MarkLogic Server

Concepts Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-4, January, 2018

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

Concepts Guide

1.0 Overview of MarkLogic Server 5
1.1 Relational Data Model vs. Document Data Model 5
1.2 XML Schemas 6
1.3 High Performance Transactional Database 7
1.4 Rich Search Features 7
1.5 Text and Structure Indexes 7
1.6 Semantics Support 8
1.7 Binary Document Support 9
1.8 MarkLogic APIs and Communication Protocols 9
1.9 High Performance 10
1.10 Clustered 11
1.11 Cloud Capable 11

2.0 How is MarkLogic Server Used? 13
2.1 Publishing/Media Industry 13
2.2 Government / Public Sector 14
2.3 Financial Services Industry 15
2.4 Healthcare Industry 15
2.5 Other Industries 17

3.0 Indexing in MarkLogic 19
3.1 The Universal Index 19
3.1.1 Word Indexing 20
3.1.2 Phrase Indexing 21
3.1.3 Relationship Indexing 22
3.1.4 Value Indexing 23
3.1.5 Word and Phrase Indexing 23
3.2 Other Types of Indexes 24
3.2.1 Range Indexing 24
3.2.1.1 Range Queries 27
3.2.1.2 Extracting Values 28
3.2.1.3 Optimized "Order By" 28
3.2.1.4 Using Range Indexes for Joins 29
3.2.2 Word Lexicons 30
3.2.3 Reverse Indexing 31
3.2.3.1 Reverse Query Constructor 31
3.2.3.2 Reverse Query Use Cases 32
3.2.3.3 A Reverse Query Carpool Match 32

3.2.3.4 The Reverse Index 34
3.2.3.5 Range Queries in Reverse Indexes 36
3.2.4 Triple Index 37
3.2.4.1 Triple Index Basics 37
3.2.4.2 Triple Data and Value Caches 38
3.2.4.3 Triple Values and Type Information 39
3.2.4.4 Triple Positions 39
3.2.4.5 Index Files 40
3.2.4.6 Permutations 40
3.3 Index Size 40
3.4 Fields 41
3.5 Reindexing 41
3.6 Relevance 41
3.7 Indexing Document Metadata 41
3.7.1 Collection Indexes 42
3.7.2 Directory Indexes 42
3.7.3 Security Indexes 42
3.7.4 Properties Indexes 42
3.8 Fragmentation of XML Documents 43

4.0 Data Management 45
4.1 What's on Disk 45
4.1.1 Databases, Forests, and Stands 46
4.1.2 Tiered Storage 46
4.1.3 Super Databases and Super Clusters 47
4.1.4 Partitions, Partition Keys, and Partition Ranges 50
4.2 Ingesting Data 52
4.3 Modifying Data 54
4.4 Multi-Version Concurrency Control 54
4.5 Point-in-time Queries 55
4.6 Locking 55
4.7 Updates 55
4.8 Isolating an update 56
4.9 Documents are Like Rows 56
4.10 MarkLogic Data Loading Mechanisms 57
4.11 Content Processing Framework (CPF) 58
4.12 Organizing Documents 59
4.12.1 Directories 59
4.12.2 Collections 59
4.12.3 Unprotected Collections 60
4.12.4 Protected Collections 60
4.13 Database Rebalancing 60
4.14 Bitemporal Documents 62
4.14.1 Bitemporal Data Management 62
4.14.2 Bitemporal Queries 63
4.15 Managing Semantic Triples 63

5.0 Searching in MarkLogic Server 65
5.1 High Performance Full Text Search 65
5.2 Search APIs 66
5.3 Support for Multiple Query Styles 67
5.4 Full XPath Search Support in XQuery 68
5.5 Lexicon and Range Index-Based APIs 69
5.6 Alerting API and Built-Ins 69
5.7 Semantic Searches 69
5.8 Template Driven Extraction (TDE) 70
5.9 Where to Find Additional Search Information 71

6.0 MarkLogic Server Security Model 73
6.1 Role-Based Security Model (Authorization) 74
6.2 The Security Database 75
6.3 Security Administration 77
6.4 SSL and TLS Support 77
6.5 External Authentication (LDAP and Kerberos) 78
6.6 Secure Credentials 78
6.7 Encryption at Rest 79
6.8 Element Level Security 79
6.9 Other Security Tools 79
6.9.1 Protected Collections 80
6.9.2 Compartments 80

7.0 Clustering and Caching 81
7.1 MarkLogic E-nodes and D-nodes 81
7.2 Communication Between Nodes 83
7.3 Communication Between Clusters 83
7.4 Cluster Management 84
7.5 Caching 85
7.6 Cache Partitions 86
7.7 No Need for Global Cache Invalidation 86
7.8 Locks and Timestamps in a Cluster 86

8.0 Application Development on MarkLogic Server 89
8.1 Server-side XQuery and XSLT APIs 90
8.2 Server-side JavaScript API 90
8.3 REST API 91
8.4 Java and Node.js Client APIs 92
8.5 XML Contentbase Connector (XCC) 93
8.6 SQL Support 93
8.7 Optic API 93
8.8 HTTP Functions to Access Internal and External Web Services 94
8.9 Output Options 94

8.10 Remote Filesystem Access 95
8.11 Query Console for Remote Coding 95
8.12 MarkLogic Connector for Hadoop 95

9.0 Administrating and Monitoring MarkLogic Server 97
9.1 Administration Interface 97
9.2 Administration APIs 98
9.3 Configuration Management 99
9.4 Monitoring Tools 99
9.5 Telemetry 100

10.0 High Availability and Disaster Recovery 101
10.1 Managing Backups 101
10.1.1 Typical Backup 101
10.1.2 Backup with Journal Archiving 102
10.1.3 Incremental Backup 103
10.2 Failover and Database Replication 104
10.2.1 Local- and Shared-Disk Failover 105
10.2.1.1 Local-Disk Failover 105
10.2.1.2 Shared-Disk Failover 105
10.2.2 Database Replication 106
10.2.2.1 Bulk Replication 107
10.2.2.2 Bootstrap Hosts 107
10.2.2.3 Inter-cluster Communication 108
10.2.2.4 Replication Lag 108
10.2.2.5 Master and Replica Database Index Settings 109
10.3 Flexible Replication 110
10.4 Query-Based Flexible Replication 110

11.0 Technical Support 113

12.0 Copyright 115
12.0 COPYRIGHT 115

1.0 Overview of MarkLogic Server

MarkLogic is a database designed from the ground up to make massive quantities of heterogenous data easily accessible through search. The design philosophy behind the evolution of MarkLogic is that storing data is only part of the solution. The data must also be quickly and easily retrieved and presented in a way that makes sense to different types of users. Additionally, the data must be reliably maintained by an enterprise grade, scalable software solution that runs on commodity hardware. The purpose of this guide is to describe the mechanisms in MarkLogic that are used to achieve these objectives.

MarkLogic fuses together database internals, search-style indexing, and application server behaviors into a unified system. It uses XML and JSON documents as its data model, and stores the documents within a transactional repository. It indexes the words and values from each of the loaded documents, as well as the document structure. And, because of its unique Universal Index, MarkLogic does not require advance knowledge of the document structure and adherence to a particular schema. Through its application server capabilities, it is programmable and extensible.

MarkLogic clusters on commodity hardware using a shared-nothing architecture and supports massive scale, high-availability, and very high performance. Customer deployments have scaled to hundreds of terabytes of source data while maintaining sub-second query response time.

This chapter contains the following topics:

- Relational Data Model vs. Document Data Model
- XML Schemas
- High Performance Transactional Database
- Rich Search Features
- Text and Structure Indexes
- Semantics Support
- Binary Document Support
- MarkLogic APIs and Communication Protocols
- High Performance
- Clustered
- Cloud Capable

1.1 Relational Data Model vs. Document Data Model

The main characteristic of the relational data model used in a typical relational database managment system (RDBMS) is the need for a schema to define what data is to be stored, how the data is to be categorized into tables and the relationship between those tables in the database. In MarkLogic, documents are the data and there is no need for a schema. However, in MarkLogic, you can define a document data model (the basic structure of the XML or JSON in your

documents) and configure indexes that leverage the data model to improve the speed and precision of your search queries. Your document data model can be as "loose" or as "strict" as you like. Unlike an RDBMS schema, one of the major benefits of a document data model is that it can be created after the documents are loaded into MarkLogic and then later refined as your needs evolve.

