 Interop Demo (Product Comparisons)	 200
27.8.7.2 MongoDB, CouchDB, MySQL Compare Grid	
27.8.7.3 Comparing Mongo DB and Couch DB	
27.8.8 Philosophy	
27.8.9 Licensing	
27.9 International Documentation	

27.10 Documentation Index .		203
-----------------------------	--	-----

1.1 Development Cycle

Error formatting macro: redirect: java.lang.NullPointerException

Creating and Deleting Indexes

Error formatting macro: redirect: java.lang.NullPointerException

Django and MongoDB

Error formatting macro: redirect: java.lang.NullPointerException

Older Downloads

Error formatting macro: redirect: java.lang.NullPointerException

PyMongo and mod_wsgi

Error formatting macro: redirect: java.lang.NullPointerException

Python Tutorial

Error formatting macro: redirect: java.lang.NullPointerException

Recommended Production Architectures

Error formatting macro: redirect: java.lang.NullPointerException

Shard v0.7

Simple auto-sharding suitable for things such as file attachment chunks.

Easy addition and removal of nodes from live systems.

v0.8 Details

Existing Core Functionality

- · Basic Mongo database functionality: inserts, deletes, queries, indexing.
- Master / Slave Replication
- Replica Pairs
- Server-side javascript code execution

New to v0.8

- Drivers for Java, C++, Python, Ruby.
- · db shell utility
- · (Very) basic security
- \$or
- Clean up logging
- Performance test baseline
- getlasterror
- Large capped collections
- Bug fixes (compound index keys, etc.)
- Import/Export utility
- Allow any _id that is unique, and verify uniqueness

Wanted, but may not make it

- AMI's
- Unlock eval()?
- Better disk full handling
- better replica pair negotiation logic (for robustness)

v0.9 Details

v0.9 adds stability and bug fixes over v0.8.

The biggest addition to v0.9 is a completely new and improved query optimizer.

v1.0 Details

v1.5 Details

v2.0 Details

Building SpiderMonkey

Error formatting macro: redirect: java.lang.NullPointerException

Documentation

Error formatting macro: redirect: java.lang.NullPointerException

Dot Notation

Error formatting macro: redirect: java.lang.NullPointerException

Dot Notation

Error formatting macro: redirect: java.lang.NullPointerException

Getting the Software

Placeholder - \$\$\$ TODO

Language Support

Error formatting macro: redirect: java.lang.NullPointerException

Mongo Administration Guide

Error formatting macro: redirect: java.lang.NullPointerException

Working with Mongo Objects and Classes in Ruby

Error formatting macro: redirect: java.lang.NullPointerException

MongoDB Language Support

Error formatting macro: redirect: java.lang.NullPointerException

Community Info

Error formatting macro: redirect: java.lang.NullPointerException

Internals

Cursors

Tailable Cursors

See p/db/dbclient.h for example of how, on the client side, to support tailable cursors.

Set

Option CursorTailable = 2

in the queryOptions int field to indicate you want a tailable cursor.

If you get back no results when you query the cursor, keep the cursor live if cursorid is still nonzero. Then, you can issue future <code>getMore</code> requests for the cursor.

If a getMore request has the resultFlag ResultFlag_CursorNotFound set, the cursor is not longer valid. It should be marked as "dead"

ResultFlag_CursorNotFound = 1

See the Queries and Cursors section of the Mongo Developers' Guide for more information about cursors.

See Also

· The Queries and Cursors section of the Mongo Developers' Guide for more information about cursors

TreeNavigation

Old Pages

Storing Data

Error formatting macro: redirect: java.lang.NullPointerException

Indexes in Mongo

Error formatting macro: redirect: java.lang.NullPointerException

HowTo

Error formatting macro: redirect: java.lang.NullPointerException

Searching and Retrieving

Error formatting macro: redirect: java.lang.NullPointerException

Locking

Error formatting macro: redirect: java.lang.NullPointerException

Mongo Developers' Guide

Error formatting macro: redirect: java.lang.NullPointerException

Locking in Mongo

Error formatting macro: redirect: java.lang.NullPointerException

Mongo Database Administration

Error formatting macro: redirect: java.lang.NullPointerException

Mongo Concepts and Terminology

Error formatting macro: redirect: java.lang.NullPointerException

MongoDB - A Developer's Tour

Error formatting macro: redirect: java.lang.NullPointerException

Updates

Error formatting macro: redirect: java.lang.NullPointerException

Structuring Data for Mongo

Error formatting macro: redirect: java.lang.NullPointerException

Design Overview

Error formatting macro: redirect: java.lang.NullPointerException

Document-Oriented Datastore

Error formatting macro: redirect: java.lang.NullPointerException

Why so many "Connection Accepted" messages logged?

Error formatting macro: redirect: java.lang.NullPointerException

Why are my datafiles so large?

Error formatting macro: redirect: java.lang.NullPointerException

Storing Files

Error formatting macro: redirect: java.lang.NullPointerException

Introduction - How Mongo Works

Error formatting macro: redirect: java.lang.NullPointerException

Optimizing Mongo Performance

Error formatting macro: redirect: java.lang.NullPointerException

Mongo Usage Basics

Error formatting macro: redirect: java.lang.NullPointerException

Server-Side Processing

Error formatting macro: redirect: java.lang.NullPointerException

Home

Combining the best features of document databases, key-value stores, and RDBMSes.

MongoDB (from "humongous") is a scalable, high-performance, open source, schema-free, document-oriented database. Written in C++, MongoDB features:

- Document-oriented storage (the simplicity and power of JSON-like data schemas)
- Dynamic queries
- Full index support, extending to inner-objects and embedded arrays
- Query profiling
- Fast, in-place updates
- Efficient storage of binary data large objects (e.g. photos and videos)
- Replication and fail-over support
- Auto-sharding for cloud-level scalability
- MapReduce for complex aggregation
- Commercial Support, Hosting, and Consulting

MongoDB bridges the gap between key-value stores (which are fast and highly scalable) and traditional RDBMS systems (which provide structured schemas and powerful queries).

Quick Links

- Downloads
- Follow @mongodb on Twitter
- Production Deployments | Use Cases | Philosophy
- Drivers
- Source Code
- Blog | Articles
- Tutorial | Try MongoDB in the Browser
- Licensing | Example Snippets | BugDB (Jira) | International Docs

Support

- Support forums: mongodb-user | more...
- IRC: irc.freenode.net/#mongodb

Languages and Drivers

• C | C++ | C# | ColdFusion | Erlang | Factor | Java | Javascript | PHP | Python | Ruby | Perl | More...

Quickstart



For an even quicker start go to http://mongo.kylebanker.com/

- *NIX
 - Setup
 - OS X 32-bit
 - OS X 64-bit
 - Linux 32-bit
 - Linux 64-bit
 - Making Sure It Works
- Windows
 - Setup
 - 32-bit
 - 64-bit
 - Making Sure It Works
- See Also

Getting started with MongoDB is easy. For a longer description, please see Getting Started.

The following instructions assume a modern *NIX distribution.

If you are running an old version of Linux and the database doesn't start, try the "legacy static" version on the Downloads page.

Setup

OS X 32-bit

```
# make default directory for data
$ mkdir -p /data/db

# using curl, get the pre-built distro
$ curl -0 http://downloads.mongodb.org/osx/mongodb-osx-i386-latest.tgz

# unpack
$ tar xzf mongodb-osx-i386-latest.tgz
```

OS X 64-bit

```
# make default directory for data
$ mkdir -p /data/db

# using curl, get the pre-built distro
$ curl -O http://downloads.mongodb.org/osx/mongodb-osx-x86_64-latest.tgz

# unpack
$ tar xzf mongodb-osx-x86_64-latest.tgz
```

Linux 32-bit

```
# make default directory for data
$ mkdir -p /data/db

# using curl, get the pre-built distro
$ curl -O http://downloads.mongodb.org/linux/mongodb-linux-i686-latest.tgz

# unpack
$ tar xzf mongodb-linux-i686-latest.tgz
```

Linux 64-bit

```
# make default directory for data
$ mkdir -p /data/db

# using curl, get the pre-built distro
$ curl -0 http://downloads.mongodb.org/linux/mongodb-linux-x86_64-latest.tgz

# unpack
$ tar xzf mongodb-linux-x86_64-latest.tgz
```

Making Sure It Works

```
# run the database in the background - better would be to run in a separate terminal window if testing
./mongodb-xxxxxxx/bin/mongod &
# run the mongo shell. by default it connects to localhost:
./mongodb-xxxxxxx/bin/mongo
> db.foo.save( { a : 1 } )
> db.foo.findOne()
```

For more information about the shell, see the shell reference guide.

To get started in your language, check out a driver tutorial.

Windows

Setup

32-bit

Create a the folders data and db such that C: \data\db exists. This is the default location for database files.

Download and extract the 32-bit .zip.

64-bit

Create a the folders data and db such that C:\data\db exists. This is the default location for database files.

Download and extract the 64-bit .zip.

Making Sure It Works

Start a command line and run:

```
mongodb-xxxxxxx\bin\mongod.exe
```

Start another command line and run the Mongo shell (mongo.exe, not mongod.exe):

```
mongodb-xxxxxxx\bin\mongo.exe
> db.foo.save( { a : 1 } )
> db.foo.findOne()
```

For more information about the shell, see the shell reference guide.

To get started in your language, check out a driver tutorial.

See Also

• Starting and Stopping the Database

Downloads

MongoDB Downloads

Version	OS X 32 bit	OS X 64 bit	Linux 32 bit	Linux 64 bit	Windows 32 bit	Windows 64-bit	Solaris i86pc	Solaris 64	Source	Date	Char Log
Production (stable)											
1.2.2	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	1/27/2010	1.2.x Relea Notes chan log
nightly	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	Daily	log
Dev (unstable)											

1.3.1	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	1/20/2009	1.4 Relea Notes chan log
1.3.x nightly	os x 10.5+ os x 10.4	download	download * legacy-static	download * legacy-static	download	download	download	download	tgz zip	Daily	log
Archived Releases	list	list	list	list	list	list	list	list	list		

- See Version Numbers
- The linux legacy-static builds are only recommended for older systems. If you try to run and get a floating point exception, try the legacy-static builds. Otherwise you should use the regular ones.
- · Currently the mongod server must run on little-endian cpu (intel) so if you are using a ppc os x, mongod will not work.
- See http://buildbot.mongodb.org/waterfall for details of builds and completion times.

Included in Distributions

- The MongoDB database server
- The MongoDB shell
- Backup and restore tools
- Import and export tools
- GridFS tool
- The MongoDB C++ client

Drivers

Information on how to separately download or install the drivers and tools can be found on the Drivers page.

Language	Packages	Source	API Reference
Python	bundles	github	api
PHP	pecl	github	api
Ruby	gemcutter	github	api
Java	jar	github	api
Perl	cpan	github	api
C++	included in database	github	api

See Drivers for more information and other languages.

Source Code

Source code for MongoDB and all drivers

Packages

MongoDB is included in several different package managers:

- For MacPorts, see the **mongodb** and **mongodb-devel** packages.
- For FreeBSD, see the mongodb and mongodb-devel packages.
- For Homebrew, see the mongodb formula.
- For ArchLinux, see the mongodb package in the AUR.

Documentation

This site is available for download as HTML, PDF, or XML.

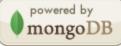
Logos

 ${\bf MongoDB\ logos\ are\ available\ for\ download\ as\ attachments\ on\ this\ page.}$

Powered By MongoDB Badges

We've made badges in beige, brown, blue and green for use on your sites that are powered by MongoDB. They are available below and in multiple sizes as attachments on this page.









1.0 Changelist

Wrote MongoDB. See documentation.

1.2.x Release Notes

New Features

- More indexes per collection
- Faster index creation
- Map/Reduce
- Stored JavaScript functions
- · Configurable fsync time
- · Several small features and fixes

DB Upgrade Required

There are some changes that will require doing an upgrade if your previous version is <= 1.0.x. If you're already using a version >= 1.1.x then these changes aren't required. There are 2 ways to do it:

- --upgrade
 - · stop your mongod process
 - run ./mongod --upgrade
 - start mongod again
- use a slave
 - start a slave on a different port and data directory
 - when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

Replication Changes

• There have been minor changes in replication. If you are upgrading a master/slave setup from <= 1.1.2 you have to update the slave first.

mongoimport

mongoimportjson has been removed and is replaced with mongoimport that can do json/csv/tsv

field filter changing

We've changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter
only changes the output, not which objects are returned. If you need that behavior, you can use \$exists

other notes

http://www.mongodb.org/display/DOCS/1.1+Development+Cycle

1.4 Release Notes

Focus

- Concurrency
- Features
- Sharding

Version Numbers

MongoDB uses the Odd-numbered versions for development releases.

There are 3 numbers in a MongoDB version: A.B.C

- · A is the major version. This will rarely change and signify very large changes
- B is the release number. This will include many changes including features and things that possible break backwards compatibility. Even Bs will be stable branches, and odd Bs will be development.
- C is the revision number and will be used for bugs and security issues.

For example:

- 1.0.0 : first GA release
- 1.0.x: bug fixes to 1.0.x highly recommended to upgrade, very little risk
- 1.1.x: development release. this will include new features that are not full finished, and works in progress. Some things may be different than 1.0
- 1.2.x : second GA release. this will be the culmintation of the 1.1.x release.

Getting Started

Getting started with MongoDB is easy:

- Download Mongo
- Install the Software
 - · Create a Directory for Data
 - Installation Layout
- Running Mongo
- Learn About MongoDB

Download Mongo

MongoDB is available either in pre-built distribution for Linux, OS X and Windows, or via source You can find the available pre-built distributions of MongoDB on the Downloads page.

Once you have the binary downloaded, either unzip or untar the distribution. For the purposes of this document, we'll call that the "mongo home directory".

Install the Software

Installation of MongoDB is easy. Once you've downloaded the software and unpacked the distribution into the MongoDB home directory, you'll need to create a directory for MongoDB to store its data files.

Create a Directory for Data

By default, MongoDB will store data in /data/db on Unix-like systems (e.g. Linux and OS X), and in c:\data\db in Windows. MongoDB will not create these directories, so please create them. They'll need to have read, write and directory creation permissions for Mongo to perform all of its usual operations. You can also specify a different directory with the --dbpath flag.

You may also choose to get one of the many drivers or other tools to work with MongoDB, although the included mongo shell is enough to start experimenting.

Installation Layout

Once installed, you should see the following general structure.

```
-- bin
   -- mongo
                                            (the database shell)
    |-- mongod
                                            (the core database server)
   |-- mongos
                                            (auto-sharding process)
    |-- mongodump
                                            (dump/export utility)
                                            (restore/import utility)
    `-- mongorestore
                                            (c++ driver include files)
 -- include
    `-- mongo
        |-- client
       |-- db
       |-- grid
        `-- util
-- lib
|-- lib64
```

Running Mongo

Running the database is as easy as starting the server. On Linux, and assuming you are in the MongoDB home directory, just type

```
$ bin/mongod
```

which will start the database. For further information on running MongoDB, please see Mongo Administration Guide

For command line help:

```
bin/mongod --help
```

To run the shell:

```
bin/mongo [--help]
```

Learn About MongoDB

Once you have MongoDB installed and running, review the tutorial.

Drivers

MongoDB currently has client support for the following programming languages:

mongodb.org Supported

- C
- C++
- Java
- Javascript
- Perl
- PHP
- Python
- Ruby

Community Supported

- C#
- http://github.com/samus/mongodb-csharp/tree/master
- Blog post: Using MongoDB from C Sharp
- Blog post
- ColdFusion
 - Blog post: Part 1 | Part 2 | Part 3
- Erlang
 - emongo An Erlang MongoDB driver that emphasizes speed and stability. "The most emo of drivers."
 - Erlmongo an almost complete MongoDB driver implementation in Erlang

- Factor
 - http://github.com/x6j8x/mongo-factor-driver/tree/master
- Fantom
 - http://bitbucket.org/liamstask/fantomongo/wiki/Home
- http://gist.github.com/218388
- Go
- gomongo
- Groovy
- See JVM Languages Haskell
 - http://hackage.haskell.org/package/mongoDB
- **JavaScript**
 - A CommonJS JavaScript wrapper of the Mongo Java library (http://github.com/mrclash/narwhal-mongodb)
- PHP
 - · Asynchronous PHP driver using libevent
- PowerShell
 - Blog post
- Ruby
 - MongoMapper
 - RMongo another Ruby driver for MongoDB
- Scala
 - See JVM Languages

Get Involved, Write a Driver!

Writing Drivers and Tools

Driver Syntax Table

The wiki generally gives examples in JavaScript, so this chart can be used to convert those examples to any language.

JavaScript	Python	PHP	Ruby
[]		array()	
{}	{}	MongoEmptyObj	{}
{x:1}	{"x": 1}	array('x' => 1)	{'x' => 1}
connect("www.example.net")	Connection("www.example.net")	new Mongo("www.example.net")	Mongo.new("www.example.
cursor.next()	cursor.next()	\$cursor->getNext()	cursor.next_object()
cursor.hasNext()	*	\$cursor->hasNext()	*
collection.findOne()	collection.find_one()	<pre>\$collection->findOne()</pre>	collection.find_one()
db.eval()	db.eval()	\$db->execute()	db.eval()

^{*} does not exist in that language

Javascript Language Center

MongoDB can be

- Used by clients written in Javascript;
- Uses Javascript internally server-side for certain options such as map/reduce;
- Has a shell that is based on Javascript for administrative purposes.

SpiderMonkey

The MongoDB shell extends SpiderMonkey. See the MongoDB shell documentation.

V8 and node.js

- http://github.com/orlandov/node-mongodb
- http://github.com/christkv/node-mongodb-native
- http://github.com/erh/mongo-v8-driver/tree/master

Server-Side Javascript

Javascript may be executed in the MongoDB server processes for various functions such as query enhancement and map/reduce processing. See Server-side Code Execution.

JVM Languages

There are many wrappers for the Java driver for other JVM Languages.

- Clojure
 - http://github.com/somnium/congomongo
- Groovy
 - Blog post: Groovy Tutorial for MongoDB
 - Blog post: MongoDB made more Groovy
- Scala
 - http://wiki.github.com/eltimn/scamongo/

Python Language Center

Error formatting macro: redirect: java.lang.NullPointerException

PHP Language Center

Installing the PHP Driver

*NIX

Run:

sudo pecl install mongo

See Installing the PHP Driver for configuration information and OS-specific installation instructions.

Windows

Download one of the binaries and place it on your extension path. For more information, see the Windows section of the installation docs.

Using the PHP Driver

To get started, see the Tutorial. Also check out the API Documentation.

GUIs

Opricot

Opricot is a hybrid GUI/CLI/Scripting web frontend implemented in PHP to manage your MongoDB servers and databases. Use as a point-and-click adventure for basic tasks, utilize scripting for automated processing or repetitive things.

Opricot combines the following components to create a fully featured administration tool:

- · An interactive console that allows you to either work with the database through the UI, or by using custom Javascript.
- A set of simple commands that wrap the Javascript driver, and provide an easy way to complete the most common tasks.
- Javascript driver for Mongo that works on the browser and talks with the AJAX interface.
- Simple server-side AJAX interface for communicating with the MongoDB server (currently available for PHP).



PHPMoAdmin

PHPMoAdmin is a MongoDB administration tool for PHP built on a stripped-down version of the Vork high-performance framework.

- Nothing to configure place the moadmin.php file anywhere on your web site and it just works!
- Fast AJAX-based XHTML 1.1 interface operates consistently in every browser!
- Self-contained in a single 75kb file!
- Works on any version of PHP5 with the MongoDB NoSQL database installed & running.
- Option to enable password-protection for one or more users; to activate protection, just add the username-password(s) to the array at the top of the file.
- E_STRICT PHP code is formatted to the Zend Framework coding standards + fully-documented in the phpDocumentor DocBlock standard.
- All textareas can be resized by dragging/stretching the lower-right corner.
- Free & open-source! Release under the GPLv3 FOSS license!

Library and Framework Tools

CakePHP

• MongoDB datasource for CakePHP.

Drupal

• MongoDB Integration - Views (query builder) backend, a watchdog implementation (logging), and field storage.

Kohana Framework

Mango at github
 An ActiveRecord-like library for PHP, for the Kohana PHP Framework.

Lithium

Lithium supports Mongo out-of-the-box.

· Tutorial on creating a blog backend.

Memcached

 MongoNode PHP script that replicates MongoDB objects to Memcached.

Symfony 2

- Symfony 2 Logger
 A centralized logger for Symfony applications. See the blog post.
- sfStoragePerformancePlugin This plugin contains some extra storage engines (MongoDB and Memcached) that are currently missing from the Symfony (>= 1.2) core.

Vork

 Vork, the high-performance enterprise framework for PHP natively supports MongoDB as either a primary datasource or used in conjunction with an RDBMS.

Zend Framework

- ZF Cache Backend
 A ZF Cache Backend for MongoDB. It support tags and auto-cleaning.
- There is a Zend_Nosql_Mongo component proposal.

Stand-Alone Tools

MapReduce API

A MapReduce abstraction layer. See the blog post.

MongoDB-MapReduce-PHP at github

Mongodloid

A nice library on top of the PHP driver that allows you to make more natural queries (query-query('a == 13 AND b >= 8 && c % 3 == 4');), abstracts away annoying \$-syntax, and provides getters and setters.

- Project Page
- Downloads
- Documentation

Morph

A high level PHP library for MongoDB. Morph comprises a suite of objects and object primitives that are designed to make working with MongoDB in PHP a breeze.

Morph at code.google.com

simplemongophp

Very simple layer for using data objects see blog post

• simplemongophp at github

Installing the PHP Driver

• PECL

- Windows Install
 - VC8 and VC9
 - VC6
 - Install
- Manual *NIX Install
- System Oddities
 - OS X
 - MAMP and XAMPP
 - Gentoo
- Optional Configuration
- Checking Your Installation

PECL

You can download and install the driver using PECL by running:

```
$ pecl install mongo
```

Add the following line to php.ini:

```
extension=mongo.so
```

That's it, you're done!

Windows Install

The precompiled driver is available for the following setups:

VC8 and VC9

Builds for each release are available on Github. They are all build for PHP 5.3, as PHP 5.2 is no longer supported with VC9.

Binaries for the latest code are also available:

- VC8 Thread-Safe
- VC8 Non-Thread-Safe
- VC9 Thread-Safe
- VC9 Non-Thread-Safe

VC6

The latest binaries for 5.2 and 5.3 are available at php.net.

Please inquire on the mailing list if you need a different build.

Install

- 1. Download and extract the .zip file. Make sure PHP version_ matches the version of PHP you are running (e.g., if you are running PHP 5.2.6, download the driver for PHP 5.2).
- 2. Copy php_mongo.dll to your PHP extensions directory (see "extension_dir" in php.ini).
- 3. Add a line to php.ini:

```
extension=php_mongo.dll
```

Manual *NIX Install

Go to Github and click the "download" button. Then run:

```
$ tar zxvf mongodb-mongodb-php-driver-<commit_id>.tar.gz
$ cd mongodb-mongodb-php-driver-<commit_id>
$ phpize
$ ./configure
$ sudo make install
```

Make the following changes to php.ini:

1. The build will display where it is installing the PHP driver with output that looks something like:

```
Installing '/usr/lib/php/extensions/no-debug-zts-20060613/mongo.so'
```

Make sure that it is the same as the PHP extension directory by runnning:

```
$ php -i | grep extension_dir
  extension_dir => /usr/lib/php/extensions/no-debug-zts-20060613 =>
/usr/lib/php/extensions/no-debug-zts-20060613
```

If it's not, change the extension_dir in php.ini or move mongo.so.

2. To load the extension on PHP startup, add a line: Github

```
extension=mongo.so
```

System Oddities

os x

If your system is unable to find autoconf, you'll need to install Xcode (available on your installation DVD).



MAMP and XAMPP

At the moment, MAMP uses PHP 5.2 and XAMPP uses PHP 5.3. Make sure you download the driver from the correct section below, or PHP will complain about the version mismatch.

MAMP and XAMPP

MAMP and XAMPP cannot compile the driver, as they are missing some PHP header files. You can download a precompiled .so of the PHP 5.3 thread safe build from Github.

MAMP Install

Put mongo.so in MAMP's PHP extension directory and add a line to the php.ini used by MAMP that says: extension=mongo.so

XAMPP Install

- Download and extract the precompiled extension.
- Move mongo so to the XAMPP PHP extensions directory (by default, /Applications/XAMPP/xamppfiles/lib/php/php-5.3.0/extensions/no-debug-non-zts-20090626)
- Open up the php.ini file used by XAMPP in a text editor (by default, the php.ini file is in /Applications/XAMPP/xamppfiles/etc/php.ini)
- Add a line (anywhere in the file) that says extension=mongo.so
- Restart (or start) the XAMPP server
- Point your browser at localhost and select your language, then phpinfo() from the left menu
- Make sure there's a Mongo section

You should be all set to use Mongo from XAMPP!

Gentoo

On Gentoo using PECL you might get an error that libtool is the wrong version. Compiling from source you'll need to run aclocal and autoconf.

```
phpize && aclocal && autoconf && ./configure && make && make install
```

(Thanks to @riklaunim)

Optional Configuration

There are also a number of options that can be configured by adding lines to php.ini:

```
[mongo]
; If the driver should reconnect to mongo
mongo.auto_reconnect = true
; Whether to allow persistent connections
mongo.allow_persistent = On
; Maximum number of persistent connections (-1 means unlimited)
mongo.max\_persistent = -1
; Maximum number of links (persistent and non-persistent, -1 means unlimited)
mongo.max\_connections = -1
; Default host for mongo connection
mongo.default_host = www.example.com
; Default port for mongo database
mongo.default_port = 42
; When saving files to the database, size of chunks to split them into
mongo.chunk\_size = 1024
; Specify an alternate character to $ to use for special db functions ($set, $push, $exists, etc.)
mongo.cmd = "$"
```

If you do not add any of the above lines, the defaults are:

```
mongo.auto_reconnect = Off
mongo.allow_persistent = On
mongo.max_persistent = -1
mongo.max_connections = -1
mongo.default_host = localhost
mongo.default_port = 27017
mongo.chunk_size = 262400
```

There are some utility PHP scripts at Github. These can be added to your include path if you would like to use them. They include:

- 1. Auth.php: includes the MongoAuth class, for authenticating database connections
- 2. Admin.php: MongoAdmin class, for authenticated connections to the admin database
- 3. Util.php: MongoUtil, which contains a number of helpful constants.

Checking Your Installation

To check that the extension is working, restart Apache and create a PHP page containing:

```
<?php
phpinfo();
?>
```

Visit the page, scroll down, and you should see a "mongo" section that looks similar to:

mongo

MongoDB Support	enabled		
Version	1.0.3		
Active Persistent Connections	0		
Active Connections	0		

Directive	Local Value	Master Value		
mongo.allow_persistent	On	On		
mongo.auto_reconnect	On	On		
mongo.chunk_size	262144	262144		
mongo.cmd	\$	\$		
mongo.default_host	localhost	localhost		
mongo.default_port	27017	27017		
mongo.max_connections	Unlimited	Unlimited		
mongo.max_persistent	Unlimited	Unlimited		

Your driver should now be fully armed and operational! See the PHP Tutorial to start using it.

PHP - Storing Files and Big Data



This page is no longer actively maintained. See the PHP manual for the most up-to-date documentation.

The PHP driver can be used to easily store and retrieve files.

About GridFS

GridFS is a storage specification all supported drivers implement. Basically, it defines two collections:

files - file metadata

chunks - file content. If the file is large, it will automatically be split into smaller chunks and each chunk will be saved as a document in the chunks collection.

Each document in the files collection contains the filename, upload date, md5 hash, etc. It also contains a unique _id, which can be used to query the chunks collection for the file's content. Each document in the chunks collection contains a chunk of binary data, a files_id field that matches its file's _id, and the position of this chunk in the overall file.

An example

The files document:

```
array("_id" => 123456789, "filename" => "foo.txt", "chunkSize" => 3, "length" => 12)
```

The chunks documents:

```
array("files_id" => 123456789, "n" => 0, "data" => new MongoBinData("abc"));
array("files_id" => 123456789, "n" => 1, "data" => new MongoBinData("def"));
array("files_id" => 123456789, "n" => 2, "data" => new MongoBinData("ghi"));
array("files_id" => 123456789, "n" => 3, "data" => new MongoBinData("jkl"));
```

Of course, the default chunk size is thousands of bytes, but that makes an unweildy example.

For more information, see the spec.

The MongoGridFS Family

MongoGridFS represents the files and chunks collections. MongoGridFS extends MongoCollection, and an instance of MongoGridFS has access to all MongoCollection methods, which act on the files collection:

```
$grid = $db->getGridFS();
$grid->update(array("filename" => "foo"), $newObj)); // update on the files collection
```

You can also access the chunks collection from an instance of MongoGridFS:

```
$chunks = $grid->chunks; // $chunks is a normal MongoCollection
$chunks->insert(array("x" => 4));
```

There are some methods for MongoGridFS with the same name as MongoCollection methods, that behave slightly differently. For example, MongoGridFS::remove(\$criteria) will remove any objects that match the criteria from the files collection and their content from the chunks collection.

To store something new in GridFS, there are a couple options. If you have a filename, you can say:

```
$grid->storeFile($filename, array("whatever" => "metadata", "you" => "want"));
```

If you have a string of bytes that isn't a file, you can also store that using MongoGridFS::storeBytes() (version 0.9.4+):

```
$grid->storeBytes($bytes, array("whatever" => "metadata", "you" => "want");
```

Querying a MongoGridFS collection returns a MongoGridFSCursor, which behaves like a normal MongoCursor except that it returns MongoGridFSFiles instead of associative arrays.

MongoGridFSFiles can be written back to disc using \$file->write() or retrieved in memory using \$file->getBytes(). There is currently no method that automatically streams chunks, but it would be fairly easy to write by querying the \$grid->chunks collection.

 $MongoGridFSFile\ objects\ contain\ a\ field\ {\tt file}\ which\ contains\ any\ file\ metadata.$

Troubleshooting the PHP Driver

- I made all of the configuration changes to php.ini, but nothing has any effect.
- When I load a page, I get an error message that says it can't connect to the server.
- I get a blank page when I try to use any Mongo stuff / I get an error message that PHP can't find mongo_connect / I checked the Apache error logs and saw that a module is failing to load when Apache starts up.

I made all of the configuration changes to ${\tt php.ini}$, but nothing has any effect.

Apache may be using a different php.ini file than the one you edited. Create and load a page containing:

```
<?php
phpinfo();
?>
```

to determine which php.ini file Apache is using.

When I load a page, I get an error message that says it can't connect to the server.

If you get something like:

```
Warning: couldn't connect to server localhost:27017 127.0.0.1:27017 in Mongo::__construct on line 21
```

This means that it couldn't connect to the database. Are you sure you started Mongo up? Is it running locally on the default port?

I get a blank page when I try to use any Mongo stuff / I get an error message that PHP can't find mongo_connect / I checked the Apache error logs and saw that a module is failing to load when Apache starts up.

Look at your Apache error log. If you see something like:

```
PHP Warning: PHP Startup: mongo: Unable to initialize module
Module compiled with module API=20060613, debug=0, thread-safety=0
PHP compiled with module API=20060613, debug=1, thread-safety=1
These options need to match
in Unknown on line 0
```

This means that the PHP used by your Apache was compiled using different options than was used when you compiled the driver. You will need to change some PHP settings and then recompile the driver. Find your PHP include files (/usr/local/include/php, by default) and modify main/php_config.h as follows:

- Set ZEND_DEBUG to whatever the value is in line 3 of the warning.
- If and only if thread-safety=1 in line 3 of the warning, add a line defining ZTS (Zend Thread Safe):

```
#define ZTS 1
```

If thread-safety=0, remove any lines defining ZTS, even if they define it as 0.

Now, go to the driver directory and type:

```
$ scons -c # scons equivalent of "make clean"
$ sudo scons install
```

Ruby Language Center

This is an overview of the available tools and suggested practices for using Ruby with MongoDB. Those wishing to skip to more detailed discussion should check out the Ruby Driver Tutorial, Rails - Getting Started, and MongoDB Data Modeling and Rails. There are also a number of good external resources worth checking out.

- Ruby Driver
 - Installing / Upgrading
 - C Extension
- Object MappersNotable Projects

Ruby Driver



Install the C extension for any performance-critical applications.

The MongoDB Ruby driver is the 10gen-supported driver for MongoDB. It's written in pure Ruby, with a recommended C extension for speed. The driver is optimized for simplicity. It can be used on its own, but it also serves as the basis for various object-mapping libraries.

- Tutorial
- API Documentation
- Source Code

Installing / Upgrading

The ruby driver uses Gemcutter for gem hosting. Before installing the driver, make sure you've installed Gemcutter:

```
$ gem install gemcutter
$ gem tumble
```

Once you've added the Gemcutter repository, you can install or upgrade like so:

```
$ gem install mongo
```

To stay on the bleeding edge, check out the latest source from github:

```
$ git clone git://github.com/mongodb/mongo-ruby-driver.git
$ cd mongo-ruby-driver/
```

Then, install the driver from there:

```
$ rake gem:install
```

C Extension

There's a separate gem containing optional, but recommended, C extensions that significantly increase driver performance. To use the extensions just install the **mongo_ext** gem:

```
$ gem install mongo_ext
```

Or, to install from source:

```
$ rake gem:install_extensions
```

As long it's in Ruby's load path, mongo_ext will be loaded automatically when you require mongo. To learn more about the Ruby driver, see the Ruby Tutorial.

Object Mappers

If you need validations, associations, and other high-level data modeling functions, consider using one of the available object mappers. Many of these exist in the Ruby ecosystem; here we host a list of the most popular ones.

Notable Projects

Tools for working with MongoDB in Ruby are being developed daily. A partial list can eb found in the Projects and Libraries section of our external resources page.

If you're working on a project that you'd like to have included, let us know.

Ruby Tutorial

This tutorial gives common examples of with MongoDB using the Ruby driver. If you're looking for information on data modeling, see MongoDB Data Modeling and Rails. Links to the various object mappers are listed on our object mappers page.

The latest source for the Ruby driver can be found on github.

- Installation
- A Quick Tour
 - Using the RubyGem
 - Making a Connection
 - Listing All Databases
 - Dropping a Database
 - Authentication (Optional)
 - Getting a List Of Collections
 - Getting a Collection
 - Inserting a Document
 - Finding the First Document In a Collection using find_one()
 - Adding Multiple Documents
 - Counting Documents in a Collection
 - Using a Cursor to get all of the Documents
 - Getting a Single Document with a Query
 - · Getting a Set of Documents With a Query
 - Querying with Regular Expressions
 - Creating An Index
 - Getting a List of Indexes on a Collection
 - Database Administration
- See Also

Installation

The mongo-ruby-driver gem is served through Gemcutter. If you haven't installed gemcutter:

```
gem install gemcutter
gem tumble
```

Next, install the mongo and mongo_ext rubygems:

```
gem install mongo
gem install mongo_ext
```

After installing, you may want to look at the examples directory included in the source distribution. These examples walk through some of the basics of using the Ruby driver.

The full API documentation can be viewed here.

A Quick Tour

Using the RubyGem

All of the code here assumes that you have already executed the following Ruby code:

```
require 'rubygems' # not necessary for Ruby 1.9
require 'mongo'
```

Making a Connection

An Mongo::Connection instance represents a connection to MongoDB. You use a Connection instance to obtain an Mongo:DB instance, which represents a named database. The database doesn't have to exist - if it doesn't, MongoDB will create it for you.

You can optionally specify the MongoDB server address and port when connecting. The following example shows three ways to connect to the database "mydb" on the local machine:

```
db = Mongo::Connection.new.db("mydb")
db = Mongo::Connection.new("localhost").db("mydb")
db = Mongo::Connection.new("localhost", 27017).db("mydb")
```

At this point, the db object will be a connection to a MongoDB server for the specified database. Each DB instance uses a separate socket connection to the server.

If you're trying to connect to a replica pair, see Replica Pairs in Ruby.

Listing All Databases

```
m = Mongo::Connection.new # (optional host/port args)
m.database_names.each { |name | puts name }
m.database_info.each { |info| puts info.inspect}
```

Dropping a Database

```
m.drop_database('database_name')
```

Authentication (Optional)

MongoDB can be run in a secure mode where access to databases is controlled through name and password authentication. When run in this mode, any client application must provide a name and password before doing any operations. In the Ruby driver, you simply do the following with the connected mongo object:

```
auth = db.authenticate(my_user_name, my_password)
```

If the name and password are valid for the database, auth will be true. Otherwise, it will be false. You should look at the MongoDB log for further information if available.

Getting a List Of Collections

Each database has zero or more collections. You can retrieve a list of them from the db (and print out any that are there):

```
db.collection_names.each { |name| puts name }
```

and assuming that there are two collections, name and address, in the database, you would see

```
name
address
```

as the output.

Getting a Collection

You can get a collection to use using the collection method:

```
coll = db.collection("testCollection")
```

This is aliased to the [] method:

```
coll = db["testCollection"]
```

Once you have this collection object, you can now do things like insert data, query for data, etc.

Inserting a Document

Once you have the collection object, you can insert documents into the collection. For example, lets make a little document that in JSON would be represented as

Notice that the above has an "inner" document embedded within it. To do this, we can use a Hash or the driver's OrderedHash (which preserves key order) to create the document (including the inner document), and then just simply insert it into the collection using the insert() method.

Finding the First Document In a Collection using find_one()

To show that the document we inserted in the previous step is there, we can do a simple find_one() operation to get the first document in the collection. This method returns a single document (rather than the Cursor that the find() operation returns), and it's useful for things where there only is one document, or you are only interested in the first. You don't have to deal with the cursor.

```
my_doc = coll.find_one()
puts my_doc.inspect
```

and you should see:

```
{"_id"=>#<Mongo::ObjectID:0x118576c ...>, "name"=>"MongoDB", "info"=>{"x"=>203, "y"=>102}, "type"=>
"database", "count"=>1}
```

Note the <u>_id</u> element has been added automatically by MongoDB to your document. Remember, MongoDB reserves element names that start with <u>_</u> for internal use.

Adding Multiple Documents

In order to do more interesting things with queries, let's add multiple simple documents to the collection. These documents will just be

```
{
    "i" : value
}
```

and we can do this fairly easily:

```
100.times { |i| coll.insert("i" => i) }
```

Notice that we can insert documents of different "shapes" into the same collection. These records are in the same collection as the complex record we inserted above. This aspect is what we mean when we say that MongoDB is "schema-free".

Counting Documents in a Collection

Now that we've inserted 101 documents (the 100 we did in the loop, plus the first one), we can check to see if we have them all using the count () method.

```
puts coll.count()
```

and it should print 101.

Using a Cursor to get all of the Documents

In order to get all the documents in the collection, we will use the find() method. The find() method returns a Cursor object which allows us to iterate over the set of documents that matched our query. The Ruby driver's Cursor is enumerable. So to query all of the documents and print them out:

```
coll.find().each { |row| puts row.inspect }
```

and that should print all 101 documents in the collection.

Getting a Single Document with a Query

We can create a *query* hash to pass to the find() method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the "i" field is 71, we would do the following;

```
coll.find("i" => 71).each { |row| puts row.inspect }
```

and it should just print just one document:

```
{"_id"=>#<Mongo::ObjectID:0x117de90 ...>, "i"=>71}
```

Getting a Set of Documents With a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write:

```
coll.find("i" => {"$gt" => 50}).each { |row| puts row }
```

which should print the documents where i > 50. We could also get a range, say 20 < i <= 30:

```
coll.find("i" => {"$gt" => 20, "$lte" => 30}).each { |row| puts row }
```

Querying with Regular Expressions

Regular expressions can be used to query MongoDB. To find all names that begin with 'a':

```
coll.find({"name" => /^a/})
```

You can also construct a regular expression dynamically. To match a given search string:

```
search_string = params['search']

# Constructor syntax
coll.find({"name" => Regexp.new(search_string)})

# Literal syntax
coll.find({"name" => /#{search_string}/})
```

Although MongoDB isn't vulnerable to anything like SQL-injection, it may be worth checking the search string for anything malicious.

Creating An Index

MongoDB supports indexes, and they are very easy to add on a collection. To create an index, you specify an index name and an array of field names to be indexed, or a single field name. The following creates an ascending index on the "i" field:

```
# create_index assumes ascending order; see method docs
# for details
coll.create_index("i")
```

To specify complex indexes or a descending index you need to use a slightly more complex syntax - the index specifier must be an Array of field name, direction pairs. Directions should be specified as Mongo::ASCENDING or Mongo::DESCENDING:

```
# explicit "ascending"
coll.create_index([["i", ASCENDING]])
```

Getting a List of Indexes on a Collection

You can get a list of the indexes on a collection using coll.index_information().

Database Administration

A DB object class has an admin method that returns an object that can perform administrative functions.

```
admin = db.admin
```

A database can have one of three profiling levels: off (:off), slow queries only (:slow_only), or all (:all). To see the database level:

```
puts admin.profiling_level # => off (the symbol :off printed as a string)
admin.profiling_level = :slow_only
```

Validating a collection will return an interesting hash if all is well or raise an exception if there is a problem.

```
p admin.validate_collection('coll_name')
```

See Also

- Ruby Driver RDoc
- MongoDB Manual

Replica Pairs in Ruby

Here follow a few considerations for those using the Ruby driver with MongoDB and replica pairing.

- Setup
- Connection Failures
- Recovery
- Testing
- Further Reading

Setup

First, make sure that you've correctly paired two mongod instances. If you want to do this on the same machine for testing, make sure you've created two data directories. The init commands are as follows:

```
./mongod --pairwith localhost:27018 --dbpath /data/left --port 27017
./mongod --pairwith localhost:27017 --dbpath /data/right --port 27018
```

When you instantiate a Ruby connection, you'll have to make sure that the driver knows about both instances:

Connection Failures

Imagine that our master node goes offline. How will the driver respond?

At first, the driver will try to send operations to what was the master node. These operations will fail, and the driver will raise a **ConnectionFailure** exception. It then becomes the client's responsibility to decide how to handle this.

If the client decides to retry, it's not guaranteed that the former slave will have been promoted to master yet, so it's still possible that the driver will raise another **ConnectionFailure**. However, once the former slave has become master, typically within a few seconds, subsequent operations will succeed.

Recovery

Driver users may wish to wrap their database calls with failure recovery code. Here's one possibility:

```
# Ensure retry upon failure
def rescue_connection_failure(max_retries=5)
    success = false
   retries = 0
    while !success
     begin
        yield
        success = true
      rescue Mongo::ConnectionFailure => ex
        retries += 1
        raise ex if retries >= max_retries
        sleep(1)
      end
    end
  end
end
# Wrapping a call to #count()
rescue_connection_failure do
  @db.collection('users').count()
```

Of course, the proper way to handle connection failures will always depend on the individual application. We encourage object-mapper and application developers to publish any promising results.

Testing

The Ruby driver (>= 0.17.2) includes some unit tests for verifying proper replica pair behavior. They reside in **tests/replica**. You can run them individually with the following rake tasks:

```
rake test:pair_count
rake test:pair_insert
rake test:pair_query
```

Make sure you have a replica pair running locally before trying to run these tests.

Further Reading

- Replica Pairs
- Pairing Internals

Rails - Getting Started

This tutorial describes how to set up a simple Rails application with MongoDB, using MongoMapper as an object mapper. We assume you're using Rails 2.3 or greater.

- Configuration
- Testing
- Coding

Using a Rails Template

All of the configuration steps listed below, and more, are encapsulated in this Rails template (based on a similar one by Ben Scofield). You can create your project with the template as follows:

```
rails project_name -m "http://gist.github.com/gists/219223.txt"
```

Be sure to replace **project_name** with the name of your project.

If you want to set up your project manually, read on.

Configuration

1. We need to tell MongoMapper which database we'll be using. Save the following to config/initializers/database.rb:

```
MongoMapper.database = "#{db_name}-#{Rails.env}"
```

Replace **db_name** with whatever name you want to give the database. The **Rails.env** variable will ensure that a different database is used for each environment.

2. If you're using Passenger, add this code to config/initializers/database.rb.

```
if defined?(PhusionPassenger)
PhusionPassenger.on_event(:starting_worker_process) do |forked|
MongoMapper.connection.connect_to_master if forked
end
end
```

- 3. Clean out config/database.yml. This file should be blank, as we're not connecting to the database in the traditional way.
- 4. Remove ActiveRecord from environment.rb.

```
config.frameworks -= [:active_record]
```

5. Add MongoMapper to the environment. This can be done by opening config/environment.rb and adding the line:

```
config.gem 'mongo_mapper'
```

Once you've done this, you can install the gem in the project by running:

```
rake gems:install
rake gems:unpack
```

Testing

It's important to keep in mind that with MongoDB, we cannot wrap test cases in transactions. One possible work-around is to invoke a **teardown** method after each test case to clear out the database.

To automate this, I've found it effective to modify ActiveSupport::TestCase with the code below.

```
# Drop all columns after each test case.

def teardown

MongoMapper.database.collections.each do \|coll\|

coll.remove
end
end

# Make sure that each test case has a teardown
# method to clear the db after each test.

def inherited(base)

base.define_method teardown do

super
end
end
```

This way, all test classes will automatically invoke the teardown method. In the example above, the teardown method clears each collection. We might also choose to drop each collection or drop the database as a whole, but this would be considerably more expensive and is only necessary if our tests manipulate indexes.

Usually, this code is added in test/test_helper.rb. See the aforementioned rails template for specifics.

Coding

If you've followed the foregoing steps (or if you've created your Rails with the provided template), then you're ready to start coding. For help on that, you can read about modeling your domain in Rails.

MongoDB Data Modeling and Rails

This tutorial discusses the development of a web application on Rails and MongoDB. MongoMapper will serve as our object mapper. The goal is to provide some insight into the design choices required for building on MongoDB. To that end, we'll be constructing a simple but non-trivial social news application. The source code for **newsmonger** is available on github for those wishing to dive right in.

- Modeling Stories
 - Caching to Avoid N+1
 - A Note on Denormalization
 - Fields as arrays
 - Atomic Updates
- Modeling Comments
 - · Linear, Embedded Comments
 - Nested, Embedded Comments
 - Comment collections
- Unfinished business

Assuming you've configured your application to work with MongoMapper, let's start thinking about the data model.

Modeling Stories

A news application relies on stories at its core, so we'll start with a Story model:

```
class Story
  include MongoMapper::Document
 key :title,
                 String
 key :url,
                 String
 key :slug,
                 String
 key :voters, Array
                Integer, :default => 0
 key :votes,
  key :relevance, Integer, :default => 0
  # Cached values.
  key :comment_count, Integer, :default => 0
  kev :username,
                 String
  # Note this: ids are of class ObjectId.
 key :user_id, ObjectId
  timestamps!
  # Relationships.
 belongs to :user
  # Validations.
  validates_presence_of :title, :url, :user_id
```

Obviously, a story needs a title, url, and user_id, and should belong to a user. These are self-explanatory.

Caching to Avoid N+1

When we display our list of stories, we'll need to show the name of the user who posted the story. If we were using a relational database, we could perform a join on users and stores, and get all our objects in a single query. But MongoDB does not support joins and so, at times, requires bit of denormalization. Here, this means caching the 'username' attribute.

A Note on Denormalization

Relational purists may be feeling uneasy already, as if we were violating some universal law. But let's bear in mind that MongoDB collections are not equivalent to relational tables; each serves a unique design objective. A normalized table provides an atomic, isolated chunk of data. A document, however, more closely represents an object as a whole. In the case of a social news site, it can be argued that a username is intrinsic to the story being posted.

What about updates to the username? It's true that such updates will be expensive; happily, in this case, they'll be rare. The read savings achieved in denormalizing will surely outweigh the costs of the occasional update. Alas, this is not hard and fast rule: ultimately, developers must evaluate their applications for the appropriate level of normalization.

Fields as arrays

With a relational database, even trivial relationships are blown out into multiple tables. Consider the votes a story receives. We need a way of recording which users have voted on which stories. The standard way of handling this would involve creating a table, 'votes', with each row referencing user_id and story_id.

With a document database, it makes more sense to store those votes as an array of user ids, as we do here with the 'voters' key.

For fast lookups, we can create an index on this field. In the MongoDB shell:

```
db.stories.ensureIndex('voters');
```

Or, using MongoMapper, we can specify the index in config/initializers/database.rb:

```
Story.ensure_index(['voters', 1])
MongoMapper.ensure_indexes!
```

To find all the stories voted on by a given user:

```
Story.all(:conditions => {:voters => @user.id})
```

Atomic Updates

Storing the voters array in the Story class also allows us to take advantage of atomic updates. What this means here is that, when a user votes on a story, we can

- 1. ensure that the voter hasn't voted yet, and, if not,
- 2. increment the number of votes and
- 3. add the new voter to the array.

MongoDB's query and update features allows us to perform all three actions in a single operation. Here's what that would look like from the shell:

```
// Assume that story_id and user_id represent real story and user ids.
db.stories.update({_id: story_id, voters: {'$ne': user_id}},
{'$inc': {votes: 1}, '$push': {voters: user_id}});
```

What this says is "get me a story with the given id whose voters array does not contain the given user id and, if you find such a story, perform two atomic updates: first, increment votes by 1 and then push the user id onto the voters array."

This operation highly efficient; it's also reliable. The one caveat is that, because update operations are "fire and forget," you won't get a response from the server. But in most cases, this should be a non-issue.

A MongoMapper implementation of the same update would look like this:

Modeling Comments

In a relational database, comments are usually given their own table, related by foreign key to some parent table. This approach is occasionally necessary in MongoDB; however, it's always best to try to embed first, as this will achieve greater query efficiency.