Another characteristic of the relational data model is normalization, meaning that related data is distributed across separate, interrelated tables. Normalization has the benefit of minimizing duplicate data, but the downside is that it is necessary for the database to join the data in the related tables together in response to certain queries. Such joins are time-consuming, so most modern databases have "denormalization" techniques that minimize the need to repeatedly do joins for often-used queries. A materialized view is an example of denormalized data. Though denormalizing the data is preferable to doing repeated joins on the data, there is still a performance cost.

In a document data model, related data is typically all contained in the same document, so the data is already denormalized. Financial contracts, medical records, legal filings, presentations, blogs, tweets, press releases, user manuals, books, articles, web pages, metadata, sparse data, message traffic, sensor data, shipping manifests, itineraries, contracts, emails, and so on are all naturally modeled as documents. In some cases the data might start formatted as XML or JSON documents (for example, Microsoft Office 2007 documents or financial products written in FpML), but if not, documents can be transformed to XML or JSON during ingestion. Relational databases, in contrast, with their table-centric data models, cannot represent data like this as naturally and so either have to spread the data out across many tables (adding complexity and hurting performance) or keep this data as unindexed BLOBs or CLOBs.

1.2 XML Schemas

For those familiar with RDBMS schemas, the term "schema" can be confusing when discussing XML content. Though the data in MarkLogic Server does not have to conform to a schema as understood in the RDBMS context, you can define "schemas" for your XML content to describe a content model for a class of documents. This model defines the expected elements and attributes within XML documents and their permitted structure. Documents that do not conform to the schema are considered invalid and are rejected during ingestion into MarkLogic Server. MarkLogic does not support schemas for JSON content.

Schemas are useful in situations where certain documents must contain specific content and conform to a particular structure. For example, the schema for a company's employee directory might describe a valid `<Employee>` as consisting of an `<EmployeeID>` element, followed by a `<FirstName>`, `<LastName>`, and `<Position>` element. The content of an `<EmployeeID>` might have a datatype constraint that it consist of a sequence of exactly five digits.

XML schemas are loaded into the Schemas database or placed in the `Config` directory. You can then configure a group of App Servers or a specific App Server to use these schemas. Each XML schema is identified by a namespace. To use a schema, you declare its associated namespace in your documents and code.

For more detail on custom XML schemas, see Understanding and Defining Schemas in the *Administrator's Guide* and Loading Schemas in the *Application Developer's Guide*. MarkLogic also supports "virtual" SQL schemas that provide the naming context for SQL views, as described in the *SQL Data Modeling Guide*.

1.3 High Performance Transactional Database

MarkLogic stores documents within its own transactional repository that was purpose-built with a focus on maximum performance and data integrity. Because the MarkLogic database is transactional, you can insert or update a set of documents as an atomic unit and have the very next query able to see those changes with zero latency. MarkLogic supports the full set of ACID properties:

- Atomicity — a set of changes either takes place as a whole or doesn't take place at all.
- Consistency — system rules are enforced, such as that no two documents can have the same identifier.
- Isolation — uncompleted transactions are not otherwise visible.
- Durability — once a commit is made the changes are not lost.

ACID transactions are considered commonplace for relational databases but they are not common for document-centric databases and search-style queries.

1.4 Rich Search Features

MarkLogic Server search supports a wide range of full-text features. These features include phrase search, boolean search, proximity, stemming, tokenization, decompounding, wildcarded searches, punctuation-sensitive search, diacritic-sensitive/insensitive searches, case-sensitive/insensitive searches, spelling correction functions, thesaurus functions, geospatial searches, advanced language and collation support, document quality settings, numerous relevance algorithms, individual term weighting, topic clustering, faceted navigation, custom-indexed fields, and much more. These features are all designed to build off of each other and work together in an extensible and flexible way.

Search is covered in more detail in "Searching in MarkLogic Server" on page 65.

1.5 Text and Structure Indexes

MarkLogic indexes both text and structure, and the two can be queried together efficiently. For example, consider the challenge of querying and analyzing intercepted message traffic for threat analysis:

> Find all messages sent by IP 74.125.19.103 between April 11th and April 13th where the message contains both "wedding cake" and "empire state building" (case and punctuation insensitive) where the phrases have to be within 15 words of each other but the message cannot contain another key phrase such as "presents" (stemmed so "present" matches also). Exclude any message that has a subject equal to "Congratulations." Also exclude

> any message where the matching phrases were found within a quote block in the email. Then, for matching messages, return the most frequent senders and recipients.

By using XML and/or JSON documents to represent each message and the structure-aware indexing to understand what is an IP, what is a date, what is a subject, and which text is quoted and which is not, a query like this is actually easy to write and highly performant in MarkLogic. Or consider some other examples.

Find hidden financial exposure:

> Extract footnotes from any XBRL financial filing where the footnote contains "threat" and is found within the balance sheet section.

Review images:

> Extract all large-format images from the 10 research articles most relevant to the phrase ‘herniated disc.” Relevance should be weighted so that phrase appearance in a title is 5 times more relevant than body text, and appearance in an abstract is 2 times more relevant.

Find a person's phone number from their emails:

> From a large corpus of emails find those sent by a particular user, sort them reverse chronological, and locate the last email they sent which had a footer block containing a phone number. Return the phone number.

Indexing is covered in more detail in “Indexing in MarkLogic” on page 19.

1.6 Semantics Support

Semantic technologies refer to a family of W3C standards that allow the exchange of data (and information about relationships in data) in machine readable form, whether it resides on the Web or within organizations. Semantics requires a flexible data model (Resource Description Format or RDF), a query language (SPARQL), a language to manage RDF data (SPARQL Update), and a common markup language for the data (such as Turtle, RDFa, or N-Triples).

MarkLogic Server supports searching, storing, and managing semantic data in the form of RDF triples, both as standalone graphs and as triples embedded in documents. With MarkLogic you can use native SPARQL and SPARQL Update, or use JavaScript, XQuery, and REST, to store, search, and manage RDF triples.

For details about MarkLogic semantics, see Introduction to Semantics in MarkLogic in the *Semantics Developer’s Guide*. For more information on semantic queries, see “Semantic Searches” on page 69. For details on the triple index, see “Triple Index” on page 37. For information on SPARQL Update, see “Managing Semantic Triples” on page 63.

1.7 Binary Document Support

Binary documents require special consideration in MarkLogic Server because binary content is often much larger than text, JSON, or XML content. MarkLogic Server provides a number of mechanisms that enable you optimize the management of binary data. You can set thresholds that determine where a binary document is stored, depending on its size. For example, smaller binary documents can be efficiently stored in the same manner as non-binary documents. Larger binaries can be stored in a special Large Data Directory, which may use a different class of disk than your regular filesystem. The largest binary documents are typically stored outside MarkLogic Server in external storage. When binary documents are stored externally, pointers are maintained by MarkLogic to the documents so it is easy to manage security on them within MarkLogic.

When a small binary is cached by MarkLogic Server, the entire document is cached in memory. When a large or external binary is cached, the content is fetched into a compressed tree cache either fully or in chunks, as needed.

For details on how to manage binary documents on MarkLogic Server, see Working With Binary Documents in the *Application Developer's Guide*.

1.8 MarkLogic APIs and Communication Protocols

MarkLogic supports the following application programming languages:

- XQuery (native, server-side support)
- JavaScript (native, server-side support)
- XSLT (native, server-side support)
- Java
- Node.js
- C#
- SQL (native, server-side support)
- SPARQL (native, server-side support)
- REST interfaces

XQuery, JavaScript, XSLT, SQL, and SPARQL are native langauges. Appplication code written in a native lanuage runs directly in the database. All of the other supported languages are abstractions that are implemented on top of a native language. Applications written in a native language avoid expensive marshalling and unmarshalling across processes and enable the query planner to aggressively optimize evaluation of complex queries. For details, see "Application Development on MarkLogic Server" on page 89.

MarkLogic operates as a single multi-threaded process per host. It opens various socket ports for external communication. When configuring new socket ports for your application to use, you can choose between three distinct protocols:

HTTP and HTTPS Web Protocols

MarkLogic natively communicates via HTTP and HTTPS. Incoming web calls can run JavaScript, XQuery, XDBC, or XSLT scripts the same way other servers invoke PHP, JSP, or ASP.NET scripts. These scripts can accept incoming data, perform updates, and generate output. Using these scripts you can write full web applications or RESTful web service endpoints, with no need for a front-end layer.

XDBC Protocol

XDBC enables programmatic access to MarkLogic from other language contexts, similar to what JDBC and ODBC provide for relational databases. MarkLogic officially supports a Java client libraries, named XCC. There are open source libraries in other languages. XDBC and the XCC client libraries make it easy to integrate MarkLogic into an existing application stack.

ODBC Protocol

ODBC enables SQL access to MarkLogic Server. An ODBC server accepts SQL queries from a PostgreSQL front end and returns the relational-style data needed by Business Intelligence (BI) tools.

WebDAV File Protocol

WebDAV is a protocol that lets a MarkLogic repository look like a filesystem to WebDAV clients, of which there are many including built-in clients in most operating systems. With a WebDAV mount point you can drag-and-drop files in and out of MarkLogic as if it were a network filesystem. This can be useful for small projects; large projects usually create an ingestion pipeline and send data over XDBC or HTTP.

1.9 High Performance

MarkLogic is designed to maximize speed and scale. Many MarkLogic applications compose advanced queries across terabytes of data that make up many millions of documents and return answers in less than a second. The largest live deployments now exceed 200 terabytes and a billion documents. The largest projects now under development will exceed a petabyte.

For more detail on performance, see the *Query Performance and Tuning Guide*.

1.10 Clustered

To achieve speed and scale beyond the capabilities of one server, MarkLogic clusters across commodity hardware connected on a LAN. A commodity server might be a box with 4 or 8 cores, 64 or 128 gigabytes of RAM or more, and either a large local disk or access to a SAN. On a box such as this a rule of thumb is you can store roughly 1 terabyte of data, sometimes more and sometimes less, depending on your use case.

MarkLogic hosts are typically configured to specialize in one of two operations. Some hosts (Data Managers, or D-nodes) manage a subset of data. Other hosts (Evaluators, or E-nodes) handle incoming user queries and internally federate across the D-nodes to access the data. A load balancer spreads queries across E-nodes. As you load more data, you add more D-nodes. As your user load increases, you add more E-nodes. Note that in some cluster architecture designs, some hosts may act as both a D-node and an E-node. In a single-host environment that is always the case.

Clustering enables high availability. In the event an E-node should fail, there is no host-specific state to lose, just the in-process requests (that can be retried), and the load balancer can route traffic to the remaining E-nodes. Should a D-node fail, that subset of the data needs to be brought online by another D-node. You can do this by using either a clustered filesystem (allowing another D-node to directly access the failed D-node's storage and replay its journals) or intra-cluster data replication (replicating updates across multiple D-node disks, providing in essence a live backup).

For more detail on clustering, see “Clustering and Caching” on page 81.

1.11 Cloud Capable

MarkLogic clusters can be deployed on Amazon Elastic Compute Cloud (EC2). An EC2 deployment of MarkLogic enables you to quickly get started with any sized cluster (or clusters) with a minimum upfront investment. You can then scale your clusters up or down, as your needs evolve.

In the EC2 environment, a MarkLogic Server is deployed as an EC2 instance. MarkLogic instances of various sizes and capabilities, as well as their related resources, can be easily created by using the Cloud Formation templates available from http://developer.marklogic.com/products/aws. EC2 provides elastic load balancers (ELBs) to automatically distribute and balance application traffic among multiple EC2 instances of MarkLogic, along with a wide range of tools and services to manage MarkLogic clusters in the EC2 environment.

In EC2, data centers are physically distributed into regions and zones. Each region is a separate geographic area, such as US west and US east. Each region contains multiple zones, which are separate data centers in the same geological area that are connected through low-latency links. To ensure high availability in the EC2 environment, you can place D-Nodes in different availability

zones in the same region and configure them for local-disk failover to ensure that each transaction is written to one or more replicas. For optimum availability, D-Nodes and E-Nodes can be split evenly between two availability zones. For disaster recovery, you can place D-Nodes in different regions and use database replication between the D-Nodes in each region.

For more information on MarkLogic in EC2, see the *MarkLogic Server on Amazon EC2 Guide*. For more information on local-disk failover and database replication, see "Failover and Database Replication" on page 104.

For more detail on deploying MarkLogic on EC2, see the *MarkLogic Server on Amazon Web Services (AWS) Guide*.

2.0 How is MarkLogic Server Used?

This chapter includes some customer stories. These stores are based on actual customer use-cases, though the customer names are fictious and some of the details of their internal operations have been changed.

MarkLogic Server currently operates in a variety of industries. Though the data stored in and extracted from MarkLogic is different in each type of industry, many customers have similar data-managment challenges.

Common themes include:

- Rapid application develpment and deployment
- Ability to store heterogenous data from multiple sources in a single repository and make it immediately available for search
- Accurate and efficient search
- Enterprise-grade features
- Low cost

The topics in this chapter are:

- Publishing/Media Industry
- Government / Public Sector
- Financial Services Industry
- Healthcare Industry
- Other Industries

2.1 Publishing/Media Industry

BIG Publishing receives data feeds from publishers, wholesalers, and distributors and sells its information in data feeds, web services, and websites, as well as through other custom solutions. Demand for the vast amount of information housed in the company's database was high and the company's search solution working with a conventional relational database were not effectively meeting that demand. The company recognized that a new search solution was necessary to help customers retrieve relevant content from its enormous database.

The database had to handle 600,000 to 1 million updates a day while it is being searched and while new content is being loaded. The company was typically six to eight days behind from when a particular document would come in to when it would be available to its customers.

MarkLogic combines full-text search with the W3C-standard XQuery language. The MarkLogic platform can concurrently load, query, manipulate and render content. When content is loaded into MarkLogic, it is automatically converted into XML and indexed, so it is immediately available for search. Employing MarkLogic enabled the company to improve its search capabilities through a combination of XML element query, XML proximity search, and full-text search. MarkLogic's XQuery interface searches the content and the structure of the XML data, making that XML content more easily accessible. It took only about 4 to 5 months for the company to develop the solution and implement it.

The company discovered that the way in which MarkLogic stores data makes it easier for them to make changes in document structure and add new content when desired. With the old relational database and search tools, it was very difficult to add different types of content. Doing so, used to require them to rebuild the whole database and that would take 3 to 4 weeks. With MarkLogic, they can now restructure documents and drop in new document types very quickly.

Another key benefit is the cost savings the company has realized as a result of the initiative. The company needed a full-time employee on staff to manage their old infrastructure. Now, the company has an employee who spends one-quarter of his time managing the MarkLogic infrastructure. The company saves on the infrastructure side internally and their customers get the content more quickly.

2.2 Government / Public Sector

The Quakezone County government wants to make it easier for county employees, developers and residents to access real-time information about zoning changes, county land ordinances, and property history. The county has volumes of data in disparate systems and in different formats and need to provide more efficient access to the data, while maintaining the integrity of the record data. They need a solution that fits within county IT infrastructure, that can be quickly implemented, and that keeps the hardware and licensing costs both low and predictable.

The solution is to migrate all of existing PDF, Word, or CAD files from the county's legacy systems into MarkLogic, which provides a secure repository for all of the record data, easy-to-use search and the ability to display the results in a geospatial manner on a map.

By having their data centralized in MarkLogic, county clerks can access all of the data they need from one central repository. MarkLogic enables the county to transform and enrich the data, as well as to view and correlate it in multiple ways by multiple applications. Tasks that once took days or weeks to accomplish can now be completed in seconds or minutes. Additionally, Quakezone County can make this information even more accessible to its constituents by deploying a public-facing web portal with powerful search capabilities on top of the same central MarkLogic repository.