Linear, Embedded Comments

Linear, non-threaded comments should be embedded. Here are the most basic MongoMapper classes to implement such a structure:

```
class Story
include MongoMapper::Document
many :comments
end
```

```
class Comment
include MongoMapper::EmbeddedDocument
key :body, String
belongs_to :story
end
```

If we were using the Ruby driver alone, we could save our structure like so:

Essentially, comments are represented as an array of objects within a story document. This simple structure should be used for any one-to-many relationship where the many items are linear.

Nested, Embedded Comments

But what if we're building threaded comments? An admittedly more complicated problem, two solutions will be presented here. The first is to represent the tree structure in the nesting of the comments themselves. This might be achieved using the Ruby driver as follows:

Representing this structure using MongoMapper would be tricky, requiring a number of custom mods.

But this structure has a number of benefits. The nesting is captured in the document itself (this is, in fact, how Business Insider represents comments). And this schema is highly performant, since we can get the story, and all of its comments, in a single query, with no application-side processing for constructing the tree.

One drawback is that alternative views of the comment tree require some significant reorganizing.

Comment collections

We can also represent comments as their own collection. Relative to the other options, this incurs a small performance penalty while granting us the greatest flexibility. The tree structure can be represented by storing the unique path for each leaf (see Mathias's original post on the idea). Here are the relevant sections of this model:

```
include MongoMapper::Document
key :body,
             String
key :path,
             String, :default => ""
# Note: we're intentionally storing parent_id as a string
key :parent_id, String
key :story_id, ObjectId
timestamps!
# Relationships.
belongs_to :story
# Callbacks.
after_create :set_path
private
# Store the comment's path.
def set path
 unless self.parent_id.blank?
   parent = Comment.find(self.parent_id)
   self.story_id = parent.story_id
   self.depth = parent.depth + 1
             = parent.path + ":" + parent.id
   self.path
 save
```

The path ends up being a string of object ids. This makes it easier to display our comments nested, with each level in order of karma or votes. If we specify an index on story_id, path, and votes, the database can handle half the work of getting our comments in nested, sorted order.

The rest of the work can be accomplished with a couple grouping methods, which can be found in the newsmonger source code.

It goes without saying that modeling comments in their own collection also facilitates various site-wide aggregations, including displaying the latest, grouping by user, etc.

Unfinished business

Document-oriented data modeling is still young. The fact is, many more applications will need to be built on the document model before we can say anything definitive about best practices. So the foregoing should be taken as suggestions, only. As you discover new patterns, we encourage you to document them, and feel free to let us know about what works (and what doesn't).

Developers working on object mappers and the like are encouraged to implement the best document patterns in their code, and to be wary of recreating relational database models in their apps.

Object Mappers for Ruby and MongoDB

Although it's possible to use the Ruby driver by itself, sometimes you want validations, associations, and many of the other conveniences provided by ActiveRecord. Here, then, is a list of the most popular object mappers available for working with Ruby and MongoDB.

- MongoMapper
- Mongoid
- MongoRecord
- MongoDoc

MongoMapper

John Nunemaker's OM. Production-ready and actively-developed.

Installation:

· gem install mongo_mapper

Source:

mongo_mapper on github

Documentation:

- MongoMapper on google groups
- #mongomapper on freenode.

Articles:

- Getting Started with MongoMapper
- MongoMapper and Rails
- More MongoMapper Awesomeness.

Mongoid

Durran Jordan's OM. Production-ready and actively-developed.

Installation:

· gem install mongoid

Source:

· mongoid on github

Documentation:

· Docs at mongoid.org

MongoRecord

10gen's original OM.

Notes:

MongoRecord is an ActiveRecord-like OM, and the first of its kind developed for MongoDB. Favored by a contingent of developers for its simplicity, MongoRecord currently receives a lot of love from Nate Wiger.

Installation:

• gem install mongo_record

Source:

• mongo-record on github

MongoDoc

Les Hill's OM. Currently alpha but actively-developed.

Notes:

MongoDoc is optimized for speed and simplicity. Originally built as a BSON library that worked like Ruby's JSON gem, MongoDoc has grown into its own OM.

Installation:

· gem install mongodoc

Source:

• mongodoc on github

Ruby External Resources

There are a number of good resources appearing all over the web for learning about MongoDB and Ruby. A useful selection is listed below. If you know of others, do let us know.

- Screencasts
- Presentations
- Articles
- Projects
- Libraries

Screencasts

Introduction to MongoDB - Part I

An introduction to MongoDB via the MongoDB shell.

Introduction to MongoDB - Part II

In this screencast, Joon You teaches how to use the Ruby driver to build a simple Sinatra app.

Introduction to MongoDB - Part III

For the final screencast in the series, Joon You introduces MongoMapper and Rails.

RailsCasts: MongoDB & MongoMapper

Ryan Bates' RailsCast introducing MongoDB and MongoMapper.

Presentations

Introduction to MongoDB (Video)

Mike Dirolf's introduction to MongoDB at Pivotal Labs, SF.

MongoDB: A Ruby Document Store that doesn't rhyme with 'Ouch' (Slides)

Wynn Netherland's introduction to MongoDB with some comparisons to CouchDB.

MongoDB (is) for Rubyists (Slides)

Kyle Banker's presentation on why MongoDB is for Rubyists (and all human-oriented programmers).

Introduction to Mongoid and MongoDB (Video)

Durran Jordan discusses Mongoid, MongoDB, and how HashRocket uses these tools in production.

Articles

Why I Think Mongo is to Databases What Rails was to Frameworks

What if a key-value store mated with a relational database system?

John Nunemaker's articles on MongoDB.

A series of articles on aggregation with MongoDB and Ruby:

- 1. Part I: Introduction of Aggregation in MongoDB
- 2. Part II: MongoDB Grouping Elaborated
- 3. Part III: Introduction to Map-Reduce in MongoDB

Projects

Mongo Admin

A Rails plugin for browsing and managing MongoDB data. See the live demo.

Sinatra Resource

Resource Oriented Architecture (REST) for Sinatra and MongoMapper.

Shorty

A URL-shortener written with Sinatra and the MongoDB Ruby driver.

NewsMonger

A simple social news application demonstrating MongoMapper and Rails.

Data Catalog API

From Sunlight Labs, a non-trivial application using MongoMapper and Sinatra.

Watchtower

An example application using Mustache, MongoDB, and Sinatra.

Shapado

A question and answer site similar to Stack Overflow. Live version at shapado.com.

Libraries

ActsAsTree (MongoMapper)

ActsAsTree implementation for MongoMapper.

Machinist adapter (MongoMapper)

Machinist adapter using MongoMapper.

Mongo-Delegate

A delegation library for experimenting with production data without altering it. A quite useful pattern.

Remarkable Matchers (MongoMapper)

Testing / Matchers library using MongoMapper.

OpenIdAuthentication, supporting MongoDB as the datastore

Brandon Keepers' fork of OpenIdAuthentication supporting MongoDB.

MongoTree (MongoRecord)

MongoTree adds parent / child relationships to MongoRecord.

Merb MongoMapper

a plugin for the Merb framework for supporting MongoMapper models.

Mongolytics (MongoMapper)

A web analytics tool.

Rack-GridFS

A Rack middleware component that creates HTTP endpoints for files stored in GridFS.

Java Language Center

- Tutorial
- API Documentation
- Downloads

Specific Topics

- Concurrency
- Saving Objects
- Data Types

3rd Party Tools

- log4j appender
- pojo to MongoDB

Wrappers for other JVM Languages

JVM Languages

If there is a project missing here, just add a comment or email the list and we'll add it.

Java Driver Concurrency

The Java MongoDB driver is thread safe. If you are using in a web serving environment, for example, you should create a single Mongo instance, and you can use it in every request.

However, if you want to ensure complete consistency in a "session" (maybe an http request), you probably want the driver to use the same socket for that session (which isn't necessarily the case since Mongo instances have built-in connection pooling). This is only necessary for a write heavy environment, where you might read data that you wrote.

To do that, you would do something like:

```
DB db...;
db.requestStart();
code....
db.requestDone();
```

Java - Saving Objects Using DBObject

The Java driver provides a DBObject interface to save custom objects to the database.

For example, suppose one had a class called Tweet that they wanted to save:

```
public class Tweet implements DBObject {
    /* ... */
}
```

Then you can say:

```
Tweet myTweet = new Tweet();
myTweet.put("user", userId);
myTweet.put("message", msg);
myTweet.put("date", new Date());
collection.insert(myTweet);
```

When a document is retrieved from the database, it is automatically converted to a DBObject. To convert it to an instance of your class, use DBCollection.setObjectClass():

```
collection.setObjectClass(Tweet);
Tweet myTweet = (Tweet)collection.findOne();
```

Java Tutorial

- Introduction
- A Quick Tour
 - Making A Connection
 - Authentication (Optional)
 - Getting A List Of Collections
 - Getting A Collection
 - Inserting a Document
 - Finding the First Document In A Collection using findOne()
 - Adding Multiple Documents
 - Counting Documents in A Collection
 - Using a Cursor to Get All the Documents
 - Getting A Single Document with A Query
 - Getting A Set of Documents With a Query
 - Creating An Index
 - Getting a List of Indexes on a Collection
- Quick Tour of the Administrative Functions
 - Getting A List of Databases
 - Dropping A Database

Introduction

This page is a brief overview of working with the MongoDB Java Driver.

For more information about the Java API, please refer to the online API Documentation for Java Driver

A Quick Tour

Using the Java driver is very simple. First, be sure to include the driver jar mongo.jar in your classpath. The following code snippets come from the examples/QuickTour.java example code found in the driver.

Making A Connection

To make a connection to a MongoDB, you need to have at the minimum, the name of a database to connect to. The database doesn't have to exist - if it doesn't, MongoDB will create it for you.

Additionally, you can specify the server address and port when connecting. The following example shows three ways to connect to the database *mydb* on the local machine :

```
import com.mongodb.Mongo;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCobject;
import com.mongodb.DBObject;
import com.mongodb.DBCursor;

Mongo m = new Mongo();
Mongo m = new Mongo( "localhost" );
Mongo m = new Mongo( "localhost" , 27017 );

DB db = m.getDB( "mydb" );
```

At this point, the db object will be a connection to a MongoDB server for the specified database. With it, you can do further operations.

Authentication (Optional)

MongoDB can be run in a secure mode where access to databases is controlled through name and password authentication. When run in this mode, any client application must provide a name and password before doing any operations. In the Java driver, you simply do the following with the connected mongo object:

```
boolean auth = db.authenticate(myUserName, myPassword);
```

If the name and password are valid for the database, auth will be true. Otherwise, it will be false. You should look at the MongoDB log for further information if available.

Getting A List Of Collections

Each database has zero or more collections. You can retrieve a list of them from the db (and print out any that are there):

```
Set<String> colls = db.getCollectionNames();

for (String s : colls) {
        System.out.println(s);
}
```

and assuming that there are two collections, name and address, in the database, you would see

```
name
address
```

as the output.

Getting A Collection

To get a collection to use, just specify the name of the collection to the getCollection(String collectionName) method:

```
DBCollection coll = db.getCollection("testCollection");
```

Once you have this collection object, you can now do things like insert data, query for data, etc

Inserting a Document

Once you have the collection object, you can insert documents into the collection. For example, lets make a little document that in JSON would be represented as

Notice that the above has an "inner" document embedded within it. To do this, we can use the BasicDBObject class to create the document (including the inner document), and then just simply insert it into the collection using the insert() method.

```
BasicDBObject doc = new BasicDBObject();

doc.put("name", "MongoDB");
doc.put("type", "database");
doc.put("count", 1);

BasicDBObject info = new BasicDBObject();

info.put("x", 203);
info.put("y", 102);

doc.put("info", info);
coll.insert(doc);
```

Finding the First Document In A Collection using findOne()

To show that the document we inserted in the previous step is there, we can do a simple findOne() operation to get the first document in the collection. This method returns a single document (rather than the DBCursor that the find() operation returns), and it's useful for things where there only is one document, or you are only interested in the first. You don't have to deal with the cursor.

```
DBObject myDoc = coll.findOne();
System.out.println(myDoc);
```

and you should see

```
{ "_id" : "49902cde5162504500b45c2c" , "name" : "MongoDB" , "type" : "database" , "count" : 1 , "info" : { "x" : 203 , "y" : 102} , "_ns" : "testCollection"}
```

Note the _id and _ns elements have been added automatically by MongoDB to your document. Remember, MongoDB reserves element names that start with _ for internal use.

Adding Multiple Documents

In order to do more interesting things with queries, let's add multiple simple documents to the collection. These documents will just be

```
{
    "i" : value
}
```

and we can do this fairly efficiently in a loop

```
for (int i=0; i < 100; i++) {
     coll.insert(new BasicDBObject().append("i", i));
}</pre>
```

Notice that we can insert documents of different "shapes" into the same collection. This aspect is what we mean when we say that MongoDB is "schema-free"

Counting Documents in A Collection

Now that we've inserted 101 documents (the 100 we did in the loop, plus the first one), we can check to see if we have them all using the getCount() method.

```
System.out.println(coll.getCount());
```

and it should print 101.

Using a Cursor to Get All the Documents

In order to get all the documents in the collection, we will use the find() method. The find() method returns a DBCursor object which allows us to iterate over the set of documents that matched our query. So to query all of the documents and print them out:

```
DBCursor cur = coll.find();

while(cur.hasNext()) {
    System.out.println(cur.next());
}
```

and that should print all 101 documents in the collection.

Getting A Single Document with A Query

We can create a *query* to pass to the find() method to get a subset of the documents in our collection. For example, if we wanted to find the document for which the value of the "i" field is 71, we would do the following;

```
BasicDBObject query = new BasicDBObject();

query.put("i", 71);

cur = coll.find(query);

while(cur.hasNext()) {
    System.out.println(cur.next());
}
```

and it should just print just one document

```
{ "_id" : "49903677516250c1008d624e" , "i" : 71 , "_ns" : "testCollection"}
```

Getting A Set of Documents With a Query

We can use the query to get a set of documents from our collection. For example, if we wanted to get all documents where "i" > 50, we could write :

```
query = new BasicDBObject();
    query.put("i", new BasicDBObject("$gt", 50)); // e.g. find all where i > 50
    cur = coll.find(query);
    while(cur.hasNext()) {
        System.out.println(cur.next());
    }
```

which should print the documents where i > 50. We could also get a range, say 20 < i <= 30:

```
query = new BasicDBObject();
    query.put("i", new BasicDBObject("$gt", 20).append("$lte", 30)); // i.e. 20 < i <= 30
    cur = coll.find(query);
    while(cur.hasNext()) {
        System.out.println(cur.next());
    }</pre>
```

Creating An Index

MongoDB supports indexes, and they are very easy to add on a collection. To create an index, you just specify the field that should be indexed, and specify if you want the index to be ascending (1) or descending (-1). The following creates an ascending index on the "i" field:

```
coll.createIndex(new BasicDBObject("i", 1)); // create index on "i", ascending
```

Getting a List of Indexes on a Collection

You can get a list of the indexes on a collection :

```
List<DBObject> list = coll.getIndexInfo();

for (DBObject o : list) {
        System.out.println(o);
    }
```

and you should see something like

```
{ "name" : "i_1" , "ns" : "mydb.testCollection" , "key" : { "i" : 1} , "_ns" : "system.indexes"}
```

Quick Tour of the Administrative Functions

Getting A List of Databases

You can get a list of the available databases:

```
Mongo m = new Mongo();

for (String s : m.getDatabaseNames()) {
        System.out.println(s);
}
```

Dropping A Database

You can drop a database by name using the Mongo object:

```
m.dropDatabase("my_new_db");
```

Java Types

Object Ids

com.mongodb.ObjectId is used to autogenerate unique ids.

```
ObjectId id = new ObjectId();
ObjectId copy = new ObjectId(id);
```

Regular Expressions

The Java driver uses java.util.regex.Pattern for regular expressions.

```
Pattern john = Pattern.compile("joh?n", CASE_INSENSITIVE);
BasicDBObject query = new BasicDBObject("name", john);

// finds all people with "name" matching /joh?n/i
DBCursor cursor = collection.find(query);
```

Dates/Times

The java.util.Date class is used for dates.

```
Date now = new Date();
BasicDBObject time = new BasicDBObject("ts", now);
collection.save(time);
```

Database References

com.mongodb.DBRef can be used to save database references.

```
DBRef addressRef = new DBRef(db, "foo.bar", address_id);
DBObject address = addressRef.fetch();

DBObject person = BasicDBObjectBuilder.start()
    .add("name", "Fred")
    .add("address", addressRef)
    .get();
collection.save(person);

DBObject fred = collection.findOne();
// address reference is returned as a DBObject, not a DBRef
DBObject addressObj = (DBObject)fred.get("address");
```

Binary Data

An array of bytes (byte[]) can be used for binary data.

C++ Language Center

A C++ driver is available for communicating with the MongoDB. As the database is written in C++, the driver actually uses some core MongoDB code -- this is the same driver that the database uses itself for replication.

The driver has been compiled successfully on Linux, OS X, Windows, and Solaris.

- API Documentation
- C++ Tutorial
- HOWTO
 - Connecting
 - Tailable Cursors
- Mongo Database and C++ Driver Source Code (at github)

C++ Tutorial

This document is an introduction to usage of the MongoDB database from a C++ program.

First, install Mongo -- see Getting Started for details.

Next, you may wish to take a look at the Developer's Tour guide for a language independent look at how to use MongoDB. Also, we suggest some basic familiarity with the mongo shell -- the shell is one's primary database administration tool and is useful for manually inspecting the contents of a database after your C++ program runs.

Installing the Driver Library and Headers

A good source for general information about setting up a MongoDB development environment on various operating systems is the building page.

Unix

From Source

For Unix, the Mongo driver library is libmongoclient.a.

When installing from source, use scons install to install the libraries. By default library and header files are installed in /usr/local. You can use --prefix to change the install path: scons --prefix /opt/mongo install.

From Distribution

The normal db distribution includes the C++ driver. You can just unzip wherever you like.

Windows

The MongoDB Windows binary packages include a file lib/mongoclient.lib. Include this library in your client project.

For more information on Boost setup see the Building for Windows page.

Compiling

The C++ drivers requires the boost libraries to compile. Be sure boost is on your include and lib paths.

Writing Client Code

Note: for brevity, the examples below are simply inline code. In a real application one will define classes for each database object typically.

Connecting

Let's make a tutorial.cpp file that connects to the database (see client/examples/tutorial.cpp for full text of the examples below):

```
#include <iostream>
#include "client/dbclient.h"
using namespace mongo;
void run() {
 DBClientConnection c;
  c.connect("localhost");
}
int main() {
  try {
    run();
    cout << "connected ok" << endl;</pre>
  } catch( DBException &e ) {
    cout << "caught " << e.what() << endl;</pre>
  }
  return 0;
}
```

If you are using gcc on Linux or OS X, you would compile with something like this, depending on location of your include files and libraries:

```
$ g++ tutorial.cpp -lmongoclient -lboost_thread-mt -lboost_filesystem -o tutorial
$ ./tutorial
connected ok
$
```



depending on your boost version you might need to link against the *boost_system* library as well: **-lboost_system**. Also, you may need to append "-mt" to boost_filesystem.

BSON

The Mongo database stores data in BSON format. BSON is a binary object format that is JSON-like in terms of the data which can be stored (some extensions exist, for example, a Date datatype).

To save data in the database we must create objects of class BSONObj. The components of a BSONObj are represented as BSONElement objects. We use BSONObjBuilder to make BSON objects, and BSONObjIterator to enumerate BSON objects.

Let's now create a BSON "person" object which contains name and age. We might invoke:

```
BSONObjBuilder b;
b.append("name", "Joe");
b.append("age", 33);
BSONObj p = b.obj();
```

We can also create objects with a stream-oriented syntax:

```
BSONObjBuilder b;
b << "name" << "Joe" << "age" << 33;
BSONObj p = b.obj();
```

The macro BSON lets us be even more compact:

```
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
```

Use the GENOID helper to add an object id to your object. The server will add an _id automatically if it is not included explicitly.

```
BSONObj p = BSON( GENOID << "name" << "Joe" << "age" << 33 );
// result is: { _id : ..., name : "Joe", age : 33 }
```

GENOID should be at the beginning of the generated object.

Saving Data

We now save our person object in a persons collection in the database:

```
c.insert("tutorial.persons", p);
```

The first parameter to insert is the namespace. tutorial is the database and persons is the collection name.

Retrieving Data

Let's now fetch all objects from the persons collection, and display them. We'll also show here how to use count().

```
cout << "count:" << c.count("tutorial.persons") << endl;

auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", emptyObj);
while( cursor->more() )
    cout << cursor->next().toString() << endl;</pre>
```

emptyObj is the empty BSON object -- we use it to represent {} which indicates an empty query pattern (an empty query is a query for all objects).

We use BSONObj::toString() above to print out information about each object retrieved. BSONObj::toString is a diagnostic function which prints an abbreviated JSON string representation of the object. For full JSON output, use BSONObj::jsonString.

Let's now write a function which prints out the name (only) of all persons in the collection whose age is a given value:

```
void printIfAge(DBClientConnection&c, int age) {
   auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY( "age" << age ) );
   while( cursor->more() ) {
     BSONObj p = cursor->next();
     cout << p.getStringField("name") << endl;
   }
}</pre>
```

getStringField() is a helper that assumes the "name" field is of type string. To manipulate an element in a more generic fashion we can retrieve the particular BSONElement from the enclosing object:

```
BSONElement name = p["name"];
// or:
//BSONElement name = p.getField("name");
```

See the api docs, and jsobj.h, for more information.

Our query above, written as JSON, is of the form

```
{ age : <agevalue> }
```

Queries are BSON objects of a particular format -- in fact, we could have used the BSON() macro above instead of QUERY(). See class Query in dbclient.h for more information on Query objects, and the Sorting section below.

In the mongo shell (which uses javascript), we could invoke:

```
use tutorial;
db.persons.find( { age : 33 } );
```

Indexing

Let's suppose we want to have an index on age so that our queries are fast. We would use:

```
c.ensureIndex("tutorial.persons", fromjson("{age:1}"));
```

The ensureIndex method checks if the index exists; if it does not, it is created. ensureIndex is intelligent and does not repeat transmissions to the server; thus it is safe to call it many times in your code, for example, adjacent to every insert operation.

In the above example we use a new function, fromjson. fromjson converts a JSON string to a BSONObj. This is sometimes a convenient way to specify BSON. Alternatively we could have written:

```
c.ensureIndex("tutorial.persons", BSON( "age" << 1 ));</pre>
```

Sorting

Let's now make the results from printlfAge sorted alphabetically by name. To do this, we change the query statement from:

```
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY( "age" << age ) );</pre>
```

```
to auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", QUERY( "age" << age ).sort("name")
);
```

Here we have used Query::sort() to add a modifier to our query expression for sorting.

Further Reading

This overview just touches on the basics of using Mongo from C++. There are many more capabilities. For further exploration:

- See the language-independent Developer's Tour;
- Experiment with the mongo shell;
- Review the doxygen API docs;
- See connecting pooling information in the API docs;
- See GridFS file storage information in the API docs;
- See the HOWTO pages under the C++ Language Center
- · Consider getting involved to make the product (either C++ driver, tools, or the database itself) better!

Connecting

The C++ driver includes several classes for managing collections under the parent class DBClientInterface.

In general, you will want to instantiate either a DBClientConnection object, or a DBClientPaired object. DBClientConnection is our normal connection class for a connection to a single MongoDB database server (or shard manager). We use DBClientPaired to connect to database replica pairs.

Perl Language Center

- Installing
 - CPAN
 - Manual (Non-CPAN) Installation
 - Big-Endian Systems
- Next Steps
- Other MongoDB Perl Tools
 - MojoX::Session::Store::MongoDB



Start a MongoDB server instance (mongod) before installing so that the tests will pass. Some tests may be skipped if you are not running a recent version of the database (>= 1.1.3).

Installing

CPAN

```
$ sudo cpan MongoDB
```

The Perl driver is available through CPAN as the package Mongode. It should build cleanly on *NIX and Windows (via Strawberry Perl).

Manual (Non-CPAN) Installation

If you would like to try the latest code or are contributing to the Perl driver, it is available at Github. There is also documentation generated after every commit.

You can see if it's a good time to grab the bleeding edge code by seeing if the build is green.

To build the driver, run:

```
$ perl Makefile.PL
$ make
$ make
$ make test # make sure mongod is running, first
$ sudo make install
```

Please note that the tests will not pass without a mongod process running.

Big-Endian Systems

The driver will work on big-endian machines, but the database will not. The tests assume that mongod will be running on localhost unless %ENV{MONGOD} is set. So, to run the tests, start the database on a little-endian machine (at, say, "example.com") and then run the tests with:

MONGOD=example.com make test

A few tests that require a database server on "localhost" will be skipped.

Next Steps

There is a tutorial and API documentation on CPAN.

If you're interested in contributing to the Perl driver, check out Contributing to the Perl Driver.

Other MongoDB Perl Tools

MojoX::Session::Store::MongoDB

MojoX::Session::Store::MongoDB is a store for MojoX::Session that stores a session in a MongoDB database. Created by Ask Bjørn Hansen.

Contributing to the Perl Driver

The easiest way to contribute is to file bugs and feature requests on Jira.

If you would like to help code the driver, read on...

Finding Something to Help With

Fixing Bugs

You can choose a bug on Jira and fix it. Make a comment that you're working on it, to avoid overlap.

Writing Tests

The driver could use a lot more tests. We would be grateful for any and all tests people would like to write.

Adding Features

If you think a feature is missing from the driver, you're probably right. Check on IRC or the mailing list, then go ahead and create a Jira case and add the feature. The Perl driver was a bit neglected for a while (although it's now getting a lot of TLC) so it's missing a lot of things that the other drivers have. You can look through their APIs for ideas.

Contribution Guildlines

The best way to make changes is to create an account on Github, fork the driver, make your improvements, and submit a merge request.

To make sure your changes are approved and speed things along:

- · Write tests. Lots of tests.
- Document your code.
- · Write POD, when applicable.

Bonus (for C programmers, particularly):

• Make sure your change works on Perl 5.8, 5.10, Windows, Mac, Linux, etc.

Code Layout

The important files:

```
| perl_mongo.c # serialization/deserialization
| mongo_link.c # connecting to, sending to, and receiving from the database
- lib
- MongoDB
| Connection.pm # connection, queries, inserts... everything comes through here
| Database.pm
| Collection.pm
| Cursor.pm
| OID.pm
| GridFS.pm
- GridFS
| File.pm
- xs
| Mongo.xs
| Connection.xs
| Cursor.xs
| OID.xs
```

Perl Tutorial

Error formatting macro: redirect: java.lang.NullPointerException

Online API Documentation

MongoDB API and driver documentation is available online. It is updated daily.

- Java Driver API Documentation
- C++ Driver API Documentation
- Python Driver API Documentation
- Ruby Driver API Documentation
- PHP Driver API Documentation

Developer Zone

- Tutorial
- Shell
- Manual
 - Databases
 - Collections
 - Indexes
 - Data Types and Conventions
 - GridFS
 - Inserting
 - Updating
 - Querying
 - Removing
 - Optimization
- Developer FAQ

If you have a comment or question about anything, please contact us through IRC (freenode.net#mongodb) or the mailing list, rather than leaving a comment at the bottom of a page. It is easier for us to respond to you through those channels.

Introduction

MongoDB is a collection-oriented, schema-free document database.

By *collection-oriented*, we mean that data is grouped into sets that are called 'collections'. Each collection has a unique name in the database, and can contain an unlimited number of documents. Collections are analogous to tables in a RDBMS, except that they don't have any defined schema.

By schema-free, we mean that the database doesn't need to know anything about the structure of the documents that you store in a collection. In fact, you can store documents with different structure in the same collection if you so choose.

By document, we mean that we store data that is a structured collection of key-value pairs, where keys are strings, and values are any of a rich

set of data types, including arrays and documents. We call this data format "BSON" for "Binary Serialized dOcument Notation."

MongoDB Operational Overview

MongoDB is a server process that runs on Linux, Windows and OS X. It can be run both as a 32 or 64-bit application. We recommend running in 64-bit mode, since Mongo is limited to a total data size of about 2GB for all databases in 32-bit mode.

The MongoDB process listens on port 27017 by default (note that this can be set at start time - please see Command Line Parameters for more information.

Clients connect to the MongoDB process, optionally authenticate themselves if security is turned on, and perform a sequence of actions, such as inserts, queries and updates.

MongoDB stores its data in files (default location is /data/db/), and uses memory mapped files for data management for efficiency.

MongoDB can also be configured for automatic data replication , as well as automatic fail-over .

For more information on MongoDB administration, please see Mongo Administration Guide.

MongoDB Functionality

As a developer, MongoDB drivers offer a rich range of operations:

- Queries: Search for documents based on either query objects or SQL-like "where predicates". Queries can be sorted, have limited return sizes, can skip parts of the return document set, and can also return partial documents.
- Inserts and Updates: Insert new documents, update existing documents.
- · Index Management : Create indexes on one or more keys in a document, including substructure, deleted indexes, etc
- General commands: Any MongoDB operation can be managed via DB Commands over the regular socket.

See Also

- The Mongo Architecture
- The starting point for all Home

Tutorial

- · Getting the Database
- Getting A Database Connection
- Inserting Data into A Collection
- Accessing Data From a Query
- · Specifying What the Query Returns
- findOne() Syntactic Sugar
- Limiting the Result Set via limit()
- What Next

Getting the Database

Download the database, unpack it, and start the mongod process:

\$ bin/mongod

Getting A Database Connection

Let's now try manipulating the database with the database shell. (Note we could perform similar operations from any programming language using an appropriate driver. The shell is convenient for interactive use.)

Start the MongoDB JavaScript shell with:

\$ bin/mongo

(By default the shell connects to an assumed database on localhost.) You then see:

```
MongoDB shell version: 0.9.8
url: test
connecting to: test
type "help" for help
>
```

"connecting to:" tells you the name of the database the shell is using. To switch databases, type:

```
> use mydb
```

To see a list of handy commands, type help.



Tip for Developers with Experience in Other Databases

You may notice, in the examples below, that we never create a database or collection. MongoDB does not require that you do so. As soon as you insert something, MongoDB creates the underlying collection and database. If you query a collection that does not exist, Mongo treats it as an empty collection.

Inserting Data into A Collection

Let's create a test collection and insert some data into it. We will create two objects, j and t, and then save them in the collection things.

In the following examples, '>' indicates commands typed at the shell prompt.

```
> j = { name: "mongo"};
{"name": "mongo"}
> t = { x : 3 };
{ "x" : 3 }
> db.things.save(j);
> db.things.save(t);
> db.things.find();
in cursor for : DBQuery: example.things ->
{"name": "mongo", "_id": "497cf60751712cf7758fbdbb"}
{"x" : 3 , "_id": "497cf61651712cf7758fbdbc"}
>
```

A few things to note:

- We did not predefine the collection. The database creates it automatically on the first insert.
- The documents we store can have any "structure" in fact in this example, the documents have no common data elements at all. In practice, one usually stores documents of the same structure within collections. However, this flexibility means that schema migration and augmentation are very easy in practice rarely will you need to write scripts which perform "alter table" type operations.
- Upon being inserted into the database, objects are assigned an object ID (if they do not already have one) in the field _id.
- When you run the above example, your ObjectID values will be different.

Let's add some more records to this collection:

```
> for( var i = 1; i < 10; i++ ) db.things.save( { x:4, j:i } );
> db.things.find();
in cursor for: DBQuery: example.things ->
    {"name": "mongo", "_id": "497cf60751712cf7758fbdbb"}
    {"x": 3 , "_id": "497cf61651712cf7758fbdbc"}
    {"x": 4 , "j": 1 , "_id": "497cf87151712cf7758fbdbd"}
    {"x": 4 , "j": 2 , "_id": "497cf87151712cf7758fbdbe"}
    {"x": 4 , "j": 3 , "_id": "497cf87151712cf7758fbdbf"}
    {"x": 4 , "j": 5 , "_id": "497cf87151712cf7758fbdc0"}
    {"x": 4 , "j": 5 , "_id": "497cf87151712cf7758fbdc0"}
    {"x": 4 , "j": 6 , "_id": "497cf87151712cf7758fbdc2"}
    {"x": 4 , "j": 7 , "_id": "497cf87151712cf7758fbdc2"}
    {"x": 4 , "j": 8 , "_id": "497cf87151712cf7758fbdc2"}
    {"x": 4 , "j": 8 , "_id": "497cf87151712cf7758fbdc4"}
    has more
```

Note that not all documents were shown - the shell limits the number to 10 when automatically iterating a cursor. Since we already had 2

documents in the collection, we only see the first 8 of the newly-inserted documents.

If we want to return the next set of results, there's the it shortcut. Continuing from the code above:

```
{"x" : 4 , "j" : 7 , "_id" : "497cf87151712cf7758fbdc3"}

{"x" : 4 , "j" : 8 , "_id" : "497cf87151712cf7758fbdc4"}

has more

> it

{"x" : 4 , "j" : 9 , "_id" : "497cf87151712cf7758fbdc5"}

{"x" : 4 , "j" : 10 , "_id" : "497cf87151712cf7758fbdc6"}
```

Technically, find() returns a cursor object. But in the cases above, we haven't assigned that cursor to a variable. So, the shell automatically iterates over the cursor, giving us an initial result set, and allowing us to continue iterating with the **it** command.

But we can also work with the cursor directly; just how that's done is discussed in the next section.

Accessing Data From a Query

Before we discuss queries in any depth, lets talk about how to work with the results of a query - a cursor object. We'll use the simple find() query method, which returns everything in a collection, and talk about how to create specific queries later on.

In order to see all the elements in the collection when using the mongo shell, we need to explicitly use the cursor returned from the find() operation.

Lets repeat the same query, but this time use the cursor that find() returns, and iterate over it in a while loop:

```
> var cursor = db.things.find();
> while (cursor.hasNext()) { print(tojson(cursor.next())); }
{"name" : "mongo" , "_id" : "497cf60751712cf7758fbdbb"}
{"x" : 3 , "_id" : "497cf61651712cf7758fbdbc"}
{"x" : 4 , "j" : 1 , "_id" : "497cf87151712cf7758fbdbd"}
{"x" : 4 , "j" : 2 , "_id" : "497cf87151712cf7758fbdbe"}
{"x" : 4 , "j" : 3 , "_id" : "497cf87151712cf7758fbdbf"}
{"x" : 4 , "j" : 3 , "_id" : "497cf87151712cf7758fbdc0"}
{"x" : 4 , "j" : 5 , "_id" : "497cf87151712cf7758fbdc1"}
{"x" : 4 , "j" : 6 , "_id" : "497cf87151712cf7758fbdc2"}
{"x" : 4 , "j" : 7 , "_id" : "497cf87151712cf7758fbdc3"}
{"x" : 4 , "j" : 8 , "_id" : "497cf87151712cf7758fbdc4"}
{"x" : 4 , "j" : 9 , "_id" : "497cf87151712cf7758fbdc5"}
>
```

The above example shows cursor-style iteration. The hasNext() function tells if there are any more documents to return, and the next() function returns the next document. We also used the built-in tojson() method to render the document in a pretty JSON-style format.

When working in the JavaScript shell, we can also use the functional features of the language, and just call forEach on the cursor. Repeating the example above, but using forEach() directly on the cursor rather than the while loop:

```
> db.things.find().forEach( function(x) { print(tojson(x));});
{"name" : "mongo" , "_id" : "497cf60751712cf7758fbdbb"}
{"x" : 3 , "_id" : "497cf61651712cf7758fbdbc"}
{"x" : 4 , "j" : 1 , "_id" : "497cf87151712cf7758fbdbd"}
{"x" : 4 , "j" : 2 , "_id" : "497cf87151712cf7758fbdbe"}
{"x" : 4 , "j" : 3 , "_id" : "497cf87151712cf7758fbdbf"}
{"x" : 4 , "j" : 3 , "_id" : "497cf87151712cf7758fbdc0"}
{"x" : 4 , "j" : 5 , "_id" : "497cf87151712cf7758fbdc1"}
{"x" : 4 , "j" : 6 , "_id" : "497cf87151712cf7758fbdc2"}
{"x" : 4 , "j" : 7 , "_id" : "497cf87151712cf7758fbdc3"}
{"x" : 4 , "j" : 8 , "_id" : "497cf87151712cf7758fbdc3"}
{"x" : 4 , "j" : 9 , "_id" : "497cf87151712cf7758fbdc5"}
>
```

In the case of a forEach() we must define a function that is called for each document in the cursor.

In the mongo shell, you can also treat cursors like an array:

```
> var cursor = db.things.find();
> print (tojson(cursor[4]));
{"x" : 4 , "j" : 3 , "_id" : "497cf87151712cf7758fbdbf"}
```

When using a cursor this way, note that all values up to the highest accessed (cursor[4] above) are loaded into RAM at the same time. This is inappropriate for large result sets, as you will run out of memory. Cursors should be used as an iterator with any query which returns a large number of elements.

In addition to array-style access to a cursor, you may also convert the cursor to a true array:

```
> var arr = db.things.find().toArray();
> arr[5];
{"x" : 4 , "j" : 4 , "_id" : "497cf87151712cf7758fbdc0"}
```

Please note that these array features are specific to mongo - The Interactive Shell, and not offered by all drivers.

MongoDB cursors are not snapshots - operations performed by you or other users on the collection being queried between the first and last call to next() of your cursor *may or may not* be returned by the cursor. Use explicit locking to perform a snapshotted query.

Specifying What the Query Returns

Now that we know how to work with the cursor objects that are returned from queries, lets now focus on how to tailor queries to return specific things.

In general, the way to do this is to create "query documents", which are documents that indicate the pattern of keys and values that are to be matched.

These are easier to demonstrate than explain. In the following examples, we'll give example SQL queries, and demonstrate how to represent the same query using MongoDB via the mongo shell. This way of specifying queries is fundamental to MongoDB, so you'll find the same general facility in any driver or language.

```
SELECT * FROM things WHERE name="mongo"

> db.things.find({name:"mongo"}).forEach(function(x) { print(tojson(x));});
{"name" : "mongo" , "_id" : "497cf60751712cf7758fbdbb"}
>
```

```
SELECT * FROM things WHERE x=4

> db.things.find({x:4}).forEach(function(x) { print(tojson(x));});

{"x" : 4 , "j" : 1 , "_id" : "497cf87151712cf7758fbdbd"}

{"x" : 4 , "j" : 2 , "_id" : "497cf87151712cf7758fbdbe"}

{"x" : 4 , "j" : 3 , "_id" : "497cf87151712cf7758fbdbf"}

{"x" : 4 , "j" : 4 , "_id" : "497cf87151712cf7758fbdc0"}

{"x" : 4 , "j" : 5 , "_id" : "497cf87151712cf7758fbdc0"}

{"x" : 4 , "j" : 6 , "_id" : "497cf87151712cf7758fbdc2"}

{"x" : 4 , "j" : 7 , "_id" : "497cf87151712cf7758fbdc2"}

{"x" : 4 , "j" : 8 , "_id" : "497cf87151712cf7758fbdc3"}

{"x" : 4 , "j" : 9 , "_id" : "497cf87151712cf7758fbdc4"}

{"x" : 4 , "j" : 9 , "_id" : "497cf87151712cf7758fbdc5"}

>
```

The query expression is an document itself. A query document of the form { a:A, b:B, ... } means "where a==A and b==B and ...". More information on query capabilities may be found in the Queries and Cursors section of the Mongo Developers' Guide.

MongoDB also lets you return "partial documents" - documents that have only a subset of the elements of the document stored in the database. To do this, you add a second argument to the find() query, supplying a document that lists the elements to be returned.

To illustrate, lets repeat the last example find({x:4}) with an additional argument that limits the returned document to just the "i" elements:

```
SELECT j FROM things WHERE x=4

> db.things.find({x:4}, {j:true}).forEach(function(x) { print(tojson(x));});

{"j" : 1 , "_id" : "497cf87151712cf7758fbdbd"}

{"j" : 2 , "_id" : "497cf87151712cf7758fbdbe"}

{"j" : 3 , "_id" : "497cf87151712cf7758fbdbf"}

{"j" : 4 , "_id" : "497cf87151712cf7758fbdc0"}

{"j" : 5 , "_id" : "497cf87151712cf7758fbdc1"}

{"j" : 6 , "_id" : "497cf87151712cf7758fbdc2"}

{"j" : 7 , "_id" : "497cf87151712cf7758fbdc3"}

{"j" : 8 , "_id" : "497cf87151712cf7758fbdc4"}

{"j" : 9 , "_id" : "497cf87151712cf7758fbdc5"}

>
```

Note that the "_id" field is always returned.

findOne() - Syntactic Sugar

For convenience, the mongo shell (and other drivers) lets you avoid the programming overhead of dealing with the cursor, and just lets you retrieve one document via the findOne() function. findOne() takes all the same parameters of the find() function, but instead of returning a cursor, it will return either the first document returned from the database, or null if no document is found that matches the specified query.

As an example, lets retrieve the one document with name=='mongo'. There are many ways to do it, including just calling next() on the cursor (after checking for null, of course), or treating the cursor as an array and accessing the 0th element.

However, the findOne() method is both convenient and efficient:

```
> var mongo = db.things.findOne({name:"mongo"});
> print(tojson(mongo));
{"name" : "mongo" , "_id" : "497cf60751712cf7758fbdbb"}
>
```

This is more efficient because the client requests a single object from the database, so less work is done by the database and the network. This is the equivalent of find (name: "mongo" }) .limit(1).

Limiting the Result Set via limit()

You may limit the size of a query's result set by specifing a maximum number of results to be returned via the limit() method.

This is highly recommended for performance reasons, as it limits the work the database does, and limits the amount of data returned over the network. For example:

```
> db.things.find().limit(3);
in cursor for : DBQuery: example.things ->
{"name" : "mongo" , "_id" : "497cf60751712cf7758fbdbb"}
{"x" : 3 , "_id" : "497cf61651712cf7758fbdbc"}
{"x" : 4 , "j" : 1 , "_id" : "497cf87151712cf7758fbdbd"}
>
```

What Next

After completing this tutorial the next step to learning MongoDB is to dive into the manual for more details.

Manual

This is the MongoDB manual. Except where otherwise noted, all examples are in JavaScript for use with the mongo shell. There is a table available giving the equivalent syntax for each of the drivers.

- Connections
- Databases
 - Commands
 - Clone Database
 - fsync Command
 - Index-Related Commands

- Last Error Commands
- Viewing and Terminating Current Operation
- Validate Command
- getLastError
- List of Database Commands
- Mongo Metadata
- Collections
 - Capped Collections
 - Using a Large Number of Collections
- Data Types and Conventions
 - Internationalized Strings
 - Object IDs
 - DB Ref
- GridFS
- Indexes
 - Indexing as a Background Operation
 - Multikeys
- Inserting
 - Legal Key Names
 - Schema Design
 - Trees in MongoDB
- Optimization
 - · Optimizing Storage of Small Objects
 - Query Optimizer
- Querying
 - Advanced Queries
 - Dot Notation (Reaching into Objects)
 - Full Text Search in Mongo
 - min and max Query Specifiers
 - OR operations in query expressions
 - Queries and Cursors
 - Tailable Cursors
 - Server-side Code Execution
 - · Sorting and Natural Order
 - Aggregation
- Removing
- Updating
 - Atomic Operations
 - findandmodify Command
 - Updating Data in Mongo
- MapReduce
- Data Processing Manual

Connections



The format described on this page is an eventual goal for the drivers. Refer to you driver's documentation to see how much (if any) of the standard connection format is supported. Currently, only the PHP and Python drivers support subsets of this functionality.

Standard Connection Format

mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/database]

- mongodb: // is a required prefix to identify that this is a string in the standard connection format.
- username: password@ are optional. If given, the driver will attempt to login to a database after connecting to a database server.
- host1 is the only required part of the URI. It identifies a server address to connect to.
- :portX is optional and defaults to :27017 if not provided.
- /database is the name of the database to login to and thus is only relevant if the username:password@ syntax is used. If not specified the "admin" database will be used by default.

As many hosts as necessary may be specified (for connecting to replica pairs/sets).

Examples

Connect to a database server running locally on the default port:

mongodb://localhost

Connect and login to the admin database as user "fred" with password "foobar":

mongodb://fred:foobar@localhost

Connect and login to the "baz" database as user "fred" with password "foobar":

mongodb://fred:foobar@localhost/baz

Connect to a replica pair, with one server on example1.com and another server on example2.com:

mongodb://example1.com:27017,example2.com:27017

Connect to a replica set with three servers running on localhost (on ports 27017, 27018, and 27019):

mongodb://localhost,localhost:27018,localhost:27019

Databases

Each MongoDB server can support multiple *databases*. Each database is independent, and the data for each database is stored separately, for security and ease of management.

A database consists of one or more *collections*, the *documents* (objects) in those collections, and an optional set of security credentials for controlling access.

- Commands
 - Clone Database
 - fsync Command
 - Index-Related Commands
 - Last Error Commands
 - Viewing and Terminating Current Operation
 - Validate Command
 - getLastError
 - List of Database Commands
- Mongo Metadata

Commands

Introduction

The Mongo database has a concept of a *database command*. Database commands are ways to ask the database to perform special operations, or to request information about its current operational status.

- Introduction
- Privileged Commands
- Getting Help Info for a Command
- More Command Documentation

A command is sent to the database as a query to a special collection namespace called \$cmd. The database will return a single document with the command results - use findOne() for that if your driver has it.

The general command syntax is:

```
db.$cmd.findOne( { <commandname>: <value> [, options] } );
```

The shell provides a helper function for this:

```
db.runCommand{ <commandname>: <value> [, options] } );
```

For example, to check our database's current profile level setting, we can invoke:

```
> db.runCommand({profile:-1});
{
    "was" : 0.0 ,
    "ok" : 1.0
}
```

For many db commands, some drivers implement wrapper methods are implemented to make usage easier. For example, the mongo shell offers

```
> db.getProfilingLevel()
0.0
```

Let's look at what this method is doing:

```
> print( db.getProfilingLevel )
function () {
    var res = this._dbCommand({profile:-1});
    return res ? res.was : null;
}
> print( db._dbCommand )
function (cmdObj) {
    return this.$cmd.findOne(cmdObj);
}
```

Many commands have helper functions - see your driver's documentation for more information.

Privileged Commands

Certain operations are for the database administrator only. These privileged operations may only be performed on the special database named admin.

```
> use admin;
> db.runCommand("shutdown"); // shut down the database
```

If the db variable is not set to 'admin', you can use _adminCommand to switch to the right database automatically (and just for that operation):

```
> db._adminCommand("shutdown");
```

(For this particular command there is also a shell helper function, db.shutdownServer.)

Getting Help Info for a Command

Use commandHelp in shell to get help info for a command:

```
> db.commandHelp("datasize") help for: datasize example: { datasize:"blog.posts", keyPattern:\{x:1\}, min:\{x:10\}, max:\{x:55\} } NOTE: This command may take awhile to run
```

(Help is not yet available for some commands.)

More Command Documentation

- Clone Database
- fsync Command
- Index-Related Commands
- Last Error Commands
- Viewing and Terminating Current Operation
- Validate Command
- getLastError
- List of Database Commands

Clone Database

MongoDB includes commands for copying a database from one server to another.

```
// copy an entire database from one name on one server to another
// name on another server. omit <from_hostname> to copy from one
// name to another on the same server.
db.copyDatabase(<from_dbname>, <to_dbname>, <from_hostname>);
// in "command" syntax (runnable from any driver):
db.runCommand( { copydb : 1, fromdb : ..., todb : ..., fromhost : ... } );

// clone the current database (implied by 'db') from another host
var fromhost = ...;
print("about to get a copy of database " + db + " from " + fromhost);
db.cloneDatabase(fromhost);
// in "command" syntax (runnable from any driver):
db.runCommand( { clone : fromhost } );
```

fsync Command

- Basics
- Lock, Snapshot and Unlock
- See Also



Version 1.3.1 and higher

The fsync command allows us to flush all pending writes to datafiles. More importantly, it also provides a lock option that makes backups easier.

Basics

The fsync command forces the database to flush all datafiles:

```
> use admin
> db.runCommand({fsync:1});
```

By default the command returns after synchronizing. To return immediately use:

```
> db.runCommand({fsync:1,async:true});
```

To fsync on a regular basis, use the --syncdelay command line option (see mongod --help output). By default a full flush is forced every 60 seconds.

Lock, Snapshot and Unlock

The fsync command supports a lock option that allows one to safely snapshot the database's datafiles. While locked, all write operations are blocked, although read operations are still allowed. After snapshotting, use the unlock command to unlock the database and allow locks again. Example:

```
> use admin
switched to db admin
> db.runCommand({fsync:1,lock:1})
 "info" : "now locked against writes",
 "ok" : 1
> db.currentOp()
 "inprog" : [
1.
 "fsyncLock" : 1
}
> // do some work here: for example, snapshot datafiles...
> db.$cmd.sys.unlock.findOne();
{ "ok" : 1, "info" : "unlock requested" }
> // unlock is now requested. it may take a moment to take effect.
> db.currentOp()
{ "inprog" : [ ] }
```

See Also

Backups

Index-Related Commands

Create Index

ensureIndex() is the helper function for this. Its implementation creates an index by adding its info to the system.indexes table.

```
> db.myCollection.ensureIndex(<keypattern>);
> // same as:
> db.system.indexes.insert({ name: "name", ns: "namespaceToIndex",
   key: <keypattern> });
```

You can query system.indexes too, for example to see all indexes for a table foo:

```
>db.system.indexes.find( { ns: "foo" } );
```

In some drivers, <code>ensureIndex()</code> remembers if it has recently been called, and foregoes the insert operation in that case. Even if this is not the case, <code>ensureIndex()</code> is a cheap operation, so it may be invoked often to ensure that an index exists.

Dropping an Index

From the shell:

```
db.mycollection.dropIndex(<name_or_pattern>)
  db.mycollection.dropIndexes()

// example:
  t.dropIndex( { name : 1 } );
```

From a driver (raw command object form; many drivers have helpers):

```
{ deleteIndexes: <collection_name>, index: <index_name> }
// "*" for <index_name> will drop all indexes except _id
```

Index Namespace

Each index has a namespace of its own for the btree buckets. The namespace is:

```
<collectionnamespace>.$<indexname>
```

This is an internal namespace that cannot be queried directly.

Last Error Commands

getlasterror

Check for an error on the last database operation for this connection.

```
> db.$cmd.findOne({getlasterror:1})
{
    "err" : null , // err is null if no error
    "ok" : 1.0
}
...
> db.$cmd.findOne({getlasterror:1})
{
    "err" : "''an error msg''" ,
    "ok" : 1.0
}
```

getpreverror

Check for a previous error, even if not the immediately preceding operation. nPrev indicates how many operations back in time the error occurred.

Clear any flagged error on this db connection. Useful in conjunction with ${\tt getpreverror}.$

```
> db.$cmd.findOne({reseterror:1})
{
    "ok" : 1.0
}
```

Viewing and Terminating Current Operation

- View Current Operation(s) in Progress
- Terminate (Kill) an Operation in Progress

View Current Operation(s) in Progress

^{*} reseterror

Fields:

- opid an incrementing operation number. Use with killOp().
- op the operation type (query, update, etc.)
- ns namespace for the operation (database + collection name)
- · query the query spec, if operation is a query

NOTE: currentOp's output format varies from version 1.0 and version 1.1 of MongoDB. The format above is for 1.1 and higher.

Terminate (Kill) an Operation in Progress

```
// <= v1.2
> db.killOp()
> // same as: db.$cmd.sys.killop.findOne()
{"info" : "no op in progress/not locked"}

// v>= 1.3
> db.killOp(1234/*opid*/)
> // same as: db.$cmd.sys.killop.findOne({op:1234})
```

Validate Command

Use this command to check that a collection is valid (not corrupt) and to get various statistics.

This command scans the entire collection and its indexes and will be very slow on large datasets.

From the mongo shell:

```
> db.foo.validate()
{"ns" : "test.foo" , "result" : "
validate
  details: 08D03C9C ofs:963c9c
  firstExtent:0:156800 ns:test.foo
  lastExtent:0:156800 ns:test.foo
  # extents:1
  datasize?:144 nrecords?:3 lastExtentSize:2816
  padding:1
  first extent:
   loc:0:156800 xnext:null xprev:null
   ns:test.foo
   size:2816 firstRecord:0:1568b0 lastRecord:0:156930
  3 objects found, nobj:3
  192 bytes data w/headers
  144 bytes data wout/headers
  deletedList: 0000000100000000000
  deleted: n: 1 size: 2448
 nIndexes:1
    test.foo.$x_1 keys:3
  , "ok" : 1 , "valid" : true , "lastExtentSize" : 2816}
```

From a driver one might invoke the driver's equivalent of:

```
> db.$cmd.findOne({validate:"foo" } );
```

validate takes an optional scandata parameter which skips the scan of the base collection (but still scans indexes).

```
> db.$cmd.findOne({validate:"foo", scandata:true});
```

getLastError

Most drivers, and the db shell, support a getlasterror capability. This lets one check the error code on the last operation.

Database commands, as well as queries, have a direct return code. While last error is set for them also, it is primarily useful for write operations. Write operations by default do not have a return code: this saves the client from waiting for client/server turnarounds during write operations. One

can always call getLastError if one wants a return code. While most operations support recording the last error, killCursors does not.

getPrevError

When performing bulk write operations, resetError() and getPrevError() can be an efficient way to check for success of the operation. For example if we are inserting 1,000 objects in a collection, checking the return code 1,000 times over the network is slow. Instead one might do something like this:

```
db.resetError();
for( loop 1000 times... )
   db.foo.save(something...);
if( db.getPrevError().err )
   print("didn't work!");
```

Last Error in the Shell

The database shell performs a resetError() before each read/eval/print loop command evaluation - and automatically prints the error, if one occurred, after each evaluation. Thus, after an error, at the shell prompt db.getLastError() will return null. However, if called before returning to the prompt, the result is as one would expect:

```
> try { db.foo.findOne() } catch(e) { print("preverr:" + tojson(db.getPrevError())); print("lasterr:"
+ tojson(db.getLastError()));}
preverr:{"err" : "unauthorized" , "nPrev" : 1 , "ok" : 1}
lasterr:"unauthorized"
```

FSync with GetLastError

Include the fsync option to force the database to fsync all files before returning (v1.3+):

```
> db.runCommand({getlasterror:1,fsync:true})
{ "err" : null, "n" : 0, "fsyncFiles" : 2, "ok" : 1 }
```

List of Database Commands

A work in progress.

Key:

- Auth requires authentication
- Admin must be run from the admin database
- Slave Okay can be run on a slave db server

Command	Requires Auth	Admin Only	Slave Okay	Description
{buildinfo : 1}	No	Yes	Yes	Returns: {version : dbVersion, gitVersion : gitCommitId, sysInfo : osInfo, ok : 1}
{listDatabases : 1}	No	Yes	Yes	Returns a list of database names and sizes.
{ fsync : 1[, async : true] }	No	Yes	Yes	{numFiles : N, ok : 1}

{ clone : from_host }	Yes	No	No	Copy the current database from another host to here. DocPage
{ copydb :, fromdb :, todb :, fromhost : }	Yes	Yes	No	Copy a database, potentially from another host. DocPag
{shutdown : 1}	Yes	Yes	Yes	Shuts down the database server. Returns nothing. DocPage
{reseterror : 1}	No	No	Yes	Clears previous database errors. Returns:
{getlasterror : 1}	No	No	Yes	Returns if there was an error on the previous database operation. DocPage {err : error, n : numberOfErrors, ok : 1}
{getpreverror : 1}	No	No	Yes	Returns when the last database error occured. {err : error, n : numberOfErrors, nPrev : numberOfOperationsAgo, ok : 1}
{forceerror : 1}	No	No	Yes	Forces a database error. {assertion : "forced error", errmsg : "db assertion failure", ok : 0}
{dropDatabase : 1}	No	No	No	Drops this database. {ok : 1}
{repairDatabase : 1, preserveClonedFilesOnFailure : boolean, backupOriginalFiles : boolean}	No	No	Yes	Repairs/compacts this database. {ok: 1}

{profile : integer}	No	No	Yes	If passed -1, gets the profiling level, if passed 0-2, sets to profiling level. {was : previousLevel, ok : 1}
{serverStatus : 1}	No	No	Yes	uptime/lock info/memory info - since 1.1.2
{assertinfo : 1}	No	No	Yes	Info on any assertions thrown: {dbasserted : boolean, assert : string, assert : string, assertmsg : string, assertmsg : string, assertuser : string, ok : 1}
{getoptime : 1}	No	No	Yes	{optime : Timestamp, ok : 1}
{opLogging : 1}	No	Yes	Yes	Checks opLogging settings: {ok : 1}
{deleteIndexes : collection_name, index : index_name}	No	No	?	Deletes one or all (if index_name is "*") indexes. {nIndexesWas : N, ok : 1}
{drop : collection_name}	No	No	No	<pre>Drops a collection. {nIndexesWas : N, msg : "all indexes deleted for collection", ns : namespace, "ok" : 1}</pre>
{collstats : collection_name}	No	No	No	Gets statistics about a collection.
{queryTraceLevel : N}	No	Yes	Yes	{ok : 1}
{distinct : collection_name, key : key_name}	No	No	Yes	Finds all distinct values for a given key.

<pre>{filemd5 : object_id, root : file_root}</pre>	No	No	Yes	Finds the MD5 hash of a GridFS file. file_root is the prefix of the files and chunks collections. For example, with the default fs.files and fs.chunks collections, it would be "fs".
--	----	----	-----	---

Mongo Metadata

The system. namespaces in Mongo are special and contain database system information. System collections include:

- system.namespaces lists all namespaces.
- system.indexes lists all indexes.
- Additional namespace / index metadata exists in the database.ns files, and is opaque.
- system.profile stores database profiling information.
- system.users lists users who may access the database.
- local.sources stores replica slave configuration data and state.
- Information on the structure of a stored object is stored within the object itself. See BSON .

There are several restrictions on manipulation of objects in the system collections. Inserting in system.indexes adds an index, but otherwise that table is immutable (the special drop index command updates it for you). system.users is modifiable. system.profile is droppable.

Note: \$ is a reserved character. Do not use it in namespace names or within field names. Internal collections for indexes use the \$ character in their names. These collection store b-tree bucket data and are not in BSON format (thus direct querying is not possible).

Collections

MongoDB collections are essentially named groupings of documents. You can think of them as roughly equivalent to relational database tables.

Details

A MongoDB collection is a collection of BSON documents. These documents are usually have the same structure, but this is not a requirement since MongoDB is a *schema-free* database. You may store a heterogeneous set of documents within a collection, as you do not need predefine the collection's "columns" or fields.

A collection is created when the first document is inserted.

Collection names should begin with letters or an underscore and may include numbers; \$ is reserved. Collections can be organized in namespaces; these are named groups of collections defined using a dot notation. For example, you could define collections blog.posts and blog.authors, both reside under "blog". Note that this is simply an organizational mechanism for the user -- the collection namespace is flat from the database's perspective.

Programmatically, we access these collections using the dot notation. For example, using the mongo shell:

```
if( db.blog.posts.findOne() )
    print("blog.posts exists and is not empty.");
```

See also:

- Capped Collections
- Using a Large Number of Collections

Capped Collections

Capped collections are fixed sized collections that have a very high performance auto-LRU age-out feature (age out is based on insertion order).

In addition, capped collections automatically, with high performance, maintain insertion order for the objects in the collection; this is very powerful for certain use cases such as logging.

Creating a Fixed Size (capped) Collection

Unlike a standard collection, you must explicitly create a capped collection, specifying a collection size in bytes. The collection's data space is then preallocated. Note that the size specified includes database headers.

```
db.createCollection("mycoll", {capped:true, size:100000})
```

Usage and Restrictions

- You may insert new objects in the capped collection.
- You may update the existing objects in the collection. However, the objects must not grow in size. If they do, the update will fail. (There are some possible workarounds which involve pre-padding objects; contact us in the support forums for more information, if help is needed.)
- The database does not allow deleting objects from a capped collection. Use the drop() method to remove all rows from the collection.
 Note: After the drop you must explicitly recreate the collection.
- Maximum size for a capped collection is currently 1e9 bytes on a thirty-two bit machine. The maximum size of a capped collection on a sixty-four bit machine is constrained only by system resources.

Behavior

- Once the size constraint is reached, newly added objects will replace the oldest objects in the collection.
- If you perform a find() on the collection with no ordering specified, the objects will always be returned in insertion order. Reverse order is always retrievable with find().sort({\$natural:-1}).

Applications

- Logging. Capped collections are the preferred logging mechanism for Mongo. Mongo does not use log "files." Rather, log events are stored in the database. Capped collections provide a high-performance means for storing these objects in the database. Inserting objects in an unindexed capped collection will be close to the speed of logging to a filesystem. Additionally, with the built-in LRU mechanism, you are not at risk of using excessive disk space for the logging.
- Caching. If you wish to cache a small number of objects in the database, perhaps cached computations of information, the capped tables provide a convenient mechanism for this. Note that for this application you will likely use an index on the capped table as there will be more reads than writes.
- Auto Archiving. If you know you want data to automatically "roll out" over time as it ages, a capped collection can be an easier way to support than writing manual archival cron scripts.

Recommendations

- For maximum performance, do not create indexes on a capped collection. If the collection will be written to much more than it is read from, it is better to have no indexes. Note that you may create indexes on a capped collection; however, you are then moving from "log speed" inserts to "database speed" inserts -- that is, it will still be quite fast by database standards.
- Use natural ordering to retrieve the most recently inserted elements from the collection efficiently. This is (somewhat) analogous to tail on a log file.

Capping the Number of Objects

You may also cap the number of objects in the collection. Once the limit is reached, items roll out on a least recently inserted basis.

To cap on number of objects, specify a max: parameter on the createCollection() call.

Note: When specifying a cap on the number of objects, you must also cap on size. Be sure to leave enough room for your chosen number of objects or items will roll out faster than expected. You can use the validate() utility method to see how much space an existing collection uses, and from that estimate your size needs.

```
db.createCollection("mycoll", {capped:true, size:100000, max:100});
db.mycoll.validate();
```

Tip: When programming, a handy way to store the most recently generated version of an object can be a collection capped with max=1.

Preallocating space for a normal collection

The createCollection command may be used for non capped collections as well. For example:

```
db.createCollection("mycoll", {size:10000000});
db.createCollection("mycoll", {size:10000000, autoIndexId:false});
```

Explicitly creating a non capped collection via createCollection allows parameters of the new collection to be specified. For example, specification of a collection size causes the corresponding amount of disk space to be preallocated for use by the collection. The autoIndexId field may be set to true or false to explicitly enable or disable automatic creation of a unique key index on the _id object field. By default, such an index is created for non capped collections but is not created for capped collections.

See Also

• The Sorting and Natural Order section of this Guide

Using a Large Number of Collections

A technique one can use with MongoDB in certain situations is to have several collections to store information instead of a single collection. By doing this, certain repeating data no longer needs to be stored in every object, and an index on that key may be eliminated. More importantly for performance (depending on the problem), the data is then clustered by the grouping specified.

For example, suppose we are logging objects/documents to the database, and want to have M logs: perhaps a dev log, a debug log, an ops log, etc. We could store them all in one collection 'logs' containing objects like:

```
{ log : 'dev', ts : ..., info : ... }
```

However, if the number of logs is not too high, it might be better to have a collection per log. We could have a 'logs.dev' collection, a 'logs.debug' collection, 'logs.ops', etc.:

```
// logs.dev:
{ ts : ..., info : ... }
```

Of course, this only makes sense if we do not need to query for items from multiple logs at the same time.

Generally, having a large number of collections has no significant performance penalty, and results in very good performance.

Limits

By default MongoDB has a limit of approximately 24,000 *namespaces* per database. Each collection counts as a namespace, as does each index. Thus if every collection had one index, we can create up to 12,000 collections. Use the --nssize parameter to set a higher limit.

Be aware that there is a certain minimum overhead per collection -- a few KB. Further, any index will require at least 8KB of data space as the b-tree page size is 8KB.

--nssize

If more collections are required, run mongod with the --nssize parameter specified. This will make the <database>.ns file larger and support more collections. Note that --nssize sets the size used for newly created .ns files -- if you have an existing database and wish to resize, after running the db with --nssize, run the db.repairDatabase() command from the shell to adjust the size.

Maximum .ns file size is 2GB.

Data Types and Conventions

MongoDB (BSON) Data Types

Mongo uses special data types in addition to the basic JSON types of string, integer, boolean, double, null, array, and object. These types include date, object id, binary data, regular expression, and code. Each driver implements these types in language-specific ways, see your driver's documentation for details.

See BSON for a full list of database types.

Internationalization

• See Internationalized strings

Database References

• See DB Ref and Schema Design

Internationalized Strings

MongoDB supports UTF-8 for strings in stored objects and queries. (Specifically, BSON strings are UTF-8.)

Generally, drivers for each programming language convert from the language's string format of choice to UTF-8 when serializing and deserializing

BSON. For example, the Java driver converts Java Unicode strings to UTF-8 on serialization.

In most cases this means you can effectively store most international characters in MongoDB strings. A few notes:

- MongoDB regex queries support UTF-8 in the regex string.
- Currently, sort() on a string uses strcmp: sort order will be reasonable but not fully international correct. Future versions of MongoDB
 may support full UTF-8 sort ordering.

Object IDs

Documents in MongoDB are required to have a key, _id, which uniquely identifies them.

- · Document IDs: id
- The BSON ObjectId Datatype
 - BSON ObjectID Specification
 - Document Timestamps

Document IDs: _id

Every MongoDB document has an _id field as its first attribute. This value usually a BSON ObjectId. Such an id must be unique for each member of a collection; this is enforced if the collection has an index on _id, which is the case by default.

If a user tries to insert a document without providing an id, the database will automatically generate an _object id and store it the _id field.

Users are welcome to use their own conventions for creating ids; the _id value may be of any type so long as it is a unique.

The BSON Objected Datatype

Although _id values can be of any type, a special BSON datatype is provided for object ids. This type is a 12-byte binary value designed to have a reasonably high probability of being unique when allocated. All of the officially-supported MongoDB drivers use this type by default for _id values. Also, the Mongo database itself uses this type when assigning _id values on inserts where no _id value is present.

In the MongoDB shell, ObjectId() may be used to create ObjectIds. ObjectId(string) creates an object ID from the specified hex string.

```
> x={ name: "joe" }
{ name : "joe" }
> db.people.save(x)
{ name : "joe" , _id : ObjectId( "47cc67093475061e3d95369d" ) }
> x
{ name : "joe" , _id : ObjectId( "47cc67093475061e3d95369d" ) }
> x
{ name : "joe" , _id : ObjectId( "47cc67093475061e3d95369d" ) }
> db.people.findOne( { _id: ObjectId( "47cc67093475061e3d95369d" ) } )
{ _id : ObjectId( "47cc67093475061e3d95369d" ) , name : "joe" }
> db.people.findOne( { _id: "47cc67093475061e3d95369d" } )
{ _id : ObjectId( "47cc67093475061e3d95369d" ) , name : "joe" }
```

BSON ObjectID Specification

A BSON ObjectID is a 12-byte value consisting of a 4-byte timestamp, a 3-byte machine id, a 2-byte process id, and a 3-byte counter. Here's the schema:

```
0 1 2 3 4 5 6 7 8 9 10 11
time machine pid inc
```

Document Timestamps

One useful consequence of this specification is that it provides documents with a creation timestamp for free. All of the drivers implement methods for extracting these timestamps; see the relevant api docs for details.

DB Ref

DBRef is a specification for creating references between documents.

- Specification
- Using in Different Languages
 - C++
 - Java
 - Javascript (mongo shell)
 - PHP
 - Python

Specification

Database references are references from one document (object) to another within a database. A database reference is a standard embedded object: we are defining a convention, not a special type. By having a standard way to represent, drivers and data frameworks can add helper methods which manipulate the references in standard ways.



This is just one way to store foreign references, not the only way. Currently the only advantage is that it allows optional automatic **client-side** dereferencing with some drivers, although more features may be added later. In many cases, you can just get away with storing the _id as a reference then dereferencing manually. For example, using usernames as _ids:

```
{_id:"scientists", users:[ "aeinstein", "anewton" ]}
```

Syntax for a DBRef reference value is

```
{ $ref : <collname>, $id : <idvalue>[, $db : <dbname>] }
```

where <collname> is the collection name referenced (without the database name), and <idvalue> is the value of the _id field for the object referenced. \$db is optional (currently unsupported by many of the drivers) and allows for references to documents in other databases (specified by <dbname>).



The ordering for DBRefs does matter, fields must be in the order specified above.

The old BSON DBRef datatype is deprecated.

Using in Different Languages

C++

The C++ driver does not yet provide a facility for automatically traversing DBRefs. However one can do it manually of course.

.lava

Java supports DB references using the DBRef class.

Javascript (mongo shell)

Example:

```
> x = \{ name : 'Biology' \}
{ "name" : "Biology" }
> db.courses.save(x)
{ "name" : "Biology", "_id" : ObjectId("4b0552b0f0da7d1eb6f126a1") }
> stu = { name : 'Joe', classes : [ new DBRef('courses', x._id) ] }
// or we could write:
// stu = { name : 'Joe', classes : [ {$ref:'courses',$id:x._id} ] }
> db.students.save(stu)
> stu
{
        "name" : "Joe",
        "classes" : [
                {
                         "$ref" : "courses",
                         "$id" : ObjectId("4b0552b0f0da7d1eb6f126a1")
        "_id" : ObjectId("4b0552e4f0da7d1eb6f126a2")
}
> stu.classes[0]
{ "$ref" : "courses", "$id" : ObjectId("4b0552b0f0da7dleb6f126a1") }
> stu.classes[0].fetch()
  "_id" : ObjectId("4b0552b0f0da7dleb6f126a1"), "name" : "Biology" }
```

PHP

PHP supports DB references with the MongoDBRef class, as well as creation and deferencing methods at the database (MongoDB::createDBRef and MongoDB::getDBRef) and collection (MongoCollection::createDBRef and MongoCollection::getDBRef) levels.

Python

To create a DB reference in python use the pymongo.dbref.DBRef class. You can also use the dereference method on Database instances to make dereferencing easier.

Python also supports auto-ref and auto-deref - check out the auto_reference example.

Ruby

Ruby also supports DB references using the DBRef class and a dereference method on DB instances. For example:

```
@db = Connection.new.db("blog")
@user = @db["users"].save({:name => "Smith"})
@post = @db["posts"].save({:title => "Hello World", :user_id => @user.id})
@ref = DBRef.new("users", @posts.user_id)
assert_equal @user, @db.dereference(@ref)
```

GridFS

GridFS is a specification for storing large files in MongoDB.

The database supports native storage of binary data within BSON objects. However, BSON objects in MongoDB are limited to 4MB in size. Further, even if unlimited, for gigabyte datasets we would want to be able to perform range operations -- such as fetching only the first N bytes of a file.

To facilitate this, a standard is specified for the chunking of files. Each file has a metadata object in a files collection, and one or more chunk objects in a chunks collection. Details of how this is stored can be found in the GridFS Specification; however, you do not really need to read that, instead, just look at the GridFS API in each language's client driver, and also the GridFS tools.

Language Support

Most drivers include GridFS implementations; for languages not listed below, check the driver's API documentation. (If a language does not include support, see the GridFS specification -- implementing a handler is usually quite easy.)

- C++
- Java
- PHP
- Python
- Ruby

Command Line Tools

Command line tools are available to write and read GridFS files from and to the local filesystem.

Indexes

Indexes enhance query performance, often dramatically. It's important to think about the kinds of queries your application will need so that you can define relevant indexes. Once that's done, actually creating the indexes in MongoDB is relatively easy.

- Basics
 - Default Indexes
 - Embedded Keys
 - Documents as Keys
- Compound Keys Indexes
- Unique Indexes
 - Missing Keys
 - Duplicate Values
- Background Index Building
- Dropping Indexes
- ReIndex
- Additional Notes on Indexes
 - Index Performance
 - Using sort() without an Index

Basics

An index is a data structure that collects information about the values of the specified fields in the documents of a collection. This data structure is used by Mongo's query optimizer to quickly sort through and order the documents in a collection. Formally speaking, these indexes are implemented as "B-Tree" indexes.

In the shell, you can create an index by calling the <code>ensureIndex()</code> function, and providing a document that specifies one or more keys to index. Referring back to our *examples* database from Mongo Usage Basics, we can index on the 'j' field as follows:

```
db.things.ensureIndex({j:1});
```

The <code>ensureIndex()</code> function only creates the index if it does not exist.

Once a collection is indexed on a key, random access on query expressions which match the specified key are fast. Without the index, MongoDB has to go through each document checking the value of specified key in the query:

```
db.things.find(\{j:2\}); // fast - uses index db.things.find(\{x:3\}); // slow - has to check all because 'x' isn't indexed
```

Default Indexes

An index is always created on _id. This index is special and cannot be deleted. The _id index enforces uniqueness for its keys.

Embedded Keys

With MongoDB you can even index on a key inside of an embedded document. For example:

```
db.things.ensureIndex({"address.city": 1})
```

Documents as Keys

Indexed fields may be of any type, including documents:

```
db.factories.save( { name: "xyz", metro: { city: "New York", state: "NY" } } );
  db.factories.ensureIndex( { metro : 1 } );
```

You may not as of yet specify a sort order for an object key; thus, at this time, object keys are useful for random access retrieval but less so for ordered access. (For the latter, consider using a compound key index.)

Compound Keys Indexes

In addition to single-key basic indexes, MongoDB also supports multi-key "compound" indexes. Just like basic indexes, you use the <code>ensureIndex()</code> function in the shell to create the index, but instead of specifying only a single key, you can specify several:

```
db.things.ensureIndex({j:1, name:-1});
```

When creating an index, the number associated with a key specifies the direction of the index, so it should always be 1 (ascending) or -1 (descending). Direction doesn't matter for single key indexes or for random access retrieval but is important if you are doing sorts or range queries on compound indexes.

If you have a compound index on multiple fields, you can use it to query on the beginning subset of fields. So if you have an index on

```
a,b,c
```

you can use it query on

```
a,b,c
```

Unique Indexes

MongoDB supports unique indexes, which guarantee that no documents are inserted whose values for the indexed keys match those of an existing document. To create an index that guarantees that no two documents have the same values for both firstname and lastname you would do:

```
db.things.ensureIndex({firstname: 1, lastname: 1}, {unique: true});
```

Missing Keys

When a document is saved to a collection with unique indexes, any missing indexed keys will be inserted with null values. Thus, it won't be possible to insert multiple documents missing the same indexed key.

```
db.things.ensureIndex({firstname: 1}, {unique: true});
  db.things.save({lastname: "Smith"});

  // Next operation will fail because of the unique index on firstname.
  db.things.save({lastname: "Jones"});
```

Duplicate Values

A unique index cannot be created on a key that has duplicate values. If you would like to create the index anyway, keeping the first document the database indexes and deleting all subsequent documents that have duplicate values, add the dropDups option.

```
db.things.ensureIndex({firstname : 1}, {unique : true, dropDups : true})
```

Background Index Building

By default, building an index blocks other database operations. v1.3.2 and higher has a background index build option.

Dropping Indexes

To delete all indexes on the specified collection:

```
db.collection.dropIndexes();
```

To delete a single index:

```
db.collection.dropIndex({x: 1, y: -1})
```

Running directly as a command without helper:

```
// note: command was "deleteIndexes", not "dropIndexes", before MongoDB v1.3.2
// remove index with key pattern {y:1} from collection foo
db.runCommand({dropIndexes:'foo', index : {y:1}})
// remove all indexes:
db.runCommand({dropIndexes:'foo', index : '*'})
```

ReIndex

The relndex command will rebuild all indexes for a collection.

```
db.myCollection.reIndex()
// same as:
db.runCommand( { reIndex : 'myCollection' } )
```

Usually this is unnecessary. You may wish to do this if the size of your collection has changed dramatically or the disk space used by indexes seems oddly large.

Repair database recreates all indexes in the database.

Additional Notes on Indexes

- MongoDB indexes (and string equality tests in general) are case sensitive.
- When you update an object with save(), if the object fits in its previous allocation area, only those indexes whose keys have changed
 are updated. This improves performance. Note that if the object has grown and must move, all index keys must then update, which is
 slower.
- MongoDB supports Multikeys which allows indexing of arrays of values.
- Index information is kept in the system.indexes collection, run db.system.indexes.find() to see example data.

Index Performance

Indexes make retrieval by a key, including ordered sequential retrieval, very fast. Updates by key are faster too as MongoDB can find the document to update very quickly.

However, keep in mind that each index created adds a certain amount of overhead for inserts and deletes. In addition to writing data to the base collection, keys must then be added to the B-Tree indexes. Thus, indexes are best for collections where the number of reads is much greater than the number of writes. For collections which are write-intensive, indexes, in some cases, may be counterproductive. Most collections are read-intensive, so indexes are a good thing in most situations.

Using sort() without an Index

You may use sort() to return data in order without an index if the data set to be returned is small (less than four megabytes). For these cases it is best to use limit() and sort() together.

Indexing as a Background Operation

By default the <code>ensureIndex()</code> operation is blocking, and will stop other operations on the database from proceeding until completed. However, in v1.3.2+, a background indexing option is available.

To build an index in the background, add background: true to your index options. Examples:

```
> db.things.ensureIndex({x:1}, {background:true});
> db.things.ensureIndex({name:1}, {background:true, unique:true, dropDups:true});
```

With background mode enabled, other operations, including writes, will not be obstructed during index creation. The index is not used for queries until the build is complete.

Although the operation is 'background' in the sense that other operations may run concurrently, the command will not return to the shell prompt until completely finished. To do other operations at the same time, open a separate mongo shell instance.

Please note that background mode building uses an incremental approach to building the index which is slower than the default foreground mode: time to build the index will be greater.

While the build progresses, it is possible to see that the operation is still in progress with the db.currentOp() command (will be shown as an insert to system.indexes). You may also use db.killOp() to terminate the build process.

While the build progresses, the index is visible in system.indexes, but it is not used for queries until building completes.

Notes:

- Only one index build at a time is permitted per collection.
- Some administrative operations, such as repairDatabase, are disallowed while a background indexing job is in progress.
- v1.3.2 : alpha (of this feature)
- v1.3.3 : beta
- v1.4 : production

Multikeys

MongoDB provides an interesting "multikey" feature that can automatically index arrays of an object's values. A good example is tagging. Suppose you have an article tagged with some category names:

We can easily perform a query looking for a particular value in the tags array:

```
> db.articles.find( { tags: 'april' } )
{"name" : "Warm Weather" , "author" : "Steve" ,
"tags" : ["weather","hot","record","april"] , "_id" : "497ce4051ca9ca6d3efca323"}
```

Further, we can index on the tags array. Creating an index on an array element indexes results in the database indexing each element of the array:

```
> db.articles.ensureIndex( { tags : 1 } )
true
> db.articles.find( { tags: 'april' } )
{"name" : "Warm Weather" , "author" : "Steve" ,
  "tags" : ["weather", "hot", "record", "april"] , "_id" : "497ce4051ca9ca6d3efca323"}
> db.articles.find( { tags: 'april' } ).explain()
{"cursor" : "BtreeCursor tags_1" , "startKey" : {"tags" : "april"} ,
  "endKey" : {"tags" : "april"} , "nscanned" : 1 , "n" : 1 , "millis" : 0 }
```

Embedded object fields in an array

Additionally the same technique can be used for fields in embedded objects:

Querying on all values in a given set

By using the \$all query option, a set of values may be supplied each of which must be present in a matching object field. For example:

```
> db.articles.find( { tags: { $all: [ 'april', 'record' ] } } )
{"name" : "Warm Weather" , "author" : "Steve" ,
   "tags" : ["weather","hot","record","april"] , "_id" : "497ce4051ca9ca6d3efca323"}
> db.articles.find( { tags: { $all: [ 'april', 'june' ] } } )
> // no matches
```

See Also

The Multikeys section of the Full Text Search in Mongo document for information about this feature.

Inserting

When we insert data into MongoDB, that data will always be in document-form. Documents are data structure analogous to JSON, Python dictionaries, and Ruby hashes, to take just a few examples. Here, we discuss more about document-orientation and describe how to insert data into MongoDB.

- Document-Orientation
- JSON
- Mongo-Friendly Schema
 - Store Example

Document-Orientation

When we describe MongoDB as "document-oriented", we mean it's in the class of databases for which the primary storage unit is a collection possibly structured - of data, most likely as key/value pairs.

Some examples of document formats are JSON, XML, and simple sets of key/value pairs.

The documents stored in Mongo DB are JSON-like. To be efficient, the database uses a format called BSON which is a binary representation of this data. The goal is a format that is both compact and reasonably fast to scan.

Client drivers serialize data to BSON, then transmit the data over the wire to the db. Data is stored on disk in BSON format. Thus, on a retrieval, the database does very little translation to send an object out, allowing high efficiency. The client driver unserialized a received BSON object to its native language format.

JSON

For example the following "document" can be stored in Mongo DB:

This document is a blog post, so we can store in a "posts" collection using the shell:

```
> doc = { author : 'joe', created : new Date('03-28-2009'), ... }
> db.posts.insert(doc);
```

MongoDB understands the internals of BSON objects -- not only can it store them, it can query on internal fields and index keys based upon them. For example the query

```
> db.posts.find( { "comments.author" : "jim" } )
```

is possible and means "find any blog post where at least one comment subjobject has author == 'jim'".

Mongo-Friendly Schema

Mongo can be used in many ways, and one's first instincts when using it are probably going to be similar to how one would write an application with a relational database. While this work pretty well, it doesn't harness the real power of Mongo. Mongo is designed for and works best with a rich object model.

Store Example

If you're building a simple online store that sells products with a relation database, you might have a schema like:

```
item
title
price
sku
item_features
sku
feature_name
feature_value
```

You would probably normalize it like this because different items would have different features, and you wouldn't want a table with all possible features. You could model this the same way in mongo, but it would be much more efficient to do

```
item : {
          "title" : <title> ,
          "price" : <price> ,
          "sku" : <sku> ,
          "features" : {
                "optical zoom" : <value> ,
                ...
          }
}
```

This does a few nice things

- · you can load an entire item with 1 dbquery
- all the data for an item is on the same place on disk, so its only 1 seek to load it all

Now, at first glance there might seem to be some issues, but we've got them covered.

• you might want to insert or update a single feature. mongo lets you operate on embedded files like:

```
db.items.update( { sku : 123 } , { "$set" : { "features.zoom" : "5" } } )
```

 Does adding a feature requires moving the entire object on disk? No. mongo has a padding heuristic that adapts to your data so it will leave some empty space for the object to grow. This will prevent indexes from being changed, etc.

Legal Key Names

Key names in inserted documents are limited as follows:

- The '\$' character must not be the first character in the key name.
- The '.' character must not appear anywhere in the key name.

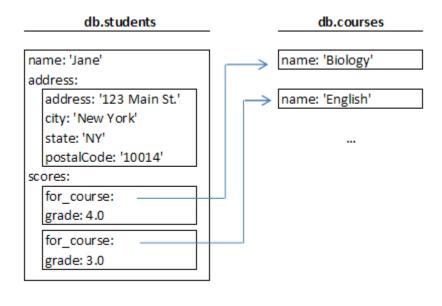
Schema Design

- Introduction
- Embed vs. Reference
- Use Cases
- Index Selection
- How Many Collections?
- See Also

Introduction

With Mongo, you do less "normalization" than you would perform designing a relational schema because there are no server-side "joins". Generally, you will want one database collection for each of your top level objects.

You do not want a collection for every "class" - instead, embed objects. For example, in the diagram below, we have two collections, students and courses. The student documents embed address documents and the "score" documents, which have references to the courses.



Compare this with a relational schema, where you would almost certainly put the scores in a separate table, and have a foreign-key relationship back to the students.

Embed vs. Reference

The key question in Mongo schema design is "does this object merit its own collection, or rather should it embed in objects in other collections?" In relational databases, each sub-item of interest typically becomes a separate table (unless denormalizing for performance). In Mongo, this is not recommended - embedding objects is much more efficient. Data is then colocated on disk; client-server turnarounds to the database are eliminated. So in general the question to ask is, "why would I not want to embed this object?"

So why are references slow? Let's consider our students example. If we have a student object and perform:

```
print( student.address.city );
```

This operation will always be fast as address is an embedded object, and is always in RAM if student is in RAM. However for

```
print( student.scores[0].for_course.name );
```

if this is the first access to scores[0], the shell or your driver must execute the query

```
// pseudocode for driver or framework, not user code
student.scores[0].for_course = db.courses.findOne({_id:_course_id_to_find_});
```

Thus, each reference traversal is a query to the database. Typically, the collection in question is indexed on _id. The query will then be reasonably fast. However, even if all data is in RAM, there is a certain latency given the client/server communication from appserver to database. In general, expect 1ms of time for such a query on a ram cache hit. Thus if we were iterating 1,000 students, looking up one reference per student would be quite slow - over 1 second to perform even if cached. However, if we only need to look up a single item, the time is on the order of 1ms, and completely acceptable for a web page load. (Note that if already in db cache, pulling the 1,000 students might actually take much less than 1 second, as the results return from the database in large batches.)

Some general rules on when to embed, and when to reference:

- "First class" objects, that are at top level, typically have their own collection.
- Line item detail objects typically are embedded.
- Objects which follow an object modelling "contains" relationship should generally be embedded.
- Many to many relationships are generally by reference.
- Collections with only a few objects may safely exist as separate collections, as the whole collection is quickly cached in application server memory.
- Embedded objects are harder to reference than "top level" objects in collections, as you cannot have a DBRef to an embedded object (at least not yet).
- It is more difficult to get a system-level view for embedded objects. For example, it would be easier to query the top 100 scores across all students if Scores were not embedded.
- If the amount of data to embed is huge (many megabytes), you may read the limit on size of a single object.
- If performance is an issue, embed.

Use Cases

Let's consider a few use cases now.

- 1. Customer / Order / Order Line-Item
- orders should be a collection. customers a collection. line-items should be an array of line-items embedded in the order object.
- 1. Blogging system.
- posts should be a collection. post author might be a separate collection, or simply a field within posts if only an email address. comments should be embedded objects within a post for performance.

Index Selection

A second aspect of schema design is index selection. As a general rule, where you want an index in a relational database, you want an index in Mongo.

- \bullet $\,$ The $\,$ _id field is automatically indexed.
- · Fields upon which keys are looked up should be indexed.
- Sort fields generally should be indexed.

The MongoDB profiling facility provides useful information for where an index should be added that is missing.

Note that adding an index slows writes to a collection, but not reads. Use lots of indexes for collections with a high read: write ratio (assuming one does not mind the storage overage). For collections with more writes than reads, indexes are very expensive.

How Many Collections?

As Mongo collections are polymorphic, one could have a collection objects and put everything in it! This approach is taken by some object databases. For performance reasons, we do not recommend this approach. Data within a Mongo collection tends to be contiguous on disk. Thus, table scans of the collection are possible, and efficient. Collections are very important for high throughput batch processing.

See Also

- DBRef
- Trees in MongoDB
- MongoDB Data Modeling and Rails Next: Advanced Queries

Trees in MongoDB

Work in Progress

- Basics
- Styles
 - Full Tree in Single Document
 - Full Path in Each Node
 - Parent Links
 - Child Links
- Separating Structure from Data
- Thanks

Basics

There are many ways to represent trees in MongoDB. Just as there is no one-size-fits-all database, there is no one-size-fits-all way to represent trees. I'll present a few methods that should cover most needs.

Styles

Full Tree in Single Document

Pros:

- · Single document to fetch per page
- One location on disk for whole tree
- You can see full structure easily

Cons:

- · Hard to search for details
- Can get unwieldy if you need a huge tree (there is a 4MB per doc limit)

You could also store this as a flat list with a "depth" field to mark nesting. This will help with searching, but will require fetching and replacing the full list whenever you want to add a child node.

Full Path in Each Node

Parent Links

Child Links

Separating Structure from Data

Thanks

Thanks to Sean Cribbs for this blog post which is the source for much of this page.

Optimization

- Optimizing A Simple Example
 - Optimization #1: Create an index
 - Optimization #2: Limit results
 - Optimization #3: Select only relevant fields
- Using the Profiler
- Optimizing Statements that Use count()

- Increment Operations
- Circular Fixed Size Collections
- Server Side Code Execution
- Explain
- Hint
- See Also

Optimizing A Simple Example

This section describes proper techniques for optimizing database performance.

Let's consider an example. Suppose our task is to display the front page of a blog - we wish to display headlines of the 10 most recent posts. Let's assume the posts have a timestamp field ts.

The simplest thing we could write might be:

```
articles = db.posts.find().sort({ts:-1}); // get blog posts in reverse time order

for (var i=0; i< 10; i++) {
    print(articles[i].getSummary());
}</pre>
```

Optimization #1: Create an index

Our first optimization should be to create an index on the key that is being used for the sorting:

```
db.posts.ensureIndex({ts:1});
```

With an index, the database is able to sort based on index information, rather than having to check each document in the collection directly. This is much faster.

Optimization #2: Limit results

MongoDB cursors return results in groups of documents that we'll call 'chunks'. The chunk returned might contain more than 10 objects - in some cases, much more. These extra objects are a waste of network transmission and resources both on the app server and the database.

As we know how many results we want, and that we do not want all the results, we can use the limit() method for our second optimization.

```
articles = db.posts.find().sort({ts:-1}).limit(10); // 10 results maximum
```

Now, we'll only get 10 results returned to client.

Optimization #3: Select only relevant fields

The blog post object may be very large, with the post text and comments embedded. Much better performance will be achieved by selecting only the fields we need:

```
articles = db.posts.find({}, {ts:1,title:1,author:1,abstract:1}).sort({ts:-1}).limit(10);
articles.forEach( function(post) { print(post.getSummary()); } );
```

The above code assumes that the getSummary() method only references the fields listed in the find() method.

Note if you fetch only select fields, you have a partial object. An object in that form cannot be updated back to the database:

```
a_post = db.posts.findOne({}, Post.summaryFields);
a_post.x = 3;
db.posts.save(a_post); // error, exception thrown
```

Using the Profiler

MongoDB includes a database profiler which shows performance characteristics of each operation against the database. Using the profiler you can find queries (and write operations) which are slower than they should be; use this information, for example, to determine when an index is needed. See the Performance Tuning section of the Mongo Developers' Guide for more information.

Optimizing Statements that Use count()

To speed operations that rely on count (), create an index on the field involved in the count guery expression.

```
db.posts.ensureIndex({author:1});
db.posts.find({author:"george"}).count();
```

Increment Operations

MongoDB supports simple object field increment operations; basically, this is an operation indicating "increment this field in this document at the server". This can be much faster than fetching the document, updating the field, and then saving it back to the server and are particularly useful for implementing real time counters. See the Updates section of the Mongo Developers' Guide for more information.

Circular Fixed Size Collections

MongoDB provides a special circular collection type that is pre-allocated at a specific size. These collections keep the items within well-ordered even without an index, and provide very high-speed writes and reads to the collection. Originally designed for keeping log files - log events are stored in the database in a circular fixed size collection - there are many uses for this feature. See the Capped Collections section of the Mongo Developers' Guide for more information.

Server Side Code Execution

Occasionally, for maximal performance, you may wish to perform an operation in process on the database server to eliminate client/server network turnarounds. These operations are covered in the Server-Side Processing section of the Mongo Developers' Guide.

Explain

A great way to get more information on the performance of your database queries is to use the \$explain feature. This will display "explain plan" type info about a query from the database.

When using the mongo - The Interactive Shell, you can find out this "explain plan" via the explain() function called on a cursor. The result will be a document that contains the "explain plan".

```
db.collection.find(query).explain();
```

provides information such as the following:

```
{
    "cursor" : "BasicCursor",
    "nscanned" : 57594.0,
    "n" : 3.0 ,
    "millis" : 108.0
}
```

This will tell you the type of cursor used (BtreeCursor is another type), the number of records the DB had to examine as part of this query, the number of records returned by the query, and the time in milliseconds the query took to execute.

Hint

While the mongo query optimizer often performs very well, explicit "hints" can be used to force mongo to use a specified index, potentially improving performance in some situations. When you have a collection indexed and are querying on multiple fields (and some of those fields are indexed), pass the indexe as a hint to the query. You can do this in two different ways. You may either set it per query, or set it for the entire collection.

To set the hint for a particular query, call the hint() function on the cursor before accessing any data, and specify a document with the key to be used in the query:

```
db.collection.find({user:u, foo:d}).hint({user:1});
```



Be sure to Index

For the above hints to work, you need to have run <code>ensureIndex()</code> to index the collection on the user field.

See Also

- · Optimizing Storage of Small Objects
- Query Optimizer
- currentOp()

Optimizing Storage of Small Objects

MongoDB records have a certain amount of overhead per object (BSON document) in a collection. This overhead is normally insignificant, but if your objects are tiny (just a few bytes, maybe one or two fields) it would not be. Below are some suggestions on how to optimize storage efficiently in such situations.

Using the _id Field Explicitly

Mongo automatically adds an object ID to each document and sets it to a unique value. Additionally this field in indexed. For tiny objects this takes up significant space.

The best way to optimize for this is to use _id explicitly. Take one of your fields which is unique for the collection and store its values in _id. By doing so, you have explicitly provided IDs. This will effectively eliminate the creation of a separate _id field. If your previously separate field was indexed, this eliminates an extra index too.

Using Small Field Names

Consider a record

```
{ last_name : "Smith", best_score: 3.9 }
```

The strings "last_name" and "best_score" will be stored in each object's BSON. Using shorter strings would save space:

```
{ lname : "Smith", score : 3.9 }
```

Would save 9 bytes per document. This of course reduces expressiveness to the programmer and is not recommended unless you have a collection where this is of significant concern.

Field names are not stored in indexes as indexes have a predefined structure. Thus, shortening field names will not help the size of indexes. In general it is not necessary to use short field names.

Combining Objects

Fundamentally, there is a certain amount of overhead per document in MongoDB. One technique is combining objects. In some cases you may be able to embed objects in other objects, perhaps as arrays of objects. If your objects are tiny this may work well, but will only make sense for certain use cases.

Query Optimizer

The MongoDB query optimizer generates query plans for each query submitted by a client. These plans are executed to return results.

The database uses an interesting approach to query optimization. Traditional approaches (which tend to be cost-based and statistical) are not used, as these approaches have a couple of problems.

First, the optimizer might consistently pick a bad query plan. For example, there might be correlations in the data of which the optimizer is unaware. In a situation like this, the developer might use a query hint.

Also with the traditional approach, query plans can change in production with negative results. No one thinks rolling out new code without testing is a good idea. Yet often in a production system a query plan can change as the statistics in the database change on the underlying data. The query plan in effect may be a plan that never was invoked in QA. If it is slower than it should be, the application could experience an outage.

The Mongo query optimizer is different. It is not cost based -- it does not model the cost of various queries. Instead, the optimizer simply tries different query plans and learn which ones work well. Of course, when the system tries a really bad plan, it may take an extremely long time to run. To solve this, when testing new plans, MongoDB executes multiple query plans in parallel. As soon as one finishes, it terminates the other executions, and the system has learned which plan is good. This works particularly well given the system is non-relational, which makes the space of possible query plans much smaller (as there are no joins).

Sometimes a plan which was working well can work poorly -- for example if the data in the database has changed, or if the parameter values to the query are different. In this case, if the query seems to be taking longer than usual, the database will once again run the query in parallel to try different plans.

This approach adds a little overhead, but has the advantage of being much better at worst-case performance.

Querying

MongoDB features a rich language for performing dynamic queries.

Query Selectors

MongoDB supports a number of query expressions for fetching data. Queries are expressed in a JSON-style notation and transmitted to the database server as BSON. Queries on document keys include the ability to match against both embedded objects and arrays within a document.

The database's query optimizer analyzes each query and generates an appropriate plan, using indexes when available.

Cursors

Database queries, performed with the find() method, return a *cursor*. Cursors are then used to iteratively retrieve all the documents returned by the query. For example, in the mongo shell:

```
> var cur = db.example.find();
> cur.forEach( function(x) { print(tojson(x))});
{"n" : 1 , "_id" : "497ce96f395f2f052a494fd4"}
{"n" : 2 , "_id" : "497ce971395f2f052a494fd5"}
{"n" : 3 , "_id" : "497ce973395f2f052a494fd6"}
>
```

See Also

- Queries and Cursors
- Advanced Queries
- Query Optimizer

Advanced Queries

- Introduction
- Conditional Operators : <, <=, >, >=
- Conditional Operator : \$ne
- Conditional Operator : \$in
- Conditional Operator : \$nin
- Conditional Operator : \$mod
- Conditional Operator: \$all
- Conditional Operator : \$size
- · Conditional Operator : \$exists
- Regular Expressions
- Value in an Array
- Conditional Operator: \$elemMatch
- Value in an Embedded Object
- Full language expressions and \$where
- sort()
- limit()
- skip()
- snapshot()
- count()
- group()

Introduction

MongoDB offers a rich query environment with lots of features. This page lists some of those features.

Queries in MongoDB are represented as JSON-style objects, very much like the documents we actually store in the database. For example:

```
db.things.find( \{ \ x : 3, \ y : "foo" \ \} ); // i.e., select * from things where x==3 and y=="foo"
```

Note that any of the operators on this page can be combined in the same query document. For example, to find all document where j is not equal to 3 and k is greater than 10, you'd query like so:

```
db.things.find({j: {$ne: 3}, k: {$gt: 10} });
```

Conditional Operators: <, <=, >, >=

Use these special forms for greater than and less than comparisons in queries, since they have to be represented in the query document:

```
db.collection.find({ "field" : { $gt: value } } );  // greater than : field > value
db.collection.find({ "field" : { $lt: value } } );  // less than : field < value
db.collection.find({ "field" : { $gte: value } } );  // greater than or equal to : field >= value
db.collection.find({ "field" : { $lte: value } } );  // less than or equal to : field <= value</pre>
```

For example:

```
db.things.find({j : {$lt: 3}});
db.things.find({j : {$gte: 4}});
```

You can also combine these operators to specify ranges:

```
db.collection.find({ "field" : { $gt: value1, $lt: value2 } } ); // value1 < field < value
```

Conditional Operator: \$ne

Use \$ne for "not equals".

```
db.things.find( { x : { $ne : 3 } } );
```

Conditional Operator: \$in

The \$in operator is analogous to the SQL IN modifier, allowing you to specify an array of possible matches.

```
db.collection.find( { "field" : { $in : array } } );
```

Let's consider a couple of examples. From our *things* collection, we could choose to get a subset of documents based upon the value of the 'j' key:

```
db.things.find({j:{$in: [2,4,6]}});
```

Suppose the collection updates is a list of social network style news items; we want to see the 10 most recent updates from our friends. We might invoke:

```
db.updates.ensureIndex( { ts : 1 } ); // ts == timestamp
var myFriends = myUserObject.friends; // let's assume this gives us an array of DBRef's of my friends
var latestUpdatesForMe = db.updates.find( { user : { $in : myFriends } } ).sort( { ts : -1 }
).limit(10);
```

Conditional Operator: \$nin

The \$nin operator is similar to \$in except that it selects objects for which the specified field does not have any value in the specified array. For example

```
db.things.find({j:{$nin: [2,4,6]}});
```

would match $\{j:1,b:2\}$ but not $\{j:2,c:9\}$.