2.3 Financial Services Industry

TimeTrader Services Inc. provides financial research to customers on a subscription basis. Because every second counts in the fast-paced world of stock trading, the firm needs to deliver new research to its subscribers as quickly as possible to help them make better decisions about their trades.

Unfortunately, these efforts were hampered by the firm's legacy infrastructure. Because of shortcomings with the current tool they were not able to easily respond to new requirements or to fully leverage the documents that were being created. Additionally they could not meet their goals for delivering alerts in a timely fashion.

TimeTrader Services replaced its legacy system with a MarkLogic Server. Now the firm can take full advantage of the research information. The solution drastically reduces alert latency and delivers information to the customer's portal and email. In addition, the ability to create triple indexes and do semantic searches has vastly improved the user experience.

Thanks to the new system, TimeTrader Services delivers timely research to 80,000 users worldwide, improving customer satisfaction and competitive advantage. By alerting customers to the availability of critical new research more quickly, financial traders gain a definite edge in the office and on the trading floor.

2.4 Healthcare Industry

HealthSmart is a Health Information Exchange (HIE) that is looking into using new technologies as a differentiating factor for success. They seek a technology advantage to solve issues around managing and gaining maximum use of a large volume of complex, varied, and constantly changing data. The number of requests for patient data and the sheer volume of that data are growing exponentially, in communities large and small, whether serving an integrated delivery network (IDN), hospital, or a large physician practice.

These challenges include aggregating diverse information types, finding specific information in a large dataset, complying with changing formats and standards, adding new sources, and maintaining high performance and security, all while keeping costs under control.

HIE solutions that solve big data challenges must meet the strict requirements of hospitals, IDNs and communities to lead to an effective and successful exchange. To develop a successful HIE, communities need to embrace technologies that help with key requirements around several important characteristics:

- Performance – The system should be able to provide real time results. As a result, doctors can get critical test results without delay.
- Scalability – As data volumes grow, the system should be able to scale quickly on commodity hardware with no loss in performance. Hospitals can then easily accommodate data growth in systems critical to patient care.

- Services – An effective exchange should have the option of rich services such as search, reporting, and analytics. Doctors will be notified if a new flu trend has developed in the past week in a certain geographic location.
- Systems – both of which will impact the quality, risks and costs of care.
- Interoperability – It should be easy to integrate different systems from other members of the community into the exchange through a common application programming interface (API). Community members can leverage the exchange sooner to share data and improve patient care.
- Security – Only authenticated and authorized users will be allowed to view private data. Community members want to ensure patient privacy and also comply with regulations such as HIPAA.
- Time to delivery – Implementation should be measured in weeks, not months or years and overhead should remain low. Healthcare can save millions in maintenance and hardware with low overhead, next generation technology.
- Total cost of ownership – The system must make economic sense. It should help to cut healthcare costs, not increase them.

By leveraging MarkLogic, HealthSmart gained a significant performance boost that reduced queries and transformations to sub-second response times, which was critical for accomplishing their mission. In addition, MarkLogic's flexible data model enabled integration of new sources of data in a matter of days instead of weeks or months.

MarkLogic can efficiently manage billions of documents in the hundreds of terabytes range. It offers high speed indexes and optimizations on modern hardware to deliver sub-second responses to users. HealthSmart leverages the performance and scalability benefits of MarkLogic not only to quickly deliver information to users, but also to grow the deployment as the load grows.

Consequently, HealthSmart provides the wide range of functionality required in information heavy healthcare environments today. Features such as full-text search, granular access to information, dynamic transformation capabilities, and a web services framework all lower the development overhead typically required with other technologies. This makes HealthSmart a feature-rich HIE solution that does not make the tradeoffs that other solutions make.

HealthSmart is particularly advantageous with regard to interoperability. Since MarkLogic is based on XML, and XML is widely used in healthcare systems, the technology fit is ideal. The ability to load information "as is" dramatically lowers the barrier of adding new systems in an HIE community. This, in part, enabled HealthSmart to build the patient registry component in a mere two months, far faster than any other vendor. HealthSmart's dynamic transformation capabilities facilitate compliance with transport standards and regulatory legacy interfaces. And MarkLogic's services-oriented architecture (SOA) and its support for building REST endpoints enable an easy and standardized way to access information.

As an enterprise-class database, MarkLogic supports the security controls needed to keep sensitive information private. MarkLogic is used in top secret installations in the Federal Government, and provides access controls to ensure classified data is only accessible to authorized personnel.

Finally, HealthSmart and MarkLogic help to significantly lower time-to-delivery and the total cost of ownership. The lower overhead in adding new community members directly leads to quick adoption and cost savings. MarkLogic's optimization on modern, commodity hardware enables exchanges to benefit from lower cost hardware systems. High performance enables systems with fewer hardware servers, and scalability allows growth by simply adding more commodity servers, rather than replacing existing servers with larger, high cost servers.

2.5 Other Industries

Other industries benefiting from deployments of MarkLogic Server include:

- Legal — Laws, regional codes, public records, case files, and so on.
- Government Intelligence — Identify patterns and discover connections from massive amounts of heterogeneous data.
- Airlines — Flight manuals, service records, customer profiles, and so on.
- Insurance — Claims data, actuary data, regulatory data, and so on.
- Education — Student records, test assembly, online instructional material, and so on.

For more information on customers and their uses of MarkLogic, see http://www.marklogic.com/solutions/.

3.0 Indexing in MarkLogic

MarkLogic makes use of multiple types of indexes to resolve queries. As described in "Overview of MarkLogic Server" on page 5, this combination of indexes is referred to as the Universal Index. This chapter describes the indexing model utilized by MarkLogic Server.

The main topics are:

- The Universal Index
- Other Types of Indexes
- Index Size
- Fields
- Reindexing
- Relevance
- Indexing Document Metadata
- Fragmentation of XML Documents

3.1 The Universal Index

The universal index indexes the XML elements and JSON properties in the loaded documents. By default, MarkLogic Server builds a set of indexes that is designed to yield the fast query performance in general usage scenarios. You can configure MarkLogic to index additional data to speed up specific types of searches. In general, the cost of supporting additional indexes is increased disk space and document load times. As more and more indexes are maintained, search performance increases and document load speed decreases.

MarkLogic indexes the XML or JSON structure it sees during ingestion, whatever that structure might be. It does not have to be told what schema to expect, any more than a search engine has to be told what words exist in the dictionary. MarkLogic sees the challenge of querying for structure or for text as fundamentally the same. At the index level, matching the XPath expression `/a/b/c` can be performed similarly to matching the phrase `"a b c"`.

The MarkLogic universal index captures multiple data elements, including full text, markup structure, and structured data. The universal index is optimized to allow text, structure and value searches to be combined into a single process for extremely high performance.

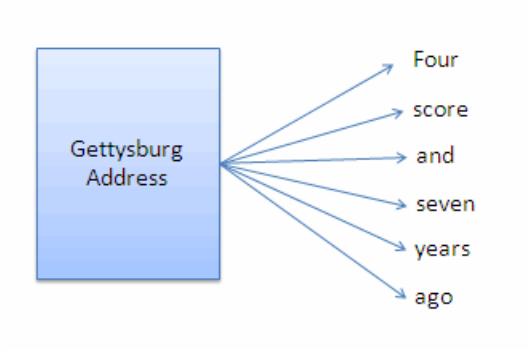
This section describes the types of indexes used by MarkLogic in the Universal Index:

- Word Indexing
- Phrase Indexing
- Relationship Indexing
- Value Indexing
- Word and Phrase Indexing

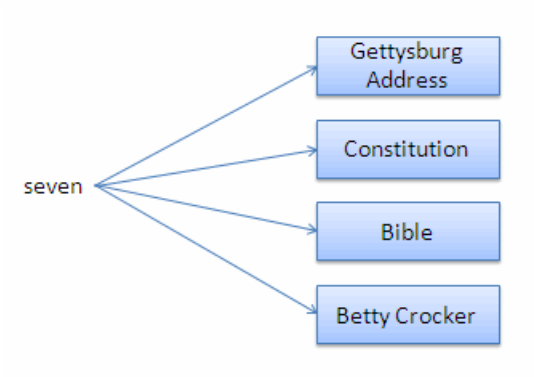
3.1.1 Word Indexing

By default, a MarkLogic database is enabled for fast element word searches. This feature enables MarkLogic Server to resolve word queries by means of an inverted index, which is a list of all of the words in all the documents in the database and, for each word, a list of which documents have that word. An inverted index is called "inverted" because it inverts the relationship of words to documents.