Conditional Operator: \$mod

The \$mod operator allows you to do fast modulo queries to replace a common case for where clauses. For example, the following \$where query:

```
db.things.find( "this.a % 10 == 1")
```

can be replaced by:

```
db.things.find( { a : { $mod : [ 10 , 1 ] } } )
```

Conditional Operator: \$all

The \$all operator is similar to \$in, but instead of matching any value in the specified array all values in the array must be matched. For example, the object

```
{ a: [ 1, 2, 3 ] }
```

would be matched by

```
db.things.find( { a: { $all: [ 2, 3 ] } );
```

but not

```
db.things.find( { a: { $all: [ 2, 3, 4 ] } } );
```

Conditional Operator: \$size

The \$size operator allows specification of the size of an array. The following example would match the object {a:[99]}:

```
db.things.find( { a : { $size: 1 } } );
```

Conditional Operator: \$exists

Check for existence (or lack thereof) of a field.

```
db.things.find( { a : { $exists : true } } ); // return object if a is present
db.things.find( { a : { $exists : false } } ); // return if a is missi
```

Regular Expressions

You may use regular expressions in database query expressions:

```
db.customers.find( { name : /acme.*corp/i } );
```

For regular expressions of the form /^''normalchars''.*/, the database will use an index when available and appropriate (much like most SQL databases that use indexes for a LIKE 'foo%' expression).

Value in an Array

To look for the value "red" in an array field colors:

```
db.things.find( { colors : "red" } );
```

That is, when "colors" is inspected, if it is an array, each value in the array is checked. This technique may be mixed with the embedded object technique below.

Conditional Operator: \$elemMatch

Version 1.3.1 and higher.

Use \$elemMatch to check if an element in an array matches the specified match expression.

Note that a single array element must match all the criteria specified; thus, the following query is semantically different in that each criteria can match a different element in the x array:

```
> t.find( { "x.a" : 1, "x.b" : { $gt : 1 } } )
```

Value in an Embedded Object

For example, to look author.name=="joe" in a postings collection with embedded author objects:

```
db.postings.find( { "author.name" : "joe" } );
```

Full language expressions and \$where

In addition to the structured query syntax shown so far, you may write query "where" expressions in the style of a SQL-like "WHERE" predicate. To do so, pass a string containing a Javascript expression to find(), or assign such a string to the query object member \$where. The database will evaluate this expression for each object scanned. When the result is true, the object is returned in the query results.

For example, the following three statements all do the same thing:

```
db.myCollection.find( { a : { $gt: 3 } } );
  db.myCollection.find( { $where: "this.a > 3" } );
  db.myCollection.find("this.a > 3");
```

See the server-side processing page for more information.

sort()

sort() is analogous to the ORDER BY statement in SQL - it requests that items be returned in a particular order. We pass sort() a key pattern which indicates the desired order for the result.

```
db.myCollection.find().sort( { ts : -1 } ); // sort by ts, descending order
```

sort() may be combined with the limit() function. In fact, if you do not have a relevant index for the specified key pattern, limit() is recommended as there is a limit on the size of sorted results when an index is not used.

limit()

limit() is analogous to the LIMIT statement in MySQL: it specifies a maximum number of results to return. For best performance, use limit() whenever possible. Otherwise, the database may return more objects than are required for processing.

```
db.students.find().limit(10).forEach( function(student) { print(student.name + ""); } );
```

skip()

The skip() expression allows one to specify at which object the database should begin returning results. This is often useful for implementing "paging". Here's an example of how it might be used in a JavaScript application:

```
function printStudents(pageNumber, nPerPage) {
   print("Page: " + pageNumber);
   db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach( function(student) {
   print(student.name + ""); } );
}
```

snapshot()

Indicates use of snapshot mode for the query. Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query's execution (even if the object were updated). If an object is new during the query, or deleted during the query, it may or may not be returned, even with snapshot mode.

Note that short guery responses (less than 1MB) are always effectively snapshotted.

Currently, snapshot mode may not be used with sorting or explicit hints.

count()

The count () method returns the number of objects matching the query specified. It is specially optimized to perform the count in the MongoDB server, rather than on the client side for speed and efficiency:

```
nstudents = db.students.find({'address.state' : 'CA'}).count();
```

Note that you can achieve the same result with the following, but the following is slow and inefficient as it requires all documents to be put into memory on the client, and then counted. Don't do this:

```
nstudents = db.students.find(\{'address.state' : 'CA'\}).toArray().length; // VERY BAD: slow and uses \\ excess memory
```

group()

The <code>group()</code> method is analogous to GROUP BY in SQL. <code>group()</code> is more flexible, actually, allowing the specification of arbitrary reduction operations. See the Aggregation section of the Mongo Developers' Guide for more information.

Dot Notation (Reaching into Objects)

MongoDB is designed for store JSON-style objects. The database understands the structure of these objects and can reach into them to evaluate query expressions.

Let's suppose we have some objects of the form:

```
> db.persons.findOne()
{ name: "Joe", address: { city: "San Francisco", state: "CA" } ,
likes: [ 'scuba', 'math', 'literature' ] }
```

Querying on a top-level field is straightforward enough using Mongo's JSON-style query objects:

```
> db.persons.find( { name : "Joe" } )
```

But what about when we need to reach into embedded objects and arrays? This involves a bit different way of thinking about queries than one would do in a traditional relational DBMS. To reach into embedded objects, we use a "dot notation":

```
> db.persons.find( { "address.state" : "CA" } )
```

Reaching into arrays is implicit: if the field being queried is an array, the database automatically assumes the caller intends to look for a value within the array:

```
> db.persons.find( { likes : "math" } )
```

We can mix these styles too, as in this more complex example:

We can also create indexes of keys on these fields:

```
db.persons.ensureIndex( { "address.state" : 1 } );
db.blogposts.ensureIndex( { "comments.by" : 1 } );
```

Dot Notation vs. Subobjects

Suppose there is an author id, as well as name. To store the author field, we can use an object:

```
> db.blog.save({ title : "My First Post", author: {name : "Jane", id : 1}})
```

If we want to find any authors named Jane, we use the notation above:

```
> db.blog.findOne({"author.name" : "Jane"})
```

To match only objects with these exact keys and values, we use an object:

```
db.blog.findOne({"author" : {"name" : "Jane", "id" : 1}})
```

Note that

```
db.blog.findOne({"author" : {"name" : "Jane"}})
```

will not match, as subobjects have to match exactly (it would match an object with one field: {"name" : "Jane"}).

Array Element by Position

Array elements also may be accessed by specific array position:

```
// i.e. comments[0].by == "Abe"
> db.blogposts.find( { "comments.0.by" : "Abe" } )
```

(The above examples use the mongo shell's Javascript syntax. The same operations can be done in any language for which Mongo has a driver available.)

Full Text Search in Mongo

- Introduction
- · Multikeys (Indexing Values in an Array)
- Text Search
- Comparison to Full Text Search Engines
- Real World Examples

Introduction

Mongo provides some functionality that is useful for text search and tagging.

Multikeys (Indexing Values in an Array)

The Mongo multikey feature can automatically index arrays of values. Tagging is a good example of where this feature is useful. Suppose you

have an article object/document which is tagged with some category names:

```
obj = {
   name: "Apollo",
   text: "Some text about Apollo moon landings",
   tags: [ "moon", "apollo", "spaceflight" ]
}
```

and that this object is stored in db.articles. The command

```
db.articles.ensureIndex( { tags: 1 } );
```

will index all the tags on the document, and create index entries for "moon", "apollo" and "spaceflight" for that document.

You may then query on these items in the usual way:

```
> print(db.articles.findOne( { tags: "apollo" } ).name);
Apollo
```

The database creates an index entry for each item in the array. Note an array with many elements (hundreds or thousands) can make inserts very expensive. (Although for the example above, alternate implementations are equally expensive.)

Text Search

It is fairly easy to implement basic full text search using multikeys. What we recommend is having a field that has all of the keywords in it, something like:

```
{ title : "this is fun" ,
_keywords : [ "this" , "is" , "fun" ]
}
```

Your code must split the title above into the keywords before saving. Note that this code (which is not part of Mongo DB) could do stemming, etc. too. (Perhaps someone in the community would like to write a standard module that does this...)

Comparison to Full Text Search Engines

MongoDB has interesting functionality that makes certain search functions easy. That said, it is not a dedicated full text search engine.

For example, dedicated engines provide the following capabilities:

- built-in text stemming
- ranking of queries matching various numbers of terms (can be done with MongoDB, but requires user supplied code to do so)
- bulk index building

Bulk index building makes building indexes fast, but has the downside of not being realtime. MongoDB is particularly well suited for problems where the search should be done in realtime. Traditional tools are often not good for this use case.

Real World Examples

The Business Insider web site uses MongoDB for its blog search function in production.

Mark Watson's opinions on Java, Ruby, Lisp, AI, and the Semantic Web - A recipe example in Ruby.

min and max Query Specifiers

The $\min()$ and $\max()$ functions may be used in conjunction with an index to constrain query matches to those having index keys between the min and max keys specified. The $\min()$ and $\max()$ functions may be used individually or in conjunction. The index to be used may be specified with a $\operatorname{hint}()$ or one may be inferred from pattern of the keys passed to $\min()$ and/or $\max()$.

```
db.f.find().min({name:"barry"}}.max({name:"larry"}).hint({name:1});
  db.f.find().min({name:"barry"}}.max({name:"larry"});
  db.f.find().min({last_name:"smith",first_name:"john"}};
```

The min() value is included in the range and the max() value is excluded.

Normally, it is much preferred to use \$gte and \$It rather than to use min and max, as min and max require a corresponding index. Min and max are primarily useful for compound keys: it is difficult to express the last example above without this feature (it can be done using \$where).

min and max exist primarily to support the mongos (sharding) process.

OR operations in query expressions

Query objects in Mongo by default AND expressions together. Mongo currently does not include an OR operator for such queries, however there are ways to express such queries.

\$in

The \sin operator indicates a "where value in ..." expression. For expressions of the form x == a OR x == b, this can be represented as

```
{ x : { $in : [ a, b ] } }
```

\$where

We can provide arbitrary Javascript expressiosn to the server via the \$where operator. This provides a means to perform OR operations. For example in the mongo shell one might invoke:

```
db.mycollection.find( \{ $where : function() \{ return this.a == 3 || this.b == 4; \} \} );
```

The following syntax is briefer and also works; however, if additional structured query components are present, you will need the \$where form:

```
db.mycollection.find( function() { return this.a == 3 || this.b == 4; } );
```

See Also

Advanced Queries

Queries and Cursors

Modes

The find() method on a collection returns a "cursor," which is an object that allows you to iterate through the documents returned for a query.

Cursor Mode

In cursor mode, use hasNext() and next() methods.

```
for( var c = db.parts.find(); c.hasNext(); ) {
   print( c.next());
}
```

In the dbshell, for Each() may be used with a cursor:

```
db.users.find().forEach( function(u) { print("user: " + u.name); } );
```

Array Mode

Note that in some languages, like JavaScript, the driver supports an "array mode". Please check your driver documentation for specifics.

In the db shell, to use the cursor in array mode, use array index [] operations and the length property.

Array mode will load all data into RAM up to the highest index requested. Thus it should not be used for any query which can return very large

amounts of data: you will run out of memory.

You may also call toArray() on a cursor. toArray() will load all objects queries into RAM.

Getting a Single Item

Use findOne() to fetch a single item. Null is returned if no item is found.

findOne() is equivalent in functionality to:

```
function findOne(coll, query) {     var cursor = coll.find(query).limit(1);     return cursor.hasNext() ?
    cursor.next() : null; }
```

Tip: If you only need one row back and multiple match, findOne() is efficient, as it performs the limit() operation, which limits the objects returned from the database to one.

Querying Embedded Objects

To find an exact match of an entire embedded object, simply query for that object:

```
db.order.find( { shipping: { carrier: "usps" } } );
```

The above query will work if { carrier: "usps" } is an exact match for the entire contained shipping object. If you wish to match any subobject with shipping.carrier == "usps", use this syntax:

```
db.order.find( { "shipping.carrier" : "usps" } );
```

Greater Than / Less Than

```
db.myCollection.find( { a : { $gt : 3 } } );
db.myCollection.find( { a : { $gt e : 3 } } );
db.myCollection.find( { a : { $lt : 3 } } );
db.myCollection.find( { a : { $lt e : 3 } } ); // a <= 3</pre>
```

Latent Cursors and Snapshotting

A latent cursor has (in addition to an initial access) a latent access that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete).

Under most circumstances, the database supports these operations.

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

Mongo DB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations or not.

Auditing allocated cursors

Information on allocated cursors may be obtained using the {cursorInfo:1} command.

See Also

- Advanced Queries
- Multikeys in the HowTo

Tailable Cursors

MongoDB has a feature known as tailable cursors which are similar to the Unix "tail -f" command.

Tailable means cursor is not closed when the last data is retrieved. rather, the cursor marks the final object's position. you can resume using the cursor later, from where it was located, if more data was received.

Like any "latent cursor", the cursor may become invalid at some point -- for example if the final object it references were deleted. Thus, you should be prepared to requery if the cursor is dead.



If the field you wish to "tail" is indexed, simply requerying for "field > value" is already quite efficient. Tailable will be slightly faster in situations such as that. However, if the field is not indexed, tailable provides a huge improvement in performance.

Mongo replication uses this feature to follow the end of the master server's replication op log collection -- the tailable feature eliminates the need to create an index for the oplog at the master, which would slow log writes.

C++ example:

```
#include "client/dbclient.h"
#include "util/goodies.h"
using namespace mongo;
/* "tail" the namespace, outputting elements as they are added.
   For this to work _id values should be increasing when items are
   added.
*/
void tail(DBClientBase& conn, const char *ns) {
  BSONElement lastId = minKey.firstElement();
  Query query = Query().sort("_id");
  while( 1 ) {
    auto_ptr<DBClientCursor> c =
      conn.query(ns, query, 0, 0, 0, Option_CursorTailable);
    while( 1 ) {
     if( !c->more() ) {
 if( c->isDead() ) {
   // we need to requery
 sleepsecs(1); // all data (so far) exhausted, wait for more
}
      BSONObj o = c->next();
      lastId = o["_id"];
      cout << o.toString() << endl;</pre>
    // prepare to requery
query = QUERY( "_id" << GT << lastId ).sort("_id");</pre>
}
```

Server-side Code Execution

- Introduction
- \$where Clauses and Functions in Queries
 - Restrictions
- Using db.eval()
 - Examples
- Storing functions server-side
- Map/Reduce
- Notes on Concurrency

Introduction

Mongo supports the execution of code inside the database process.

\$where Clauses and Functions in Queries

In addition to the regular document-style query specification for find() operations, you can also express the query either as a string containing a SQL-style WHERE predicate clause, or a full JavaScript function.

When using this mode of query, the database will call your function, or evaluate your predicate clause, for each object in the collection.

In the case of the string, you must represent the object as "this" (see example below). In the case of a full JavaScript function, you use the normal

JavaScript function syntax.

The following four statements in mongo - The Interactive Shell are equivalent:

```
db.myCollection.find( { a : { $gt: 3 } } );
db.myCollection.find( { $where: "this.a > 3" });
db.myCollection.find( "this.a > 3" );
db.myCollection.find( { $where: function() { return this.a > 3;}});
```

The first statement is the preferred form. It will be at least slightly faster to execute because the query optimizer can easily interpret that query and choose an index to use.

You may mix data-style find conditions and a function. This can be advantageous for performance because the data-style expression will be evaluated first, and if not matched, no further evaluation is required. Additionally, the database can then consider using an index for that condition's field. To mix forms, pass your evaluation function as the \$where field of the query object. For example:

```
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < 0; } );</pre>
```

You may mix data-style find conditions and a function. This can be advantageous for performance because the data-style expression will be evaluated first, and if not matched, no further evaluation is required. Additionally, the database can then consider using an index for that condition's field. For example:

```
db.myCollection.find( { active: true, $where: "this.credits - this.debits < 0" } );
```

Restrictions

Do not write to the collection being inspected from the \$where expression.

Using db.eval()

db.eval() is used to evaluate a function (written in JavaScript) at the database server.

This is useful if you need to touch a lot of data lightly. In that scenario, network transfer of the data could be a bottleneck.

db.eval() returns the return value of the function that was invoked at the server. If invocation fails an exception is thrown.

For a trivial example, we can get the server to add 3 to 3:

```
> db.eval( function() { return 3+3; } );
6
>
```

Let's consider an example where we wish to erase a given field, foo, in every single document in a collection. A naive client-side approach would be something like

Calling my_erase() on the client will require the entire contents of the collection to be transmitted from server to client and back again.

Instead, we can pass the function to eval(), and it will be called in the runtime environment of the server. On the server, the db variable is set to the current database:

```
db.eval(my_erase);
```

Examples

```
> myfunc = function(x){ return x; };
> db.eval( myfunc, {k:"asdf"} );
{ k : "asdf" }
> db.eval( myfunc, "asdf" );
"asdf"
> db.eval( function(x){ return x; }, 2 );
```

If an error occurs on the evaluation (say, a null pointer exception at the server), an exception will be thrown of the form:

```
{ dbEvalException: { errno : -3.0 , errmsg : "invoke failed" , ok : 0.0 } }
```

Example of using eval() to do equivalent of the Mongo count() function:

```
function mycount(collection) {
 return db.eval( function(){return db[collection].find({},{_id:ObjId()}).length();} );
```

Example of using db.eval() for doing an atomic increment, plus some calculations:

```
function inc( name , howMuch ){
    return db.eval(
        function(){
            var t = db.things.findOne( { name : name } );
            t = t || { name : name , num : 0 , total : 0 , avg : 0 };
            t.num++;
            t.total += howMuch;
            t.avg = t.total / t.num;
            db.things.save( t );
            return t;
        }
    );
}
db.things.remove( {} );
print( tojson( inc( "eliot" , 2 )) );
print( tojson( inc( "eliot" , 3 )) );
```

Storing functions server-side



in version 1.1.1 and above

There is a special system collection called system. js that can store JavaScript function to be re-used. To store a function, you would do:

```
db.system.js.save( { _id : "foo" , value : function( x , y ) { return x + y; } } );
```

_id is the name of the function, and is unique per database.

Once you do that, you can use foo from any JavaScript context (db.eval, \$where, map/reduce)

See http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js for a full example

Map/Reduce

MongoDB supports Javascript-based map/reduce operations on the server. See the map/reduce documentation for more information.

Notes on Concurrency

eval() blocks the entire mongod process while running. Thus, its operations are atomic but prevent other operations from processing.

When more concurrency is needed consider using map/reduce instead of eval().

Sorting and Natural Order

"Natural order" is defined as the database's native ordering of objects in a collection.

When executing a find() with no parameters, the database returns objects in forward natural order.

For standard tables, natural order is not particularly useful because, although the order is often close to insertion order, it is not *guaranteed* to be. However, for Capped Collections, natural order is guaranteed to be the insertion order. This can be very useful.

In general, the natural order feature is a very efficient way to store and retrieve data in insertion order (much faster than say, indexing on a timestamp field). But remember, the collection must be capped for this to work.

In addition to forward natural order, items may be retrieved in reverse natural order. For example, to return the 50 most recently inserted items (ordered most recent to less recent) from a capped collection, you would invoke:

```
> c=db.cappedCollection.find().sort({$natural:-1}).limit(50)
```

Sorting can also be done on arbitrary keys in any collection. For example, this sorts by 'name' ascending, then 'age' descending:

```
> c=db.collection.find().sort({name : 1, age : -1})
```

See Also

- The Capped Collections section of this Guide
- Advanced Queries
- The starting point for all Home

Aggregation

Mongo includes utility functions which provide server-side count, distinct, and group by operations. More advanced aggregate functions can be crafted using MapReduce.

Count

count () returns the number of objects in a collection or matching a query. If a document selector is provided, only the number of matching documents will be returned.

size() is like count() but takes into consideration any limit() or skip() specified for the query.

```
db.collection.count(selector);
```

For example:

```
print( "# of objects: " + db.mycollection.count() );
print( db.mycollection.count( {active:true} );
```

count is faster if an index exists for the condition in the selector. For example, to make the count on active fast, invoke

```
db.mycollection.ensureIndex( {active:1} );
```

Distinct

distinct(key) returns a list of distinct values for the given key across a collection.

```
db.addresses.save({"zip-code": 10010})
db.addresses.save({"zip-code": 10010})
db.addresses.save({"zip-code": 99701})

db.addresses.distinct("zip-code");
[ 10010, 99701 ]
```

distinct may also reference a nested key:

```
db.comments.save({"user": {"points": 25}})
db.comments.save({"user": {"points": 31}})
db.comments.save({"user": {"points": 25}})

db.comments.distinct("user.points");
[ 25, 31 ]
```

You can add an optional query parameter to distinct as well

```
db.address.distinct( "zip-code" , { age : 30 } )
```

Group

group returns an array of grouped items. The command is similar to SQL's group by. The SQL statement

```
select a,b,sum(c) csum from coll where active=1 group by a,b
```

corresponds to the following in MongoDB:

One caveat: the array resulting from the group command must fit in RAM; otherwise, you'll get an exception.

group takes a single object parameter containing the following fields:

- key: Fields to group by.
- reduce: The reduce function aggregates (reduces) the objects iterated. Typical operations of a reduce function include summing and counting. reduce takes two arguments: the current document being iterated over and the aggregation counter object. In the example above, these arguments are named obj and prev.
- initial: initial value of the aggregation counter object.
- keyf: An optional function returning a "key object" to be used as the grouping key. Use this instead of key to specify a key that is not an existing member of the object (or, to access embedded members). Set in lieu of key.
- cond: An optional condition that must be true for a row to be considered. This is essentially a find() query expression object. If null, the reduce function will run against all rows in the collection.
- finalize: An optional function to be run on each item in the result set just before the item is returned. Can either modify the item (e.g., add an average field given a count and a total) or return a replacement object (returning a new object with just _id and average fields). See jstests/group3.js for examples.

To order the grouped data, simply sort it client-side upon return. The following example is an implementation of <code>count()</code> using <code>group()</code>.

Examples

The examples assume data like this:

```
{ domain: "www.mongodb.org"
, invoked_at: {d:"2009-11-03", t:"17:14:05"}
, response_time: 0.05
, http_action: "GET /display/DOCS/Aggregation"
}
```

Show me stats for each http_action in November 2009:

```
db.test.group(
    { cond: {"invoked_at.d": {$gt: "2009-11", $lt: "2009-12"}}
    , key: {http_action: true}
    , initial: {count: 0, total_time:0}
    , reduce: function(doc, out){ out.count++; out.total_time+=doc.response_time }
    , finalize: function(out){ out.avg_time = out.total_time / out.count }
    });

[
    {
        "http_action": "GET /display/DOCS/Aggregation",
        "count": 1,
        "total_time": 0.05,
        "avg_time": 0.05
    }
}
```

Show me stats for each domain for each day in November 2009:

```
db.test.group(
    { cond: {"invoked_at.d": {$gt: "2009-11", $lt: "2009-12"}}
    , key: {domain: true, invoked_at.d: true}
    , initial: {count: 0, total_time:0}
    , reduce: function(doc, out){ out.count++; out.total_time+=doc.response_time }
    , finalize: function(out){ out.avg_time = out.total_time / out.count }
    });

[
    {
        "http_action" : "GET /display/DOCS/Aggregation",
        "count" : 1,
        "total_time" : 0.05,
        "avg_time" : 0.05
    }
}
```

Using Group from Various Languages

Some language drivers provide a group helper function. For those that don't, one can manually issue the db command for group. Here's an

example using the Mongo shell syntax:

See Also

- jstests/eval2.js for an example of group() usage
- Advanced Queries

Removing

Removing Objects from a Collection

To remove objects from a collection, use the remove() function in the mongo shell. (Other drivers offer a similar function, but may call the function "delete". Please check your driver's documentation).

remove() is like find() in that it takes a JSON-style query document as an argument to select which documents are removed. If you call remove() without a document argument, or with an empty document {}, it will remove all documents in the collection. Some examples:

```
db.things.remove({});  // removes all
db.things.remove({n:1}); // removes all where n == 1
```

If you have a document in memory and wish to delete it, the most efficient method is to specify the item's document _id value as a criteria:

```
db.things.remove({_id: myobject._id});
```

You may be tempted to simply pass the document you wish to delete as the selector, and this will work, but it's inefficient.



References

If a document is deleted, any pre-existing references to the document will still exist in the database. These references will return null when evaluated.

Concurrency and Remove

v1.3+ supports concurrent operations while a remove runs. If a simultaneous update (on the same collection) grows an object which matched the remove criteria, the updated object may not be removed (as the operations are happening at approximately the same time, this may not even be surprising). In situations where this is undesireable, pass {\$atomic : true} in your filter expression:

```
db.videos.remove( { rating : { $lt : 3.0 }, $atomic : true } )
```

The remove operation is then completely atomic - however, it will also block other operations while executing.

Updating

MongoDB supports atomic, in-place updates as well as more traditional updates for replacing an entire document.

- Update
- save() in the mongo shell
- Modifier Operations
 - \$inc
 - \$set\$unset
 - \$unse
 - \$push

- \$pushAll
- \$pop
- \$pull
- \$pullAll
- Upserts
- Pushing a Unique Value
- Checking the Outcome of an Update Request
- A Note About Padding
- See Also

Update

Here's the MongoDB shell syntax for update():

```
db.collection.update( criteria, objNew, upsert, multi )
```

Arguments:

- criteria query which selects the record to update;
- objNew updated object or \$ operators (e.g., \$inc) which manipulate the object
- upsert if this should be an "upsert"; that is, if the record does not exist, insert it
- multi if all documents matching criteria should be updated (the default is to only update the first document found)

${\tt save}$ () in the mongo shell

The save() command in the mongo shell provides a shorthand syntax to perform a single object update with upsert:

```
// x is some JSON style object
db.mycollection.save(x); // updates if exists; inserts if new
```

save() does an upsert if x has no _id field and an insert if it does. Thus, normally, you will not need to explicitly request upserts, just use save().

Upsert means "update if present; insert if missing".

```
myColl.update( { name: "Joe" }, { name: "Joe", age: 20 }, true );
```

Modifier Operations

Modifier operations are highly-efficient and useful when updating existing values; for instance, they're great for incrementing a number.

So, while a conventional implementation does work:

```
var j=myColl.findOne( { name: "Joe" } );
j.n++;
myColl.save(j);
```

a modifier update has the advantages of avoiding the latency involved in querying and returning the object. The modifier update also features operation atomicity and very little network data transfer.

To perform an atomic update, simply specify any of the special update operators (which always start with a '\$' character) with a relevant update document:

```
db.people.update( { name:"Joe" }, { $inc: { n : 1 } } );
```

The preceding example says, "Find the first document where 'name' is 'Joe' and then increment 'n' by one."

\$inc

```
{ $inc : { field : value } }
```

increments field by the number value if field is present in the object, otherwise sets field to the number value.

\$set

```
{ $set : { field : value } }
```

sets field to value. All datatypes are supported with \$set.

\$unset

```
{ $unset : { field : 1} }
```

Deletes a given field. v1.3+

\$push

```
{ $push : { field : value } }
```

appends value to field, if field is an existing array, otherwise sets field to the array [value] if field is not present. If field is present but is not an array, an error condition is raised.

\$pushAll

```
{ $pushAll : { field : value_array } }
```

appends each value in $value_array$ to field, if field is an existing array, otherwise sets field to the array $value_array$ if field is not present. If field is present but is not an array, an error condition is raised.

\$pop

```
{ $pop : { field : 1 } }
```

removes the last element in an array (ADDED in 1.1)

```
{ $pop : { field : -1 } }
```

removes the first element in an array (ADDED in 1.1) |

\$pull

```
{ $pull : { field : _value } }
```

removes all occurrences of value from field, if field is an array. If field is present but is not an array, an error condition is raised.

\$pullAll

```
{ $pullAll : { field : value_array } }
```

removes all occurrences of each value in $value_array$ from field, if field is an array. If field is present but is not an array, an error condition is raised.

Upserts

You may use upsert with a modifier operation. In such a case, the modifiers will be applied to the update <code>criteria</code> member and the resulting object will be inserted. The following upsert example may insert the object {name: "Joe", x:1, y:1}.

```
db.people.update( { name:"Joe" }, { $inc: { x:1, y:1 } }, true );
```

There are some restrictions. A modifier may not reference the _id field, and two modifiers within an update may not reference the same field, for example the following is not allowed:

```
db.people.update( { name:"Joe" }, { $inc: { x: 1 }, $set: { x: 5 } } );
```

Pushing a Unique Value

To add a value to an array only if not already present, add \$ne : <value> to your query expression:

```
update( {_id:'joe', tags: {"$ne": "baseball"}},
{"$push": { tags : "baseball" } } );
```

Checking the Outcome of an Update Request

As described above, a non-upsert update may or may not modify an existing object. An upsert will either modify an existing object or insert a new object. The client may determine if its most recent message on a connection updated an existing object by subsequently issuing a getlasterror command (findOne({getlasterror:1})). If the result of the getlasterror command contains an updatedExisting field, the last message on the connection was an update request. If the updatedExisting field's value is true, that update request caused an existing object to be updated; if updatedExisting is false, no existing object was updated.

A Note About Padding

When you update an object in MongoDB, the update occurs in-place if the object has not grown in size. This is good for insert performance if the collection has many indexes.

Mongo also adaptively learns if objects in a collection tend to grow, and if they do, it adds some padding to prevent excessive movements.

See Also

- · findandmodify Command
- Atomic Operations

Atomic Operations

- Modifier operations
- "Update if Current"
- Find and Modify (or Remove)
- Updating Multiple Objects Atomically

MongoDB's philosophy towards atomic operations is quite different from traditional databases. Mongo DB does not support traditional locking for a number of reasons:

- First, in sharded environments, distributed locks could be expensive and slow. Mongo DB's goal is to be lightweight and fast.
- We dislike the concept of deadlocks. We want the system to be simple and predictable without these sort of surprises.
- We want Mongo DB to work well for realtime problems. If an operation may execute which locks large amounts of data, it might stop some small light queries for an extended period of time. (We don't claim Mongo DB is perfect yet in regards to being "real-time", but we certainly think locking would make it even harder.)

That said, we clearly need ways to atomically update documents in the database. We will look at a few here now.

Modifier operations

The Mongo DB update command supports several modifiers which automically update an element in a document. They include:

- \$set set a particular value
- \$unset set a particular value (since 1.3.0)
- \$inc increment a particular value by a certain amount
- \$push append a value to an array
- \$pushAll append several values to an array
- \$pull remove a value(s) from an existing array
- \$pullAll remove several value(s) from an existing array

These modifiers are convenient ways to certain operations atomically.

"Update if Current"

Another strategy for atomic updates is "Update if Current". For this we

- 1. Fetch the object.
- 2. Modify the object locally.
- 3. Send an update request that says "update the object to this new value if it still matches its old value".

Should the operation fail, we might then want to try again from step 1.

For example, suppose we wish to fetch one object from inventory. We want to see that an object is available, and if it is, deduct it from the inventory. The following code demonstrates this using mongo shell syntax (similar functions may be done in any language):

```
> t=db.inventory
> s = t.findOne({sku:'abc'})
{"_id" : "49df4d3c9664d32c73ea865a" , "sku" : "abc" , "qty" : 30}
> qty_old = s.qty;
> --s.qty;
> t.update({_id:s._id, qty:qty_old}, s); db.$cmd.findOne({getlasterror:1});
{"err" : , "updatedExisting" : true , "n" : 1 , "ok" : 1} // it worked
```

For the above example, we likely don't care the exact sku quantity as long as it is as least as great as the number to deduct. Thus the following code is better, although less general -- we can get away with this as we are using a predefined modifier operation (\$inc). For more general updates, the "update if current" approach shown above is recommended.

```
> t.update({sku:"abc",qty:{$gt:0}}, { $inc : { qty : -1 } } ) ; db.$cmd.findOne({getlasterror:1})
    {"err" : , "updatedExisting" : true , "n" : 1 , "ok" : 1} // it worked
    > t.update({sku:"abcz",qty:{$gt:0}}, { $inc : { qty : -1 } } ) ; db.$cmd.findOne({getlasterror:1})
    {"err" : , "updatedExisting" : false , "n" : 0 , "ok" : 1} // did not work
```

Find and Modify (or Remove)

See the findandmodify Command documentation for more information.

Updating Multiple Objects Atomically

Generally, Mongo DB does not support updating several documents atomically in one operation. However, you can nest objects making them effectively one document for atomicity purposes. The db.eval() statement provides a way to automically perform several operations at once; however, its use for this is not recommended as this eval() atomicity will not be supported for certain cases in sharded environments.

findandmodify Command

Find and Modify (or Remove)



In 1.3.0 and higher

MongoDB 1.3+ supports a "find, modify, and return" command. This command is useful to atomically change an object and then get back the results. It also includes a sort option which is useful when storing queue-like data. The general form is

The mongo shell includes a helper method findAndModify() for this purpose. Some drivers provide helpers also.

As an example, to fetch the highest priority job that hasn't been grabbed yet and atomically mark it as grabbed:

```
job = db.jobs.findAndModify({
    query: {inprogress:false},
    sort:{priority:-1},
    update: {$set: {inprogress: true}}
});
```

You could also simply remove the object to be returned:

```
job = db.jobs.findAndModify({sort:{priority:-1}, remove:true}}});
```

See the tests for more examples.

One of the update or remove parameter is required; other arguments are optional.

Argument	Description	Default
query	a filter for the query	{}
sort	if multiple docs match, choose the first one in the specified sort order as the object to manipulate	{}
remove	set to a true to remove the object before returning	N/A
update	a modifier object	N/A
new	set to true if you want to return the modified object rather than the original. Ignored for remove.	false

If your driver doesn't yet provide a helper function for this command, run the command directly with something like this:

See Also

Atomic Operations

Updating Data in Mongo

- Updating a Document in the mongo Shell with save()
- Embedding Documents Directly in Documents
- Database References

Updating a Document in the mongo Shell with save()

As shown in the previous section, the <code>save()</code> method may be used to save a new document to a collection. We can also use <code>save()</code> to update an existing document in a collection.

Continuing with the example database from the last section, lets add new information to the document {name: "mongo"} that already is in the collection.

```
> var mongo = db.things.findOne({name:"mongo"});
> print(tojson(mongo));
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo"}
> mongo.type = "database";
database
> db.things.save(mongo);
> db.things.findOne({name:"mongo"});
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database"}
>
```

This was a simple example, adding a string valued element to the existing document. When we called <code>save()</code>, the method saw that the document already had an "_id" field, so it simply performed an update on the document.

In the next two sections, we'll show how to embed documents within documents (there are actually two different ways), as well as show how to query for documents based on values of embedded documents.

Embedding Documents Directly in Documents

As another example of updating an existing document, lets embed a document within an existing document in the collection. We'll keep working with the original {name: "mongo"} document for simplicity.

```
> var mongo = db.things.findOne({name:"mongo"});
> print(tojson(mongo));
{"_id" : "497da93d4ee47b3a675d2d9b" , "name" : "mongo", "type" : "database"}
> mongo.data = { a:1, b:2};
{"a" : 1 , "b" : 2}
> db.things.save(mongo);
> db.things.findOne({name:"mongo"});
{"_id" : "497da93d4ee47b3a675d2d9b" , "name" : "mongo" , "type" : "database", "data" : {"a" : 1 , "b" : 2}}
>
```

As you can see, we added new data to the mongo document, adding $\{a:1, b:2\}$ under the key "data".

Note that the value of "data" is a document itself - it is embedded in the parent mongo document. With BSON, you may nest and embed documents to any level. You can also query on embedded document fields, as shown here:

```
> db.things.findOne({ "data.a" : 1});
{"_id" : "497da93d4ee47b3a675d2d9b" , "name" : "mongo" , "data" : {"a" : 1 , "b" : 2}}
> db.things.findOne({ "data.a" : 2});
>
```

Note that the second findOne() doesn't return anything, because there are no documents that match.

Database References

Alternatively, a document can reference other documents which are not embedded via a *database reference*, which is analogous to a foreign key in a relational database. A database reference (or "DBRef" for short), is a reference implemented according to the DB Ref. Most drivers support helpers for creating DBRefs. Some also support additional functionality, like dereference helpers and auto-referencing. See specific driver documentation for examples / more information

In the previous section we saw that the statement

```
obj_a.x = obj_b;
```

Lets repeat the above example, but create a document and place in a different collection, say *otherthings*, and embed that as a reference in our favorite "mongo" object under the key "otherdata":

```
// first, save a new doc in the 'otherthings' collection
> var other = { s : "other thing", n : 1};
> db.otherthings.save(other);
> db.otherthings.find();
{"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 1}
// now get our mongo object, and add the 'other' doc as 'otherthings'
> var mongo = db.things.findOne();
> print(tojson(mongo));
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database" , "data" : {"a" : 1 , "b"
: 2}}
> mongo.otherthings = new DBRef( 'otherthings' , other._id );
\label{eq:special} \mbox{ $\{$"s" : "other thing" , "n" : 1 , "\_id" : "497dbcb36b27d59a708e89a4"\} }
> db.things.save(mongo);
> db.things.findOne().otherthings.fetch();
: 2} , "otherthings" : {"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 1}}
// now, lets modify our 'other' document, save it again, and see that when the dbshell
// gets our mongo object and prints it, if follows the dbref and we have the \operatorname{new} value
> other.n = 2;
> db.otherthings.save(other);
> db.otherthings.find();
{"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 2}
> db.things.findOne().otherthings.fetch();
{"\_id"} : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" : "database" , "data" : {"a" : 1 , "b" | family | fam
: 2} , "otherthings" : {"_id" : "497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 2}}
```

MapReduce

MongoDB supports map/reduce queries for performing sophisticated and simple aggregations over a collection.

- Overview
 - Map Function
 - Reduce Function
 - Finalize Function
- Examples
 - Shell Example 1
 - Shell Example 2
 - More Examples
 - Note on Permanent Collections

Overview



Version 1.1.1 and above

In sharded environments, data processing runs in parallel on all shards. (NOTE 2Feb10: sharded map/reduce pending and available soon.)

map/reduce is invoked via a database command. The database creates a temporary collection to hold output of the operation. The collection is cleaned up when the client connection closes, or when explicitly dropped. Alternatively, one can specify a permanent output collection name. map and reduce functions are written in JavaScript and execute on the server.

Command syntax:

```
db.runCommand(
    { mapreduce : <collection>,
        map : <mapfunction>,
        reduce : <reducefunction>
        [, query : <query filter object>]
        [, sort : <sort the query. useful for optimization>]
        [, limit : <number of objects to return from collection>]
        [, out : <output-collection name>]
        [, keeptemp: <true|false>]
        [, finalize : <finalizefunction>]
        [, scope : <object where fields go into javascript global scope >]
        [, verbose : true]
    }
}
```

- keeptemp if true, the generated collection is not treated as temporary. Defaults to false. When out is specified, the collection is automatically made permanent.
- finalize function to apply to all the results when finished
- verbose provide statistics on job execution time
- scope can pass in variables that can be access from map/reduce/finalize example mr5

Result:

```
{ result : <collection_name>,
    counts : {
        input : <number of objects scanned>,
        emit : <number of times emit was called>,
        output : <number of items in output collection>
    },
    timeMillis : <job_time>,
    ok : <1_if_ok>,
    [, err : <errmsg_if_error>]
}
```

A command helper is available in the MongoDB shell:

```
db.collection.mapReduce(mapfunction,reducefunction[,options]);
```

map, reduce, and finalize functions are written in JavaScript.

Map Function

The map function references the variable this to inspect the current object under consideration. A map function must call emit(key, value) at least once, but may be invoked any number of times, as may be appropriate.

```
function map(void) -> void
```

Reduce Function

The reduce function receives a key and an array of values. To use, reduce the received values, and return a result.

```
function reduce(key, value_array) -> value
```

The MapReduce engine may invoke reduce functions iteratively; thus, these functions must be idempotent. That is, the following must hold for your reduce function:

```
for all k,vals : reduce( k, [reduce(k,vals)] ) == reduce(k,vals)
```

If you need to perform an operation only once, use a finalize function.

Note: Currently, the return value from a reduce function cannot be an array (it's typically an object or a number).

Finalize Function

A finalize function may be run after reduction. Such a function is optional and is not necessary for many map/reduce cases. The finalize function takes a key and a value, and returns a finalized value.

```
function finalize(key, value) -> final_value
```

Examples

Shell Example 1

The following example assumes we have an events collection with objects of the form:

```
{ time : <time>, user_id : <userid>, type : <type>, ... }
```

We then use MapReduce to extract all users who have had at least one event of type "sale":

```
> m = function() { emit(this.user_id, 1); }
> r = function(k,vals) { return 1; }
> res = db.events.mapReduce(m, r, { query : {type:'sale'} });
> db[res.result].find().limit(2)
{ "_id" : 8321073716060 , "value" : 1 }
{ "_id" : 7921232311289 , "value" : 1 }
```

If we also wanted to output the number of times the user had experienced the event in question, we could modify the reduce function like so:

```
> r = function(k,vals) {
...  var sum=0;
...  for(var i in vals) sum += vals[i];
...  return sum;
... }
```

Note, here, that we cannot simply return vals.length, as the reduce may be called multiple times.

Shell Example 2

```
$ ./mongo
> db.things.insert( { _id : 1, tags : ['dog', 'cat'] } );
> db.things.insert( { _id : 2, tags : ['cat'] } );
> db.things.insert( { _id : 3, tags : ['mouse', 'cat', 'dog'] } );
> db.things.insert( { _id : 4, tags : [] } );
> // map function
> m = function(){
      this.tags.forEach(
         function(z){
               emit( z , { count : 1 } );
. . .
. . .
     );
...};
> // reduce function
> r = function( key , values ){
      var total = 0;
     for ( var i=0; i<values.length; i++ )
        total += values[i].count;
     return { count : total };
. . . } ;
> res = db.things.mapReduce(m,r);
> res
{"timeMillis.emit" : 9 , "result" : "mr.things.1254430454.3" ,
 "numObjects" : 4 , "timeMillis" : 9 , "errmsg" : "" , "ok" : 0}
> db[res.result].find()
{"_id" : "cat" , "value" : {"count" : 3}}
{"_id" : "dog" , "value" : {"count" : 2}}
{"_id" : "mouse" , "value" : {"count" : 1}}
> db[res.result].drop()
```

More Examples

- example mr1
- Finalize example: example mr2

Note on Permanent Collections

Even when a permanent collection name is specified, a temporary collection name will be used during processing. At map/reduce completion, the temporary collection will be renamed to the permanent name atomically. Thus, one can perform a map/reduce job periodically with the same target collection name without worrying about a temporary state of incomplete data. This is very useful when generating statistical output collections on a regular basis.

Data Processing Manual

DRAFT - TO BE COMPLETED.

This guide provides instructions for using MongoDB batch data processing oriented features including map/reduce.

By "data processing", we generally mean operations performed on large sets of data, rather than small interactive operations.

Import

One can always write a program to load data of course, but the mongoimport utility also works for some situations. mongoimport supports importing from json, csv, and tsv formats.

A common usage pattern would be to use mongoimport to load data in a relatively raw format and then use a server-side script (db.eval() or map/reduce) to reduce the data to a more clean format.

See Also

• Import/Export Tools

- Server-Side Code Execution
- Map/Reduce

mongo - The Interactive Shell

MongoDB Interactive Shell

The MongoDB distribution includes bin/mongo, the MongoDB interactive shell. This utility is a JavaScript shell that allows you to issue commands to MongoDB from the command line.

The shell is useful for:

- · inspecting a database's contents
- testing queries
- creating indices
- · other administrative functions.

When you see sample code in this wiki and it looks like JavaScript, assume it is a shell example.

More Information

- Shell Overview
- Shell Reference
- Shell API

Overview - The MongoDB Interactive Shell

Starting the Shell

The interactive shell is included in the standard MongoDB distribution. To start the shell, go into the root directory of the distribution and type

```
./bin/mongo
```

It might be useful to add mongo_distribution_root/bin to your PATH so you can just type mongo from anywhere.

If you start with no parameters, it connects to a database named "test" running on your local machine on the default port (27017). You can see the db to which you are connecting by typing db:

```
./mongo
type "help" for help
> db
test
```

You can pass mongo an optional argument specifying the address, port and even the database to initially connect to:

./mongo foo	connects to the foo database on your local machine
./mongo 192.168.13.7/foo	connects to the foo database on 192.168.13.7
./mongo dbserver.mydomain.com/foo	connects to the foo database on dbserver.mydomain.com
./mongo 192.168.13.7:9999/foo	connects to the £00 database on 192.168.13.7 on port 9999

Connecting

If you have not connected via the command line, you can use the following commands:

```
conn = new Mongo(host);
db = conn.getDB(dbname);
db.auth(username,password);
```

where host is a string that contains either the name or address of the machine you want to connect to (e.g. "192.168.13.7") or the machine and port (e.g. "192.168.13.7:9999"). Note that host in an optional argument, and can be omitted if you want to connect to the database instance

running on your local machine. (e.g. conn = new Mongo())

Basics Commands

The following are three basic commands that provide information about the available databases, and collections in a given database.

show dbs	displays all the databases on the server you are connected to
use db_name	switches to db_name on the same server
show collections	displays a list of all the collections in the current database

Querying

mongo uses a JavaScript API to interact with the database. Because mongo is also a complete JavaScript shell, db is the variable that is the current database connection.

To query a collection, you simply specify the collection name as a property of the db object, and then call the find() method. For example:

```
db.foo.find();
```

This will display the first 10 objects from the foo collection.

Inserting Data

In order to insert data into the database, you can simply create a JavaScript object, and call the <code>save()</code> method. For example, to save an object { <code>name: "sara"}</code> in a collection called *foo*, type:

```
db.foo.save({ name : "sara"});
```

Note that MongoDB will implicitly create any collection that doesn't already exist.

Modifying Data

Let's say you want to change someone's address. You can do this using the following mongo commands:

```
person = db.people.findOne( { name : "sara" } );
person.city = "New York";
db.people.save( person );
```

Deleting Data

db.foo.drop()	drop the entire £00 collection
db.foo.remove()	remove all objects from the collection
db.foo.remove({ name : "sara" })	remove objects from the collection where name is sara

Indexes

db.foo.getIndexKeys()	get all fields that have indexes on them
<pre>db.foo.ensureIndex({ _field_ : 1 })</pre>	create an index on field if it doesn't exist

Line Continuation

If a line contains open '(' or '{' characters, the shell will request more input before evaluating:

```
> function f() {
... x = 1;
... }
>
```

You can press Ctrl-C to escape from "..." mode and terminate line entry.

See Also

• MongoDB Shell Reference

dbshell Reference

Special Command Helpers

Non-javascript convenience macros:

show dbs	Print a list of all databases on this server
use dbname	Set the db variable to represent usage of dbname on the server
show collections	Print a list of all collections for current database
show users	Print a list of users for current database
show profile	Print most recent profiling operations that took >= 1ms

Basic Shell Javascript Operations

db	The variable that references the current database object / connection. Already defined for you in your instance.
db.auth(user,pass)	Authenticate with the database (if running in secure mode).
coll = db.collection	Access a specific <i>collection</i> within the database.
<pre>cursor = coll.find();</pre>	Find all objects in the collection. See queries.
<pre>coll.remove(objpattern);</pre>	Remove matching objects from the collection. objpattern is an object specifying fields to match. E.g.: coll.remove({ name: "Joe" });
coll.save(object);	Save an object in the collection, or update if already there. If your object has a presave method, that method will be called before the object is saved to the db (before both updates and inserts)
coll.insert(object);	Insert object in collection. No check is made (i.e., no upsert) that the object is not already present in the collection.
coll.update()	Update an object in a collection. See the Updating documentation; update() has many options.
<pre>coll.ensureIndex({ name : 1 })</pre>	Creates an index on tab. name. Does nothing if index already exists.
coll.update()	
coll.drop()	Drops the collection coll
db.getSisterDB(name)	Switch to another database using this same connection. Simpilar to "use name" but works as a normal javascript expression.

Queries

coll.find()	Find all.
it	Continue iterating the last cursor returned from find().

<pre>coll.find(criteria);</pre>	Find objects matching criteria in the collection. E.g.: coll.find({ name: "Joe" });
<pre>coll.findOne(criteria);</pre>	Find and return a single object. Returns null if not found. If you want only one object returned, this is more efficient than just find() as limit(1) is implied. You may use regular expressions if the element type is a string, number, or date: coll.find({ name: /joe/i });
<pre>coll.find(criteria, fields);</pre>	Get just specific fields from the object. E.g.: coll.find({}, {name:true});
<pre>coll.find().sort({field:1[, field :1] });</pre>	Return results in the specified order (field ASC). Use -1 for DESC.
<pre>coll.find(criteria).sort({ field : 1 })</pre>	Return the objects matching criteria, sorted by field.
<pre>coll.find().limit(n)</pre>	Limit result to n rows. Highly recommended if you need only a certain number of rows for best performance.
<pre>coll.find().skip(n)</pre>	Skip n results.
coll.count()	Returns total number of objects in the collection.
coll.find().count()	Returns the total number of objects that match the query. Note that the number ignores limit and skip; for example if 100 records match but the limit is 10, count() will return 100. This will be faster than iterating yourself, but still take time.

More information: see queries.

Error Checking

db.getLastError()	Returns error from the last operation.
db.getPrevError()	Returns error from previous operations.
db.resetError()	Clear error memory.

Administrative Command Helpers

db.cloneDatabase(fromhost)	Clone the current database from the other host specified. fromhost database must be in noauth mode.
<pre>db.copyDatabase(fromdb, todb, fromhost)</pre>	Copy fromhost/fromdb to todb on this server. fromhost must be in noauth mode.
db.repairDatabase()	Repair and compact the current database. This operation can be very slow on large databases.
db.addUser(user,pwd)	Add user to current database.
db.getCollectionNames()	get list of all collections.
db.dropDatabase()	Drops the current database.