For example, documents contain a series of words in a particular order. The document-words relationship can be described by the following picture:



An inverted index inverts the document-words relationship described above into a word-documents relationship. For example, you can use an inverted index to identify which documents contain the word "seven," as shown in the following picture:



Each entry in the inverted index is called a *term list*. A term can be a word or a phrase consisting of multiple words. A database can be configured to index only the shortest stem of each term (run), all stems of each term (run, runs, ran, running), or all stems of each term and smaller component terms of large compound terms. Each successive level of stemming improves recall of word searches, but also causes slower document loads and larger database files.

No matter which word is used in a query, MarkLogic can quickly locate the associated documents by finding the right term in the term list. This is how MarkLogic resolves simple word queries.

For queries that ask for documents that contain two different words, MarkLogic simply searches the term list to find all document ids with the first word, then all document ids with the second, and then intersects the lists. By sorting the term lists in advance and storing them in a "skip list," MarkLogic Server does this intersection efficiently and in logarithmic time. This is how MarkLogic resolves simple boolean queries.

For negative queries that ask for documents that contain one word but exclude those documents with another word, you can use the indexes to find all document ids with the first word, all document ids with the second word, and do a list subtraction.

3.1.2 Phrase Indexing

There are multiple ways MarkLogic Server can be configured to handle a query to locate all of the documents that contain two-word phrase:

- Use term lists to find documents that contain both words and then look inside the candidate documents to determine if the two words appear together in a phrase.
- Enable word positions to include position information for each term in the term list. That way you know at what location in the document each word appears. By looking for term

hits with adjoining positions, you can find the answer using just indexes. This avoids having to look inside any documents.

Enable fast phrase searches to include two-word phrases as terms to the term list. When a user queries include a two-word phrase MarkLogic Server can find the term list for that two-word phrase and immediately know which documents contain that phrase without further processing. Because the fast phrase searches index doesn't expand the inverted index very much, it's a good choice when phrase searches are common.

For example, if a document contains “the quick brown fox,” the following terms will be stored in the term list:

- the quick
- quick brown
- brown fox

If a user queries for “the quick brown fox,” then the documents that include these three phrases will be returned. However, this does not necessarily mean these phrases appear in the specified order. For example, a document containing “The clown spotted the **quick brown** monkey and the slow **brown fox** in **the quick** clown car” would also be returned in the result set. However, if both “word positions” and “fast phrase searches” are enabled, only documents containing “the quick brown fox” would be returned.

The query results produced are the same regardless of whether “fast phrase searches” are enabled; it's just a matter of index choices and performance tradeoffs. After index resolution, MarkLogic optionally filters the candidate documents to check if they are actual matches.

The procedure for producing query results is as follows:

1. Look at the query and decide what indexes can help.

2. Use the indexes to narrow the results down to a set of candidate documents. The more index options enabled, the tighter the candidate result set.

3. Optionally filter the documents to confirm the match.

3.1.3 Relationship Indexing

The indexing techniques described so far are typical of most standard search engines. This section describes how MarkLogic goes beyond simple search to index document structure.

In additon to creating a term list for each word, MarkLogic creates a term list for each XML element or JSON property in documents. For example, a query to locate all of the documents that have a `<title>` element within them can be return the correct list of documents very rapidly.

A more complex query might be to find documents matching the XPath `/book/metadata/title` and for each return the title node. That is, we want documents having a `<book>` root element, with a `<metadata>` child, and a `<title>` subchild. With the simple element term list from above you can find documents having `<book>`, `<metadata>`, and `<title>` elements. However, the hierarchical relationship between the elements is unknown.

MarkLogic automatically indexes element relationships to keep track of parent-child element hierarchies. Without configuring any index settings, MarkLogic, in this example, indexes the `book/metadata` and `metadata/title` relationships to produce a smaller set of candidate documents. Now you can see how the XPath `/a/b/c` can be resolved very much like the phrase "a b c". The parent-child index lets you search against an XPath even when you don't know the path in advance.

Note that, even with the parent-child index, there's still the potential for documents to be in the candidate set that aren't an actual match. Knowing that somewhere inside a document there's a `<book>` parent of `<metadata>` and a `<metadata>` parent of `<title>` doesn't mean it's the same `<metadata>` between them. That's where filtering comes in: MarkLogic confirms each of the results by looking inside the document before the programmer sees them. While the document is open MarkLogic also extracts any nodes that match the query.

3.1.4 Value Indexing

MarkLogic automatically creates an index that maintains a term list for each XML element or JSON property value. To understand how element and property values are indexed in MarkLogic, consider a query that starts with the same XPath but requires the title equal the phrase “Good Will Hunting.” This could be expressed with the following XPath:

```
/book/metadata/title[. = "Good Will Hunting"]
```

Given the XPath shown above, MarkLogic Server searches the element-value index to locate a term list for documents having a `<title>` of “Good Will Hunting ” and immediately resolves which documents have that element value.

Element-value indexing is made efficient by hashing. Instead of storing the full element name and text, MarkLogic hashes the element name and the text down to a succinct integer and uses that as the term list lookup key. No matter how long the element name and text string, it is only a small entry in the index. MarkLogic Server uses hashes to store all term list keys, element-value or otherwise, for sake of efficiency. The element-value index has proven to be so efficient that it's always and automatically enabled within MarkLogic.

3.1.5 Word and Phrase Indexing

MarkLogic allows you to enable a “fast element word searches” index that maintains a term list for tracking element names along with the individual words within the element. For example, consider a query for a title containing a word or phrase, like “Good Will.” The element-value index above isn't any use because it only matches full values, like “Good Will Hunting.” When enabled, the “fast element word searches” index maintains term lists for tracking element names

along with the individual words within those elements. For example, the “fast element word searches” index adds a term list entry when it sees a `<title>` element with the word “Good,” another for `<title>` with the word “Will,” and another for `<title>` with the word “Hunting.” Using the “fast element word searches” index, MarkLogic Server can quickly locate which documents contain the words "Good" and "Will" under a `<title>` element to narrow the list of candidate documents.

The “fast element word searches” index alone does not know if the words “Good Will” are together in a phrase. To configure indexing to resolve at that level of granularity, MarkLogic allows you to enable the “fast element phrase searches” and “element word positions” index options. The “fast element phrase searches” option, when enabled, maintains a term list for every element and pair of words within the element. For example, it will have a term list for times when a `<title>` has the phrases "Good Will" and “Will Hunting” in its text. The “element word positions” maintains a term list for every element and its contained words, and tracks the positions of all the contained words. Either or both of these indexes can be used to ensure the words are together in a phrase under the `<title>` element. The “fast element phrase searches” and “element word positions” indexes are most beneficial should the phrase “Good Will” frequently appear in elements other than `<title>`.

Whatever indexing options you have enabled, MarkLogic will automatically make the most of them in resolving queries. If you're curious, you can use the `xdmp:plan` function to see the constraints for any particular XPath or `cts:search` expression.

3.2 Other Types of Indexes

MarkLogic Server makes use of other types of indexes that are not part of the Universal Index.

- Range Indexing
- Word Lexicons
- Reverse Indexing
- Triple Index

3.2.1 Range Indexing

The previously described indexes enable you to rapidly search the text, structure, and combinations of the text and structure within collections of XML and JSON documents. In some cases, however, documents can incorporate numeric, date or other typed information. Queries against these documents may include search conditions based on inequalities (for example, price < 100.00 or date ≥ 2007-01-01). Specifying range indexes for these elements and/or attributes will substantially accelerate the evaluation of these queries.