Miscellaneous

<pre>c = connect("<host>:<port>/<dbname>")</dbname></port></host></pre>	Open a new database connection. One may have multiple connections within a single shell, however, automatic getLastError reporting by the shell is done for the 'db' variable only. See here for an example of connect().
Object.bsonsize(db.foo.findOne())	prints the bson size of a db object (mongo version 1.3 and greater)
db.foo.findOne().bsonsize()	prints the bson size of a db object (mongo versions predating 1.3)

For a full list of functions, see the shell API.

Developer FAQ

- How do I copy all objects from one database collection to another?
- If you remove an object attribute is it deleted from the store?
- Are null values allowed?
- Does an update fsync to disk immediately?
- · How do I do transactions/locking?
- How do I do equivalent of SELECT count * and GROUP BY?
- What are so many "Connection Accepted" messages logged?
- Why are my data files so large?
 - Preallocation
 - Deleted Space
 - · Checking Size of a Collection

How do I copy all objects from one database collection to another?

See below. The code below may be ran server-side for high performance with the eval() method.

```
db. myoriginal. find(). for Each( function(x) \{db. mycopy. save(x)\} ); \\
```

If you remove an object attribute is it deleted from the store?

Yes, you remove the attribute and then re-save() the object.

Are null values allowed?

For members of an object, yes. You cannot add null to a database collection though as null isn't an object. You can add {}, though.

Does an update fsync to disk immediately?

No, writes to disk are lazy. A write may hit disk a couple of seconds later. For example, if the database receives a thousand increments to an object within one second, it will only be flushed to disk once.

How do I do transactions/locking?

MongoDB does not use traditional locking or complex transactions with rollback, as it is designed to be lightweight and fast and predictable in its performance. It can be thought of as analogous to the MySQL MyISAM autocommit model. By keeping transaction support extremely simple, performance is enhanced, especially in a system that may run across many servers.

The system provides alternative models for atomically making updates that are sufficient for many common use cases. See the wiki page Atomics Operations for detailed information.

How do I do equivalent of SELECT count * and GROUP BY?

See aggregation.

What are so many "Connection Accepted" messages logged?

If you see a tremendous number of connection accepted messages in the mongod log, that means clients are repeatedly connecting and disconnected. This works, but is inefficient.

With CGI this is normal. If you find the speed acceptable for your purposes, run mongod with --quiet to suppress these messages in the log. If you need better performance, which to a solution where connections are pooled -- such as an Apache module.

Why are my data files so large?

You may notice that for a given set of data the MongoDB datafiles in /data/db are larger than the data set inserted into the database. There are several reasons for this.

Preallocation

Each datafile is preallocated to a given size. (This is done to prevent file system fragmentation, among other reasons.) The first file for a database is <dbname>.0, then <dbname>.1, etc. <dbname>.0 will be 64MB, <dbname>.1 128MB, etc., up to 2GB. Once the files reach 2GB in size, each successive file is also 2GB.

Thus if the last datafile present is say, 1GB, that file might be 90% empty if it was recently reached.

Additionally, on Unix, mongod will preallocate an additional datafile in the background and do background initialization of this file. These files are prefilled with zero bytes. This inititialization can take up to a minute (less on a fast disk subsystem) for larger datafiles; without prefilling in the

background this could result in significant delays when a new file must be prepopulated.

You can disable preallocation with the --noprealloc option to the server. This flag is nice for tests with small datasets where you drop the db after each test. It shouldn't be used on production servers.

For large databases (hundreds of GB or more) this is of no signficant consequence as the unallocated space is small.

Deleted Space

MongoDB maintains deleted lists of space within the datafiles when objects or collections are deleted. This space is reused but never freed to the operating system.

To compact this space, run db.repairDatabase() from the mongo shell (note this operation will block and is slow).

When testing and investigating the size of datafiles, if your data is just test data, use db.dropDatabase() to clear all datafiles and start fresh.

Checking Size of a Collection

Use the validate command to check the size of a collection -- that is from the shell run:

```
> db.<collectionname>.validate();

> // these are faster:
> db.<collectionname>.dataSize(); // just data size for collection
> db.<collectionname>.storageSize(); // allocation size including unused space
> db.<collectionname>.totalSize(); // data + index
> db.<collectionname>.totalIndexSize(); // index data size
```

This command returns info on the collection data but note there is also data allocated for associated indexes. These can be checked with validate too, if one looks up the index's namespace name in the system.namespaces collection. For example:

```
> db.system.namespaces.find()
{"name" : "test.foo"}
{"name" : "test.system.indexes"}
{"name" : "test.foo.$_id_"}
> > db.foo.$_id_.validate()
{"ns" : "test.foo.$_id_" , "result" : "
validate
 details: 0xb3590b68 ofs:83fb68
  firstExtent:0:8100 ns:test.foo.$_id_
 lastExtent:0:8100 ns:test.foo.$_id_
 datasize?:8192 nrecords?:1 lastExtentSize:131072
 padding:1
  first extent:
   loc:0:8100 xnext:null xprev:null
   ns:test.foo.$ id
   size:131072 firstRecord:0:81b0 lastRecord:0:81b0
  1 objects found, nobj:1
  8208 bytes data w/headers
  8192 bytes data wout/headers
 deletedList: 000000000001000000
 deleted: n: 1 size: 122688
 nIndexes:0
 , "ok" : 1 , "valid" : true , "lastExtentSize" : 131072}
```

Compare Order for Types

MongoDB allows objects in the same collection which have values which may differ in type. When comparing values from different types, a convention is utilized as to which value is less than the other. This (somewhat arbitary but well defined) ordering is listed below.

Note that some types are treated as equivalent for comparison purposes -- specifically numeric types which undergo conversion before comparison.

See also the BSON specification.

Null

- Numbers (ints, longs, doubles)
- Symbol, String
- Object
- Array
- BinData
- ObjectID
- Boolean
- Date, Timestamp
- Regular Expression

Example (using the mongo shell):

```
> t = db.mycoll;
> t.insert({x:3});
> t.insert( {x : 2.9} );
> t.insert( {x : new Date()} );
> t.insert( {x : new Date()} );
> t.insert( {x : true } )
> t.find().sort({x:1})
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
```

MinKey and MaxKey

In addition to the above types MongoDB internally uses a special type for MinKey and MaxKey which are less than, and greater than all other possible BSON element values, respectively.

From the mongo Javascript Shell

For example we can continue our example from above adding two objects which have x key values of MinKey and MaxKey respectively:

```
> t.insert( { x : MaxKey } )
> t.insert( { x : MinKey } )
> t.find().sort({x:1})
{ "_id" : ObjectId("4b04094b7c65b846e2090112"), "x" : { $minKey : 1 } }
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
{ "_id" : ObjectId("4b0409487c65b846e2090111"), "x" : { $maxKey : 1 } }
```

From C++

See also the Tailable Cursors page for an example of using MinKey from C++. See also minKey and maxKey definitions in jsobj.h.

Admin Zone

- Hosting Center
- Administration Basics
- DBA Operations from the Shell
- Sharding
- Replication
- Production Notes
- Security and Authentication
- Mongo Architecture
- How to do Snapshotted Queries in the Mongo Database
- Backups
- Database Profiler
- Diagnostics and Troubleshooting
- Durability and Repair

See Also

Commands in Developer Zone

Hosting Center

Setup Instructions

- Amazon EC2
- Joyent
- Webfaction
- Linode

Hosted MongoDB

MongoHQ provides cloud-style hosted MongoDB instances

One-Click Installation

· Dreamhost offers instant configuration and deployment of MongoDB

Amazon EC2

- Instance Types
- Linux
- EC2 TCP Port Management
- EBS Snapshotting

MongoDB runs well on Amazon EC2. This page includes some notes in this regard.

Instance Types

MongoDB works on most EC2 types including Linux and Windows. We recommend you use a 64 bit instance as this is required for all databases of significant size .

Linux

One can download a binary or build from source. Generally it is easier to download a binary. We can download and run the binary without being root. For example on 64 bit Linux:

```
[~]$ curl -0 http://downloads.mongodb.org/linux/mongodb-linux-x86_64-1.0.1.tgz
[~]$ tar -xzf mongodb-linux-x86_64-1.0.1.tgz
[~]$ cd mongodb-linux-x86_64-1.0.1/bin
[bin]$ ./mongod --version
```

Before running the database one should decide where to put datafiles. Run df -h to see volumes. On some images /mnt will be the many locally attached storage volume. Alternatively you may want to use Elastic Block Store which will have a different mount point. Regardless, create a directory in the desired location and then run the database:

```
mkdir /mnt/db
./mongod --fork --logpath ~/mongod.log --dbpath /mnt/db/
```

EC2 TCP Port Management

By default the database will now be listening on port 27017. The web administrative UI will be on port 28017.

EBS Snapshotting



v1.3.1+

If your datafiles are on an EBS volume, you can snapshot them for backups. Use the fsync lock command to lock the database to prevent writes. Then, snapshot the volume. Then use the unlock command to allow writes to the database again. See the fsync documentation for more information.

This method may also be used with slave databases.

Joyent

The prebuilt MongoDB Solaris 64 binaries work with Joyent accelerators.

Some newer gcc libraries are required to run -- see sample setup session below.

```
$ # assuming a 64 bit accelerator
$ /usr/bin/isainfo -kv
64-bit amd64 kernel modules
$ # get mongodb
$ # note this is 'latest' you may want a different version
$ curl -0 http://downloads.mongodb.org/sunos5/mongodb-sunos5-x86_64-latest.tgz
$ gzip -d mongodb-sunos5-x86_64-latest.tgz
$ tar -xf mongodb-sunos5-x86_64-latest.tgz
$ mv "mongodb-sunos5-x86_64-2009-10-26" mongo
$ cd mongo
$ # get extra libraries we need (else you will get a libstdc++.so.6 dependency issue)
$ curl -0 http://downloads.mongodb.org.s3.amazonaws.com/sunos5/mongo-extra-64.tgz
$ gzip -d mongo-extra-64.tgz
$ tar -xf mongo-extra-64.tar
$ # just as an example - you will really probably want to put these somewhere better:
$ export LD_LIBRARY_PATH=mongo-extra-64
$ bin/mongod --help
```

Administration Basics

- Starting and Stopping Mongo
- Command Line Parameters
- File Based Configuration
- Import Export Tools
- GridFS Tools
- Http Interface

Starting and Stopping Mongo

- Starting Mongo
 - Default Data Directory, Default Port
 - Alternate Data Directory, Default Port
 - Alternate Port
 - Running as a Daemon
- Stopping Mongo
 - Control-C
 - Sending shutdownServer() message from the mongo shell
 - Sending a Unix TERM signal
- Memory Usage

Mongo is run as a standard program from the command line. Please see Command Line Parameters for more information on those options.

Here are some common use cases for starting mongo

The following examples assume that you are in the directory where the Mongo executable is, and the Mongo executable is called mongod.

Starting Mongo

Default Data Directory, Default Port

To start Mongo in default mode, where data will be stored in the /data/db directory (or c:\data\db on Windows), and listening on port 27017, just type

```
$ ./mongod
```

Alternate Data Directory, Default Port

To specify a directory for Mongo to store files, use the --dbpath option:

```
$ ./mongod --dbpath /var/lib/mongodb/
```

Note that you must create the directory and set its permissions appropriately ahead of time -- Mongo will not create the directory if it doesn't exist.

Alternate Port

You can specify a different port for Mongo to listen on for connections from clients using the --port option

```
$ ./mongod --port 12345
```

This is useful if you want to run more than one instance of Mongo on a machine (e.g., for running a master-slave pair).

Running as a Daemon

Note: these options are only available in MongoDB version 1.1 and later.

This will fork the Mongo server and redirect its output to a logfile. As with --dbpath, you must create the log path yourself, Mongo will not create parent directories for you.

```
$ ./mongod --fork --logpath /var/log/mongodb.log --logappend
```

Stopping Mongo

Control-C

If you have Mongo running in the foreground in a terminal, you can simply "Ctrl-C" the process. This will cause Mongo to do a clean exit, flushing and closing it's data files. Note that it will wait until all ongoing operations are complete.

Sending shutdownServer() message from the mongo shell

The shell can request that the server terminate.

```
$ ./mongo
> db.shutdownServer()
```

This command only works from localhost, or, if one is authenticated.

From a driver (where the helper function may not exist), one can run the command

```
{ "shutdown" : 1 }
```

Sending a Unix TERM signal

You can cleanly stop Mongo via a TERM signal on Unix-like systems via the kill command. Mongo will do a clean shutdown when it receives a TERM.

```
kill -2 PID
```

Please note that sending a KILL signal (9) will probably cause damage as Mongo will not be able to cleanly exit. (In such a scenario, run the repairDatabase command.)

Memory Usage

Mongo uses memory mapped files to access data, which results in large numbers being displayed in tools like top for the mongod process. This is not a concern, and is normal when using memory-mapped files. Basically, the size of mapped data is shown in the virtual size parameter, and resident bytes shows how much data is being cached in RAM.

You can get a feel for the "inherent" memory footprint of Mongo by starting it fresh, with no connections, with an empty /data/db directory and looking at the resident bytes. (Running with --nojni option will result in even lower core memory usage.)

Command Line Parameters

MongoDB can be configured via command line parameters in addition to File Based Configuration. You can see the currently supported set of command line options by running the database with -h [--help] as a single parameter:

```
$ ./mongod -h
```

Information on usage of these parameters can be found in Starting and Stopping Mongo.

The following list of options is not complete; for the complete list see the usage information as described above.

Basic Options

-h help	Shows all options
-f config <file></file>	Specify a configuration file to use
port <portno></portno>	Specifies the port number on which Mongo will listen for client connections. Default is 27017
bind_ip <ip></ip>	Specifies a single IP that the database server will listen for
dbpath <path></path>	Specifies the directory for datafiles. Default is /data/db or c:\data\db
directoryperdb	Specify use of an alternative directory structure, in which files for each database are kept in a unique directory.
quiet	Reduces amount of log output
logpath <file></file>	File to write logs to (instead of stdout). You can rotate the logs by sending SIGUSR1 to the server.
logappend	Append to existing log file, instead of overwritting
repairpath <path></path>	Root path for temporary files created during database repair. Default is dbpath value.
fork	Fork the server process
cpu	Enables periodic logging of CPU utilization and I/O wait
noauth	Turns off security. This is currently the default
auth	Turn on security
-v[v[v[v[v]]]] verbose	Verbose logging output (-vvvvv is most verbose, -v ==verbose)
objcheck	Inspect all client data for validity on receipt (useful for developing drivers)
quota	Enable db quota management
diaglog <n></n>	Set oplogging level where n is 0=off (default) 1=W 2=R 3=both 7=W+some reads
nocursors	Diagnostic/debugging option
nohints	Ignore query hints
nohttpinterface	Disable the HTTP interface (localhost:27018)
noscripting	Turns off server-side scripting. This will result in greatly limited functionality
notablescan	Turns off table scans. Any query that would do a table scan fails
noprealloc	Disable data file preallocation
nssize <mb></mb>	Specifies .ns file size for new databases
sysinfo	Print system info as detected by Mongo and exit
upgrade	Upgrade database files to new format if necessary (required when upgrading from <= 1.0 to 1.1+)
mms-token <arg></arg>	Accout token for Mongo monitoring server

mms-name <server></server>	Server name for Mongo monitoring server
mms-interval <sec></sec>	Ping interval for Mongo monitoring server

Replication Options

master	Designate this server as a master in a master-slave setup			
slave	Designate this server as a slave in a master-slave setup			
source <server:port></server:port>	Specify the source (master) for a slave instance			
only <db></db>	Slave only: specify a single database to replicate			
pairwith <server:port></server:port>	Address of a server to pair with			
arbiter <server:port></server:port>	Address of arbiter server			
autoresync	Automatically resync if slave data is stale			
oplogSize <mb></mb>	Custom size for replication operation log			
opIdMem <bytes></bytes>	Size limit for in-memory storage of op ids			

File Based Configuration

In addition to accepting Command Line Parameters, MongoDB can also be configured using a configuration file. A configuration file to use can be specified using the -f or --config command line options. The following example configuration file demonstrates the syntax to use:

```
# This is an example config file for MongoDB.

dbpath = /var/lib/mongodb
bind_ip = 127.0.0.1
noauth = true # use 'true' for options that don't take an argument
```

Notes

- Lines starting with octothorpes are comments
- Options are case sensitive
- The syntax is assignment of a value to an option name
- · All command line options are accepted

Import Export Tools

- mongoimportjson
- mongoimport
- mongoexport
- mongodump
- mongorestore

mongoimportjson



replaced in 1.1.2 with mongoimport

This utility takes a single file that contains 1 JSON string per line and inserts it. You have to specify a database and a collection.

mongoimport



new in 1.1.2

This utility takes a single file that contains 1 JSON/CSV/TSV string per line and inserts it. You have to specify a database and a collection.

mongoexport

This utility takes a collection and exports to either JSON or CSV. You can specify a filter for the query, or a list of fields to output.

If you want to output CSV, you have to specify the fields in the order you want them.

Example

```
master erh@ERH-OFC-MP.local ~/work/mongo -> ./mongoexport -d test -c test1 --csv -f "name,num"
connected to: 127.0.0.1
name,num
"sara",2
"eliot",1
"eliot",1
```

```
export options:

--help produce help message
-h [--host] arg mongo host to connect to
-d [--db] arg database to use
-c [--collection] arg collection to use (some commands)
--dbpath arg directly access mongod data files in this path,
instead of connecting to a mongod instance
-q [--query] arg query filter
-f [--fields] arg comma seperated list of field names e.g. -f=name,age
--csv export to csv instead of json
-o [--out] arg output file; if not specified, stdout is used
```

mongodump

This takes a database and outputs it in a binary representation. This is mostly used for doing hot backups of a database.

mongorestore

This takes the output from mongodump and restores it.

```
restore options:

--help produce help message
-h [ --host ] arg mongo host to connect to
-d [ --db ] arg database to use
-c [ --collection ] arg collection to use (some commands)
--dbpath arg directly access mongod data files in this path,
instead of connecting to a mongod instance
--dir arg directory to restore from
```

GridFS Tools

File Tools

mongofiles is a tool for manipulating GridFS from the command line.

A simple interaction.

```
$ ./mongofiles list
connected to: 127.0.0.1
$ ./mongofiles put libmongoclient.a
connected to: 127.0.0.1
done!
$ ./mongofiles list
connected to: 127.0.0.1
libmongoclient.a 12000964
$ cd /tmp/
$ ~/work/mon/mongofiles get libmongoclient.a
$ ~/work/mongo/mongofiles get libmongoclient.a
connected to: 127.0.0.1
done write to: libmongoclient.a
$ md5 libmongoclient.a
MD5 (libmongoclient.a) = 23a52d361cfa7bad98099c5bad50dc41
$ md5 ~/work/mongo/libmongoclient.a
MD5 (/Users/erh/work/mongo/libmongoclient.a) = 23a52d361cfa7bad98099c5bad50dc41
```

Http Interface

MongoDB provides a simple http interface listing information of interest to administrators. This interface may be accessed at the port with numeric value 1000 more than the configured mongod port; the default port for the http interface is 28017. To access the http interface an administrator may, for example, point a browser to http://127.0.0.1:28017 if mongod is running with the default port on the local machine.

Here is a description of the informational elements of the http interface:

element	description
db version	database version information
git hash	database version developer tag
sys info	mongod compilation environment
dblocked	indicates whether the primary mongod mutex is held
uptime	time since this mongod instance was started
assertions	any software assertions that have been raised by this mongod instance
replinfo	information about replication configuration
currentOp	most recent client request
# databases	number of databases that have been accessed by this mongod instance
curclient	last database accessed by this mongod instance
Cursors	describes outstanding client cursors
master	whether this mongod instance has been designated a master
slave	whether this mongod instance has been designated a slave
initialSyncCompleted	whether this slave or repl pair node has completed an initial clone of the mongod instance it is replicating
DBTOP	Displays the total time the mongod instance has devoted to each listed collection, as well as the percentage of available time devoted to each listed collection recently and the number of reads, writes, and total calls made recently
dt	Timing information about the primary mongod mutex

A screen capture of browser rendered output from the http interface follows:

mongodb Aarons-computer.local:27017

```
db version v0.9.1+, pdfile version 4.4
git hash: 3d03daec09fb9635e8cca3cc9741224c3c8b52a2
sys info: Darwin Aarons-computer.local 9.6.0 Darwin Kernel Version 9.6.0: Mon Nov 24 17:37:00 PST 2008; root:xnu-1228.9.59-1/RELEASE_I386 i386

dblocked: 0 (initial)
uptime: 17 seconds

assertions:

replInfo:
time to get dblock: 0ms
currentOp: { opid: 1, active: 0, op: "query", ns: "local.system.namespaces", query: "{ name: /^local.temp./ }", inLock: 1 }

# databases: 1
Cursors byLoc.size(): 0

replication
master: 0
slave: 0
initialSyncCompleted: 1

DBTOP
```

DBTOP					
Namespace	윰	Reads	Writes	Calls	Time
admin	0.00	0	0	0	00:00:00.000024
local	0.00	0	0	0	00:00:00.071435
local.dbinfo	0.00	0	0	0	00:00:00.071192
local.dbinfo.admin	0.00	0	0	0	00:00:00.008763
local.dbinfo.local	0.00	0	0	0	00:00:00.043639
local.dbinfo.test	0.00	0	0	0	00:00:00.018790
local.system	0.00	0	0	0	00:00:00.000146
local.system.namespaces	0.00	0	0	0	00:00:00.000146
test	0.00	0	0	0	00:00:00.000020

```
dt tlocked
4000 0 09
4000 0 09
4000 0 09
4000 0 09
```

Security

If security is configured for a mongod instance, authentication is required for a client to access the http interface from another machine.

REST interface (experimental)

To get the contents of a collection (note the trailing slash):

```
http://127.0.0.1:28017/databaseName/collectionName/
```

To add a limit:

```
http://127.0.0.1:28017/databaseName/collectionName/?limit=-10
```

To skip:

```
http://127.0.0.1:28017/databaseName/collectionName/?skip=5
```

To query for {a : 1}:

```
http://127.0.0.1:28017/databaseName/collectionName/?filter_a=1
```

Separate conditions with an &:

```
http://127.0.0.1:28017/databaseName/collectionName/?filter_a=1&limit=-10
```

Web Interface JSON

The web interface uses strict JSON (as opposed to the shell, which uses Dates, regular expressions, etc.). To display non-JSON types, the web interface wraps them in objects and uses the key for the type. For example:

```
# ObjectIds just become strings
"_id" : "4a8acf6e7fbadc242de5b4f3"

# dates
"date" : { "$date" : 1250609897802 }

# regular expressions
"match" : { "$regex" : "foo", "$options" : "ig" }
```

The code type has not been implemented yet and causes the DB to crash if you try to display it in the browser.

See Mongo Extended JSON for details.

DBA Operations from the Shell

This page lists common DBA-class operations that one might perform from the MongoDB shell.

Note one may also create .js scripts to run in the shell for administrative purposes.

```
help
                            show help
show dbs
                            show database names
                         show collections in current database
show collections
show users
                           show users in current database
show profile
                           show most recent system.profile entries with time >= 1ms
use <db name>
                           set curent database to <db name>
db.addUser (username, password)
db.removeUser(username)
db.cloneDatabase(fromhost)
db.copyDatabase(fromdb, todb, fromhost)
db.createCollection(name, { size : ..., capped : ..., max : ... } )
db.getName()
db.dropDatabase()
db.printCollectionStats()
db.currentOp() displays the current operation in the db
db.killOp() kills the current operation in the db
db.getProfilingLevel()
db.setProfilingLevel(level) 0=off 1=slow 2=all
db.getReplicationInfo()
db.printReplicationInfo()
db.printSlaveReplicationInfo()
db.repairDatabase()
db.version() current version of the server
db.shutdownServer()
```

Commands for manipulating and inspecting a collection:

```
db.foo.drop() drop the collection
db.foo.dropIndex(name)
db.foo.dropIndexes()
db.foo.getIndexes()
db.foo.ensureIndex(keypattern,options) - options object has these possible
                                         fields: name, unique, dropDups
db.foo.find( [query] , [fields])
                                       - first parameter is an optional
                                         query filter. second parameter
                                         is optional
                                         set of fields to return.
                                    e.g. db.foo.find(
                                                  \{ x : 77 \},
                                                  \{ name : 1 , x : 1 \} )
db.foo.find(...).count()
db.foo.find(...).limit(n)
db.foo.find(...).skip(n)
db.foo.find(...).sort(...)
db.foo.findOne([query])
db.foo.getDB() get DB object associated with collection
db.foo.count()
db.foo.group( { key : ..., initial: ..., reduce : ...[, cond: ...] } )
db.foo.renameCollection( newName ) renames the collection
db.foo.stats()
db.foo.dataSize()
db.foo.storageSize() - includes free space allocated to this collection
db.foo.totalIndexSize() - size in bytes of all the indexes
db.foo.totalSize() - storage allocated for all data and indexes
db.foo.validate() (slow)
db.foo.insert(obj)
db.foo.update(query, object[, upsert_bool])
db.foo.save(obi)
db.foo.remove(query)
                                       - remove objects matching query
                                         remove({}) will remove all
```

Sharding

MongoDB includes an auto-sharding module ("mongos"). Auto-sharding allows one to build a large horizontally scalable database cluster that can incorporate additional machines dynamically.

Alpha 2 of sharding is now available (sharding alpha 2 is part of MongoDB 1.1+). Please see the Limitations page for alpha restrictions.

Please note: the documentation below is for alpha 2; certain commands have changed since alpha 1. Please be sure you are using MongoDB v1.1 or higher.

- Introduction and Overview
- Restrictions and Limits
- Operational
 - Configuring
 - Administrative
 - Failover Information
- FAQ
- Internals

Sharding FAQ

- · Where do unsharded collections go if sharding is enabled for a database?
- When will data be on more than one shard?
- What happens if I try to update a document on a chunk that is being migrated?
- What if a shard is down or slow and I do a query?

Where do unsharded collections go if sharding is enabled for a database?

In alpha 2 unsharded data goes to the "primary" for the database specified (query config.databases to see details). Future versions will parcel out unsharded collections to different shards (that is, a collection could be on any shard, but will be on only a single shard if unsharded).

When will data be on more than one shard?

MongoDB sharding is range based. So all the objects in a collection get put into a chunk. Only when there is more than 1 chunk is there an option for multiple shards to get data. Right now, the chunk size is 50mb, so you need at least 50mb for a migration to occur.

What happens if I try to update a document on a chunk that is being migrated?

The update will go through immediately on the old shard, and then the change will be replicated to the new shard before ownership transfers.

What if a shard is down or slow and I do a query?

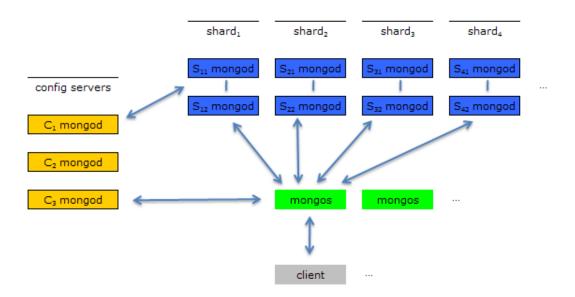
If a shard is down, the query will return an error. If a shard is responding slowly, mongos will wait for it. You won't get partial results.

Sharding Introduction

- Shards
- Chunks
- Config Servers
- mongos Process
- Shard Keys
- Operation Types
- Server Layout
- See Also

MongoDB includes auto-sharding support as part of the system. This document provides an architectural overview of sharding and how it can be used.

Be sure to see the limitations document for limitations of Alpha release of sharding.



A MongoDB cluster consists of a number of shards, mongos routing processes, one or more config servers, and clients which use the cluster.

Shards

Each shard consists of one or two servers and stores data using the mongod process, which is the core MongoDB database process. Normally, one uses two servers per shard to ensure availability. The two servers (e.g., S_{11} and S_{12}) replicate to one another using a slightly enhanced version of MongoDB's replica pair functionality. (Future versions will support more than two replicas per shard; however for the near future the limit is two copies.)

Data is partitioned by collection in an order preserving manner. This allows us to perform range queries by shard key efficiently.

Chunks

A chunk is a contiguous range of data (documents) from a particular collection. (collection, minKey, maxKey) describes a chunk, where the shard key K of a given document meets the condition minKey >= K > maxKey.

Chunks grow to a maximum size, for example 50MB. Once enough documents are in a chunk to reach that approximate size, the chunk *splits* into two new chunks. When a particular shard has excess data, chunks will then *migrate* to other shards in the system. Likewise, chunks migrate when one adds additional servers (shards).

Config Servers

The config server(s) store the cluster's metadata, which includes basic information on each shard and server, and chunk information.

Chunk information is the key data stored by the config servers. Each config server has a complete copy of all chunk information.

mongos Process

The mongos process can be thought of as a routing and coordination process that makes the various components of the cluster look like a single system. mongos has no persistent state, and can run on any server desired (some might choose to run it on the shard servers themselves, but one could also run it elsewhere such as on the client servers).

mongos fetches metadata from the config server(s) to get started. Then, when receiving a client request, it routes the request to the appropriate server(s) and compiles a result to send back to the client.

A given system can have any number of mongos instances. Each instance requires some RAM for metadata storage, otherwise there is no limitation as there is no coordination between mongos instances (all coordination is from a single mongos to the shard servers and config servers; in addition shard servers speak to one another and to the config servers).

Shard Keys

To partition a collection, we specify a shard key pattern. This pattern is similar to the key pattern used to define an index: it names one or more fields that will be the key upon which we distribute data. Some example shard key patterns:

```
{ name : 1 }
{ _id : 1 }
{ lastname : 1, firstname : 1 }
{ tag : 1, timestamp : -1 }
```

MongoDB partitioning is order-preserving: adjacent data by shard key tends to be on the same server (in the same chunk). The config database stores chunk information such as:

```
collection minkey maxkey location
users { name : 'Miller' } { name : 'Nessman' } shard<sub>2</sub>
users { name : 'Nessman' } { name : 'Ogden' } shard<sub>4</sub>
```

Note that shard key values should be *granular*. in the above example, we should not have say, 1 million people all with the same name. Otherwise a chunk becomes to large and cannot split. In a case like that, use a compound shard key, and add an additional field which provides further discrimination of the values.

Operation Types

On a sharded system we have two styles of operations: global and targeted.

For targeted operations, mongos communicates with a very small number of shards -- often a single shard. These operations are very efficient.

Global operations involve the mongos process reaching out to all (or most) shards in the system.

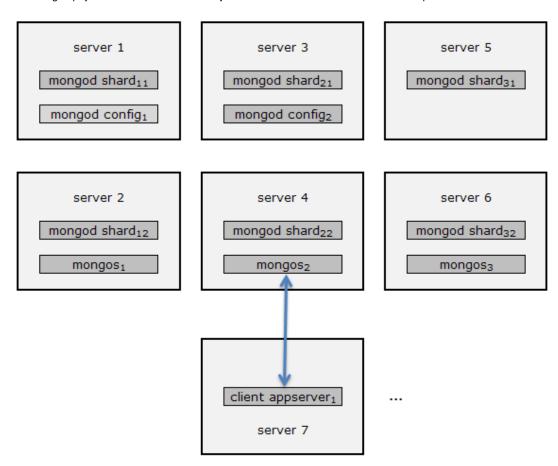
The following table shows various operations and their type. For the examples below, assume a shard key of $\{x:1\}$.

Operation	Туре	Comments
db.foo.find({ x : 300 })	Targeted	Queries a single shard.

db.foo.find({ x : 300, age : 40 })	Targeted	Queries a single shard.
db.foo.find({ age : 40 })	Global	Queries all shards.
db.foo.find()	Global	sequential
db.foo.find().count()		Same as the corresponding find() operation
db.foo.find().sort({ age : 1 })	Global	parallel
db.foo.find().sort({ x : 1 })	Global	sequential
db.foo.count()	Global	parallel
db.foo.insert(<object>)</object>	Targeted	
db.foo.update({ x : 100 }, <object>) db.foo.remove({ x : 100 })</object>	Targeted	
db.foo.update({ age : 40 }, <object>)</object>	Global	
db.foo.remove({ age : 40 })		
db.getLastError()	?	
db.getPrevError()		
db.foo.ensureIndex()	Global	

Server Layout

Machines may be organized in a variety of fashions. First, one could have separate servers for each config db process, mongos process, and mongod process. However, this is likely overkill as load may be low on certain things such as the config db's. Below we show an example where some sharing of physical machines is used to lay out a cluster without excess instances required.



Other configurations are possible too. For example, one might choose to run a mongos process on all of servers 1-6; or, one could run the mongos process on each app server (server 7). There is some potential benefit to running it at the app server as the communications between appserver and mongos are then over the localhost interface.

See Also

Configuring

Configuring Sharding

Start the relevant processes.

- Run mongod on the shard servers. Use the --shardsvr command line parameter. For replica pairs, use the --pairwith command line
 option. (To get started with a simple test we recommend just running a single mongod process per shard without failure.)
- Run mongod on the config server(s) with the --configsvr command line parameter. If the config servers are running on a shared machine
 other processes, assign it to separate a separate dbpath (--dbpath command line parameter).
- Run mongos on the servers of your choice. Specify --configdb parameter to indicate location of the config database(s).

Configure Sharding from the Shell

You may want to automate or record your steps below in a .js file for replay in the shell anytime needed.

To run these commands, connect to a mongos that was started above, and run all configuration commands through it. Note you must use the special admin database for these commands:

```
./mongo <mongos-hostname>:<mongos-port>/admin
> db
admin
>
```

Adding a Shard

Each shard consist of either two servers (a replica pair) or a single mongod server instance. (Alpha 2 supports single-server shards only.) To add a shard:

```
> db.runCommand( { addshard : "<serverhostname>[:<port>]" } );
{"ok" : 1 , "added" : ...}
```

Add pairs by comma separating two server[:port] names in the addshard command (alpha 3 and beyond).

There is an optional "maxSize" parameter that lets you tell the system how much disk space to use on that shard. If you do not specify that, it will just use the entire disk.

To see existing shards:

```
> db.runCommand( { listshards : 1 } );
```

Enabling for a Database

We must enable sharding for a database -- otherwise data for the database is stored entirely on one shard.

```
> db.runCommand( { enablesharding : "<dbname>" } );
```

Once enabled, mongos will place different collections for the database on different shards. However, unless the collection is sharded (see below), all data from one collection will be located on a single shard.

Sharding a Collection

Use the shardcollection command to shard a collection. An index is automatically created for the specified key.

For example, to shard the GridFS chunks collection (which is a great candidate as this can get very large) on the test database one would invoke:

```
> db.runCommand( { shardcollection : "test.fs.chunks", key : { _id : 1 } } ) { "ok" : 1}
```

You can also make your shard key unique:

```
db.runCommand( \{ shardcollection : "test.users" , key : \{ email : 1 \} , unique : true \} \} \}
```

Examples

- A sample configuration session
- The following example shows how to run a simple shared setup on a single machine for testing purposes:
 - http://github.com/mongodb/mongo-snippets/blob/master/sharding/first.js.

See Also

- Administration
- TCP Port Numbers

A Sample Configuration Session

The following example uses two shards (one server each), one config db, and one mongos process, all running on a single test server.

```
$ mkdir /data/db/a
$ mkdir /data/db/b
$ mkdir /data/db/config
$ ./mongod --dbpath /data/db/a --port 10000 > /tmp/sharda.log &
$ cat /tmp/sharda.log
$ ./mongod --dbpath /data/db/b --port 10001 > /tmp/shardb.log &
$ cat /tmp/shardb.log
$ ./mongod --dbpath /data/db/config --port 20000 > /tmp/configdb.log &
$ cat /tmp/configdb.log
$ ./mongos --configdb localhost:20000 > /tmp/mongos.log &
$ cat /tmp/mongos.log
$ # we connect to mongos process
$ ./mongo
MongoDB shell version: 1.1.0-
url: test
connecting to: test
type "help" for help
> use admin
switched to db admin
> db.runCommand( { addshard : "localhost:10000", allowLocal : true } )
{"ok" : 1 , "added" : "localhost:10000"}
> db.runCommand( { addshard : "localhost:10001", allowLocal : true } )
{"ok" : 1 , "added" : "localhost:10001"}
> config = connect("localhost:20000")
> config = config.getSisterDB("config")
> test = db.getSisterDB("test")
test
> db.runCommand( { enablesharding : "test" } )
> db.runCommand( { shardcollection : "test.people", key : {name : 1} } )
{"ok" : 1}
> db.runCommand({listshards:1})
{"servers" : [{"_id" : ObjectId( "4a9d40c981ba1487ccfaa634") , "host" : "localhost:10000"},
             {"_id" : ObjectId( "4a9d40df81ba1487ccfaa635") , "host" : "localhost:10001"}] ,
"ok" : 1}
```

Sharding Administration

See also Configuring.

```
> // test if we are speaking to a mongos process or
> // straight to a mongod process
> db.$cmd.findOne({isdbgrid:1});

> // mongos returns { ismaster: 0.0, msg: "isdbgrid" }
> db.$cmd.findOne({ismaster:1});
```

List Existing Shards

```
> db.runCommand( { listshards : 1 } )
{"servers" :
    [{"_id" : ObjectId( "4a9d40c981ba1487ccfaa634") ,
        "host" : "localhost:10000"},
    {"_id" : ObjectId( "4a9d40df81ba1487ccfaa635") ,
        "host" : "localhost:10001"}
    l,
        "ok" : 1
}
```

List Which Databases are Sharded

Here we query the config database (through mongos – your shell connection is connected to a mongos and it connects to the config database automatically).

```
> config = db.getSisterDB("config")
> config.system.namespaces.find()
```

See full sharding setup

```
> printShardingStatus( db.getSisterDB( "config" ) );
```

Moving a chunk manually

has to be run on admin db

```
db.runCommand( \{ movechunk : <full ns> , find : <something in the chunk> , to : <shard name> \} )
```

example:

```
db.runCommand( { movechunk : "test.blog.posts" , find : { title : "The Cool Post" } , to : "192.168.1.2" } )
```

More

```
> db.runCommand({netstat:1})
implementation pending...
```

Sharding and Failover

Failure of a mongod process within a shard (PLANNED)

If one half of a replica pair for a shard is down, read and write operations are still permitted. However, chunk migrations to that shard will be suspended until the down node is fixed.

Failure of all mongod processes within a shard (PLANNED)

If both replicas within a shard are down, the data within that shard is (at least temporarily) unavailable. Queries which can be resolved at other shards continue to work properly. Queries that require the down shard will return an error.

Failure of a config server

When one of the config servers is down, split and migrate operations are not possible.

Sharding Limits

Sharding Alpha 2 (MongoDB v1.1)

- Work is still underway on replication/failover. For Alpha 2, please use unreplicated shards (and config servers) for testing. This will be available in Alpha 3.
- group() is not supported in alpha 2. In generally, we recommend using the map/reduce facility in sharded environments.
- Sharding must be ran in trusted security mode, without explicit security.

Shard Keys

- Shard keys are immutable in the current version.
- Compound shard keys are not yet supported, but will be in the future.

Differences from Unsharded Configurations

Sharded databases behave differently in certain circumstances from a standalone, single-server mongod instance.

\$where

\$where works with sharding. However do not reference the db object from the \$where function (one normally does not do this anyway).

db.eval

db.eval() may not be used with sharded collections. However, you may use db.eval() if the evaluation function accesses unsharded collections within your database. In future versions, a map/reduce facility will replace eval() for sharded collection manipulation.

aetPrevError

getPrevError is unsupported for sharded databases, and may remain so in future releases (TBD).

Unique Indexes

For a sharded collection, you may only (optionally) specify a unique constraint on the shard key. Other secondary indexes work (via a global operation) as long as no unique constraint is specified.

Counts

Count is supported with sharding; however, a "count all in collection" will not be instantaneous for a sharded collection as it is for an unsharded collection.

Scale Limits

Long term goals are for systems of up to 1,000 shards. Testing for Alpha 2 will be limited to clusters with a modest number of shards (e.g., 20). More information will be reported here later on any scaling limitations which are encountered.

MongoDB sharding supports two styles of operations -- targeted and global. On giant systems, global operations will be of less applicability.

Replication

The Mongo database supports replication of data between servers.

The replication is an enhanced master-slave configuration: that is, only one server is active for writes (the master) at a given time. The primary goal of replication is failover and redundancy.

Mongo supports two forms of replication: simple master-slave configurations, and additionally a replica pair concept. Please see the following

pages for more details:

- · Master-Slave Replication
- Replica Pairs
- · Limited Master-Master Replication

Security

When security is enabled, one must configure a user account for the local database that exists on both servers.

The slave-side of a replication connection first looks for a user repl in local.system.users. If present, that user is used to authenticate against the local database on the source side of the connection. If repl user does not exist, the first user object in local.system.users is tried.

The local database works like the admin database: an account for local has access to the entire server.

Diagnostics

Check master status from the mongo shell with:

```
db.printReplicationInfo()
```

Check slave status from the mongo shell with:

```
db.printSlaveReplicationInfo()
```

Master Slave

Setup of a Manual Master/Slave Configuration

To configure an instance of Mongo to be a master database in a master-slave configuration, you'll need to start two instances of the database, one in *master* mode, and the other in *slave* mode.



Data Storage

The following examples explicitly specify the location of the data files on the command line. This is unnecessary if you are running the master and slave on separate machines, but in the interest of the readers who are going try this setup on a single node, they are supplied in the interest of safety.

```
$ bin/mongod --master [--dbpath /data/masterdb/]
```

As a result, the master server process will create a local.oplog. \$main collection. This is the "transaction log" which queues operations which will be applied at the slave.

To configure an instance of Mongo to be a slave database in a master-slave configuration:

```
$ bin/mongod --slave --source <masterhostname>[:<port>] [--dbpath /data/slavedb/]
```

Details of the source server are then stored in the slave's local.sources collection. Instead of specifying the --source parameter, on can add an object to local.sources which specifies information about the master server:

```
$ bin/mongo <slavehostname>/local
> db.sources.find(); // confirms the collection is empty. then:
> db.sources.save( { host: <masterhostname> } );
```

- host: masterhostname is the IP address or FQDN of the master database machine. Append :port to the server hostname if you wish to run on a nonstandard port number.
- · only: databasename (optional) if specified, indicates that only the specified database should replicate

A slave can pull from multiple upstream masters. In such a situation add multiple configuration objects to the local.sources collection.

A server can be slave and a master at same time.

A slave may become out of sync with a master if it falls far behind the data updates available from that master, or if the slave is terminated and then restarted some time later when relevant updates are no longer available from the master. If a slave becomes out of sync, replication will terminate and operator intervention is required by default if replication is to be restarted. An operator may restart replication using the {resync:1} command. Alternatively, the command line option --autoresync causes a slave to restart replication automatically (after ten second pause) if it becomes out of sync. If the --autoresync option is specified, the slave will not attempt an automatic resync more than once in a ten minute periond.

The --oplogSize command line option may be specified (along with --master) to configure the amount of disk space in megabytes which will be allocated for storing updates to be made available to slave nodes. If the --oplogSize option is not specified, the amount of disk space for storing updates will be the maximum of 1 gigabyte and five percent of available disk space for sixty-four bit machines, or fifty megabytes for thirty-two bit machines.

Security

Example security configuration when security is enabled:

```
$ dbshell <slavehostname>/admin -u <existingadminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
^c
$ dbshell <masterhostname>/admin -u <existingadminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
```

Replica Pairs

- · Setup of Replica Pairs
- Consistency
- Security
- · Replacing a Replica Pair Server
- Querying the slave
- What is and when should you use an arbiter?
- Working with an existing (non-paired) database
- See Also

Setup of Replica Pairs

Mongo supports a concept of *replica pairs*. These databases automatically coordinate which is the master and which is the slave at a given point in time.

At startup, the databases will negotiate which is master and which is slave. Upon an outage of one database server, the other will automatically take over and become master from that point on. In the event of another failure in the future, master status would transfer back to the other server. The databases manage this themselves internally.

Note: Generally, start with empty /data/db directories for each pair member when creating and running the pair for the first time. See section on Existing Databases below for more information.

To start a pair of databases in this mode, run each as follows:

```
$ ./mongod --pairwith <remoteserver> --arbiter <arbiterserver>
```

where

- remoteserver is the hostname of the other server in the pair. Append :port to the server hostname if you wish to run on a
 nonstandard port number.
- arbiterserver is the hostname (and optional port number) of an arbiter. An arbiter is a Mongo database server that helps negotiate which member of the pair is master at a given point in time. Run the arbiter on a third machine; it is a "tie-breaker" effectively in determining which server is master when the members of the pair cannot contact each other. You may also run with no arbiter by not including the --arbiter option. In that case, both servers will assume master status if the network partitions.

One can manually check which database is currently the master:

```
$ ./mongo
> db.$cmd.findOne({ismaster:1});
{ "ismaster" : 0.0 , "remote" : "192.168.58.1:30001" , "ok" : 1.0 }
```

(Note: When security is on, remote is only returned if the connection is authenticated for the admin database.)

However, Mongo drivers with replica pair support normally manage this process for you.

Consistency

Members of a pair are only eventually consistent on a failover. If machine L of the pair was master and fails, its last couple seconds of operations may not have made it to R - R will not have those operations applied to its dataset until L recovers later.

Security

Example security configuration when security is enabled:

```
$ ./mongo <lefthost>/admin -u <adminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
^c
$ ./mongo <righthost>/admin -u <adminusername> -p<adminpassword>
> use local
> db.addUser('repl', <replpassword>);
```

Replacing a Replica Pair Server

When one of the servers in a Mongo replica pair set fails, should it come back online, the system recovers automatically. However, should a machine completely fail, it will need to be replaced, and its replacement will begin with no data. The following procedure explains how to replace one of the machines in a pair.

Let's assume nodes (n1, n2) is the old pair and that n2 dies. We want to switch to (n1,n3).

- 1. If possible, assure the dead n2 is offline and will not come back online: otherwise it may try communicating with its old pair partner.
- 2. We need to tell n1 to pair with n3 instead of n2. We do this with a replacepeer command. Be sure to check for a successful return value from this operation.

```
n1> ./mongo n1/admin
> db.$cmd.findOne({replacepeer:1});
{
    "info" : "adjust local.sources hostname; db restart now required" ,
    "ok" : 1.0
}
```

At this point, n1 is still running but is reset to not be confused when it begins talking to n3 in the future. The server is still up although replication is now disabled.

3. Restart n1 with the right command line to talk to n3

```
nl> ./mongod --pairwith n3 --arbiter <arbiterserver>
```

4. Start n3 paired with n1.

```
n3> ./mongod --pairwith n1 --arbiter <arbiterserver>
```

Note that n3 will not accept any operations as "master" until fully synced with n1, and that this may take some time if there is a substantial amount of data on n1.

Querying the slave

You can query the slave if you set the slave ok flag. In the shell:

```
db.getMongo().setSlaveOk()
```

What is and when should you use an arbiter?

The arbiter is used in some situations to determine which side of a pair is master. In the event of a network partition (left and right are both up, but can't communicate) whoever can talk to the arbiter becomes master.

If your left and right server are on the same switch, an arbiter isn't necessary. If you're running on the same ec2 availability zone, probably not needed as well. But if you've got left and right on different ec2 availability zones, then an arbiter should be used.

Working with an existing (non-paired) database

Care must be taken when enabling a pair for the first time if you have existing datafiles you wish to use that were created from a singleton database. Follow the following procedure to start the pair. Below, we call the two servers "left" and "right".

- assure no mongod processes are running on both servers
- we assume the data files to be kept are on server left. Check that there is no local.* datafiles in left's /data/db (--dbpath) directory. If there are, remove them.
- check that there are no datafiles at all on right's /data/db directory
- start the left process with the appropriate command line including --pairwith argument
- start the right process with the appropriate paired command line

If both left and right servers have datafiles in their dbpath directories at pair initiation, errors will occur. Further, you do not want a local database (which contains replication metadata) during initiation of a new pair.

See Also

· Replica Pairs in Ruby

Master Master Replication

Mongo does not support full master-master replication. However, for certain restricted use cases master-master can be used. Generally, we recommend one does not use the database in a master-master mode.

Master-master usages is eventually consistent.

To configure master-master, simply run both databases with both the --master and --slave parameters. For example, to set up this configuration on a single machine as a test one might run:

```
$ nohup mongod --dbpath /data1/db --port 27017 --master --slave --source localhost:27018 > /tmp/dblog1
&
$ nohup mongod --dbpath /data2/db --port 27018 --master --slave --source localhost:27017 > /tmp/dblog2
&
```

This mode is safe for:

- insert operation
- delete operations by _id;
- any query

Master-master should not be used if:

• concurrent updates of single object may occur (including \$inc and other updates)

A sample test session on a single computer follows:

```
$ # terminal 1, we run a mongod on default db port (27017)
$ ./mongod --slave --master --source localhost:10000
$ # terminal 2, we run a mongod on port 10000
$ ./mongod --slave --master --dbpath /data/slave --port 10000 --source localhost
$ ./mongo
> // 'db' is now connected to localhost:27017/test
> z = connect("localhost:10000/test")
> // 'z' is now connected to localhost:10000/test db
> db.foo.insert({x:7});
> z.foo.find()
{"_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
> db.foo.find()
{"_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
> db.foo.insert({x:8})
> db.foo.find()
{"\_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
{"_id" : ObjectId( "4ab9182a938798896fd8a906") , "x" : 8}
> z.foo.find()
{"_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
{"_id" : ObjectId( "4ab9182a938798896fd8a906") , "x" : 8}
> z.foo.save({x:9})
> z.foo.find()
{"\_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
{"_id" : ObjectId( "4ab9182a938798896fd8a906") , "x" : 8}
{"_id" : ObjectId( "4ab9188ac50e4c10591ce3b7") , "x" : 9}
> db.foo.find()
{"_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
{"_id" : ObjectId( "4ab9182a938798896fd8a906") , "x" : 8}
{"_id" : ObjectId( "4ab9188ac50e4c10591ce3b7") , "x" : 9}
> z.foo.remove({x:8})
> db.foo.find()
{"_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
{"_id" : ObjectId( "4ab9188ac50e4c10591ce3b7")
> z.foo.find()
{"_id" : ObjectId( "4ab917d7c50e4c10591ce3b6") , "x" : 7}
{"_id" : ObjectId( "4ab9188ac50e4c10591ce3b7") , "x" : 9}
> db.foo.drop()
{"nIndexesWas" : 1 , "msg" : "all indexes deleted for collection" , "ns" : "test.foo" , "ok" : 1}
> db.foo.find()
> z.foo.find()
```

Production Notes

- Architecture
 - Production Options
 - Backups
- Recommended System Settings
- TCP Port Numbers
- See Also

Architecture

Production Options

- Master Slave
 - 1 master, N slaves have to handle failover manually

- Replica Pairs
 - · 2 servers, 1 is always master, auto-failover

Backups

• Import Export Tools

Recommended System Settings

- · turn off atime
- set file descriptor limit to 4+k

TCP Port Numbers

Default TCP port numbers for MongoDB processes:

- standalone mongod 27017
- mongos 27017
- shard server (mongod --shardsvr) 27018
- config server (mongod --configsvr) 27019
- web stats page for mongod add 1000 to port number

See Also

Starting and Stopping the Database

Security and Authentication

Running Without Security (Trusted Environment)

One valid way to run the Mongo database is in a trusted environment, with no security and authentication. This is the default option and is recommended. Of course, in such a configuration, one must be sure only trusted machines can access database TCP ports.

The current revision of sharding requires trusted (nonsecure) mode.

Mongo Security

The current version of Mongo supports only very basic security. One authenticates a username and password in the context of a particular database. Once authenticated, a normal user has full read and write access to the database in question while a read only user only has read access.

The admin database is special. In addition to several commands that are administrative being possible only on admin, authentication on admin gives one read and write access to all databases on the server. Effectively, admin access means root access to the server process.

Run the database (mongod process) with the --auth option to enable security. You **must** have added a user to the admin db before starting the server with --auth. To run the database with security checking for writes only, use the --authWriteOnly option.

Configuring Authentication and Security

Authentication is stored in each database's system.users collection. For example, on a database projectx, projectx.system.users will contain user information.

We should first configure an administrator user for the entire db server process. This user is stored under the special admin database.

If no users are configured in admin.system.users, one may access the database from the localhost interface without authenticating. Thus, from the server running the database (and thus on localhost), run the database shell and configure an administrative user:

```
$ ./mongo
> use admin
> db.addUser("theadmin", "anadminpassword")
```

We now have a user created for database admin. Note that if we have not previously authenticated, we now must if we wish to perform further operations, as there is a user in admin.system.users.

```
> db.auth("theadmin", "anadminpassword")
```

We can view existing users for the database with the command:

```
> db.system.users.find()
```

Now, let's configure a "regular" user for another database.

```
> use projectx
> db.addUser("joe", "passwordForJoe")
```

Finally, let's add a readonly user.

```
> use projectx
> db.addUser("guest", "passwordForGuest", true)
```

Changing Passwords

The shell addUser command may also be used to update a password: if the user already exists, the password simply updates.

Many Mongo drivers provide a helper function equivalent to the db shell's addUser method.

Deleting Users

To delete a user:

```
db.system.users.remove( { user: username } )
```

Mongo Architecture

MongoDB has two primary components to the database server. The first is the mongod process which is the core database server. In many cases, mongod may be used as a self-contained system similar to how one would use mysqld on a server. Separate mongod instances on different machines (and data centers) can replicate from one to another.

Another MongoDB process, mongos, facilitates auto-sharding. mongos can be thought of as a "database router" to make a cluster of mongod processes appear as a single database. See the sharding documentation for more information.

Database Caching

With relational databases, object caching is usually a separate facility (such as memcached), which makes sense as even a RAM page cache hit is a fairly expensive operation with a relational database (joins may be required, and the data must be transformed into an object representation). Further, memcached type solutions are more scaleable than a relational database.

Mongo eliminates the need (in some cases) for a separate object caching layer. Queries that result in file system RAM cache hits are very fast as the object's representation in the database is very close to its representation in application memory. Also, the MongoDB can scale to any level and provides an object cache and database integrated together, which is very helpful as there is no risk of retrieving stale data from the cache. In addition, the complex queries a full DBMS provides are also possible.

Next: Mongo Concepts and Terminology

How to do Snapshotted Queries in the Mongo Database



This document refers to query snapshots. For backup snapshots of the database's datafiles, see the fsync lock page.

For performance reasons, MongoDB does not currently support true point-in-time snapshotting of collections. (This may change in the future.) However, some functionality is available which is detailed below.

Cursors

A MongoDB query returns data as well as a cursor ID for additional lookups, should more data exist. Drivers lazily perform a "getMore" operation as needed on the cursor to get more data. Cursors may have latent getMore accesses that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete).

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

Mongo DB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations. In fact, it is even possible (although unlikely) to see the same object returned twice if the object were updated and grew in size (and thus moved in the datafile). To assure no update duplications, use snapshot() mode (see below).

Snapshot Mode

snapshot() mode assures that objects which update during the lifetime of a query are returned once and only once. This is most important when doing a find-and-update loop that changes the size of documents that are returned (\$inc does not change size).

```
> // mongo shell example
> var cursor = db.myCollection.find({country:'uk'}).snapshot();
```

Even with snapshot mode, items inserted or deleted during the query may or may not be returned; that is, this mode is not a true point-in-time snapshot (currently).

Note that a true point-in-time snapshot occurs for short query responses (less than 1MB).

Because snapshot mode traverses the _id index, it may not be used with sorting or explicit hints. It also cannot use any other index for the query.

You can get the same effect as snapshot by using any unique index on a field(s) that will not be modified (probably best to use explicit hint() too). If you want to use a non-unique index (such as creation time), you can make it unique by appending _id to the index at creation time.

See Also

• The Advanced Queries section of MongoDB - A Developer's Tour

Backups

- Fsync, Write Lock and Backup
- Shutdown and Backup
- Exports
- Slave Backup

Several strategies exist for backing up MongoDB databases. A word of warning: it's not safe to back up the mongod data files (by default in /data/db/) while the database is running and writes are occurring; such a backup may turn out to be corrupt. See the fsync option below for a way around that.

Fsync, Write Lock and Backup

MongoDB v1.3.1 and higher supports an fsync and lock command with which we can flush writes, lock the database to prevent writing, and then backup the datafiles.

While in this locked mode, all writes will block. If this is a problem consider one of the other methods below.

Shutdown and Backup

A simple approach is just to stop the database, back up the data files, and resume. This is safe but of course requires downtime.

Exports

The mongodump utility may be used to dump an entire database, even when the database is running and active. The dump can then be restored later if needed.

Slave Backup

Another good technique for backups is replication to a slave database. The slave polls master continuously and thus always has a nearly-up-to-date copy of master.

We then have several options for backing up the slave:

- 1. Fsync, write lock, and backup the slave.
- 2. Shut it down, backup, and restart.
- 3. Export from the slave.

For methods 1 and 2, after the backup the slave will resume replication, applying any changes made to master in the meantime.

Using a slave is advantageous because we then always have backup database machine ready in case master fails (failover). But a slave also gives us the chance to back up the full data set without affecting the performance of the master database.

Database Profiler

Mongo includes a profiling tool to analyze the performance of database operations.

See also the currentOp command.

Enabling Profiling

To enable profiling, from the mongo shell invoke:

```
> db.setProfilingLevel(2);
{"was" : 0 , "ok" : 1}
> db.getProfilingLevel()
2
```

Profiling levels are:

- 0 off
- 1 log slow operations (>100ms)
- 2 log all operations

Staring in 1.3.0, you can also enable on the command line, --profile=1

Viewing

Profiling data is recorded in the database's system.profile collection. Query that collection to see the results.

```
> db.system.profile.find()
{"ts" : "Thu Jan 29 2009 15:19:32 GMT-0500 (EST)" , "info" : "query test.$cmd ntoreturn:1 reslen:66
nscanned:0 <br>query: { profile: 2 } nreturned:1 bytes:50" , "millis" : 0}
...
```

To see output without \$cmd (command) operations, invoke:

```
db.system.profile.find( function() { return this.info.indexOf('$cmd')<0; } )</pre>
```

To view operations for a particular collection:

```
> db.system.profile.find( { info: /test.foo/ } )
{"ts": "Thu Jan 29 2009 15:19:40 GMT-0500 (EST)", "info": "insert test.foo", "millis": 0}
{"ts": "Thu Jan 29 2009 15:19:42 GMT-0500 (EST)", "info": "insert test.foo", "millis": 0}
{"ts": "Thu Jan 29 2009 15:19:45 GMT-0500 (EST)", "info": "query test.foo ntoreturn:0 reslen:102
nscanned:2 <br>
compared:2 <br/>futs": "Thu Jan 29 2009 15:21:17 GMT-0500 (EST)", "info": "query test.foo ntoreturn:0 reslen:36
nscanned:2 <br/>futs": "Thu Jan 29 2009 15:21:27 GMT-0500 (EST)", "info": "query test.foo ntoreturn:0 reslen:36
nscanned:2 <br/>futs": "Thu Jan 29 2009 15:21:27 GMT-0500 (EST)", "info": "query test.foo ntoreturn:0 exception
bytes:53", "millis": 88}
```

To view operations slower than a certain number of milliseconds:

```
> db.system.profile.find( { millis : { $gt : 5 } } )
{"ts" : "Thu Jan 29 2009 15:21:27 GMT-0500 (EST)" , "info" : "query test.foo ntoreturn:0 exception
bytes:53" , "millis" : 88}
```

To see newest information first:

```
db.system.profile.find().sort({$natural:-1})
```

The mongo shell includes a helper to see the most recent 5 profiled events that took at least 1ms to execute. Type show profile at the command prompt to use this feature.

Understanding the Output

The output reports the following values:

- ts Timestamp of the profiled operation.
- millis Time, in milliseconds, to perform the operation.
- info Details on the operation.
 - query A database query operation. The query info field includes several additional terms:
 - ntoreturn Number of objects the client requested for return from a query. For example, <code>findOne()</code> sets
 ntoreturn to 1. <code>limit()</code> sets the appropriate limit. Zero indicates no limit.
 - query Details of the query spec.
 - nscanned Number of objects scanned in executing the operation.
 - reslen Query result length in bytes.
 - nreturned Number of objects returned from query.
 - update A database update operation. <code>save()</code> calls generate either an update or insert operation.
 - fastmod Indicates a fast modify operation. See Updates. These operations are normally quite fast.
 - fastmodinsert indicates a fast modify operation that performed an upsert.
 - upsert Indicates on upsert performed.
 - moved Indicates the update moved the object on disk (not updated in place). This is slower than an in place update, and normally occurs when an object grows.
 - insert A database insert.
 - getmore For large queries, the database initially returns partial information. getmore indicates a call to retrieve further information.

Optimizing Query Performance

- If nscanned is much higher than nreturned, the database is scanning many objects to find the target objects. Consider creating an index to improve this.
- reslen A large number of bytes returned (hundreds of kilobytes or more) causes slow performance. Consider passing
 code>find()</code> a second parameter of the member names you require.

Note: There is a cost for each index you create. The index causes disk writes on each insert and some updates to the collection. If a rare query, it may be better to let the query be "slow" and not create an index. When a query is common relative to the number of saves to the collection, you will want to create the index.

Optimizing Update Performance

- Examine the nscanned info field. If it is a very large value, the database is scanning a large number of objects to find the object to
 update. Consider creating an index if updates are a high-frequency operation.
- Use fast modify operations when possible (and usually with these, an index). See Updates.

Profiler Performance

When enabled, profiling affects performance, although not severely.

Profile data is stored in the database's system.profile collection, which is a Capped Collection. By default it is set to a very small size and thus only includes recent operations.

Configuring "Slow"

Since 1.3.0 there are 2 ways to configure "slow"

- --slowms on the command line when starting mongod (or file config)
- db.setProfilingLevel(level , slowms)

```
db.setProfilingLevel( 1 , 10 );
```

will log all queries over 10ms to system.profile

See Also

- Optimization
- explain()
- Viewing and Terminating Current Operation

Diagnostics and Troubleshooting

Mongo provides the following tools for diagnostics and troubleshooting:

Profiler

This is the best way to see slow queries. See full docs

Database Record/Replay

This is a diagnostic capability.

To enable logging:

```
db._adminCommand( { diagLogging : 1 } )
```

To disable:

```
db._adminCommand( { diagLogging : 0 } )
```

Values for opLogging:

- 0 off. Also flushes any pending data to the file.
- 1 log writes
- 2 log reads
- 3 log both

Note: if you log reads, it will record the findOnes above and if you replay them, that will have an effect!

Output is written to diaglog.bin_ in the /data/db/ directory (unless --dbpath is specified).

Replaying

To replay the logged events:

```
nc ''database_server_ip'' 27017 < ''somelog.bin'' | hexdump -c
```

REVIEW THIS: No code exists yet to record these requests and actually play them back.

Durability and Repair

- Repairing a Database
- Thoughts on Durability
 - Tradeoffs
- See Also

Relational database systems achieve durability, in part, by writing all updates to a transaction log. However, transaction logging incurs a performance penalty, so MongoDB takes an alternate approach to durability: replication.

Because MongoDB forgoes transaction logging, it's possible to lose data on a hardware or process crash (or kill -9). Data can be lost when writes have occurred recently and haven't yet been synced to disk. If a database file becomes corrupted, it will be necessary to repair the

database.



Since 1.1.4, the --syncdelay option controls how often changes are flushed to disk (the default is 60 seconds). If replication is not being used, then it may be desirable to reduce this default.

Repairing a Database

In the event of a crash, we recommend running a repair - analgous to running fsck. If a slave crashes, one option is just to re-slave and start from scratch which will likely be just as fast.

From the command line mongod --repair

From the shell:

```
> db.repairDatabase();
```

Analgous to running fsck on the file system, this command will repair the database.

To check the condition of any particular collection, there's a validation command that returns a lot of useful information. For example, here we validate the users collection:

```
> db.users.validate();
 "ns" : "test.users",
 "result" : " validate
 details: 0x1243dbbdc ofs:740bdc
  firstExtent:0:178b00 ns:test.users
  lastExtent:0:178b00 ns:test.users
  # extents:1
 datasize?:44 nrecords?:1 lastExtentSize:8192
 padding:1
  first extent:
   loc:0:178b00 xnext:null xprev:null
   nsdiag:test.users
   size:8192 firstRecord:0:178bb0 lastRecord:0:178bb0
  1 objects found, nobj:1
  60 bytes data w/headers
  44 bytes data wout/headers
  deletedList: 000000010000000000
  deleted: n: 1 size: 7956
 nIndexes:2
    test.users.$ id keys:1
    test.users.$username_1 keys:1 ",
 "ok" : 1,
 "valid" : true,
 "lastExtentSize" : 8192
```

During a repair operation, mongod must store temporary files to disk. By default, mongod creates temporary directories under the dbpath for this purpose. Alternatively, the --repairpath command line option can be used to specify a base directory for temporary repair files.

Thoughts on Durability

If you question the idea of forgoing transaction logging, keep in mind that this idea is not unique to MongoDB. For example, MySQL's MyISAM storage engine also dispenses with transaction logging.

The philosophy behind MongoDB is that different styles of databases should be used for different problems. MongoDB attempts to provide high performance at some reduction of features. Of course, some problems require greater durability: we would not recommend using it for a bond trading system. However, for many problems, particularly with scaling web site infrastructure, MongoDB works quite well in practice (see our growing list of production deployments).

Tradeoffs

Full durability comes at a significant cost to write performance. For this reason, modern hard drives have hardware buffering of writes turned on by default. What's more, many traditional RDBMS solutions simply perform an fsync-style flush when flushing the transaction log. This is reasonably fast but incomplete -- on Linux for example, fsync() may return before the data is permanantly stored when the drive has write caching

enabled. (Disabling the write cache on the drive could result in a huge drops in performance).

What all this means is that even the most durable systems are susceptible to corruption and data loss in certain situations. Some research is underway with MongoDB on durability schemes that maintain very high performance. Those desiring greater durability are free to configure the --syncdelay option mentioned above and are encouraged to employ some forms of replication.

See Also

fsync Command

Contributors

- · C++ Coding Style
- Project Ideas
- UI ^{*}
- Source Code
- Building
- Database Internals
- Writing Drivers and Tools
- Contributing to the Documentation

10gen Contributor Agreement

C++ Coding Style



not totally obeyed yet

camelCase

brackets

```
if ( 0 ) {
}
```

class members

```
class Foo {
   int bar;
};
```

templates

```
set<int> s;
```

Project Ideas

If you're interested in getting involved in the MongoDB community (or the open source community in general) a great way to do so is by starting or contributing to a MongoDB related project. Here we've listed some project ideas for you to get started on. For some of these ideas projects are already underway, and for others nothing (that we know of) has been started yet.

If you are working on something or have an idea that you think belongs on this list, please let us know!

A GUI

One feature that is often requested for MongoDB is a GUI, much like CouchDB's futon or phpMyAdmin. There are a couple of projects working on this sort of thing that are worth checking out:

http://github.com/sbellity/futon4mongo

http://www.mongodb.org/display/DOCS/Http+Interface http://www.mongohq.com

We've also started to spec out the features that a tool like this should provide.

Try Mongo!

It would be neat to have a web version of the MongoDB Shell that allowed users to interact with a real MongoDB instance (for doing the tutorial, etc). A project that does something similar (using a basic MongoDB emulator) is here:

http://github.com/banker/mongulator

Real-time Full Text Search Integration

It would be interesting to try to nicely integrate a search backend like Xapian, Lucene or Sphinx with MongoDB. One idea would be to use MongoDB's oplog (which is used for master-slave replication) to keep the search engine up to date.

GridFS FUSE

There is a project working towards creating a FUSE filesystem on top of GridFS - something like this would create a bunch of interesting potential uses for MongoDB and GridFS:

http://github.com/mikejs/gridfs-fuse

GridFS Web Server Modules

There are a couple of modules for different web servers designed to allow serving content directly from GridFS:

Nginx: http://github.com/mdirolf/nginx-gridfs

Lighttpd: http://bitbucket.org/bwmcadams/lighttpd-gridfs

Framework Adaptors

Working towards adding MongoDB support to major web frameworks is a great project, and work has been started on this for a variety of different frameworks (please use google to find out if work has already been started for your favorite framework).

Logging and Session Adaptors

MongoDB works great for storing logs and session information. There are a couple of projects working on supporting this use case directly.

Logging

Zend: http://raphaelstolt.blogspot.com/2009/09/logging-to-mongodb-and-accessing-log.html

Python: http://github.com/andreisavu/mongodb-log Rails: http://github.com/peburrows/mongo_db_logger

Sessions:

web.py: http://github.com/whilefalse/webpy-mongodb-sessions

Beaker: http://pypi.python.org/pypi/mongodb_beaker

Package Managers

Add support for installing MongoDB with your favorite package manager and let us know!

Drivers

If you use an esoteric/new/awesome programming language write a driver to support MongoDB! Again, check google to see what people have started for various languages.

UI

Spec/requirements for a future MongoDB admin UI.

- list databases
 - repair, drop, clone?
 - collections
 - validate(), datasize, indexsize, clone/copy
 - indexes
 - queries explain() output
 - security: view users, adjust

- · see replication status of slave and master
- sharding
- · system.profile viewer; enable disable profiling
- curop / killop support

Source Code

All source for MongoDB, it's drivers, and tools is open source and hosted at Github.

- Mongo Database (includes C++ driver)
- Python Driver
- PHP Driver
- Ruby Driver
- Javá Driver
- Perl Driver

(Additionally, community drivers and tools also exist and will be found in other places.)

See Also

Building

Building

This section provides instructions on setting up your environment to write Mongo drivers or other infrastructure code. For specific instructions, go to the document that corresponds to your setup.

Note: see the Downloads page for prebuilt binaries!

Sub-sections of this section:

- · Building for FreeBSD
- Building for Linux
- Building for OS X
- Building for Solaris
- Building for Windows
 - Boost 1.41.0 Visual Studio 2010 Binary
- · Building Spider Monkey

See Also

• The main Database Internals page

Building for FreeBSD

On FreeBSD 8.0 and later, there is a mongodb port you can use.

For FreeBSD <= 7.2:

- 1. Get the database source: http://www.github.com/mongodb/mongo.
- 2. Update your ports tree: sudo portsnap fetch && portsnap extract (the packages that come by default on 7.2 and older are too old, you'll get weird errors when you try to run the database)
- 3. Install SpiderMonkey: cd /usr/ports/lang/spidermonkey && make && make install
- 4. Install scons: cd /usr/ports/devel/scons && make && make install
- 5. Install boost: cd /usr/ports/devel/boost-all && make && make install (it will pop up an X "GUI", select PYTHON)
- 6. Install libexecinfo: cd /usr/ports/devel/libexecinfo && make && make install
- 7. Change to the database source directory
- 8. scons .

Building for Linux

- General Instructions
- · Special Notes about Spider Monkey
- Package Requirements
 - Fedora
 - Fedora 8 or 10
 - Ubuntu

- Ubuntu 8.04
- Ubuntu 9.04 and 9.10
- See Also

General Instructions

- 1. Install Dependencies see platform specific below
- 2. get source

```
git clone git://github.com/mongodb/mongo.git
```

3. build

```
scons all
```

4. install

```
scons --prefix=/opt/mongo install
```

Special Notes about Spider Monkey

Most pre-built spider monkey binaries don't have UTF8 compiled in. Additionally, ubuntu has a weird version of spider monkey that doesn't support everything we use. If you get any warnings during compile time or runtime, we highly recommend building spider monkey from source. See Building Spider Monkey for more information.

We currently support spider monkey 1.6 and 1.7, although there is some degredation with 1.6, so we recommend using 1.7. We have not yet tested 1.8, but will once it is officially released.

Package Requirements

Fedora

Fedora 8 or 10

```
sudo yum -y install git tcsh scons gcc-c++ glibc-devel
sudo yum -y install boost-devel pcre-devel js-devel readline-devel
#for release builds:
sudo yum -y install boost-devel-static readline-static ncurses-static
```

Ubuntu

See spider monkey note above.

Ubuntu 8.04

```
apt-get -y install tcsh git-core scons g++
apt-get -y install libpcre++-dev libboost-dev libreadline-dev xulrunner-1.9-dev
apt-get -y install libboost-program-options-dev libboost-thread-dev libboost-filesystem-dev
libboost-date-time-dev
```

Ubuntu 9.04 and 9.10

```
apt-get -y install tcsh git-core scons g++
apt-get -y install libpcre++-dev libboost-dev libreadline-dev xulrunner-1.9.1-dev
```

See Also

- · The Building page for setup information for other operating systems
- The main Database Internals page

Building for OS X

- Upgrading to Snow Leopard
- Setup
 - Package Manager Setup (32bit)
 - Manual Setup
 - Install Apple developer tools
 - Install libraries (32bit option)
 - Install libraries (64bit option)
- Compiling
- XCode
- See Also

To set up your OS X computer for MongoDB development:

Upgrading to Snow Leopard

If you have installed Snow Leopard, the builds will be 64 bit -- so if moving from a previous OS release, a bit more setup may be required than one might first expect.

- 1. Install XCode tools for Snow Leopard.
- 2. Install MacPorts (snow leopard version). If you have MacPorts installed previously, we've had the most success by running rm -rf /opt/local first.
- 3. Update/install packages: sudo port install boost pcre.
- 4. Update/install SpiderMonkey with sudo port install spidermonkey. (If this fails, see the note on #2 above.)

Setup

- 1. Install git.
 - You'll want git (you may already have it). If not, download the source and run ./configure; make; sudo make install
- 2. Install gcc.
 - gcc version 4.0.1 (from XCode Tools install) works, but you will receive compiler warnings. The easiest way to upgrade gcc is to install the iPhone SDK.

Package Manager Setup (32bit)

1. Install libraries (using macports)

```
port install boost pcre++ spidermonkey
```

Manual Setup

Install Apple developer tools

Install libraries (32bit option)

1. Download boost {{boost 1.37.0 http://downloads.sourceforge.net/boost/boost_1_37_0.tar.gz}}Apply the following patch:

```
diff -u -r a/configure b/configure
--- a/configure 2009-01-26 14:10:42.000000000 -0500
+++ b/configure 2009-01-26 10:21:29.00000000 -0500
@@ -9,9 +9,9 @@
 BJAM=" "
 TOOLSET=""
-BJAM_CONFIG=""
+BJAM_CONFIG="--layout=system"
BUILD=""
PREFIX=/usr/local
EPREFIX=
diff -u -r a/tools/build/v2/tools/darwin.jam b/tools/build/v2/tools/darwin.jam
--- a/tools/build/v2/tools/darwin.jam 2009-01-26 14:22:08.000000000 -0500 +++ b/tools/build/v2/tools/darwin.jam 2009-01-26 10:22:08.000000000 -0500
@@ -367,5 +367,5 @@
 actions link.dll bind LIBRARIES
     "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name "$(<:B)$(<:S)" -L
"$(LINKPATH)" -0 "$(<)" "$(>)" "$(LIBRARIES)" -1$(FINDLIBS-SA) -1$(FINDLIBS-ST)
 \\ $(FRAMEWORK\_PATH) - framework\\ (_)\\ $(FRAMEWORK:D=:S=) \\ $(OPTIONS) \\ $(USER\_OPTIONS)$ 
     "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name
"/usr/local/lib/$(<:B)$(<:S)" -L"$(LINKPATH)" -o "$(<)" "$(>)" "$(LIBRARIES)" -1$(FINDLIBS-SA)
-1$(FINDLIBS-ST) $(FRAMEWORK_PATH) -framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
```

then.

```
./configure; make; sudo make install
```

2. Install pcre http://www.pcre.org/ (must enable UTF8)

```
./configure --enable-utf8 --enable-unicode-properties --with-match-limit=200000 --with-match-limit-recursion=4000; make; sudo make install
```

3. Install c++ unit test framework http://unittest.red-bean.com/ (optional)

```
./configure; make; sudo make install
```

Install libraries (64bit option)

(The 64bit libraries will be installed in /usr/64/{include,lib}.)

1. Download SpiderMonkey: ftp://ftp.mozilla.org/pub/mozilla.org/js/js-1.7.0.tar.gz

Apply the following patch:

```
diff -u -r js/src/config/Darwin.mk js-1.7.0/src/config/Darwin.mk
--- js/src/config/Darwin.mk 2007-02-05 11:24:49.00000000 -0500
+++ js-1.7.0/src/config/Darwin.mk 2009-05-11 10:18:37.000000000 -0400
@@ -43,7 +43,7 @@
 # Just ripped from Linux config
-CC = cc
+CC = cc -m64
CCC = g++
CFLAGS += -Wall -Wno-format
OS_CFLAGS = -DXP_UNIX -DSVR4 -DSYSV -D_BSD_SOURCE -DPOSIX_SOURCE -DDARWIN
@@ -56,9 +56,9 @@
 #.c.o:
     $(CC) -c -MD $*.d $(CFLAGS) $<
-CPU\_ARCH = \$(shell uname -m)
+CPU_ARCH = "X86_64"
ifeq (86,$(findstring 86,$(CPU_ARCH)))
-CPU\_ARCH = x86
+CPU\_ARCH = x86\_64
OS_CFLAGS+= -DX86_LINUX
endif
GFX\_ARCH = x
@@ -81,3 +81,14 @@
# Don't allow Makefile.ref to use libmath
NO\_LIBM = 1
+ifeq ($(CPU_ARCH),x86_64)
+# Use VA_COPY() standard macro on x86-64
+# FIXME: better use it everywhere
+OS_CFLAGS += -DHAVE_VA_COPY -DVA_COPY=va_copy
+endif
+ifeq ($(CPU_ARCH),x86_64)
+# We need PIC code for shared libraries
+# FIXME: better patch rules.mk & fdlibm/Makefile*
+OS_CFLAGS += -DPIC -fPIC
+endif
```

compile and install

```
cd src
make -f Makefile.ref
sudo JS_DIST=/usr/64 make -f Makefile.ref export
```

remove the dynamic library

```
sudo rm /usr/64/lib64/libjs.dylib
```

Download boost {{boost 1.37.0 http://downloads.sourceforge.net/boost/boost_1_37_0.tar.gz}}Apply the following patch:

```
diff -u -r a/configure b/configure
--- a/configure 2009-01-26 14:10:42.000000000 -0500
+++ b/configure 2009-01-26 10:21:29.00000000 -0500
@@ -9,9 +9,9 @@
BJAM=" "
TOOLSET=""
-BJAM_CONFIG=""
+BJAM_CONFIG="architecture=x86 address-model=64 --layout=system"
BUILD=""
-PREFIX=/usr/local
+PREFIX=/usr/64
EPREFIX=
LIBDIR=
INCLUDEDIR=
diff -u -r a/tools/build/v2/tools/darwin.jam b/tools/build/v2/tools/darwin.jam
--- a/tools/build/v2/tools/darwin.jam 2009-01-26 14:22:08.000000000 -0500
+++ b/tools/build/v2/tools/darwin.jam 2009-01-26 10:22:08.000000000 -0500
@@ -367,5 +367,5 @@
actions link.dll bind LIBRARIES
     "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name "$(<:B)$(<:S)" -L"$(LINKPATH)"
-o "$(<)" "$(>)" "$(LIBRARIES)" -1$(FINDLIBS-SA) -1$(FINDLIBS-ST) $(FRAMEWORK_PATH)
-framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
     "$(CONFIG_COMMAND)" -dynamiclib -Wl,-single_module -install_name "/usr/64/lib/$(<:B)$(<:B)" -L
"$(LINKPATH)" -0 "$(<)" "$(>)" "$(LIBRARIES)" -1$(FINDLIBS-SA) -1$(FINDLIBS-ST) $(FRAMEWORK_PATH)
-framework$(_)$(FRAMEWORK:D=:S=) $(OPTIONS) $(USER_OPTIONS)
}
```

then,

```
./configure; make; sudo make install
```

Install pcre http://www.pcre.org/ (must enable UTF8)

```
CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" ./configure --enable-utf8 --with-match-limit=200000 --with-match-limit-recursion=4000 --enable-unicode-properties --prefix /usr/64; make; sudo make install
```

1. Install unit test framework http://unittest.red-bean.com/ (optional)

```
CFLAGS="-m64" CXXFLAGS="-m64" LDFLAGS="-m64" ./configure --prefix /usr/64; make; sudo make install
```

Compiling

To compile 32bit, just run

```
scons
```

To compile 64bit, run

```
scons --64
```

XCode

You can open the project with:

\$ open mongo.xcodeproj/

You need to add an executable target .:

- 1. In the mongo project window, go to the Executables, right click and choose Add->NewCustomExecutable.
- 2. Name it db. Path is ./db/db. It will appear under Executables".
- 3. Double-click on it.
- 4. Under **general**, set the working directory to the project directory.
- 5. Under arguments, add run.
- 6. Go to general prefs (cmd ,), go to debugging and turn off lazy load. (Seems to be an issue that prevents breakpoints from working in debugger?)

See Also

- The Building page for setup information for other environments
- The main Database Internals page
- The starting point for all Home

Building for Solaris

These are very rough at the moment.

preregs

- g++ 4.x (SUNWgcc)
- · scons (need to install from source)
- spider monkey Building Spider Monkey
- pcre (SUNWpcre)
- boost (need to install from source)

Building for Windows

- · Prerequisites and Setup
- Visual Studio
 - · Running in Visual Studio
 - Preprocessor Defines
 - Visual Studio 2010 (beta 2)
- Building with SCons
- Common Problems

MongoDB can be compiled for Windows (32 and 64 bit) using Visual C++. SCons is used as the make mechanism, although a .vcproj/.sln is also included in the project for convenience when making changes and debugging.

There are several dependencies exist which are listed below; you may find it easier to simply download a pre-built binary.

Prerequisites and Setup

- Install Git
 - Then: git clone git://github.com/mongodb/mongo.git (more info)
- Install SCons:
 - First install Python: http://www.python.org/download/releases/2.6.4/
 - Then SCons itself: http://sourceforge.net/projects/scons/files/scons/1.2.0/scons-1.2.0.win32.exe/download
 - Add the python scripts directory (e.g., C:\Python26\Scripts) to your PATH
- Install Visual C++ Express or Visual Studio 2008
- Install Boost
 - 32-bit: http://www.boostpro.com/products/free install the prebuilt libraries for Boost version 1.35.0 (or higher generally newer is better). During installation, for release builds choose static multithread libraries for installation. The Debug version of the project uses the DLL libraries; choose all multithread libraries if you plan to do development. From the BoostPro installer, be sure to select all relevant libraries that mongodb uses -- for example, you need Filesystem, Regex, Threads, and ProgramOptions (and perhaps others).
 - 64-bit: have to use boost 1.39. compile boost: bjam variant=debug,release link=static threading=multi address-model=64 runtime-link=static
 - If boost headers are not found when compiling, add the boost home location in Tools.Options.Projects&Solutions.VC++
 Directories.Include; for example, "c:\program files\boost\boost_1_39".

- If boost libs are not found when linking, add the boost lib location in Tools.Options.Projects&Solutions.VC++
 Directories.Libraries; for example, "c:\program files\boost\boost_1_39\lib".
- Build a SpiderMonkey js engine library (js.lib) details here

Visual Studio

Mongo can be compiled on Windows with Visual Studio 2008. The Visual Studio debugger is actually a very nice way to perform debugging.

Note: After installing Visual Studio, you may want to go to Control Panel->Services and disable SQL Server Express, as we won't be using that.

Running in Visual Studio

Open the solution for Mongo. There are two options:

- 1. msvc/mongo.sln. This is the new, comprehensive .sln file.
- 2. db/db.sln. This is the old solution file. It is not as nice, but can be used if one has any issue with #1.

Preprocessor Defines

These will be set for the vcproj already, but just so you know:

- _CRT_SECURE_NO_WARNINGS suppressed warnings on Visual Studio
- HAVE_CONFIG_H for PCRE on Windows

Note: If you receive "error building program database" or other errors building, install the Visual Studio 2008 Service Pack 1 (if it is not already present).

Visual Studio 2010 (beta 2)

The project now includes a Visual Studio 2010 solution file -- db/db_10.sln (as well as .vcxproj files).

Boost 1.41.0 compiles with Visual Studio 2010 -- you will want that version of boost. A prebuilt boost binary may be found here.

Building with SCons

Run scons in the mongo project directory to build.

When using SCons you will want the Visual C++ files to be on your path; one way to do that is use Tools...Visual Studio Command Prompt to get a prompt from Visual Studio (or manually set up your PATH).

To build mongod:

scons

To build the C++ client driver library:

• scons mongoclient.lib

To build all end use components:

scons all

To build all components including components including unit tests:

• scons.

Common Problems

 Can't find jstypes.h when compiling. This file is generated when building SpiderMonkey. See the Building SpiderMonkey page for more info.

Boost 1.41.0 Visual Studio 2010 Binary

The following is a prebuilt boost binary (libraries) for Visual Studio 2010 beta 2.

The MongoDB vcxproj files assume this package is unzipped under c:\Program Files\boost\boost_1_41_0\.

http://downloads.mongodb.org/misc/boost_1_41_0_binary_vs10beta2.zipx

Note: we're not boost build gurus please let us know if there are things wrong with the build.

See also the prebuilt boost binaries at http://www.boostpro.com/download.

Building Spider Monkey

MongoDB uses SpiderMonkey for server-side Javascript execution. (V8 support is under development.)

Building js.lib

Download

```
curl -O ftp://ftp.mozilla.org/pub/mozilla.org/js/js-1.7.0.tar.gz
tar zxvf js-1.7.0.tar.gz
```

Build

```
cd js/src
export CFLAGS="-DJS_C_STRINGS_ARE_UTF8"
make -f Makefile.ref
```

SpiderMonkey does not use UTF-8 by default, so we enable before building.

An experimental SConstruct build file is available here.

Install

```
JS_DIST=/usr make -f Makefile.ref export
```

By default, the mongo scons project expects spidermonkey to be located at ../js/.

Windows

For convenience when building on Windows, a prebuilt SpiderMonkey library and headers for Win32 is attached to this document.

Database Internals

This section provides information for developers who want to write drivers or tools for MongoDB, \ contribute code to the MongoDB codebase itself, and for those who are just curious how it works internally.

Sub-sections of this section:

- Caching
- Concurrency
- Cursors
- Sharding Internals
 - Moving Chunks
 - Sharding Commands
 - Sharding Design
 - Sharding Use Cases
 - Shard Ownership
 - Splitting Shards
- Error Codes
- Internal Commands
- Replication Internals
- Smoke Tests
- Pairing Internals

Caching

MongoMemMapped_RecStore Storage Engine

This is the current default storage engine for MongoDB, and it uses memory-mapped files for all disk I/O. Using this strategy, the operating

system's virtual memory manager is in charge of caching. This has several implications:

- There is no redundancy between file system cache and database cache: they are one and the same.
- MongoDB can use all free memory on the server for cache space automatically without any configuration of a cache size.
- Virtual memory size and resident size will appear to be very large for the mongod process. This is benign: virtual memory space will be
 just larger than the size of the datafiles open and mapped; resident size will vary depending on the amount of memory not used by other
 processes on the machine.
- Caching behavior (such as LRU'ing out of pages, and laziness of page writes) is controlled by the operating system: quality of the VMM implementation will vary by OS.

CachedBasicRecStore (alternative) Storage Engine

An alternative storage engine (CachedBasicRecStore), which does not use memory-mapped files, is under development. This engine is more traditional in design with its own page cache. With this store the database has more control over the exact timing of reads and writes, and of the cache LRU strategy.

Generally, the memory-mapped store works quite well. The alternative store will be useful in cases where an operating system's virtual memory manager is behaving suboptimally.

Concurrency

This page has implementation details on mongod process concurrency.

- logging Logstream::mutex
- clientcursor maps mutex in ClientCursor
- lastAssert
- nextmessageid
- rand() calls, getNonce?

MongoMutex

Cursors

Error formatting macro: redirect: java.lang.NullPointerException

Sharding Internals

This section includes internal implementation details for MongoDB auto sharding. See also the main sharding documentation.

Note: some internals docs could be out of date -- if you see that let us know so we can fix.

Internals

- Moving Chunks
- Sharding Commands
- Sharding Design
- Sharding Use Cases
- Shard Ownership
- Splitting Shards

Unit Tests

./mongo --nodb jstests/sharding/*js

Moving Chunks

- inc version
- try to set on from
- if set is successful, have it "locked"
- start transfer
- finish transfer
- · commit result

max version for a shard is MAX(chunks on shard)

this poses slight problem when moving last chunk off of a shard, so add a special marker

Sharding Commands

All of these commands need to be run on the admin database when connected to mongos.

db management

db.runCommand({ addserver : "localhost:30001" })
 db.runCommand({ movePrimary : "foo" , to : "localhost:30001" })

partition/sharding management

- db.runCommand({ partition : "foo" }); this turns on partitioning for a db.
 - you have to turn on partitioning to be able to have different collections on different servers, or to use sharding
- db.runCommand({ shard : "foo.users" , key : { name : 1 } })

db commands relevant to the mongoshard process

- db.\$cmd.findOne({isdbgrid:1}) use to test if you are speaking to a dbgrid process or straight to a mongo db.
- db.\$cmd.findOne({ismaster:1}) normally used to check which member of a replica pair is currently master. dbgrid returns { ismaster: 0.0, msg: "isdbgrid" }
- db.\$cmd.findOne({netstat:1}) check on health of connections between dbgrid and the various db processes

Sharding Design

concepts

- · config database the top level database that stores information about servers and where things live.
- shard. this can be either a single server or a replica pair.
- database one top level namespace. a database can be partitioned or not
- chunk a region of data from a particular collection. A chunk can be though of as (collectionname, fieldname, lowvalue, highvalue). The range is inclusive on the low end and exclusive on the high end, i.e., [lowvalue, highvalue).

components and database collections

- config database
- config.servers this contains all of the servers that the system has. These are logical servers. So for a replica pair, the entry would be 192.168.0.10,192.168.0.11
- config.databases all of the databases known to the system. This contains the primary server for a database, and information about whether its partitioned or not.
 - · config.shards a list of all database shards. Each shard is a db pair, each of which runs a db process.
 - config.homes specifies which shard is home for a given client db.
- shard databases
 - client.system.chunklocations the home shard for a given client db contains a client.system.chunklocations collection. this
 collection lists where to find particular chunks; that is, it maps chunk->shard.
- mongos process
 - "routes" request to proper db's, and performs merges, can have a couple per system, or can have 1 per client server.
 - gets chunk locations from the client db's home shard. load lazily to avoid using too much mem.
 - chunk information is cached by mongos. This information can be stale at a mongos (it is always up to date at the owning shard; you cannot migrate an item if the owning shard is down). If so, the shard contacted will tell us so and we can then retry to the proper location.

db operations

- moveprimary move a database's primary server
- migrate migrate a chunk from one machine to another.
 - lock and migrate
 - shard db's coordinate with home shard to atomically pass over ownership of the chunk (two phase commit)
- split split a chunk that is growing too large into pieces. as the two new chunks are on the same machine after the split, this is really just
 a metadata update and very fast.
- reconfiguration operations
 - add shard dbgrid processes should lazy load information on a new (unknown) shard when encountered.
 - retire shard in background gradually migrate all chunks off

minimizing lock time

If a chunk is migrating and is 50MB, that might take 5-10 seconds which is too long for the chunk to be locked.

We could perform the migrate much like Cloner works, where we copy the objects and then apply all operations that happened during copying. This way lock time is minimal.

Sharding Use Cases

What specific use cases do we want to address with db partioning (and other techniques) that are challenging to scale? List here for discussion.

- video site (e.g., youtube) (also, GridFS scale-up)
 - · seems straightforward: partition by video
 - for related videos feature, see search below
- social networking (e.g., facebook)
 - this can be guite hard to partition, because it is difficult to cluster people.
- · very high RPS sites with small datasets
 - N replicas, instead of partioning, might help here
 - replicas only work if the dataset is really small as we are using/wasting the same RAM on each replica. thus, partioning might help us with ram cache efficiency even if entire data set fits on one or two drives.
- twitter
- · search & tagging

Log Processing

Use cases related to map-reduce like things.

- massive sort
- · top N queries per day
- · compare data from two nonadjacent time periods

Shard Ownership

By shard ownership we mean which server owns a particular key range.

Early draft/thoughts will change:

Contract

- the master copy of the ownership information is in the config database
- mongos instances have cached info on which server owns a shard, this information may be stale.
- mongod instances have definitive information on who owns a shard (atomic with the config db) when they know about a shards ownership

mongod

The mongod processes maintain a cache of shards the mongod instance owns:

```
map<ShardKey,state> ownership
```

State values are as follows:

- missing no element in the map means no information available. In such a situation we should guery the config database to get the state.
- 1 this instance owns the shard
- 0 this instance does not own the shard (indicates we gueried the config database and found another owner, and remembered that fact)

Initial Assignment of a region to a node.

This is trivial: add the configuration to the config db. As the ShardKey is new, no nodes have any cached information.

Splitting a Key Range

The mongod instance A which owns the range R breaks it into R1,R2 which are still owned by it. It updates the config db. We take care to handle the config db crashing or being unreachable on the split:

```
lock(R) on A update the config db -- ideally atomically perhaps with eval(). await return code. ownership[R].erase unlock(R) on A
```

After the above the cache has no information on the R,R1,R2 ownerships, and will requery configdb on the next request. If the config db crashed and failed to apply the operation, we are still consistent.

Migrate ownership of keyrange R from server A->B. We assume here that B is the coordinator of the job:

```
B copies range from A lock(R) on A and B
B copies any additional operations from A (fast)
clear ownership maps for R on A and B. B waits for a response from A on this operation.
B then updates the ownership data in the config db. (Perhaps even fsyncing.) await return code.
unlock(R) on B
delete R on A (cleanup)
unlock (R) on A
```

We clear the ownership maps first. That way, if the config db update fails, nothing bad happens, IF mongos filters data upon receipt for being in the correct ranges (or in its query parameters).

R stays locked on A for the cleanup work, but as that shard no longer owns the range, this is not an issue even if slow. It stays locked for that operation in case the shard were to quickly migrate back.

Migrating Empty Shards

Typically we migrate a shard after a split. After certain split scenarios, a shard may be empty but we want to migrate it.

Splitting Shards

For chunk [lowbound,highbound)->shard, there are two types of splits we can perform. In one case, we split the range s.t. both new chunks have about the same number of values, in another, we split the range but one of the new chunk is empty.

When at least one bound is MinKey or MaxKey, we choose a split strategy s.t. all existing values remain in one chunk. For example:

```
chunk [MinKey, MaxKey) values { 1,5,9,12 }
  becomes
  chunk [MinKey, 13) values { 1,5,9,12 }
  chunk [13, MaxKey) values { }

whereas

chunk [0 , 15) values { 1,5,9,12 }
  becomes
  chunk [0, 9) values { 1,5 }
  chunk [9, 15) values { 9,12 }
```

This approach is more efficient when items are added in key order.

Error Codes

Error Code	Description	Comments	
E10003	failing update: objects in a capped ns cannot grow		
E11000	duplicate key error	_id values must be unique in a collection	
E11001	duplicate key on update		
E12000	idxNo fails	an internal error	
E12001	can't sort with \$snapshot	the \$snapshot feature does not support sorting yet	
E12010 - E12012	can't \$inc/\$set an indexed field		

Internal Commands

Most commands have helper functions and do not require the \$cmd.findOne() syntax. These are primarily internal and administrative.

```
> db.$cmd.findOne({assertinfo:1})
{
    "dbasserted" : false , // boolean: db asserted
"asserted" : false , // boolean: db asserted or a user assert have happend
"assert" : "" , // regular assert
"assertw" : "" , // "warning" assert
"assertmsg" : "" , // assert with a message in the db log
"assertuser" : "" , // user assert - benign, generally a request that was not meaningful
}
> db.$cmd.findOne({serverStatus:1})
  "uptime" : 6 ,
  "globalLock" : {
                    "totalTime" : 6765166 ,
                    "lockTime" : 2131 ,
                    "ratio" : 0.00031499596610046226
                 } ,
   "mem" : {
                    "resident" : 3 ,
                    "virtual" : 111 ,
                    "mapped" : 32
  "ok" : 1
}
> admindb.$cmd.findOne({replacepeer:1})
    "info" : "adjust local.sources hostname; db restart now required" ,
    "ok" : 1.0
}
// close all databases. a subsequent request will reopen a db.
> admindb.$cmd.findOne({closeAllDatabases:1});
```

Replication Internals

On the *master* mongod instance, the local database will contain a collection, oplog. Smain, which stores a high-level transaction log. The transaction log essentially describes all actions performed by the user, such as "insert this object into this collection." Note that the oplog is not a low-level redo log, so it does not record operations on the byte/disk level.

The slave mongod instance polls the oplog. \$main collection from master. The actual query looks like this:

```
local.oplog.$main.find({ ts: { $gte: ''last_op_processed_time'' } }).sort({$natural:1});
```

where 'local' is the master instance's local database. oplog. \$main collection is a capped collection, allowing the oldest data to be aged out automatically.

See the Replication section of the Mongo Developers' Guide for more information.

OpTime

An OpTime is a 64-bit timestamp that we use to timestamp operations. These are stored as Javascript Date datatypes but are not JavaScript Date objects. Implementation details can be found in the OpTime class in repl.h.

Applying OpTime Operations

Operations from the oplog are applied on the slave by reexecuting the operation. Naturally, the log includes write operations only.

Note that inserts are transformed into upserts to ensure consistency on repeated operations. For example, if the slave crashes, we won't know

exactly which operations have been applied. So if we're left with operations 1, 2, 3, 4, and 5, and if we then apply 1, 2, 3, 2, 3, 4, 5, we should achieve the same results. This repeatability property is also used for the initial cloning of the replica.

Tailing

After applying operations, we want to wait a moment and then poll again for new data with our \$gteoperation. We want this operation to be fast, quickly skipping past old data we have already processed. However, we do not want to build an index on ts, as indexing can be somewhat expensive, and the oplog is write-heavy. Instead, we use a table scan in natural order, but use a tailable cursor to "remember" our position. Thus, we only scan once, and then when we poll again, we know where to begin.

Initiation

To create a new replica, we do the following:

```
t = now();
cloneDatabase();
end = now();
applyOperations(t..end);
```

cloneDatabaseeffectively exports/imports all the data in the database. Note the actual "image" we will get may or may not include data modifications in the time range (t..end). Thus, we apply all logged operations from that range when the cloning is complete. Because of our repeatability property, this is safe.

See class Cloner for more information.

Smoke Tests

```
// run basic c++ unit tests:
scons smoke

// run all unit tests (slow)

// NOTE This scons target starts a mongod instance needed by some tests.
scons smokeAll

// to run basic jstests

// first start mongod, then:
scons smokeJs
```

Full list of scons targets for running smoke tests:

smoke, smokePerf, smokeClient, mongosTest, smokeJs, smokeClone, smokeRepl, smokeDisk, smokeSharding, smokeJsPerf, smokeQuota, smokeTool

Note that before running smokeClient, smokeJs, smokeJsPerf, or smokeQuota a mongod instance must first be started.

Pairing Internals

Policy for reconciling divergent oplogs

In a paired environment, a situation may arise in which each member of a pair has logged operations as master that have not been applied to the other server. In such a situation, the following procedure will be used to ensure consistency between the two servers:

- 1. The new master will scan through its own oplog from the point at which it last applied an operation from it's peer's oplog to the end. It will create a set C of object ids for which changes were made. It will create a set M of object ids for which only modifier changes were made. The values of C and M will be updated as client operations are applied to the new master.
- 2. The new master will iterate through its peer's oplog, applying only operations that will not affect an object having an id in C.
- 3. For any operation in the peer's oplog that may not be applied due to the constraint in the previous step, if the id of the object in question is in M, the value of the whole object on the new master is logged to the new master's oplog.
- 4. The new slave applies all operations from the new master's oplog.