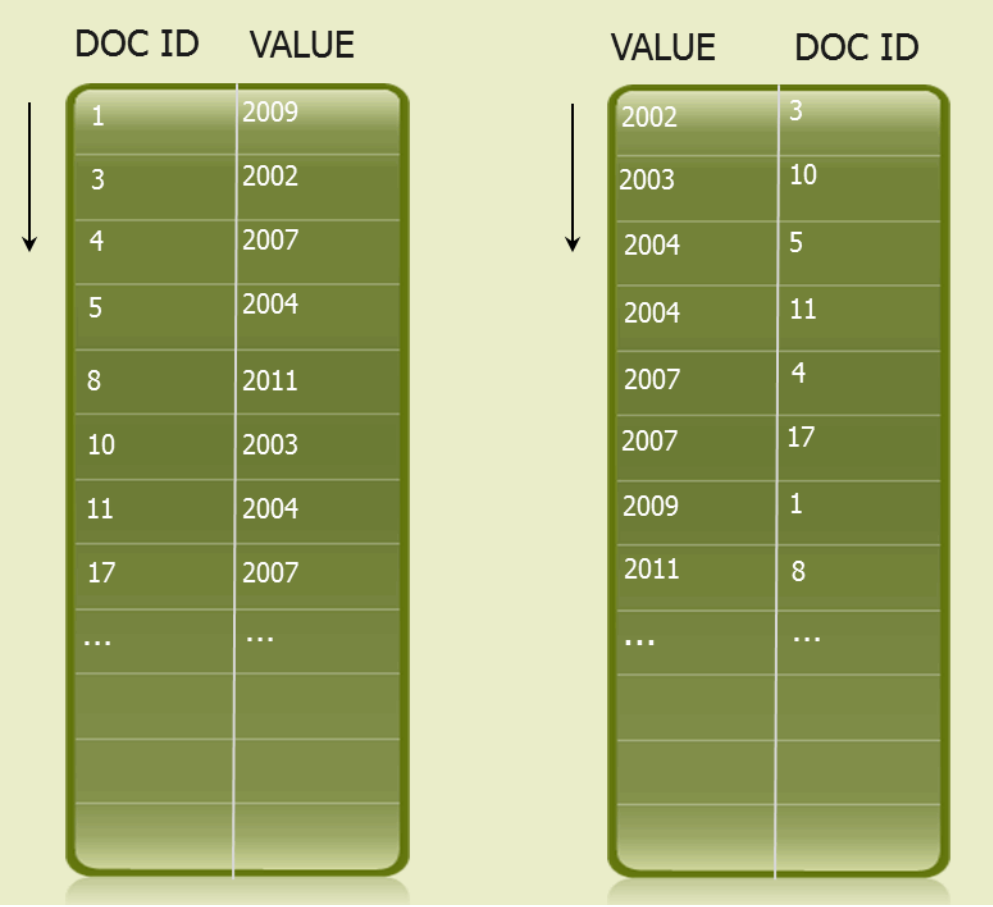
A range index enables you to do the following:

- Perform fast range queries. For example, you can provide a query constraint for documents having a date between two given endpoints.

- Quickly extract specific values from the entries in a result set. For example, you can get a distinct list of message senders from documents in a result set, as well as the frequency of how often each sender appears. These are often called facets and displayed with search results as an aid in search navigation.
- Perform optimized order by calculations. For example, you can sort a large set of product results by price.
- Perform efficient cross-document joins. For example, if you have a set of documents describing people and a set of documents describing works authored by those people, you can use range indexes to efficiently run queries looking for certain kinds of works authored by certain kinds of people.
- Quickly extract co-occurring values from the entries in a result set. For example, you can quickly get a report for which two entity values appear most often together in documents, without knowing either of the two entity values in advance.
- As described in “Bitemporal Documents” on page 62, bitemporal documents are associated with both a valid time that marks when a thing is known in the real world and a system time that marks when the thing is available for discovery in MarkLogic Server. The valid and system axis each make use of dateTime range indexes that define the start and end times. A temporal collection is a logical grouping of temporal documents that share the same axes with timestamps defined by the same range indexes.
- Create views on which SQL queries can be performed, as described in “SQL Support” on page 93.

To configure a range index, use the Admin Interface to provide the QName (the fully qualified XML or JSON name) for the element or attribute on which you want the range index to apply; a data type (int, date, string, etc); and (for strings) a collation, which is in essence a string sorting algorithm identified with a special URI. For each range index, MarkLogic creates data structures that make it easy to do two things: for any document get the document's range index value(s), or, for any range index value get the documents that have that value.

Conceptually, you can think of a range index as implemented by two data structures, both written to disk and then memory mapped for efficient access. One such data structure can be thought of as an array of document ids and values, sorted by document ids; the other as an array of values and document ids, sorted by values. It is not actually this simple or wasteful with memory and disk (in reality the values are only stored once), but it is a useful mental model.



Note: A “document id” is really a “fragment id” and is used here to avoid introducing a new concept and keep the discussion focused on range indexing. Fragments are discussed later in “Fragmentation of XML Documents” on page 43.

The examples in this section conceptualize a range index as lists of people and their birthdays. The people represent documents and their birthdays represent their associated values. Imagine this range index as being a list of birthdays mapped to people, sorted by birthday, and a list of people mapped to birthdays, sorted by person, as illustrated below.

Birthday (Value)	Person (Document)	Person (Value)	Birthday (Document)
6/16/1992	Lori Treger	Colleen Smithers	6/2/1978
3/4/1988	Mary Koo	Gordon Scott	2/2/1946
6/2/1978	Colleen Smithers	Jose Smith	9/15/1955
8/14/1972	Marty Frigger	Kim Oye	5/5/1965
4/21/1966	Kofu Kreen	Kofu Kreen	4/21/1966
5/5/1965	Kim Oye	Lori Treger	6/16/1992
3/25/1964	Travis Macy	Marty Frigger	8/14/1972
5/19/1963	Will Smith	Mary Koo	3/4/1988
5/15/1958	Nancy Wilson	Nancy Wilson	5/15/1958
9/15/1955	Jose Smith	Tom Jones	2/4/1953
2/4/1953	Tom Jones	Travis Macy	3/25/1964
12/2/1946	Gordon Scott	Will Smith	5/19/1963

3.2.1.1 Range Queries

To perform a fast range query, MarkLogic uses the “value to document id” lookup array. Because the lookup array is sorted by value, there's a specific subsequence in the array that holds the values between the two user-defined endpoints. Each of those values in the range has a document id associated with it, and the set of those document ids can be quickly gathered and used like a synthetic term list to limit the search result to documents having a matching value within the user's specified range.

For example, to find people with a birthday between January 1, 1960, and May 16, 1980, you would find the point in the date-sorted range index for the start date, then the end date. Every date in the array between those two endpoints is a birthday for someone, and it's easy to get the people's names because every birthday date has the person listed right next to it. If multiple people have the same birthday, you'll have multiple entries in the array with the same value but a different corresponding name. In MarkLogic, instead of people's names, the system tracks document ids.

Range queries can be combined with any other types of queries in MarkLogic. Say you want to limit results to those within a date range as above but also having a certain metadata tag. MarkLogic uses the range index to get the set of document ids in the range, uses a term list to get the set of document ids with the metadata tag, and intersects the sets of ids to determine the set of results matching both constraints. All indexes in MarkLogic are fully composable with each other.

3.2.1.2 Extracting Values

To quickly extract specific values from the documents in a result set, MarkLogic uses the data structure that maps document ids to values. After first using inverted indexes to resolve the document ids that match a query, MarkLogic then uses the "document id to value" lookup array to find the values associated with each document id, quickly and without touching the disk. It can also count how often each value appears.

The counts can be bucketed as well. With bucketing, instead of returning counts per value, you return counts falling within a range of values. You can, for example, get the counts per day, week, or month against source data listing specific dates. You specify the buckets as part of the XQuery call. Then, when walking down the range index, MarkLogic keeps count of how many entries occur in each bucket.

For example, you want to find the birthdays of all of the people who live in Seattle, you first use your inverted index to find people who live in Seattle. Then you use your name-to-birthday lookup array to find the birthdays of those people. You can count how many people have each birthday. If you want to group the birthdays by month, you can do that with simple comparisons.