Writing Drivers and Tools

Overview - Writing Drivers and Tools

This section contains information for developers that are working with the low-level protocols of Mongo - people who are writing drivers and higher-level tools.

Documents of particular interest :

BSON	Description of the BSON binary document format. Fundamental to how Mongo and it's client software works.
Mongo Wire Protocol	Specification for the basic socket communications protocol used between Mongo and clients.
Mongo Driver Requirements	Description of what functionality is expected from a Mongo Driver
GridFS Specification	Specification of GridFS - a convention for storing large objects in Mongo
Mongo Extended JSON	Description of the extended JSON protocol for the REST-ful interface (ongoing development)

Additionally we recommend driver authors take a look at existing driver source code as an example.

Mongo Driver Requirements

- Provisional Driver API
 - "Level 0 Driver"
 - Level 0 Driver Features
 - Required
 - Optional
 - Server Object
 - Single Server Connection
 - Pair Mode Connection
 - Database Object
 - **Database Administration**
 - Collection
 - Basic Ops
 - Index Operations
 - Misc Operations
 - Cursor Object
 - "Level 1 Driver"
 - · Level 1 Database Object
 - Level 1 Collection Object

Provisional Driver API

"Level 0 Driver"

A "Level 0" driver is one that exposes and supports the basic operations of Mongo with no value-add. I think that all operations at level 0 should be "document" operations, keeping anything that smells of an "object operation" out, and in some upper-level layer.

Level 0 Driver Features Required

- BSON serialization/deserialization
- · basic operations as listed in this section
- full cursor support (e.g. support OP_GET_MORE operation)
 close cursors via OP_KILL_CURSORS
- database command support
- handle query errors
- convert all strings to UTF-8
- hint, explain, count, \$where
- database profiling: set/get profiling level, get profiling info
- validate a collection in a database

Optional

- automatic reconnection
- buffer pooling
- advanced connection management (master-server, replica pair, Option_SlaveOk)
- Tailable cursor support

Server Object

A level 0 driver will implement the following connection options.

Single Server Connection

A driver should be able to connect to a single server. By default this must be localhost: 27017, and must also allow the server to be specified by hostname and port.

```
Mongo m = new Mongo(); // go to localhost, default port

Mongo m = new Mongo(String host, int port);
```

How the driver does this is up to the driver - make it idiomatic. However, a driver should make it explicit and clear what is going on.

Pair Mode Connection

A driver must be able to support "Pair Mode" configurations, where two instances of Mongo, labeled "left" and "right", and configured for hot-failover.

The driver should determine which of the pair is the current master, and send all operations to that server. In the event of an error, either socket error or a "not a master" error, the driver must restart the determination process. It must not assume the other server in the pair is now the master.

```
ServerPair sp = new ServerPair(INETAddr...); // should be able to give only one and it discovers the rest?

Mongo m = new Mongo(sp)
```

A driver may optionally allow a driver to connect deliberately to the "non-master" in the pair, for debugging, admin or operational purposes.

```
ServerPair sp = new ServerPair(INETAddr...); // should be able to give only one and it discovers the rest?

sp.setTarget(ServerPair.SHADOW_MASTER);

Mongo m = new Mongo(sp);
```

1. Cluster Mode Connect to master in master-slave cluster

```
ServerCluster sc = new ServerCluster(INETAddr...); // again, give one and discover?

Mongo m = new Mongo(sc);
```

Connect to slave in read-only mode in master-slave cluster

```
ServerCluster sc = new ServerCluster(INETAddr...); // again, give one and discover?
sc.setTarget(...)
Mongo m = new Mongo(sc);
or maybe make it like *Default/Simple* w/ a flag?
```

Other than that, we need a way to get a DB object :

```
Mongo m = new Mongo();

DB db = m.getDB(name);
```

And a list of db names (useful for tools...):

```
List<String> getDBNameList();
```

Database Object

Simple operations on a database object :

```
* get name of database
        dbName = db.getName();
String
/**
\mbox{\ensuremath{\star}} Get a list of all the collection names in this database
List<String> cols = db.getCollectionNames();
 * get a collection object. Can optionally create it if it
 ^{\star} doesn't exist, or just be strict. (XJDM has strictness as an option)
Collection coll = db.getCollection(string);
\star Create a collection w/ optional options. Can fault
* if the collection exists, or can just return it if it already does
Collection coll = db.createCollection( string);
Collection coll = db.createCollection( string, options);
* Drop a collection by its name or by collection object.
* Driver could invalidate any outstanding Collection objects
* for that collection, or just hope for the best.
boolean b = db.dropCollection(name);
boolean b = db.dropCollection(Collection);
* Execute a command on the database, returning the
* BSON doc with the results
* /
Document d = db.executeCommand(command);
* Close the [logical] database
void db.close();
/**
 * Erase / drop an entire database
bool dropDatabase(dbname)
```

Database Administration

These methods have to do with database metadata: profiling levels and collection validation. Each admin object is associated with a database. These methods could either be built into the Database class or provided in a separate Admin class whose instances are only available from a database instance.

```
/* get an admin object from a database object. */
Admin admin = db.getAdmin();

/**
    * Get profiling level. Returns one of the strings "off", "slowOnly", or
    * "all". Note that the database returns an integer. This method could
    * return an int or an enum instead --- in Ruby, for example, we return
    * symbols.
    */
String profilingLevel = admin.getProfilingLevel();

/**
    * Set profiling level. Takes whatever getProfilingLevel() returns.
    */
admin.setProfilingLevel("off");

/**
    * Retrieves the database's profiling info.
    */
Document profilingInfo = admin.getProfilingInfo();

/**
    * Returns true if collection is valid; raises an exception if not.
    */
boolean admin.validateCollection(collectionName);
```

Collection Basic Ops

```
* full query capabilities - limit, skip, returned fields, sort, etc
Cursor
           find(...);
void
           insert(...) // insert one or more objects into the collection, local variants on args
            remove(query) // remove objects that match the query
void
            modify(selector, modifier) // modify all objects that match selector w/ modifier object
void
void
            replace(selector, object) // replace first object that match selector w/ specified
obiect
void
            repsert(selector, object) // replace first object that matches, or insert **upsert w/
modifier makes no logical sense*
           getCount();
long
             qetCount(query);
```

Index Operations

```
void createIndex( index_info)
void dropIndex(name)
void dropIndexes()
List<info> getIndexInformation()
```

Misc Operations

```
document explain(query)
options getOptions();
string getName();
void close();
```

Cursor Object

```
document getNextDocument()
iterator getIterator() // again, local to language
bool hasMore()
void close()
```

"Level 1 Driver"

A "level 1" driver uses a "level 0" driver to add additional functionality. In terms of our history, I think that this is where "ORM"-type of functions should appear.

I think that we might have a variety of level 1 drivers :

- Level 1 "ORM" Driver: this would support stuff like save(), ensureIndex(), auto create index for _id, resolution of DB Refs, etc
- Level 1 "GridFS" Driver: this would be a separate and distinct driver that leverages a level 0 driver to do GridFS. No need to combine w/ anything else

Level 1 Database Object

```
/**
 * resolve a DBRef, with recursion. This is a potentially dangerous operation, as the
 * resulting document graph - and thus the resulting single document, can be arbitrarily
 * and surprisingly large
 */
Document resolveReference(DBRef ref, boolean recurse);
```

Level 1 Collection Object

```
* From JS, this is a caching index function that supported the
 * multiple re-entry and re-initialization model on the app server. It's a far
 * more difficult operation outside of the controlled environment of an app server, since
   one process has no clue about what other code might be doing (like dropping the index)
 * Put here for completeness
boolean coll.ensureIndex(...);
/**
    "saves" a document - this means that if it's already blessed with an _id, the doc
    in the database is updated, and if it doesn't have an _id, and _id is injected and then
     the doc is inserted into the collection.
    There are some interesting implications here now that we're considering relaxing
    the constraint on _id being a BabbleOID, and letting it be anything. There will have to
    be some kind of "PKInjector" (see the XJDM) where the programmer can give the level 1
    driver an _id factory....
 * /
void coll.save(Document doc);
```

Spec, Notes and Suggestions for Mongo Drivers

Assume that the BSON objects returned from the database may be up to 4MB. This size may change over time but for now the limit is 4MB per object. We recommend you test your driver with 4MB objects.

See Also

- Driver Requirements
- BSON
- The main Database Internals page

Feature Checklist for Mongo Drivers

Functionality Checklist

This section lists tasks the driver author might handle.

Essential

- BSON serialization/deserialization
- Basic operations: query, save, update, remove, ensureIndex, findOne, limit, sort
- Fetch more data from a cursor when necessary (dbGetMore)
- · Sending of KillCursors operation when use of a cursor has completed (ideally for efficiently these are sent in batches)
- Convert all strings to utf8
- Authentication

Recommended

- automatic doc_id generation (important when using replication)
- Database \$cmd support and helpers
- Detect { \$err: ... } response from a db query and handle appropriately --see Error Handling in Mongo Drivers
- Automatically use the proper half of a db server replica pair
- ensureIndex commands should be cached to prevent excessive communication with the database. (Or, the driver user should be informed that ensureIndex is not a lightweight operation for the particular driver.)
- Support for objects up to 4MB in size

More Recommended

- lasterror helper functions
- count() helper function
- Swhere clause
- eval()
- File chunking
- · hint fields
- explain helper
- Automatic _id index creation (maybe the db should just do this???)

More Optional

- addUser, logout helpers
- Allow client user to specify Option_SlaveOk for a query
- Tailable cursor support
- In/out buffer pooling (if implementing in a garbage collected languages)

More Optional

- · connection pooling
- · Automatic reconnect on connection failure
- DBRef Support:
 - Ability to generate easily
 - Automatic traversal

See Also

- The DOCS:Driver and Integration Center for information about the latest drivers
- The top page for this section
- The main Database Internals page
- The starting point for all Home

Conventions for Mongo Drivers

Interface Conventions

It is desirable to keep driver interfaces consistent when possible. Of course, idioms vary by language, and when they do adaptation is appropriate. However, when the idiom is the same, keeping the interfaces consistent across drivers is desirable.

Terminology

In general, use these terms when naming identifiers. Adapt the names to the normal "punctuation" style of your language -- foo_bar in C might be fooBar in Java.

- database what does this mean?
- collection

index

Mongo Wire Protocol

- Introduction
- · Messages Types and Formats
 - Standard Message Header
 - Request Opcodes
- Client Request Messages
 - OP_UPDATE
 - OP_INSERT
 - OP_QUERY
 - OP_GETMORE
 - OP_DELETE
 - OP_KILL_CURSORS
 - OP_MSG
- Database Response Messages
 - OP_REPLY

Introduction

TableOfContents

The Mongo Wire Protocol is a simple socket-based, request-response style protocol. Clients communicate with the database server through a regular TCP/IP socket.



Default Socket Port

The default port is 27017, but this is configurable and will vary.

Clients should connect to the database with a regular TCP/IP socket. Currently, there is no connection handshake.



To describe the message structure, a C-like struct is used. The types notation is the same as that used in describing BSON. The standard message header is typed as MsgHeader, and a BSON document is typed as BSON. Integer constants are in capitals (e.g. ZERO for the integer value of 0).

In the case where more than one of something is possible (like in a OP_INSERT or OP_KILL_CURSORS, an array notation is used (e.g. int64[]). This simply indicates that one or more of the specified type should be written to the socket, one after another.



Byte Ordering

Note that like BSON documents, all data in the mongo wire protocol is little-endian.

Messages Types and Formats

TableOfContents

There are two types of messages, client requests and database responses, each having a slightly different structure.

Standard Message Header

In general, each message consists of a standard message header followed by request-specific data. The standard message header is structured as follows:

```
struct {
   int32 messageLength; // total size of the message, including the 4 bytes of length
int32 requestID; // client or database-generated identifier for this message
int32 responseTo; // requestID from the original request (used in reponses from db)
int32 opCode; // request type - see table below
}
```

messageLength: This is the total size of the message in bytes. This total includes the 4 bytes that holds the message length.

requestID: This is a client or database-generated identifier that uniquely identifies this message. For the case of client-generated messages (e.g. CONTRIB:OP_QUERY and CONTRIB:OP_GET_MORE), it will be returned in the

responseTo field of the CONTRIB:OP_REPLY message. Along with the reponseTo field in responses, clients can use this to associate query responses with the originating query.

responseTo: In the case of a message from the database, this will be the requestID taken from the CONTRIB:OP_QUERY or CONTRIB:OP_GET_MORE messages from the client. Along with the requestID field in queries, clients can use this to associate query responses with the originating query.

opCode: Type of message. See the table below in the next section.

Request Opcodes

TableOfContents

The following are the currently supported opcodes:

Opcode Name	opCode value	Comment
OP_REPLY	1	Reply to a client request. responseTo is set
OP_MSG	1000	generic msg command followed by a string
OP_UPDATE	2001	update document
OP_INSERT	2002	insert new document
OP_GET_BY_OID	2003	is this used?
OP_QUERY	2004	query a collection
OP_GET_MORE	2005	Get more data from a query. See Cursors
OP_DELETE	2006	Delete documents
OP_KILL_CURSORS	2007	Tell database client is done with a cursor

Client Request Messages

TableOfContents

Clients can send all messages except for CONTRIB:OP_REPLY. This is reserved for use by the database.

Note that only the CONTRIB:OP_QUERY and CONTRIB:OP_GET_MORE messages result in a response from the database. There will be no response sent for any other message.

You can determine if a message was successful with a \$\$\$ TODO get last error command.

OP UPDATE

The OP_UPDATE message is used to update a document in a collection. The format of a OP_UPDATE message is

fullCollectionName: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

flags:

bit num	name	description
0	Upsert	If set, the database will insert the supplied object into the collection if no matching document is found.
1	MultiUpdate	If set, the database will update all matching objects in the collection. Otherwise only updates first matching doc.
2-31	Reserved	Must be set to 0.

selector: BSON document that specifies the query for selection of the document to update.

document: BSON document that specifies the fields to change in the selected document, or in the case of an upsert, the document to insert into the collection.

There is no response to an OP_UPDATE message.

OP_INSERT

The OP INSERT message is used to insert one or more documents into a collection. The format of the OP INSERT message is

fullCollectionName: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

documents: One or more documents to insert into the collection. If there are more than one, they are written to the socket in sequence, one after another.

There is no response to an OP_UPDATE message.

OP QUERY

The OP_QUERY message is used to query the database for documents in a collection. The format of the OP_QUERY message is :

```
struct {
   MsgHeader header;
                                    // standard message header
int32 opts;
                               // query options. See below for details.
                               // "dbname.collectionname"
cstring fullCollectionName;
      numberToSkip;
int32
                               // number of documents to skip when returning results
int32
         numberToReturn;
                               // number of documents to return in the first OP_REPLY
         query ;
BSON
                                // query object. See below for details.
[ BSON returnFieldSelector; ] // OPTIONAL : selector indicating the fields to return. See below
for details.
}
```

opts: query options

- None: 0
- Tailable cursor: 2
- Slave OK: 4
- Oplog replay: 8 (internal replication use only drivers should not implement)
- No cursor timeout : 16

fullCollectionName: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

numberToskip: Sets the number of documents to omit - starting from the first document in the resulting dataset - when returning the result of the query.

numberToReturn: Limits the number of documents in the first CONTRIB:OP_REPLY message to the query. However, the database will still establish a cursor and return the cursorID to the client if there are more results than numberToReturn. If the client driver offers 'limit' functionality (like the SQL LIMIT keyword), then it is up to the client driver to ensure that no more than the specified number of document are returned to the calling application. If numberToReturn is 0, the db will used the default return size. If the number is negative, then the database will return that number and close the cursor. No futher results for that query can be fetched.

query: BSON document that represents the query. The query will contain one or more elements, all of which must match for a document to be included in the result set. Please see \$\$\$ TODO QUERY for more information.

returnFieldsSelector: OPTIONAL BSON document that limits the fields in the returned documents. The returnFieldsSelector contains one or more elements, each of which is the name of a field that should be returned, and and the integer value 1. In JSON notation, a returnFieldsSelector to limit to the fields "a", "b" and "c" would be:

```
{ a : 1, b : 1, c : 1}
```

The database will respond to an OP_QUERY message with an CONTRIB:OP_REPLY message.

OP_GETMORE

The OP_GETMORE message is used to query the database for documents in a collection. The format of the OP_GETMORE message is :

```
struct {

MsgHeader header; // standard message header
int32    ZERO; // 0 - reserved for future use
cstring fullCollectionName; // "dbname.collectionname"
int32    numberToReturn; // number of documents to return
int64    cursorID; // cursorID from the OP_REPLY
}
```

fullCollectionName: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

numberToReturn: Limits the number of documents in the first CONTRIB:OP_REPLY message to the query. However, the database will still establish a cursor and return the cursorID to the client if there are more results than numberToReturn. If the client driver offers 'limit' functionality (like the SQL LIMIT keyword), then it is up to the client driver to ensure that no more than the specified number of document are returned to the calling application. If numberToReturn is 0, the db will used the default return size.

cursorID: Cursor identifier that came in the CONTRIB:OP_REPLY. This must be the value that came from the database.

The database will respond to an OP_GETMORE message with an CONTRIB:OP_REPLY message.

OP_DELETE

The OP_DELETE message is used to remove one or more messages from a collection. The format of the OP_DELETE message is :

```
struct {

MsgHeader header; // standard message header

int32 ZERO; // 0 - reserved for future use

cstring fullCollectionName; // "dbname.collectionname"

int32 ZERO; // 0 - reserved for future use

BSON selector; // query object. See below for details.

}
```

fullCollectionName: The full collection name. The full collection name is the concatenation of the database name with the collection name, using a "." for the concatenation. For example, for the database "foo" and the collection "bar", the full collection name is "foo.bar".

selector: BSON document that represent the query used to select the documents to be removed. The selector will contain one or more elements, all of which must match for a document to be removed from the collection. Please see \$\$\$ TODO QUERY for more information.

There is no reponse to an OP_DELETE message.

OP_KILL_CURSORS

The OP_KILL_CURSORS message is used to close an active cursor in the database. This is necessary to ensure that database resources are reclaimed at the end of the query. The format of the OP_KILL_CURSORS message is:

numberOfCursorIDs: The number of cursors that are in the message.

cursorIDs: "array" of cursor IDs to be closed. If there are more than one, they are written to the socket in sequence, one after another.

Note that if a cursor is read until exhausted (read until OP_QUERY or OP_GETMORE returns zero for the cursor id), there is no need to kill the cursor.

OP MSG

Deprecated. OP_MSG sends a diagnostic message to the database. The database sends back a fixed resonse. The format is

```
struct {
    MsgHeader header; // standard message header
    cstring message; // message for the database
}
```

Drivers do not need to implement OP_MSG.

Database Response Messages

TableOfContents

OP REPLY

The OP_REPLY message is sent by the database in response to an CONTRIB:OP_QUERY or CONTRIB:OP_GET_MORE message. The format of an OP_REPLY message is :

```
struct {
  MsgHeader header;
                                   // standard message header
      responseFlag;
                              // normally zero, non-zero on query failure
int32
int.64
         cursorID;
                                // id of the cursor created for this query response
                               // indicates where in the cursor this reply is starting
        startingFrom;
int32
int32
        numberReturned;
                               // number of documents in the reply
BSON[]
         documents;
                                // documents
}
```

responseFlag: Flag that indicates status of the query. Normally 0, a non-zero value indicates that the query failed for some reason. In the event of a non-zero responseFlag only one document will be returned, a database-generated document that contains error information. \$\$\$ TODO DOCUMENT THAT.

cursorID: The cursorID that this OP_REPLY is a part of. In the event that the result set of the query fits into one OP_REPLY message, cursorID will be 0. This cursorID must be used in any CONTRIB:OP_GET_MORE messages used to get more data, and also must be closed by the client when no longer needed via a CONTRIB:OP_KILL_CURSORS message.

TableOfContents

TODO:

- 1) describe object id
- 2) describe namespaces

BSON

- Introduction
 - A General-Purpose Data Format
- Language-Specific Examples
 - C
 - C++PHP
 - PHPPython
 - Ruby
- BSON Document Format
 - Data types used in the grammar
 - BSON Grammar
 - Element Data Types
- Implicit Document Types

Introduction

BSON is the data storage format for "documents" – or "objects" – in MongoDB. BSON is a binary-encoded serialization of JSON-like documents, and, like JSON, supports the embedding of objects and arrays within other objects and arrays.

Although BSON stands for "Binary JSON," the format contains extensions for representing data types not included in the JSON specification. For

instance, BSON supports BinData and Date data formats.

BSON at first seems BLOB-like, but there exists an important difference: the Mongo database understands BSON internals. This means that MongoDB can "reach inside" BSON objects, even nested ones. Among other things, this allows MongoDB to build indexes and match objects against query expressions on both top-level and nested BSON keys.

A General-Purpose Data Format

MongoDB is the first major application to use BSON; however, BSON was designed as a general-purpose data format, usable for many data-marshalling problems. Developers requiring a rich, efficient data-exchange format should seriously consider BSON.

BSON libraries are available for most languages. These libraries are currently bundled with the MongoDB client drivers, but work is under way to make the various BSON modules independent.

See also: the BSON blog post.

Language-Specific Examples

С

See http://github.com/mongodb/mongo-c-driver/blob/master/src/bson.h

C++

```
BSONObj p = BSON( "name" << "Joe" << "age" << 33 );
```

See the BSON section of the C++ Tutorial for more information.

PHP

The PHP driver includes bson_encode and bson_decode functions. bson_encode takes any PHP type and serializes it, returning a string of bytes:

```
$bson = bson_encode(null);
$bson = bson_encode(true);
$bson = bson_encode(4);
$bson = bson_encode("hello, world");
$bson = bson_encode(array("foo" => "bar"));
$bson = bson_encode(new MongoDate());
```

Mongo-specific objects (MongoId, MongoDate, MongoRegex, MongoCode) will be encoded in their respective BSON formats. For other objects, it will create a BSON representation with the key/value pairs you would get by running for (\$object as \$key => \$value).

 $\verb|bson_decode| takes a string representing a BSON object and parses it into an associative array.$

Python

PyMongo also supports "ordered dictionaries" through the **pymongo.son** module. The **BSON** class can handle **SON** instances using the same methods you would use for regular dictionaries.

Ruby

The BSON class also supports ordered hashes. Simply construct your documents using the OrderedHash class, also found in the MongoDB Ruby Driver. Examples with all of the supported BSON types can be found in the BSON unit tests.

BSON Document Format

BSON is a binary message format in which zero or more key-value pairs are stored as a single entity. We call this entity a "BSON document".

All BSON data must be serialized in little-endian format.

Data types used in the grammar

Туре	Size
byte	1 byte (8-bits)
int32	4 byte, 32-bit signed integer
int64	8 byte, 64-bit signed integer
double	8 byte, 64-bit IEEE-XX floating point number

BSON Grammar

Entity	Definition	Comment	
bson_object	obj_size element* eoo		
obj_size	int32	total size in bytes of the BSON object including 4 bytes for this field	
element	element_type element_name element_data		
e00	NULL		
element_type	byte	One of the data types listed below	
element_name	cstring	See note on element_name	
element_data	(see below)	Specific data for the type - see table below	
cstring	byte* NULL	zero or more UTF-8-encoded characters ended by NULL	
NULL	0x00	single byte of value 0	
VOID		Nothing. Not 0. Zero bits.	



Notes

element_name Element names available to user documents are constrained. Please be sure to understand the limitations as outlined below in XXXX.

Element Data Types

Note - for data representations, semantic meaning for a data type is indicated in within the brackets and after a hyphen and is described in the note for that type

Element Type	Binary Value	Data Representation	Comments
data_number	1	double	
data_string	2	int32 cstring	The int32 is the # bytes following (# of bytes in string + 1 for terminating NULL)
data_object	3	bson_object	
data_array	4	bson_object	See note on data_array
data_binary	5	int32 byte byte[]	The first int32 is the # of bytes following the byte subtype. Please see note on data_binary
data_undefined	6	VOID	Conceptually equivalent to Javascript undefined. Deprecated.
data_oid	7	byte[12]	12 byte object id.
data_boolean	8	byte	legal values: 0x00 -> false, 0x01 -> true
data_date	9	int64	value: milliseconds since epoch (e.g. new Date.getTime())
data_null	10	VOID	Mapped to Null in programming languages which have a Null value or type. Conceptually equivalent to Javascript null.
data_regex	11	cstring cstring	first ctring is regex expression, second cstring are regex options See note on data_regex
data_ref	12	int32 cstring byte[12]	Deprecated. Please use a subobject instead see page DB Ref. The int32 is the length in bytes of the cstring. The cstring is the namespace: full collection name. The byte array is a 12 byte object id. See note on data_oid.
data_code	13	int32 cstring	The int32 is the # bytes following (# of bytes in string + 1 for terminating NULL) and then the code as cstring. data_code should be supported in BSON encoders/decoders, but has been deprecated in favor of data_code_w_scope
data_symbol	14	int32 cstring	same as data_string but for languages with distinct symbol type
data_code_w_scope	15	int32 int32 cstring bson_object	The first int32 is the total # of bytes (size of cstring + size of bson_object + 8 for the two int32s). The second int 32 is the size of the cstring (# of bytes in string + 1 for terminating NULL). The cstring is the code. The bson_object is an object mapping identifiers to values, representing the scope in which the code should be evaluated.
data_int	16	int32	
data_timestamp	17	int64	Special internal type used by MongoDB replication and sharding. First 4 are a timestamp, next 4 are an incremented field. Saving a zero value for data_timestamp has special semantics.
data_long	18	int64	64 bit integer
data_min_key	-1	VOID	Special type which compares lower than all other possible BSON element values. See Comparing Types and Splitting Shards
data_max_key	127	VOID	Special type which compares greater than all other possible BSON element values. See Comparing Types and Splitting Shards



Notes

All strings are UTF-8.

data_array The data for a data_array type is a normal BSON object, with integer values for the keys, starting with 0 and continuing sequentially. For example, the array ["blue", "red", "green"] is expressed as as BSON object (using JSON notation for this example:

```
{ "0" : "blue", "1" : "red", "2" : "green" }
```

The keys in the BSON object must be in ascending numerical order.

data_regex Option characters must be stored in alphabetical order. Only options "i", "m", and "x" are supported by the database. However, it's possible for users to use BSON to store their own regex. How drivers handle non-standard options in user documents is left to the drivers. We recommend that they are simply ignored, and if possible, preserved in the document on future saves. The following table documents options characters and what they mean, so that regexs can be interpreted correctly across drivers:

Option Character	Meaning
i	case-insensitive matching
m	multiline: "^" and "\$" match the beginning / end of each line as well as the whole string
x	verbose / comments: the pattern can contain comments
I (lowercase L)	locale: \w, \W, etc. depend on the current locale
S	dotall: the "." character matches everything, including newlines
u	unicode: \w, \W, etc. match unicode

data_binary The data_binary type has the following subtypes defined:

Subtype Code	Structure	Comment
0x01	unknown	function
0x02	int32 byte[]	The int32 is the number of bytes in the following byte array.
0x03	unknown	UUID
0x05	unknown	MD5
0x80	user defined	User defined

Note that 0x02 is the commonly used "binary" type for carrying general binary data. 0x80 is user defined, and can be anything.

Implicit Document Types

The mongo wire protocol uses BSON documents for three things:

- 1. User Document: This is the regular document that the database stores. These are the BSON documents that are sent to the database via the INSERT operation. User documents have limitations on the "element name" space due to the usage of special characters in the JSON-like query language.
 - a. A user document element name cannot begin with "\$".
 - b. A user document element name cannot have a "." in the name.
 - c. The element name "_id" is reserved for use as a primary key id, but you can store anything that is unique in that field. (\$\$\$ GMJ : and it probably would be prudent to avoid starting any element name with "_".)
 - d. The element name "query" cannot currently be used.
 - The database expects that drivers will prevent users from creating documents that violate these constraints.
- 2. "Selector" Documents: Selector documents (or selectors) are BSON documents that are used in QUERY, DELETE and UPDATE operations. They are used by these operations to match against documents. Selector objects have no limitations on the "element name" space, as they must be able to supply special "marker" elements, like "\$where" and the special "command" operations.
- 3. "Modifier" Documents: Documents that contain 'modifier actions' that modify user documents in the case of an update (see Updating)
- 4. Return error messages : TODO

Mongo Extended JSON

obvious JSON mappings, and multiple representations are allowed for some such types. The REST interface supports three different modes for document output { Strict, JS, TenGen }, which serve to control the representations used. Mongo can of course understand all of these representations in REST input.

- Strict mode produces output conforming to the JSON spec http://www.json.org.
- **JS** mode uses some Javascript types to represent certain BSON types.
- TenGen mode uses some Javascript types and some 10gen specific types to represent certain BSON types.

The following BSON types are represented using special conventions:

Туре	Strict	JS	TenGen	Explanation
data_binary	{ "\$binary" : " <bindata>" , "\$type": "<t>" }</t></bindata>	{ "\$binary" : " <bindata>", "\$type" : "<t>" }</t></bindata>	{ "\$binary" : " <bindata>", "\$type" : "<t>" }</t></bindata>	<bindata> is the base64 representation of a binary string. <t> is the hexadecimal representation of a single byte indicating the data type.</t></bindata>
data_date	{ "\$date" : <date> }</date>	Date(<date>)</date>	Date(<date>)</date>	<date> is the string representation of a 64 bit unsigned integer.</date>
data_regex	{ "\$regex" : " <sregex>", "\$options" : "<soptions>" }</soptions></sregex>	/ <jregex>/<joptions></joptions></jregex>	/ <jregex>/<joptions></joptions></jregex>	<sregex> is a string of valid JSON characters. <jregex> is a string that may contain valid JSON characters and unescaped '"' characters, but may not contain unescaped '/' characters. <soptions> is a string containing letters of the alphabet. <joptions> is a string contain only the characters 'g', 'i', and 'm'. Because the JS and TenGen representations support a limited range o options, any nonconforming options will be dropped when converting to this representation.</joptions></soptions></jregex></sregex>

data_oid	{ "\$oid" : " <id>" }</id>	{ "\$oid" : " <id>" }</id>	ObjectId(" <id>")</id>	<id> is a 24 character hexadecimal string. Note that these representations require a data_oid value to have an associated field name "_id".</id>
data_ref	{ "\$ref" : " <name>", "\$id" : "<id>>" }</id></name>	{ "\$ref" : " <name>" , "\$id" : "<id>" }</id></name>	Dbref(" <name>", "<id>>")</id></name>	<name> is a string of valid JSON characters. <id> is a 24 character hexadecimal string.</id></name>

GridFS Specification

- Introduction
- Specification
 - Storage Collections
 - files
 - chunks
 - Indexing

Introduction

GridFS is a storage specification for large objects in MongoDB. GridFS takes large objects and stores them as chunks of data as well as metadata. This document specifies the requirements of a GridFS implementation.

Normally, you need not worry about the details of the format -- for information on how to use GridFS, see Storing Files.

Specification

Storage Collections

GridFS uses two collections to store data:

- files contains the object metadata
- chunks contains the binary chunks with some additional accounting information

These are "subcollections" on a "root collection". By default this is fs so for a GridFS store, the collection would be considered to be fs, with the two parts fs.files and fs.chunks.

The root collection is allowed to vary, to provide for the ability for a user to segment large objects into subsets. For example, one might partition objects by type, such as pdf, contracts, {{videos}, etc.

However, £s is the default root collection for GridFS, and must be supported by any GridFS implementation in a way that it doesn't have to be specified to perform GridFS operations. For example:

Note that the above API is for demonstration purposes only - this spec does not (at this time) recommend any API. See individual driver documentation for API specifics.

files

The structure of the object metadata document is as follows:

```
"_id" : <unspecified>,
                                              // unique ID for this file
                                    // unique ID for this file
// human name for the file
// valid mime type for the object
"filename" : data_string,
"contentType" : data_string,
"length" : data_number,
                                        // size of the file in bytes
                                // size of each of the chunks. Default is 256k
"chunkSize" : data_number,
                                        // date when object first stored
"uploadDate" : data_date,
"aliases" : data_array of data_string, // optional array of alias strings
"metadata" : data_object,
                                         // anything the user wants to store
"md5" : data_string
                                         // result of running the "filemd5" command on this file's
chunks
```

Note that the _id field can be of any type at the discretion of the spec implementor.

chunks

The structure of the chunk document is as follows:

Notes:

- The _id is whatever type you choose
- The files_id must contain the value of the _id field for the "owning" files collection entry

Indexing

GridFS implementations should create an index on

```
{ files_id:1, n:1}
```

in the ${\tt chunks}$ collection, and should count on being able to retrieve chunks efficiently via

```
db.fs.chunks.find({file_id: myFileID}).orderby({n:1});
```

Implementing Authentication in a Driver

The current version of Mongo supports only very basic authentication. One authenticates a username and password in the context of a particular database. Once authenticated, the user has full read and write access to the database in question.

The admin database is special. In addition to several commands that are administrative being possible only on admin, authentication on admin gives one read and write access to all databases on the server. Effectively, admin access means root access to the db.

Note on a single socket we may authenticate for any number of databases, and as different users. This authentication persists for the life of the database connection (barring a logout command).

The Authentication Process

Authentication is a two step process. First the driver runs a getnonce command to get a nonce for use in the subsequent authentication. We can view a sample getnonce invocation from dbshell:

```
> db.$cmd.findOne({getnonce:1})
{ "nonce":"7268c504683936e1" , "ok":1
```

The nonce returned is a hex String.

The next step is to run an authenticate command for the database on which to authenticate. The authenticate command has the form:

```
{ authenticate : 1, user : username, nonce : nonce, key : digest }
```

where

- username is a username in the database's system.users collection;
- nonce is the nonce returned from a previous getnonce command;
- digest is the hex encoding of a MD5 message digest which is the MD5 hash of the concatenation of (nonce, username, password_digest), where password_digest is the user's password value in the pwd field of the associated user object in the database's system.users collection. pwd is the hex encoding of MD5(username + ":mongo:" + password_text).

Authenticate will return an object containing

```
{ ok : 1 }
```

when successful.

Details of why an authentication command failed may be found in the Mongo server's log files.

The following code from the Mongo Javascript driver provides an example implementation:

```
DB.prototype.addUser = function( username , pass ){
    var c = this.getCollection( "system.users" );
    var u = c.findOne( { user : username } ) || { user : username };
    u.pwd = hex_md5( username + ":mongo:" + pass );
    print( tojson( u ) );
    c.save( u );
}
DB.prototype.auth = function( username , pass ){
    var n = this.runCommand( { getnonce : 1 } );
    var a = this.runCommand(
            authenticate: 1,
            user : username ,
            nonce : n.nonce ,
            key : hex_md5( n.nonce + username + hex_md5( username + ":mongo:" + pass ) )
        }
    );
    return a.ok;
}
```

Logout

Drivers may optionally implement the logout command which deauthorizes usage for the specified database for this connection. Note other databases may still be authorized.

Alternatively, close the socket to deauthorize.

```
> db.$cmd.findOne({logout:1})
{
    "ok" : 1.0
}
```

Replica Pairs and Authentication

For drivers that support replica pairs, extra care with replication is required.

When switching from one server in a pair to another (on a failover situation), you must reauthenticate. Clients will likely want to cache authentication from the user so that the client can reauthenticate with the new server when appropriate.

Be careful also with operations such as Logout - if you log out from only half a pair, that could be an issue.

Authenticating with a server in slave mode is allowed.

See Also

Security and Authentication

Notes on Pooling for Mongo Drivers

Note that with the db write operations can be sent asynchronously or synchronously (the latter indicating a getlasterror request after the write).

When asynchronous, one must be careful to continue using the same connection (socket). This ensures that the next operation will not begin until after the write completes.

Pooling and Authentication

An individual socket connection to the database has associated authentication state. Thus, if you pool connections, you probably want a separate pool for each authentication case (db + username).

Pseudo-code

The following pseudo-code illustrates our recommended approach to implementing connection pooling in a driver's connection class. This handles authentication, grouping operations from a single "request" onto the same socket, and a couple of other gotchas:

```
class Connection:
   init(pool_size, addresses, auto_start_requests):
     this.pool_size = pool_size
      this.addresses = addresses
      this.auto_start_requests = auto_start_requests
      this.thread_map = {}
      this.locks = Lock[pool_size]
      this.sockets = Socket[pool_size]
      this.socket_auth = String[pool_size][]
      this.auth = {}
      this.find_master()
   find_master():
      for address in this.addresses:
         if address.is_master():
            this.master = address
   pick_and_acquire_socket():
      choices = random permutation of [0, ..., this.pool_size - 1]
      choices.sort(order: ascending,
                   value: size of preimage of choice under this.thread_map)
      for choice in choices:
         if this.locks[choice].non_blocking_acquire():
           return choice
      sock = choices[0]
      this.locks[sock].blocking_acquire()
      return sock
   get_socket():
      if this.thread_map[current_thread] >= 0:
         sock_number = this.thread_map[current_thread]
         this.locks[sock_number].blocking_acquire()
         sock_number = this.pick_and_lock_socket()
         if this.auto_start_requests or current_thread in this.thread_map:
            this.thread_map[current_thread] = sock_number
      if not this.sockets[sock_number]:
         this.sockets[sock_number] = Socket(this.master)
      return sock_number
   \verb|send_message_without_response(message)|:
      sock_number = this.get_socket()
      this.check_auth()
      this.sockets[sock_number].send(message)
      this.locks[sock_number].release()
   send_message_with_response(message):
      sock_number = this.get_socket()
      this.check auth()
      this.sockets[sock_number].send(message)
      result = this.sockets[sock_number].receive()
      this.locks[sock_number].release()
      return result
   # start_request is only needed if auto_start_requests is False
   start_request():
      this.thread_map[current_thread] = -1
```

```
end_request():
      delete this.thread_map[current_thread]
   authenticate(database, username, password):
      # TODO should probably make sure that these credentials are valid,
      \ensuremath{\sharp} otherwise errors are going to be delayed until first op.
      this.auth[database] = (username, password)
   logout(database):
      delete this.auth[database]
   check_auth(sock_number):
      for db in this.socket_auth[sock_number]:
         if db not in this.auth.keys():
            this.sockets[sock_number].send(logout_message)
            this.socket_auth[sock_number].remove(db)
      for db in this.auth.keys():
         if db not in this.socket_auth[sock_number]:
            this.sockets[sock_number].send(authenticate_message)
            this.socket_auth[sock_number].append(db)
# somewhere we need to do error checking - if you get not master then everything
# in this.sockets gets closed and set to null and we call find_master() again.
```

```
# we also need to reset the socket_auth information - nothing is authorized yet
# on the new master.
```

See Also

- The DOCS:Driver and Integration Center for information about the latest drivers
- The top page for this section
- The main Database Internals page
- The starting point for all Home

Error Handling in Mongo Drivers

If an error occurs on a query (or getMore operation), Mongo returns an error object instead of user data.

The error object has a first field guaranteed to have the reserved key \$err. For example:

```
{ $err : "some error message" }
```

The \$err value can be of any type but is usually a string.

Drivers typically check for this return code explicitly and take action rather than returning the object to the user. The query results flags include a set bit when \$err is returned.

See Also

- The DOCS:Driver and Integration Center for information about the latest drivers
- The top page for this section
- The main Database Internals page
- The starting point for all Home

Contributing to the Documentation

Qualified volunteers are welcome to assist in editing the wiki documentation. Contact us for more information.

Mongo Documentation Style Guide

This page provides information for everyone adding to the Mongo documentation on Confluence. It covers:

- General Notes on Writing Style
- Guide to Confluence markup for specific situations
- Some general notes about doc production

General Notes on Writing Style

Voice

Active voice is almost always preferred to passive voice.

To make this work, however, you may find yourself anthropromorphizing components of the system - that is, treating the driver or the database as an agent that actually does something. ("The dbms writes the new record to the collection" is better than "the new record is written to the database", but some purists may argue that the dbms doesn't do anything - it's just code that directs the actions of the processor - but then someone else says "yes, but does the processor really do anything?" and so on and on.) It is simpler and more economical to write as if these components are actually doing things, although you as the infrastructure developers might have to stop and think about which component is actually performing the action you are describing.

Tense

Technical writers in general prefer to keep descriptions of processes in the present tense: "The dbms writes the new collection to disk" rather than "the dbms will write the new collection to disk." You save a few words that way.

MongoDB Terminology

It would be good to spell out precise definitions of technical words and phrases you are likely to use often, like the following:

- Mongo
- database (do you want "a Mongo database"? Or a Mongo database instance?)
- dbms (I have't seen this term often is it correct to talk about "the Mongo DBMS"?)
- Document
- Docord
- Transaction (I stopped myself from using this term because my understanding is the Mongo doesn't support "transactions" in the sense of
 operations that are logged and can be rolled back is this right?)

These are just a few I noted while I was editing. More should be added. It would be good to define these terms clearly among yourselves, and then post the definitions for outsiders.

Markup for terms

It's important to be consistent in the way you treat words that refer to certain types of objects. The following table lists the types you will deal with most often, describes how they should look, and (to cut to the chase) gives you the Confluence markup that will achieve that appearance.

Туре	Appearance	Markup
Object name (the type of "object" that "object-oriented programming" deals with)	monospace	{{ term }}
short code fragment inline	monospace	{{ term }}
file path/name, extension	italic	_term_
programming command, statement or expression	monospace	{{ term }}
variable or "replaceable item"	monospace italic	_term_
Placeholders in paths, directories, or other text that would be italic anyway	angle brackets around <item></item>	<item></item>
GUI element (menus menu items, buttons	bold	*term*
First instance of a technical term	italic	_term_
tag (in HTML or XML, for example)	monospace	{{ term }}
Extended code sample	code block	{code} program code
		{code}

In specifying these, I have relied on the O'Reilly Style Guide, which is at:

http://oreilly.com/oreilly/author/stylesheet.html

This guide is a good reference for situations not covered here.

I should mention that for the names of GUI objects I followed the specification in the Microsoft Guide to Technical Publications.

Other Confluence markup

If you are editing a page using Confluence's RTF editor, you don't have to worry about markup. Even if you are editing markup directly, Confluence displays a guide on the right that shows you most of the markup you will need.

References and Links

Confluence also provides you with a nice little utility that allows you to insert a link to another Confluence page by searching for the page by title or by text and choosing it from a list. Confluence handles the linking markup. You can even use it for external URLs.

The one thing this mechanism does NOT handle is links to specific locations within a wiki page. Here is what you have to know if you want to insert these kinds of links:

- Every heading you put in a Confluence page ("h2.Title", "h3.OtherTitle", etc.) becomes an accessible "anchor" for linking.
- You can also insert an anchor anywhere else in the page by inserting "{anchor\:anchorname}" where _anchorname is the unique name you will use in the link.
- To insert a link to one of these anchors, you must go into wiki markup and add the anchor name preceded by a "#". Example: if the page MyPage contains a heading or an ad-hoc anchor named GoHere, the link to that anchor from within the same page would look like [#GoHere], and a link to that anchor from a different page would look like [MyPage#GoHere]. (See the sidebar for information about adding other text to the body of the link.)

Special Characters

You will often need to insert code samples that contain curly braces. As Dwight has pointed out, Confluence gets confused by this unless
you "escape" them by preceding them with a backslash, thusly:

\{ \}

You must do the same for "[", "]", "_" and some others.

Within a {code} block you don't have to worry about this. If you are inserting code fragments inline using {{ and }}, however, you still need to escape these characters. Further notes about this:

- If you are enclosing a complex code expression with {{ and }}, do NOT leave a space between the last character of the expression and the }}. This confuses Confluence.
- Confluence also gets confused (at least sometimes) if you use {{ and }}, to enclose a code sample that includes escaped curly brackets.

About MongoDB's Confluence wiki

Confluence has this idea of "spaces". Each person has a private space, and there are also group spaces as well.

The MongoDB Confluence wiki has three group spaces defined currently:

- Mongo Documentation The publicly accessible area for most Mongo documentation
- Contributor Looks like, the publicly accessible space for information for "Contributors"
- Private a space open to MongoDB developers, but not to the public at large.
 As I said in my email on Friday, all of the (relevant) info from the old wiki now lives in the "Mongo Documentation"

Standard elements of Wiki pages

You shouldn't have to spend a lot of time worrying about this kind of thing, but I do have just a few suggestions:

• Since these wiki pages are (or can be) arranged hierarchically, you may have "landing pages" that do little more than list their child pages. I think Confluence actually adds a list of children automatically, but it only goes down to the next hierarchical level. To insert a hierarchical list of a page's children, all you have to do is insert the following Confluence "macro":

{children:all=true}

See the Confluence documentation for more options and switches for this macro.

- For pages with actual text, I tried to follow these guidelines:
 - For top-level headings, I used "h2" not "h1"
 - I never began a page with a heading. I figured the title of the page served as one.
 - I always tried to include a "See Also" section that listed links to other Mongo docs.
 - I usually tried to include a link to the "Talk to us about Mongo" page.

Community

For everyone...

User Mailing List

The user list is for general questions about using, configuring and running MongoDB and the associated tools and drivers. The list is open to everyone.

IRC chat

irc://irc.freenode.net/#mongodb

We keep a log.

Blog

http://blog.mongodb.org/

Bugtracker

File, track, and vote on bugs and feature requests. There is issue tracking for MongoDB and all supported drivers

Announcement Mailing List

http://groups.google.com/group/mongodb-announce - for release announcement and important bug fixes.

Store

Visit our Cafepress store for Mongo-related swag.

For database or driver developers...

Database Developer List

This list is for people developing drivers, tools or who are contributing to the MongoDB codebase itself.

Source

The source code for the database and drivers is available at GitHub.

Community Tools

Memcached

MongoNode - PHP script that replicates Memcached data to MongoDB.

MongoDB Commercial Services Providers

Note: if you provide consultative or support services for MongoDB and wish to be listed here, just let us know.

Support

10gen

10gen provides 24x7 commercial support and management services for MongoDB.

Hosting

MongoHQ

MongoHQ provides cloud-style hosted MongoDB instances.

Consulting

Squeejee

Squeejee builds web applications on top of MongoDB with multiple sites already in production.

User Feedback

- "I just have to get my head around that mongodb is really _this_ good" -muckster, #mongodb
- "Guys at Redmond should get a long course from you about what is the software development and support $\stackrel{\Box}{\cup}$ " -kunthar@gmail.com, mongodb-user list



"#mongoDB keep me up all night. I think I have found the 'perfect' storage for my app -elpargo, Twitter

- "Maybe we can relax with couchdb but with mongodb we are completely in dreams" -namlook, #mongodb
- "Dude, you guys are legends!" -Stii, mongodb-user list
- "Times I've been wowed using MongoDB this week: 7." -tpitale, Twitter

Community Presentations

MongoDB

Adrian Madrid at Mountain West Ruby Conference 2009, video

MongoDB - Ruby friendly document storage that doesn't rhyme with ouch Wynn Netherland at Dallas.rb Ruby Group, slides

MongoDB

inunemaker at Grand Rapids RUG, slides

Developing Joomla! 1.5 Extensions, Explained (slide 37) Mitch Pirtle at Joomla! Day New England 2009, slides

Drop Acid (slide 31) (video) Bob Ippolito at Pycon 2009

Python and Non-SQL Databases (in French, slide 21) Benoit Chesneau at Pycon France 2009, slides

Massimiliano Dessì at the Spring Framework Italian User Group

- MongoDB (in Italian)
- MongoDB and Scala (in Italian)

Presentations and Screencasts at Learnivore

Frequently-updated set of presentations and screencasts on MongoDB.

Community Blog Posts

MongoDB a Light in the Darkness...

-EngineYard

Introducing MongoDB

-Linux Magazine

Boxed Ice - Choosing a non-relational database; why we migrated from MySQL to MongoDB

Rails Tips - What If A Key/Value Store Mated With A Relational Database System? -John Nunemaker

The Other Blog - The Holy Grail of the Funky Data Model

-Tom Smith

Benchmarking

We keep track of user benchmarks on the Benchmarks page.

About

- Roadmap
- Production Deployments
- Articles
- Benchmarks
- Use Cases
- FAQ
- Product Comparisons
- Philosophy
- Licensing

Roadmap

1.4 - 2010 Q1

(7 iss	(7 issues)									
Туре	Key	Summary	Assignee	Reporter	Priority	Status	Resolution	Created	Updated	Due
+	SERVER-534	Official RPMs	Richard Kreuter	Gregg Lind	û	open	Unresolved	Jan 12, 2010	Jan 26, 2010	
a ì	SERVER-479	global read/write lock	Dwight Merriman	Eliot Horowitz	û	∛ Closed	Fixed	Dec 12, 2009	Jan 05, 2010	
>	SERVER-464	background index creation	Dwight Merriman	Eliot Horowitz	û	Resolved	Fixed	Dec 09, 2009	Feb 05, 2010	
+	SERVER-459	Find and Modify command	Mathias Stearn	Mathias Stearn	û	∛ Closed	Fixed	Dec 04, 2009	Jan 05, 2010	
+	SERVER-446	finish v8 engine	aaron	Eliot Horowitz	û	Resolved	Fixed	Nov 29, 2009	Feb 07, 2010	
>	SERVER-323	improved concurrency: read/write lock - yielding ops	Eliot Horowitz	Eliot Horowitz	û	open	Unresolved	Sep 30, 2009	Jan 20, 2010	
+	SERVER-134	\$unset (was: \$unset and \$rename)	Mathias Stearn	Kristina Chodorow	û	∛ Closed	Fixed	Jul 08, 2009	Jan 05, 2010	

1.6 - 2010 Q2

(2 iss	(2 issues)									
Туре	Key	Summary	Assignee	Reporter	Priority	Status	Resolution	Created	Updated	Due
+	SERVER-380	full-text search	Eliot Horowitz	Raj Kadam	û	open	Unresolved	Oct 21, 2009	Dec 07, 2009	
	SERVER-332	use boost asio for network layer	Eliot Horowitz	Eliot Horowitz	û	open	Unresolved	Oct 02, 2009	Dec 07, 2009	

Production Deployments

If you're using MongoDB in production, we'd love to list you here! Email mike at 10gen . com.