3.2.1.3 Optimized "Order By"

Optimized order by calculations allow MarkLogic to quickly sort a large set of results against an element for which there's a range index. The XQuery syntax has to be of a particular type, such as:

```
(
for $result in cts:search(/some/path, "some terms")
order by $result/element-with-range-index
return $result
)[1 to 10]
```

To perform optimized order by calculations, MarkLogic again uses the "document id to value" lookup array. For any given result set, MarkLogic takes the index-determined document ids and feeds them to the range index, which provides fast access to the values on which the results should be sorted. Millions of result items can be sorted in a fraction of a second because the values to be sorted come out of the range index.

For example, you want to sort the Seattle people by age, finding the ten oldest or youngest. You'd limit your list first to people from Seattle, extract their birthdays, then sort by birthday and finally return the list of people in ascending or descending order as you wish.

Performance of range index operations depends mostly on the size of the result set — how many items have to be looked up. Performance varies a bit by data type but you can get roughly 10 million lookups per second per core. Integers are faster than floats, which are faster than strings, which are faster when using the simplistic Unicode Codepoint collation than when using a more advanced collation.

For more information on optimized order by expressions and the exact rules for applying them, see the *Query Performance and Tuning Guide*.

3.2.1.4 Using Range Indexes for Joins

Range indexes are also useful for cross-document joins. Here's the technique: In XQuery code, get a set of ids matching the first query, then feed that set into a second query as a constraint. The ability to use range indexes on both ends makes the work efficient.

Understanding this technique requires a code example. Imagine you have a set of tweets, and each tweet has a date, author id, text, etc. And you have a set of data about authors, with things like their author id, when they signed up, their real name, etc. You want to find authors who've been on Twitter for at least a year and who have mentioned a specific phrase, and return the tweets with that phrase. That requires joining between author data and the tweet data.

Here is example code:

```
let $author-ids := cts:element-values(
  xs:QName("author-id"), "", (),
  cts:and-query((
    cts:collection-query("authors"),
    cts:element-range-query(
      xs:QName("signup-date"), "<=",
                current-dateTime() - xdt:yearMonthDuration("P1Y")
    )
  ))
)
for $result in cts:search(/tweet,
  cts:and-query((
    cts:collection-query("tweets"),
    "quick brown fox",
    cts:element-attribute-range-query(
      xs:QName("tweet"), xs:QName("author-id"), "=", $author-ids
    )
  ))
)[1 to 10]
return string($result/body)
```

The first block of code finds all the author ids for people who've been on Twitter for at least a year. It uses the signup-date range index to resolve the `cts:elementrange-query` constraint and an `author-id` range index for the `cts:elementvalues` retrieval. This should quickly get us a long list of `$author-ids`.

The second block uses that set of `$author-ids` as a search constraint, combining it with the actual text constraint. Now, without the capabilities of a range index, MarkLogic would have to read a separate term list for every author id to find out the documents associated with that author, with a potential disk seek per author. With a range index, MarkLogic can map author ids to document ids using just in-memory lookups. This is often called a shotgun or or (for the more politically correct) a scatter query. For long lists it is vastly more efficient than looking up the individual term lists.

3.2.2 Word Lexicons

MarkLogic Server allows you to create lexicons, which are lists of unique words or values that enable you to quickly identify a word or value in the database and how many times it appears. You can alse define lexicons that allow quick access to the document and collection URIs in the database, and you can create word lexicons on named fields.

Value lexicons are created by configuring the range indexes described in “Range Indexing” on page 24. Word lexicons can be created for the entire database or for specific elements or attributes. Lexicons containing strings (a word lexicon or a value lexicon of type string) have a collation that determines how the strings are ordered in the list.

For example, you might want to create a search-suggest application that can quickly match a partial string typed into the search box with similar strings in the documents stored in the database and produce them to the user for selection. In this case, you could enable word lexicons on the database to create a lexicon of all of the unique words across all of the documents in the database. Your application can quickly scan the collated list of words and locate those that partially contain the same string entered in the search box.

> **Note:** On large databases, the performance of using a word lexicon for suggestions will probably be slower than using a value lexicon (range index). This can be very application specific, and in some cases the performance might be good, but in general, range indexes will perform much better than word lexicons with the `search:suggest` function.

Type of Lexicon	Description
Word Lexicon	Stores all of the unique words, either in a database, in an element defined by a QName, or in an attribute defined by a QName.
Value Lexicon	Stores all of the unique values for an element or an attribute defined by a QName (that is, the entire and exact contents of the specified element or attribute).
Value Co-occurrences Lexicon	Stores all of the pairs of values that appear in the same fragment.
Geospatial Lexicon	Returns geospatial values from the geospatial index.
Range Lexicon	Stores buckets of values that occur within a specified range of values.
URI Lexicon	Stores the URIs of the documents in a database.
Collection Lexicon	Stores the URIs of all collections in a database.

3.2.3 Reverse Indexing

All the indexing strategies described up to this point execute what you might call forward queries, where you start with a query and find the set of matching documents. A reverse query does the opposite: you start with a document and find all matching queries — the set of stored queries that if executed would match this document.

3.2.3.1 Reverse Query Constructor

Programmatically, you start by storing serialized representations of queries within MarkLogic. You can store them as simple documents or as elements within larger documents. For convenience, any `cts:query` object automatically serializes as XML when placed in an XML context. This XQuery:

```
<query>{
  cts:and-query((
    cts:word-query("dog"),
    cts:element-word-query(xs:QName("name"), "Champ"),
    cts:element-value-query(xs:QName("gender"), "female")
  ))
}</query>
```

Produces this XML:

```
<query>
  <cts:and-query xmlns:cts="http://marklogic.com/cts">
    <cts:word-query>
      <cts:text xml:lang="en">dog</cts:text>
    </cts:word-query>
    <cts:element-word-query>
      <cts:element>name</cts:element>
      <cts:text xml:lang="en">Champ</cts:text>
    </cts:element-word-query>
    <cts:element-value-query>
      <cts:element>gender</cts:element>
      <cts:text xml:lang="en">female</cts:text>
    </cts:element-value-query>
  </cts:and-query>
</query>
```

Assume you have a long list of documents like this, each with different internal XML that defines some `cts:query` constraints. For a given document `$doc` you could find the set of matching queries with this XQuery call:

```
cts:search(/query, cts:reverse-query($doc))
```

It returns all the `<query>` elements containing serialized queries which, if executed, would match the document `$doc`. The root element name can, of course, be anything.

MarkLogic executes reverse queries efficiently and at scale. Even with hundreds of millions of stored queries and thousands of documents loaded per second, you can run a reverse query on each incoming document without noticeable overhead. I'll explain how that works later, but first let me describe some situations where you'll find reverse queries helpful.

3.2.3.2 Reverse Query Use Cases

One common use case for reverse queries is alerting, where you want to notify an interested party whenever a new document appears that matches some specific criteria. For example, Congressional Quarterly uses MarkLogic reverse queries to support alerting. You can ask to be notified immediately anytime someone says a particular word or phrase in Congress. As fast as the transcripts can be added, the alerts can go out.

The alerting doesn't have to be only for simple queries of words or phrases. The match criteria can be any arbitrary cts:query construct — complete with booleans, structure-aware queries, proximity queries, range queries, and even geospatial queries. Do you want to be notified immediately when a company's XBRL filing contains something of interest? Alerting gives you that.

Without reverse query indexing, for each new document or set of documents you'd have to loop over all your queries to see which match. As the number of queries increases, this simplistic approach becomes increasingly inefficient.

You can also use reverse queries for rule-based classification. MarkLogic includes an SVM (support vector machine) classifier. Details on the SVM classifier are beyond the scope of this paper, but suffice to say it's based on document training sets and finding similarity between document term vectors. Reverse queries provide a rule-based alternative to training-based classifiers. You define each classification group as a cts:query. That query must be satisfied for membership. With reverse query, each new or modified document can be placed quickly into the right classification group or groups.