Company	Use Case

sourceforge	MongoDB is now used for back-end storage on the SourceForge front pages, project pages, and download pages for all projects. See Compound Thinking and the SourceForge blog for details.
github	GitHub, the social coding site, is using MongoDB for an internal reporting application.
	MongoDB is being used for the game feeds component. It caches game data from different sources which gets served to ea.com, rupture.com and the EA download manager.
The New York Times	The New York Times is using MongoDB in a form-building application for photo submissions.
DISQUS	Disqus is an innovative blog-commenting system.
THE BUSINESS INSIDER	The Business Insider has been using MongoDB since the beginning of 2008. All of the site's data, including posts, comments, and even the images, are stored on MongoDB. Read more
GILT	Gilt Groupe is an invitation only luxury shopping site.
Pitchfork	Pitchfork is using MongoDB for their year-end readers survey and internal analytics.
HOT POTATO	Hot Potato is a social tool that organizes conversations around events.
abiquity	The Mozilla open-source Ubiquity Herd project uses MongoDB for back-end storage. Source code is available on bitbucket.
🌉 floxee	Floxee, a web toolkit for creating Twitter directories, leverages MongoDB for back-end storage. The award-winning TweetCongress is powered by Floxee.
Sailthru You Communication, Wile Contract	Sailthru is an email service provider that uses MongoDB for click-stream analysis and reporting.
silentale	Silentale keeps track of your contacts and conversations from multiple platforms and allows you to search and access them from anywhere. Silentale is using MongoDB as the back-end for indexing and searching on millions of stored messages of different types. More details on Silentale can be found in this TechCrunch article.
mongoHQ yep, it's that easy.	MongoHQ provides a hosting platform for MongoDB and also uses MongoDB as the back-end for its service. Our hosting centers page provides more information about MongoHQ and other MongoDB hosting options.
Boxed Ice	BoxedIce's server monitoring solution - Server Density - uses MongoDB for back-end storage. Check out their blog post on why they chose MongoDB.
DEFENSIO connecting and span	Defensio is a comment-spam blocker that uses MongoDB for back-end storage.
TweetSaver	TweetSaver is a web service for backing up, searching, and tagging your tweets. TweetSaver uses MongoDB for back-end storage.
BLOOM	Bloom Digital's AdGear platform is a next-generation ad platform. MongoDB is used for back-end reporting storage for AdGear.

	KLATU Networks designs, develops and markets asset monitoring solutions which helps companies manage risk, reduce operating costs and streamline operations through proactive management of the status, condition, and location of cold storage assets and other mission critical equipment. KLATU uses MongoDB to store temperature, location, and other measurement data for large wireless sensor networks. KLATU chose MongoDB over competitors for scalability and query capabilities.
MUSIC NATION	Almost all data for MusicNation, including user information, images, and music videos, are stored in MongoDB.
SHOPWIKE	Shopwiki uses Mongo as a data store for its shopping search engine, where they commit all the data generated, such as custom analytics. Mongo's performance is such that ShopWiki uses it in cases where MySQL would just not be practical. ShopWiki is also using it as a storage engine for all R&D and data-mining efforts where MongoDB's document oriented architecture offers maximum flexibility.
	songkick lets you track your favorite artists so you never miss a gig again.
Detexify ²	Detexify is a cool application to find LaTeX symbols easily. It uses MongoDB for back-end storage. Check out the blog post for more on why Detexfy is using MongoDB.
Sweer Francisco	http://sluggy.com/ is built on MongoDB, mongodb_beaker, and MongoKit .
Stylkal .	Stylesignalis using MongoDB to store opinions from social media, blogs, forums and other sources to use in their sentiment analysis system, Zeitgeist.
FetLife	FetLife is the biggest kinky/bdsm social network, with over 250k users and serving 80M pageviews / month. FetLife uses MongoDB as backing storage for the activity feed. (Warning: FetLife is NSFW)
	@trackmeet helps you take notes with twitter, and is built on MongoDB
eflyover	eFlyoverleverages the Google Earth Browser Plugin and MongoDB to provide interactive flyover tours of over two thousand golf courses worldwide.
√Hapad⊙	Shapado is a multi-topic question and answer site in the style of Stack Overflow. Shapado is written in Rails and uses MongoDB for back-end storage.
Sifino	Sifino enables students to help each other with their studies. Students can share notes, course summaries, and old exams, and can also ask and respond to questions about particular courses.
GameChanger ===	GameChanger provides mobile apps that replace pencil-and-paper scorekeeping and online tools that distribute real-time game updates for amateur sports.
SOLIMAP Servas Les PO	soliMAP is a map-based ad listings site that uses MongoDB for storage.
MyBankTracker	MyBankTracker iPhone App uses MongoDB for the iPhone app's back-end server.
	BillMonitor uses MongoDB to store all user data, including large amounts of billing information. This is used by the live site and also by BillMonitor's internal data analysis tools.
Tubricator	Tubricator allows you to create easy to remember links to YouTube videos. It's built on MongoDB and Django.



Mu.ly uses MongoDB for user registration and as a backend server for its iPhone Push notification service. MongoDB is mu.ly's Main backend database and absolute mission critical for mu.ly.

See also

User Feedback

Articles

See also the User Feedback page for community presentations, blog posts, and more.

Best of the MongoDB Blog

- What is the Right Data Model? (for non-relational databases)
- · Why Schemaless is Good
- The Importance of Predictability of Performance
- Capped Collections one of MongoDB's coolest features
- Using MongoDB for Real-time Analytics
- Using MongoDB for Logging
- http://blog.mongodb.org/tagged/best+of

Articles / Key Doc Pages

- On Atomic Operations
- · Reaching into Objects how to do sophisticated query operations on nested JSON-style objects
- Schema Design
- · Full Text Search in Mongo
- MongoDB Production Deployments

Videos

- NY MySql Meetup NoSQL, Scaling, MongoDB
- Teach Me To Code Introduction to MongoDB
- DCVIE

Benchmarks

If you've done a benchmark, we'd love to hear about it! Let us know at kristina at 10gen dot com.

January 23, 2010 - Inserts and queries against MySQL, CouchDB, and Memcached.

May 10, 2009 - MongoDB vs. CouchDB vs. Tokyo Cabinet

July 2, 2009 - MongoDB vs. MySQL

September 25, 2009 - MongoDB inserts using Java.

August 11, 2009 - MySQL vs. MongoDB vs. Tokyo Tyrant vs. CouchDB inserts and queries using PHP.

August 23, 2009 - MySQL vs. MongoDB in PHP: Part 1 (inserts), Part 2 (queries), aginst InnoDB with and without the query log and MyISAM.

November 9, 2009 - MySQL vs. MongoDB in PHP and Ruby inserts (original Russian, English translation)

Disclaimer: these benchmarks were created by third parties not affiliated with MongoDB. MongoDB does not guarantee in any way the correctness, thoroughness, or repeatability of these benchmarks.

Use Cases

See also the Production Deployments page for a discussion of how companies like Disgus, EA, Github, SourceForge, etc. use MongoDB.

Use Case Articles

- Using MongoDB for Real-time Analytics
- Using MongoDB for Logging
- Full Text Search in Mongo

Well Suited

- Operational data store of a web site. MongoDB is very good at real-time inserts, updates, and queries. Scalability and replication are
 provided which are necessary functions for large web sites' real-time data stores
- Caching. With its potential for high performance, MongoDB works well as a caching tier in an information infrastructure. The persistent
 backing of Mongo's cache assures that on a system restart the downstream data tier is not overwhelmed with cache population activity.
- "High volume, low value data". Problems where a traditional DBMS might be too expensive for the data in question. In many cases
 developers would traditionally write custom code to a filesystem instead using flat files or other methodologies.
- Problems requiring high scalability. Mongo is well suited to problems where the database must comprise tens or hundreds of servers.
 Map/Reduce engine integration is planned in the Mongo roadmap.
- Storage of program objects and JSON data (and equivalent). Mongo's BSON data format makes it very easy to store and retrieve data in
 a document-style / "schemaless" format. Addition of new properties to existing objects is easy and does not require blocking "ALTER
 TABLE" style operations.

Less Well Suited

- Highly transactional systems, such as banking systems and accounting. MongoDB, like most "NOSQL" solutions, provides lightweight
 transactionality: atomicity around single documents only. Applications with highly complex transactions are more suited to a traditional
 RDBMS
- Traditional Business Intelligence. Data warehouses are more suited to new, problem-specific BI databases. However note that
 MongoDB can work very well for several reporting and analytics problems where data is predistilled or aggregated in runtime -- but
 classic, business intelligence is not a sweet spot.
- · Problems requiring SQL.

Use Case - Session Objects

MongoDB is a good tool for storing HTTP session objects.

One implementation model is to have a sessions collection, and store the session object's id value in a browser cookie.

With its update-in-place design and general optimization to make updates fast, the database is efficient at receiving an update to the session object on every single app server page view.

Aging Out Old Sessions

The best way to age out old sessions is to use the auto-LRU facility of capped collections. The one complication is that objects in capped collections may not grow beyond their initial allocation size. To handle this, we can "pre-pad" the objects to some maximum size on initial addition, and then on further updates we are fine if we do not go above the limit. The following mongo shell javascript example demonstrates padding.

(Note: a clean padding mechanism should be added to the db so the steps below are not necessary.)

```
> db.createCollection('sessions', { capped: true, size : 1000000 } )
{"ok" : 1}
> p = "";
> for(x = 0; x < 100; x++) p += 'x';
> s1 = { info: 'example', _padding : p };
{"info" : "example" , "_padding" :
> db.sessions.save(s1)
> s1
{"info" : "example" , "_padding" :
, "_id" : ObjectId( "4aafb74a5761d147677233b0") }
> // when updating later
> s1 = db.sessions.find( { _id : ObjectId( "4aafb74a5761d147677233b0") } )
{"_id" : ObjectId( "4aafb74a5761d147677233b0") , "info" : "example" , "_padding" :
}
> delete s._padding;
true
> s.x = 3; // add a new field
3
> db.sessions.save(s);
{"_id" : ObjectId( "4aafb5a25761d147677233af") , "info" : "example" , "x" : 3}
```

FAQ

- · What kind of database is the Mongo database?
- What languages can I use to work with the Mongo database?
- Does it support SQL?
- Is caching handled by the database?
- How do I do snapshotting in Mongo?
- What language is MongoDB written in?
- What are the 32-bit limitations?

What kind of database is the Mongo database?

MongoDB is an document-oriented DBMS. Think of it as a persistent object store. It is neither a relational database, nor even "table oriented" like Amazon SimpleDB or Google BigTable. If you have used object-relational mapping layers before in your programs, you will find the Mongo interface similar to use, but faster, more powerful, and less work to set up.

What languages can I use to work with the Mongo database?

The first drivers are for Java, C++, Ruby, Python, and PHP. Server-Side Javascript (V8 engine) is under development -- ask for details.

You can also use Mongo via the Babble application server (from JavaScript, Ruby or Python).

Does it support SQL?

No. See the Tour and Advanced Queries pages for more information on how one performs operations.

Is caching handled by the database?

For simple queries (with an index) Mongo should be fast enough that you can query the database directly without needing the equivalent of memcached. The goal is for Mongo to be an alternative to an ORM/memcached/mysql stack.

How do I do snapshotting in Mongo?

A latent cursor has (in addition to an initial access) a latent access that occurs after an intervening write operation on the database collection (i.e., an insert, update, or delete). Under most circumstances, the database supports these operations.

Conceptually, a cursor has a current position. If you delete the item at the current position, the cursor automatically skips its current position forward to the next item.

Mongo DB cursors do not provide a snapshot: if other write operations occur during the life of your cursor, it is unspecified if your application will see the results of those operations or not.

What language is MongoDB written in?

MongoDB is written in C++. We chose C++ for speed and memory efficiency.

What are the 32-bit limitations?

When running on a 32-bit operating system, the total storage size for the server (data, indexes, everything) is 2gb. If you are running on a 64-bit os, there is virtually no limit to storage size.

Product Comparisons

Interop Demo (Product Comparisons)

Interop 2009 MongoDB Demo

Code: http://github.com/mdirolf/simple-messaging-service/tree/master

MongoDB, CouchDB, MySQL Compare Grid

pending...

	CouchDB	MongoDB	MySQL
Data Model	Document-Oriented (JSON)	Document-Oriented (BSON)	Relational
Data Types	string,number,boolean,array,object	string, int, double, boolean, date, bytearray, object, array, others	link
Large Objects (Files)	Yes (attachments)	Yes (GridFS)	no???
Replication	Master-master (with developer supplied conflict resolution)	Master-slave	Master-slave
Object(row) Storage	One large repository	Collection based	Table based
Query Method	Map/reduce of javascript functions to lazily build an index per query	Dynamic; object-based query language	Dynamic; SQL
Secondary Indexes	Yes	Yes	Yes
Atomicity	Single document	Single document	Yes - advanced
Interface	REST	Native drivers	Native drivers
Server-side batch data manipulation	?	Map/Reduce, server-side javascript	Yes (SQL)
Written in	Erlang	C++	С
Concurrency Control	MVCC	Update in Place	

See Also

• Comparing Mongo DB and Couch DB

Comparing Mongo DB and Couch DB

We are getting a lot of questions "how are mongo db and couch different?" It's a good question: both are document-oriented databases with schemaless JSON-style object data storage. Both products have their place -- we are big believers that databases are specializing and "one size fits all" no longer applies.

We are not CouchDB gurus so please let us know in the forums if we have something wrong.

MVCC

One big difference is that CouchDB is MVCC based, and MongoDB is more of a traditional update-in-place store. MVCC is very good for certain classes of problems: problems which need intense versioning; problems with offline databases that resync later; problems where you want a large amount of master-master replication happening. Along with MVCC comes some work too: first, the database must be compacted periodically, if there are many updates. Second, when conflicts occur on transactions, they must be handled by the programmer manually (unless the db also does conventional locking -- although then master-master replication is likely lost).

MongoDB updates an object in-place when possible. Problems require high update rates of objects are a great fit; compaction is not necessary. Mongo's replication works great but, without the MVCC model, it is more oriented towards master/slave and auto failover configurations than to complex master-master setups. With MongoDB you should see high write performance, especially for updates.

Horizontal Scalability

One fundamental difference is that a number of Couch users use replication as a way to scale. With Mongo, we tend to think of replication as a way to gain reliability/failover rather than scalability. Mongo uses (auto) sharding as our path to scalabity (sharding is in alpha). In this sense MongoDB is more like Google BigTable. (We hear that Couch might one day add partitioning too.)

Query Expression

Couch uses a clever index building scheme to generate indexes which support particular queries. There is an elegance to the approach, although one must predeclare these structures for each query one wants to execute. One can think of them as materialized views.

Mongo uses traditional dynamic queries. As with, say, MySQL, we can do queries where an index does not exist, or where an index is helpful but only partially so. Mongo includes a query optimizer which makes these determinations. We find this is very nice for inspecting the data administratively, and this method is also good when we *don't* want an index: such as insert-intensive collections. When an index corresponds perfectly to the query, the Couch and Mongo approaches are then conceptually similar. We find expressing queries as JSON-style objects in MongoDB to be quick and painless though

Atomicity

Both MongoDB and CouchDB support concurrent modifications of single documents. Both forego complex transactions involving large numbers of objects.

Durability

The products take different approaches to durability. CouchDB is a "crash-only" design where the db can terminate at any time and remain consistent. MongoDB take a different approach to durability. On a machine crash, one then would run a repairDatabase() operation when starting up again (similar to MyISAM). MongoDB recommends using replication -- either LAN or WAN -- for true durability as a given server could permanently be dead. To summarize: CouchDB is better at durability when using a single server with no replication.

Map Reduce

Both CouchDB and MongoDB support map/reduce operations. For CouchDB map/reduce is inherent to the building of all views. With MongoDB, map/reduce is only for data processing jobs but not for traditional gueries.

Javascript

Both CouchDB and MongoDB make use of Javascript. CouchDB uses Javascript extensively including in the building of views .

MongoDB supports the use of Javascript but more as an adjunct. In MongoDB, query expressions are typically expressed as JSON-style query objects; however one may also specify a javascript expression as part of the query. MongoDB also supports running arbitrary javascript functions server-side and uses javascript for map/reduce operations.

REST

Couch uses REST as its interface to the database. With its focus on performance, MongoDB relies on language-specific database drivers for access to the database over a proprietary binary protocol. Of course, one could add a REST interface atop an existing MongoDB driver at any time -- that would be a very nice community project. Some early stage REST implementations exist for MongoDB.

Performance

Philosophically, Mongo is very oriented toward performance, at the expense of features that would impede performance. We see Mongo DB being useful for many problems where databases have not been used in the past because databases are too "heavy". Features that give MongoDB good performance are:

- client driver per language: native socket protocol for client/server interface (not REST)
- use of memory mapped files for data storage

- collection-oriented storage (objects from the same collection are stored contiguously)
- update-in-place (not MVCC)
- written in C++

Use Cases

It may be helpful to look at some particular problems and consider how we could solve them.

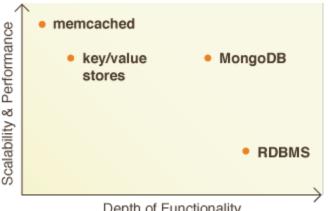
- if we were building Lotus Notes, we would use Couch as its programmer versioning reconciliation/MVCC model fits perfectly. Any problem where data is offline for hours then back online would fit this. In general, if we need several eventually consistent master-master replica databases, geographically distributed, often offline, we would use Couch.
- if we had very high performance requirements we would use Mongo. For example, web site user profile object storage and caching of data from other sources.
- if we were building a system with very critical transactions, such as financial transactions, we would not use MongoDB for those transactions -- although we might in hybrid for other data elements of the system. For something like this we would likely choose a traditional RDBMS.
- for a problem with very high update rates, we would use Mongo as it is good at that. For example, updating real time analytics counters for a web sites (pages views, visits, etc.)

Generally, we find MongoDB to be a very good fit for building web infrastructure.

Philosophy

Design Philosophy

- Databases are specializing the "one size fits all" approach no longer applies.
- By reducing transactional semantics the db provides, one can still solve an interesting set of problems where performance is very important, and horizontal scaling then becomes easier.
- The document data model is easy to code to, easy to manage (schemaless), and yields excellent performance by grouping relevant data together internally.
- A non-relational approach is the best path to database solutions which scale to many machines.
- While there is an opportunity to relax certain capabilities for better performance, there is also a need for deeper functionality than that provided by pure key/value stores.
- Database technology should run anywhere, being available both for running on your own servers or VMs, and also as a cloud pay-for-what-you-use service.



Depth of Functionality

Licensing



If you are using a vanilla MongoDB server from either source or binary packages you have NO obligations. You can ignore the rest of this page.

All of the MongoDB software is open source, under a variety of licenses. The following is a list of components and licenses:

- MongoDB: GNU AGPL v3.0
- mongodb.org Supported Drivers: Apache License v2.0 (third parties have created drivers too; licenses will vary there)
- **Documentation: Creative Commons**

From our blog post on the AGPL:

Our goal with using AGPL is to preserve the concept of copyleft with MongoDB. With traditional GPL, copyleft was associated with the concept of distribution of software. The problem is that nowadays, distribution of software is rare: things tend to run in the cloud. AGPL fixes this "loophole" in GPL by saying that if you use the software over a network, you are bound by the copyleft. Other than that, the license is virtually the same as GPL v3.

Note however that it is never required that applications using mongo be published. The copyleft applies only to the mongod and mongos database programs. This is why Mongo DB drivers are all licensed under an Apache license. Your application, even though it talks to the database, is a separate program and "work".

If you intend to modify the server and distribute or provide access to your modified version you are required to release the full source code for the

modified MongoDB server. To reiterate, you **only** need to provide the source for the MongoDB server and not your application (assuming you use the provided interfaces rather than linking directly against the server).

A few example cases of when you'd be required to provide your changes to MongoDB to external users:

Case	Required
Hosting company providing access MongoDB servers	yes
Public-facing website using MongoDB for content	yes
Internal use website using MongoDB	no
Internal analysis of log files from a web site	no

Regardless of whether you are *required* to release your changes we request that you do. The preferred way to do this is via a github fork. Then we are likely to include your changes so everyone can benefit.

International Documentation



Most documentation for MongoDB is currently written in English. We are looking for volunteers to contribute documentation in other languages. If you're interested in contributing to documentation in another language please email **mike at 10gen dot com**

Language Homepages

- DeutscheEspañolFrançais
- FrançaisItaliano

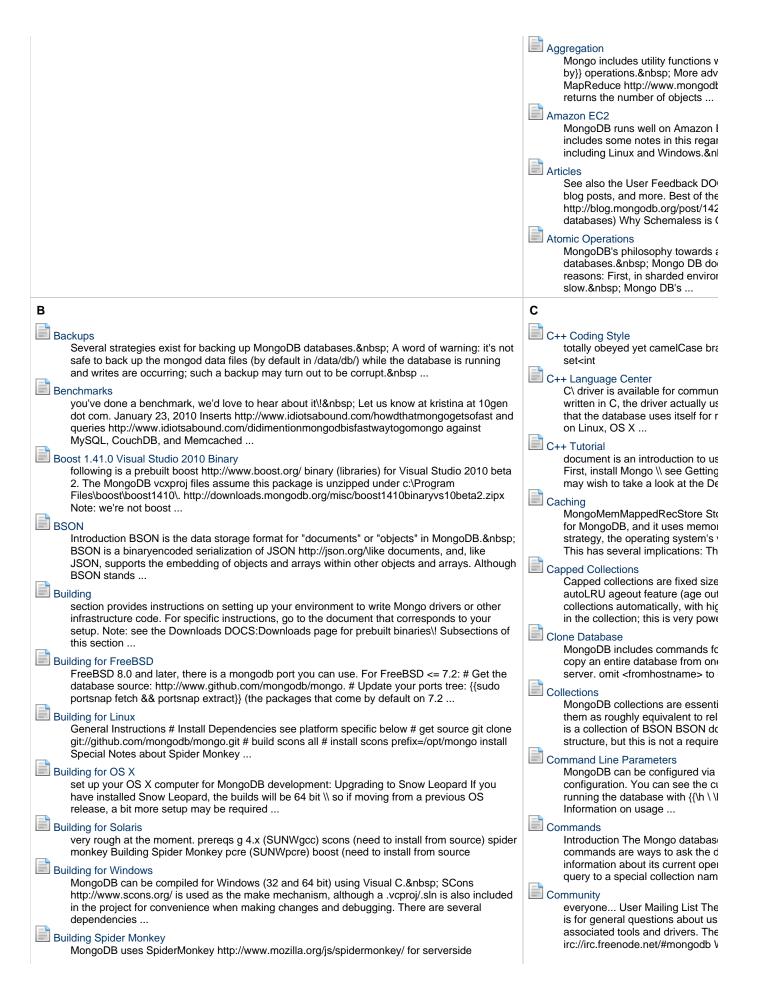
Documentation Index

Space Index

0-9 4	A 9	B 12	C 24	D 1
F 6	G 6	H 5	I 14	J 8
L 7	M 24	N 1	O 11	P ′
R 11	S 26	T 5	U 8	V {
X 0	Y 0	Z 0	!@#\$ 0	

0-9 1.0 Changelist A Sample Configuration Session Wrote MongoDB. See documentation following example uses two share process, all running on a single to 1.1 Development Cycle mkdir /data/db/config ... 1.2.x Release Notes About New Features More indexes per collection Faster index creation Map/Reduce Stored Admin Zone JavaScript functions Configurable fsync time Several small features and fixes DB Upgrade Required There are some changes that will require doing an upgrade ... See Also Commands DOCS:Con 1.4 Release Notes Administration Basics Focus Concurrency Features Sharding Advanced Queries Introduction MongoDB offers a rie lists some of those features. Que objects, very much like the docur

example: db.things.find(); // i.e. .



Javascript execution. (V8 support is under development.) Building js.lib Download curl O ftp://ftp.mozilla.org/pub/mozilla.org/js/js1.7.0.tar.gz tar zxvf js1.7.0.tar.gz Build	Community Info
Building SpiderMonkey	Community Tools Memcached MongoNode http://code.google.com/p/phpdae PHP script that replicates Memca
	Compare Order for Types MongoDB allows objects in the s type. When comparing va which value is less than the other
	Comparing Mongo DB and Couch D We are getting a lot of questions good question: both are docume object data storage. Both databases are specializing
	page has implementation details Logstream::mutex clientcursor m rand() calls, getNonce? MongoM
	Start the relevant processes. Rur \shardsvr command line paramet Pairs command line option.&nbs
	Connecting C\ driver includes several classe: DBClientInterface http://api.mongodb.org/cplusplus you will want to instantiate either http://api.mongodb.org/cplusplus a DBClientPaired http://api.mong
	Connections format described on this page is documentation to see how much Currently, only the PHP http://ww http://api.mongodb.org
	Contributing to the Documentation Qualified volunteers are welcome for more information
	Contributing to the Perl Driver easiest way to contribute is to file http://jira.mongodb.org/browse/P on Finding Something to Help \
	Contributors 10gen Contributor Agreement htt
	Conventions for Mongo Drivers Interface Conventions It is desira Of course, idioms vary by langua However, when the idiom is the s is desirable. Terminology In gene
	Creating and Deleting Indexes Cursors
D	Е
Data Processing Manual DRAFT TO BE COMPLETED. This guide provides instructions for using MongoDB batch data processing oriented features including map/reduce DOCS:MapReduce. By "data processing", we generally mean operations performed on large sets of data, rather than small	Error Codes Error Code \\ Description \\ Comr cannot grow E11000 duplicate ke E11001 duplicate key on update
Data Types and Conventions MongoDB (BSON) Data Types Mongo uses special data types in addition to the basic JSON types of string, integer, boolean, double, null, array, and object. These types include date, object id Object IDs, binary data, regular	Error Handling in Mongo Drivers an error occurs on a query (or ge instead of user data. The error ot key {{\$err}}. For example: The {{
Database Internals section provides information for developers who want to write drivers or tools for MongoDB, \ contribute code to the MongoDB codebase itself, and for those who are just curious how it works internally. Subsections of this section	
Database Profiler Mongo includes a profiling tool to analyze the performance of database operations. See also	

the currentOp DOCS:Viewing and Terminating Current Operation command. Enabling Profiling To enable profiling, from the {{mongo}} shell invoke: > db.setProfilingLevel(2); > db.getProfilingLevel() 2 Profiling ... Each MongoDB server can support multiple databases. Each database is independent, and the data for each database is stored separately, for security and ease of management. A database consists of one or more collections, the documents (objects) in those collections, and an optional set ... 🗐 DB Ref DBRef is a specification for creating references between documents. Specification Database references are references from one document (object) to another within a database. A database reference is a standard embedded object: we are defining a convention, not a special type. By having a standard ... BA Operations from the Shell page lists common DBAclass operations that one might perform from the MongoDB shell DOCS:mongo The Interactive Shell. Note one may also create .js scripts to run in the shell for administrative purposes. help show help show ... dbshell Reference Special Command Helpers Nonjavascript convenience macros: {{show dbs}} Print a list of all databases on this server {{use dbname{}}} Set the db variable to represent usage of dbname on the server {{show collections}} Print a list of all collections for current database ... Design Overview Developer FAQ How do I copy all objects from one database collection to another? See below. The code below may be ran serverside for high performance with the eval() method. db.myoriginal.find().forEach(function(x)); If you remove an object ... Developer Zone Tutorial Shell mongo The Interactive Shell Manual Databases Collections Indexes Data Types and Conventions GridFS Inserting Updating Querying Removing Optimization Developer FAQ If you have a comment or question about anything, please contact us through ... Diagnostics and Troubleshooting Mongo provides the following tools for diagnostics and troubleshooting: Profiler This is the best way to see slow queries. See full docs Database Profiler Database Record/Replay This is a diagnostic capability. To enable logging: db.adminCommand() To disable: db.adminCommand() Values for opLogging: 0 ... Django and MongoDB **Document-Oriented Datastore** Documentation **Documentation Index Dot Notation** Dot Notation Dot Notation (Reaching into Objects) MongoDB is designed for store JSONstyle objects. The database understands the structure of these objects and can reach into them to evaluate query expressions. Let's suppose we have some objects of the form: > db.persons.findOne() { name: "Joe", address: , likes: 'scuba'. 'math ... Downloads MongoDB Downloads Version OS X 32 bit OS X 64 bit Linux 32 bit Linux 64 bit Windows 32 bit Windows 64bit Solaris i86pc Solaris 64 Source Date Change Log ... Driver Syntax Table wiki generally gives examples in JavaScript, so this chart can be used to convert those examples to any language. JavaScript Python PHP Ruby Java C\ Perl \ \ \\ {{array()}} \\ {{BasicDBList}} BSONObj \\ {} {{MongoEmptyObj}} {} {{BasicDBObject}} BSONObj {} \\ ${ (array('x' => 1 ...$ Drivers MongoDB currently has client support for the following programming languages: mongodb.org Supported C http://github.com/mongodb/mongocdriver C\ C Language Center Java Java Language Center Javascript Javascript Language Center Perl Perl Language ... Durability and Repair Relational database systems achieve durability, in part, by writing all updates to a transaction log http://en.wikipedia.org/wiki/Transactionlog. However, transaction logging incurs a performance penalty, so MongoDB takes an alternate approach to durability: replication http ... F G

getLastError

What kind of database is the Mongo database? MongoDB is an documentoriented DBMS. Think of it as a persistent object store. It is neither a relational database, nor even "table oriented" like Amazon SimpleDB or Google BigTable. If you have used objectrelational mapping ... Feature Checklist for Mongo Drivers Functionality Checklist This section lists tasks the driver author might handle. Essential BSON serialization/deserialization Basic operations: {{query}}, {{save}}, {{update}}, {{remove}}, {{ensureIndex}}, {{findOne}}, {{limit}}, {{sort}} Fetch more data from a cursor when necessary ... File Based Configuration addition to accepting command line parameters, MongoDB can also be configured using a configuration file. A configuration file to use can be specified using the {{\f}} or {{\config}} command line options. The following example configuration file demonstrates the syntax to use

findandmodify Command

Find and Modify (or Remove) In 1.3.0 and higher MongoDB 1.3\ supports a "find, modify, and return" command. This command is useful to atomically change an object and then get back the results. It also includes a sort option which is useful when storing ...

Version 1.3.1 and higher The fsync command allows us to flush all pending writes to datafiles. More importantly, it also provides a lock option that makes backups easier. Basics The fsync command forces the database to flush all datafiles ...

Full Text Search in Mongo

Introduction Mongo provides some functionality that is useful for text search and tagging. Multikeys (Indexing Values in an Array) The Mongo multikey feature can automatically index arrays of values. Tagging is a good example of where this feature is useful. Suppose you ...

A Home

Н

Combining the best features of document databases, keyvalue stores, and RDBMSes. MongoDB (from "hu{}mongo{}us") is a scalable, highperformance, open source, schemafree, documentoriented database. Written in C, MongoDB features: Documentoriented storage (the simplicity and power of JSONlike DOCS ...

Hosting Center

Setup Instructions Amazon EC2 DOCS:Amazon EC2 Joyent DOCS:Joyent Webfaction http://docs.webfaction.com/software/mongodb.html Linode http://library.linode.com/databases/mongodb/ Hosted MongoDB MongoHQ

http://mongohq.com/ provides cloudstyle hosted MongoDB instances ...

How to do Snapshotted Queries in the Mongo Database

document refers to query snapshots. For backup snapshots of the database's datafiles, see the fsync lock page fsync Command. For performance reasons, MongoDB does not currently support true pointintime snapshotting of collections. (This may change in the future.) However ...

HowTo

Http Interface

MongoDB provides a simple http interface listing information of interest to administrators. This interface may be accessed at the port with numeric value 1000 more than the configured mongod port; the default port for the http interface is 28017. To access the http interface an ...

Most drivers, and the db shell, su check the error code on the last of have a direct return code.

Getting Started

Getting started with MongoDB is prebuilt distribution for Linux, OS http://github.com/mongodb/ You on the Downloads page. Once ...

Getting the Software

Placeholder \$\$\$ TODO

GridFS

GridFS is a specification for stori native storage of binary data with BSON objects in MongoDB are li for gigabyte ...

GridFS Specification

Introduction GridFS is a storage: takes large objects and stores the document specifies the requirem not worry about the details of the

GridFS Tools

File Tools {{mongofiles}} is a tool command line. A simple interaction ./mongofiles put libmongoclient.a connected to: 127.0.0.1 libmongc

Implementing Authentication in a Dri

current version of Mongo support authenticates a username and pa Once authenticated, the user has The {{admin}} database is specia

Import Export Tools

mongoimportison replaced in 1.1 contains 1 JSON string per line a collection. importjson options: he

Index-Related Commands

Create Index {{ensureIndex()}} is an index by adding its info to the db.myCollection.ensureIndex(<ke); You can query system.indexes

Indexes

Indexes enhance query performa kinds of queries your application Once that's done, actually creatir An ...

Indexes in Mongo

Indexing as a Background Operation default the {{ensureIndex()}} ope

database from proceeding until c option is available. To build an in index options. Examples: > db.th

Inserting

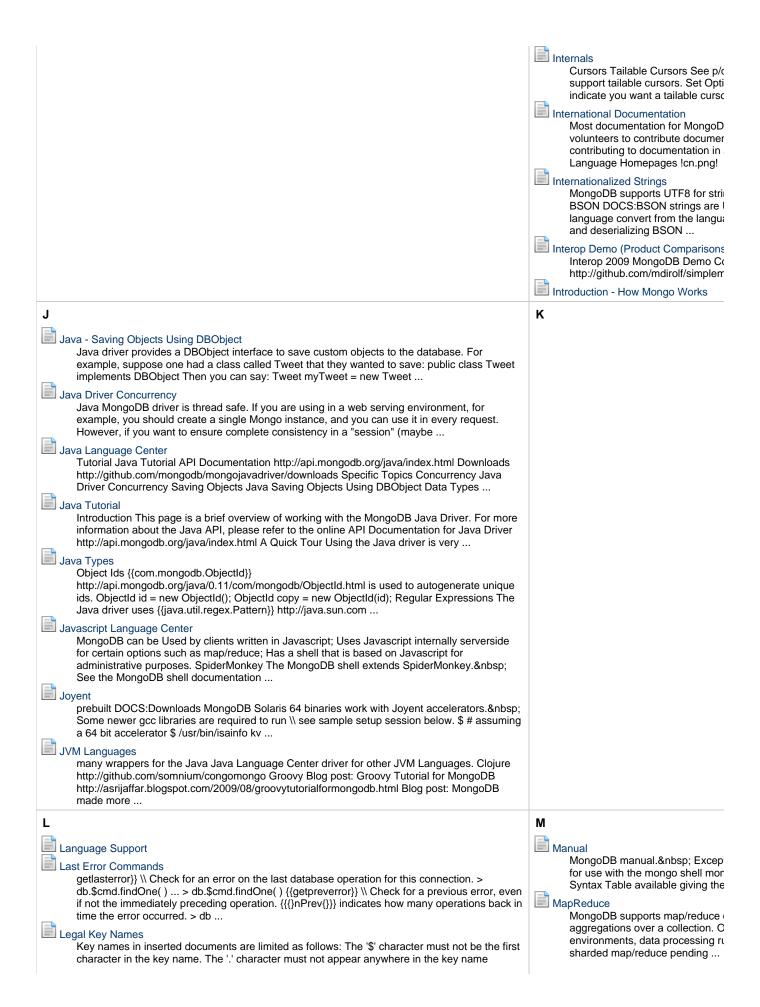
When we insert data into Mongol Documents are data structure an hashes, to take just a few examp and describe how to insert ...

Installing the PHP Driver

PECL You can download and ins mongo Add the following line to p Windows Install The precompiled VC9 Builds ...

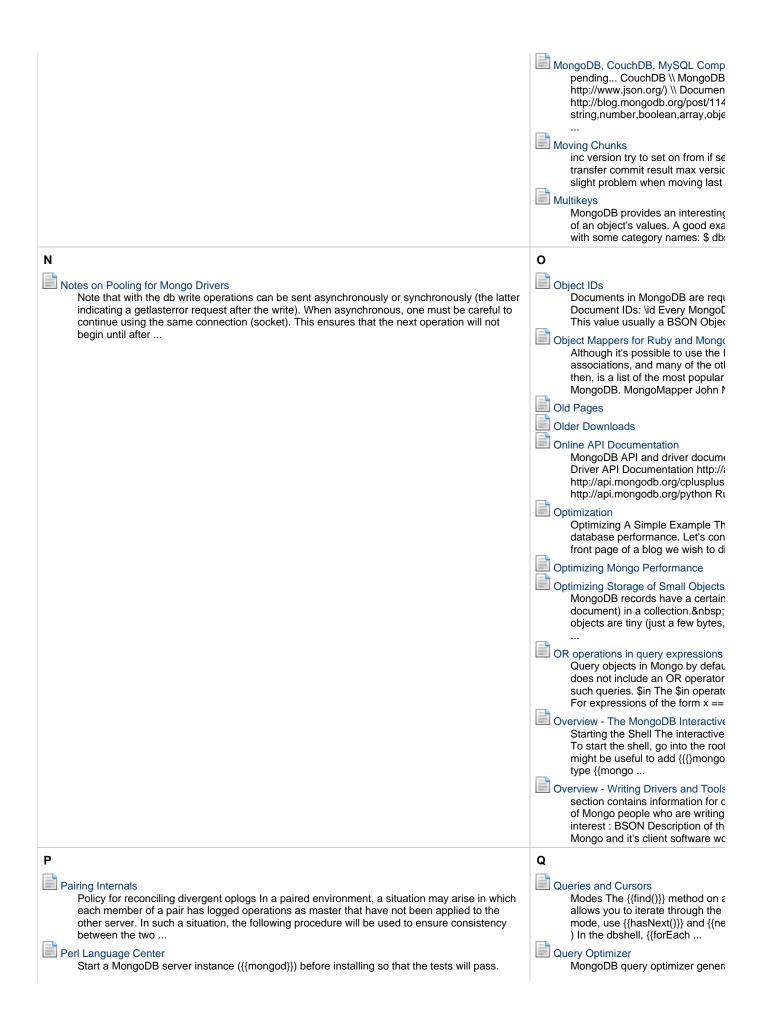
Internal Commands

Most commands Commands hav {{\$cmd.findOne()}} syntax. These db.\$cmd.findOne() > db.\$cmd.fii 1 } > admindb.\$cmd.findOne() //



Licensing Master Master Replication you are using a vanilla MongoDB server from either source or binary packages you have NO Mongo does not support full mas obligations. You can ignore the rest of this page. All of the MongoDB software is open source, restricted use cases mastermaste under a variety of licenses. The following is a list of components and licenses ... does not use the database in a m consistent. To configure ... List of Database Commands Master Slave work in progress. Key: Auth \ requires authentication Admin \ must be run from the admin database Slave Okay \ can be run on a slave db server Command \\ Requires Auth \\ Admin Setup of a Manual Master/Slave Only \\ Slave Okay \\ Description \\ No \\ Yes Yes \\ Returns ... a master database in a mastersla the database, one in master mod examples explicitly specify the lo Locking in Mongo in min and max Query Specifiers min()}} and {{max()}} functions m query matches to those having ir The {{min()}} and {{max()}} function index to be used may ... mongo - The Interactive Shell MongoDB Interactive Shell The N {{bin/mongo}}, the MongoDB inte you to issue commands to Mongi inspecting a database's contents Mongo Administration Guide Mongo Architecture MongoDB has two primary comp mongod process which is the cor may be used as a selfcontained : server ... Mongo Concepts and Terminology Mongo Database Administration Mongo Developers' Guide Mongo Documentation Style Guide page provides information for eve Confluence. It covers: #General I specific situations Some general Style Voice Active ... Mongo Driver Requirements Provisional Driver API "Level 0 D supports the basic operations of level 0 should be "document" ope Mongo Extended JSON Mongo's REST interface support representations are used for BSC and multiple representations are supports three different modes fc Mongo Metadata system. namespaces in Mongo a information. System colled namespaces. {{system.indexes}} metadata exists in the database. Mongo Usage Basics Mongo Wire Protocol Introduction #TableOfContents T requestresponse style protocol. (a regular TCP/IP socket. The def Clients should connect to the dat MongoDB - A Developer's Tour MongoDB Commercial Services Pro Note: if you provide consultative here, just let us know. Support 10 commercial support and manage MongoHQ http ... MongoDB Data Modeling and Rails tutorial discusses the developme MongoMapper will serve as our c insight into the design choices re we'll be constructing a simple but

MongoDB Language Support



Some tests may be skipped if you are not running a recent version of the database (>= 1.1.3). Installing CPAN \$ sudo cpan MongoDB The Perl driver is available through CPAN Perl Tutorial Philosophy Design Philosophy !featuresPerformace.png align=right! Databases are specializing the "one size fits all" approach no longer applies. By reducing transactional semantics the db provides, one can still solve an interesting set of problems where performance is very PHP - Storing Files and Big Data page is no longer actively maintained. See the PHP manual http://www.php.net/manual/en/class.mongogridfs.php for the most uptodate documentation. The PHP driver can be used to easily store and retrieve files. About GridFS GridFS is a storage specification all supported PHP Language Center Installing the PHP Driver NIX Run: sudo pecl install mongo See Installing the PHP Driver for configuration information and OSspecific installation instructions. Windows Download one of the binaries http://github.com/mongodb/mongophpdriver/downloads and place it on your extension Product Comparisons Production Deployments you're using MongoDB in production, we'd love to list you here\! Email mike at 10gen . com. <div mcestyle="textalign:center;margin:0" style="margin: 0pt; textalign: center;">Company Description: center; ">Company Description: Case !logosourceforge.png</div>	client. These plans are ex interesting approach to query opt be costbased and statistical) are Querying MongoDB features a rich languar MongoDB supports a number of fetching data. Queries are expres database server as BSON. Queries are expres database server as BSON. Queries are even quicker start go to http://is easy. For a longer desc instructions assume a modern \N
Production Notes Architecture Production Options Master Slave 1 master, N slaves have to handle failover manually Replica Pairs 2 servers, 1 is always master, autofailover Backups Import Export Tools Recommended System Settings turn off Project Ideas you're interested in getting involved in the MongoDB community (or the open source community in general) a great way to do so is by starting or contributing to a MongoDB related project. Here we've listed some project ideas for you to get started on. For some of these ideas	
PyMongo and mod_wsgi Python Language Center Python Tutorial	
Rails - Getting Started tutorial describes how to set up a simple Rails application with MongoDB, using MongoMapper as an object mapper. We assume you're using Rails 2.3 or greater. {}Using a Rails Template All of the configuration steps listed below, and more	S Schema Design Introduction With Mongo, you do a relational schema because the one database collection for each
Recommended Production Architectures Removing Removing Objects from a Collection To remove objects from a collection, use the {{remove()}} function in the mongo shell mongo The Interactive Shell. (Other drivers offer a similar function, but may call the function "delete". Please check your driver's documentation Replica Pairs	Searching and Retrieving Security and Authentication Running Without Security (Truste database is in a trusted environm the default option and is recomm must be sure only trusted machin
Setup of Replica Pairs Mongo supports a concept of replica pairs. These databases automatically coordinate which is the master and which is the slave at a given point in time. At startup, the databases will negotiate which is master and which is slave. Upon an outage of one database Replica Pairs in Ruby	Server-side Code Execution Introduction Mongo supports the {{\$where}} Clauses and Function query specification for {{find()}} o string containing a SQLstyle WHI
Here follow a few considerations for those using the Ruby driver Ruby Tutorial with MongoDB and replica pairing. Setup First, make sure that you've correctly paired two {{mongod}} instances. If you want to do this on the same machine for testing, make Replication Mongo database supports replication of data between servers. The replication is an enhanced	Server-Side Processing Shard Ownership shard ownership we mean which draft/thoughts will change: Contra the config database mongos inst-
masterslave configuration: that is, only one server is active for writes (the master) at a given time. The primary goal of replication is failover and redundancy. Mongo supports two forms of replication	Shard v0.7
Replication Internals master mongod instance, the {{local}} database will contain a collection, {{oplog.\$main}}, which	Simple autosharding suitable for and removal of nodes from live s Sharding

1.4 2010 Q1 1.6 2010 Q2

Ruby External Resources

number of good resources appearing all over the web for learning about MongoDB and Ruby. A useful selection is listed below. If you know of others, do let us know. Screencasts Introduction to MongoDB Part I http://www.teachmetocode.com/screencasts ...

Ruby Language Center

an overview of the available tools and suggested practices for using Ruby with MongoDB. Those wishing to skip to more detailed discussion should check out the Ruby Driver Tutorial Ruby Tutorial, Rails Getting Started Rails Getting Started, and MongoDB ...

Ruby Tutorial

tutorial gives common examples of with MongoDB using the Ruby driver. If you're looking for information on data modeling, see MongoDB Data Modeling and Rails. Links to the various object mappers are listed on our object mappers page http://www.mongodb.org/display ...

Sharding Administration

See also Configuring DOCS:Con mongos process or > // straight to returns > db.\$cmd.findOne(); Lis "ok": 1 } List Which Databases ..

Sharding and Failover

Failure of a mongod process with shard is down, read and write op migrations to that shard will be su mongod ...

Sharding Commands

All of these commands need to b mongos. db management db.run management db.runCommand(\ on partitioning to be able to have

Sharding Design

concepts config database \ the tc and where things live. shard. this database \ one top level namesp region ...

Sharding FAQ

Where do unsharded collections unsharded data goes to the "prim config.databases to see details). different shards (that is, a collection)

Sharding Internals

section includes internal impleme also the main sharding documen could be out of date \\ if you see

Sharding Introduction

MongoDB includes autosharding provides an architectural overvier the limitations Sharding Limits dc !sharding.PNG align=cen

Sharding Limits

Sharding Alpha 2 (MongoDB v1. For Alpha 2, please use unreplication will be available in Alpha 3. group

Sharding Use Cases

What specific use cases do we w techniques) that are challenging youtube) (also, GridFS scaleup): videos ...

Smoke Tests

run basic c unit tests: scons smo starts a mongod instance needec first start mongod, then: scons ...

Sorting and Natural Order

Natural order" is defined as the d When executing a {{find()}} with r natural order. For standard tables although the order is often close

Source Code

All source for MongoDB, it's drive http://github.com/mongodb. Mone http://github.com/mongodb/mong http://github.com/mongodb/mong

Spec, Notes and Suggestions for Mc

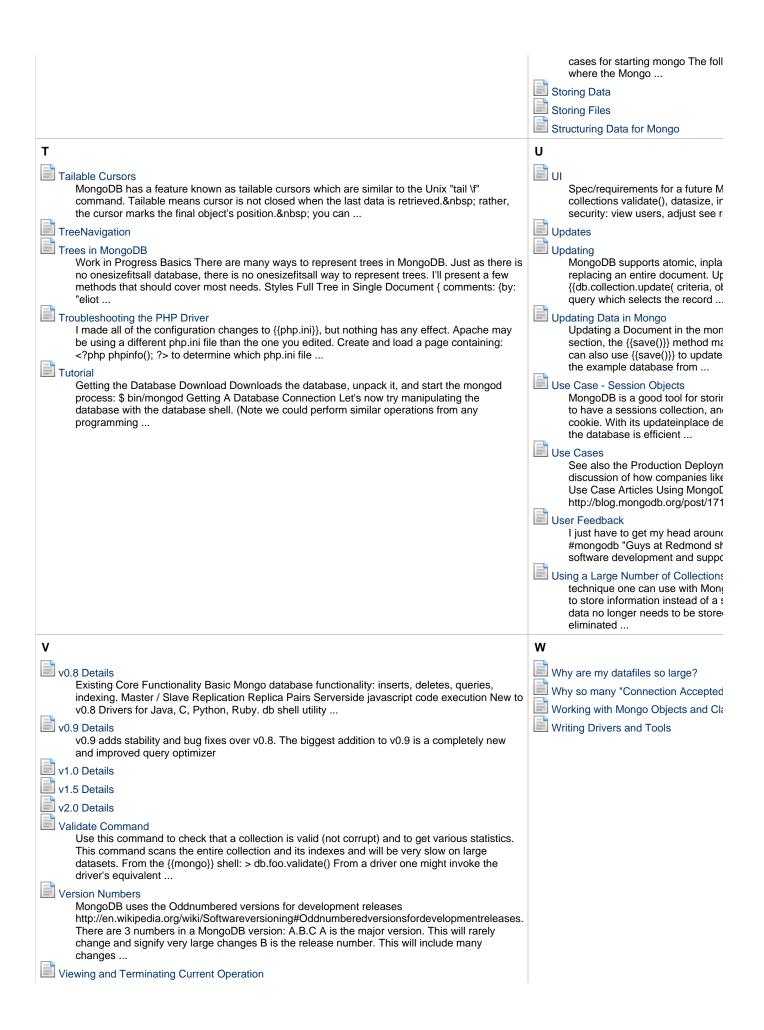
Assume that the BSON DOCS:B 4MB. This size may change over recommend you test your driver

Splitting Shards

chunk \lowbound,highbound)>sh perform. In one case, we same number of values, in anoth

Starting and Stopping Mongo

Mongo is run as a standard progl Line Parameters for more information



	View Current Operation(s) in Progress > db.currentOp(); > // same as: db.\$cmd.sys.inprog.findOne() { inprog: { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" , "query" : " " , "inLock" : 1 } } Fields: opid an incrementing operation number. Use with killOp(). op the operation type	
X		Υ
Z		!@#\$