Perhaps the most interesting and mind-bending use case for reverse queries is for matchmaking. You can match for carpools (driver/rider), employment (job/resume), medication (patient/drug), search security (document/user), love (man/woman or a mixed pool), or even in battle (target/shooter). For matchmaking you represent each entity as a document. Within that document you define the facts about the entity itself and that entity's preferences about other documents that should match it, serialized as a cts:query. With a reverse query and a forward query used in combination, you can do an efficient bi-directional match, finding pairs of entities that match each other's criteria.

3.2.3.3 A Reverse Query Carpool Match

Let's use the carpool example to make the idea concrete. You have a driver, a nonsmoking woman driving from San Ramon to San Carlos, leaving at 8AM, who listens to rock, pop, and hip-hop, and wants $10 for gas. She requires a female passenger within five miles of her start and end points. You have a passenger, a woman who will pay up to $20. Starting at "3001 Summit View

Dr, San Ramon, CA 94582" and traveling to "400 Concourse Drive, Belmont, CA 94002". She requires a non-smoking car, and won't listen to country music. A matchmaking query can match these two women to each other, as well as any other matches across potentially millions of people, in sub-second time.

This XQuery code inserts the definition of the driver — her attributes and her preferences:

```
let $from := cts:point(37.751658,-121.898387) (: San Ramon :)
let $to := cts:point(37.507363, -122.247119) (: San Carlos :)
return xdmp:document-insert(
  "/driver.xml",
  <driver>
    <from>{$from}</from>
    <to>{$to}</to>
    <when>2010-01-20T08:00:00-08:00</when>
    <gender>female</gender>
    <smoke>no</smoke>
    <music>rock, pop, hip-hop</music>
    <cost>10</cost>
    <preferences>{
      cts:and-query((
        cts:element-value-query(xs:QName("gender"), "female"),
        cts:element-geospatial-query(xs:QName("from"),
          cts:circle(5, $from)),
        cts:element-geospatial-query(xs:QName("to"),
                                     cts:circle(5, $to))
      ))
    }</preferences>
  </driver>)
```

This insertion defines the passenger — her attributes and her preferences:

```
xdmp:document-insert(
  "/passenger.xml",
  <passenger>
    <from>37.739976,-121.915821</from>
    <to>37.53244,-122.270969</to>
    <gender>female</gender>
    <preferences>{
      cts:and-query((
        cts:not-query(cts:element-word-query(xs:QName("music"),
                                             "country")),
        cts:element-range-query(xs:QName("cost"), "<=", 20),
        cts:element-value-query(xs:QName("smoke"), "no"),
        cts:element-value-query(xs:QName("gender"), "female")
      ))
    }</preferences>
  </passenger>)
```

If you're the driver, you can run this query to find matching passengers:

```
let $me := doc("/driver.xml")/driver
for $match in cts:search(/passenger,
  cts:and-query((
    cts:query($me/preferences/*),
    cts:reverse-query($me)
  ))
)
return base-uri($match)
```

It searches across passengers requiring that your preferences match them, and also that their preferences match you. The combination of rules is defined by the cts:andquery. The first part constructs a live cts:query object from the serialized query held under your preferences element. The second part constructs the reverse query constraint. It passes $me as the source document, limiting the search to other documents having serialized queries that match $me.

If you're the passenger, this finds you drivers:

```
let $me := doc("/passenger.xml")/passenger
for $match in cts:search(/driver,
  cts:and-query((
    cts:query($me/preferences/element()),
    cts:reverse-query($me)
  ))
)
return base-uri($match)
```

Again, the preferences on both parties must match each other. Within MarkLogic even a complex query such as this (notice the use of negative queries, range queries, and geospatial queries, in addition to regular term queries) runs efficiently and at scale.

3.2.3.4 The Reverse Index

To resolve a reverse query efficiently, MarkLogic uses custom indexes and a two-phased evaluation. The first phase starts with the source document (with alerting it'd be the newly loaded document) and finds the set of serialized query documents having at least one query term constraint match found within the source document. The second phase examines each of these serialized query documents in turn and determines which in fact fully match the source document, based on all the other constraints present.

To support the first phase, MarkLogic maintains a custom index. In that index MarkLogic gathers the distinct set of leaf node query terms (that is, the non-compound query terms) across all the serialized cts:query documents. For each leaf node, MarkLogic maintains a set of document ids to nominate as a potential reverse query match when that term is present in a document, and another set of ids to nominate when the term is explicitly not present.

When running a reverse query and presented with a source document, MarkLogic gathers the set of terms present in that document. It compares the terms in the document with this pre-built reverse index. This produces a set of serialized query document ids, each of which holds a query with at least one term match in the source document. For a simple one-word query this produces the final answer, but for anything more complex, MarkLogic needs the second phase to check if the source document is an actual match against the full constraints of the complex query.

For the second phase MarkLogic maintains a custom directed acyclic graph (DAG). It's a tree with potentially overlapping branches and numerous roots. It has one root for every query document id. MarkLogic takes the set of nominated query document ids, and runs them through this DAG starting at the root node for the document id, checking downward if all the required constraints are true, short-circuiting whenever possible. If all the constraints are satisfied, MarkLogic determines that the nominated query document id is in fact a reverse query match to the source document.

At this point, depending on the user's query, MarkLogic can return the results for processing as a final answer, or feed the document ids into a larger query context. In the matchmaker challenge, the cts:reverse-query() constraint represented just half of the cts:and-query(), and the results had to be intersected with the results of the forward query to find the bi-directional matches.

What if the serialized query contains position constraints, either through the use of a cts:near-query or a phrase that needs to use positions to be accurately resolved? MarkLogic takes positions into consideration while walking the DAG.

3.2.3.5 Range Queries in Reverse Indexes

What about range queries that happen to be used in reverse queries? They require special handling because with a range query there's no simple leaf node term. There's nothing to be found "present" or "absent" during the first phase of processing. What MarkLogic does is define subranges for lookup, based on the cutpoints used in the serialized range queries. Imagine you have three serialized queries, each with a different range constraint:

four.xml (doc id 4):

```
<four>{
  cts:and-query((
    cts:element-range-query(xs:QName("price"), ">=", 5),
    cts:element-range-query(xs:QName("price"), "<", 10)
  ))
}</four>
```

five.xml (doc id 5):

```
<five>{
  cts:and-query((
    cts:element-range-query(xs:QName("price"), ">=", 7),
    cts:element-range-query(xs:QName("price"), "<", 20)
  ))
}</five>
```

six.xml (doc id 6):

```
<six>{
  cts:element-range-query(xs:QName("price"), ">=", 15)
}</six>
```

For the above ranges you have cutpoints 5, 7, 10, 15, 20, and +Infinity. The range of values between neighboring cutpoints all have the same potential matching query document ids. Thus those ranges can be used like a leaf node for the first phase of processing:

Range	Present
5 to 7	4
7 to 10	4 5
10 to 15	5
15 to 20	5 6
20 to +Infinity	6

When given a source document with a price of 8, MarkLogic will nominate query document ids 4 and 5, because 8 is in the 7 to 10 subrange. With a price of 2, MarkLogic will nominate no documents (at least based on the price constraint). During the second phase, range queries act just as special leaf nodes on the DAG, able to resolve directly and with no need for the cutpoints.

Lastly, what about geospatial queries? MarkLogic uses the same cutpoint approach as for range queries, but in two dimensions. To support the first phase, MarkLogic generates a set of geographic bounding boxes, each with its own set of query document ids to nominate should the source document contain a point within that box. For the second phase, like with range queries, the geographic constraint acts as a special leaf node on the DAG, complete with precise geospatial comparisons.

Overall, the first phase quickly limits the universe of serialized queries to just those that have at least one match against the source document. The second phase checks each of those nominated documents to see if they're in fact a match, using a specialized data structure to allow for fast determination with maximum short-circuiting and expression reuse. Reverse query performance tends to be constant no matter how many serialized queries are considered, and linear with the number of actual matches discovered.

3.2.4 Triple Index

The triple index is used to index schema-valid `sem:triple` elements found anywhere in a document. The indexing of triples is performed when documents containing triples are ingested into MarkLogic or during a database reindex.

This section covers the following topics:

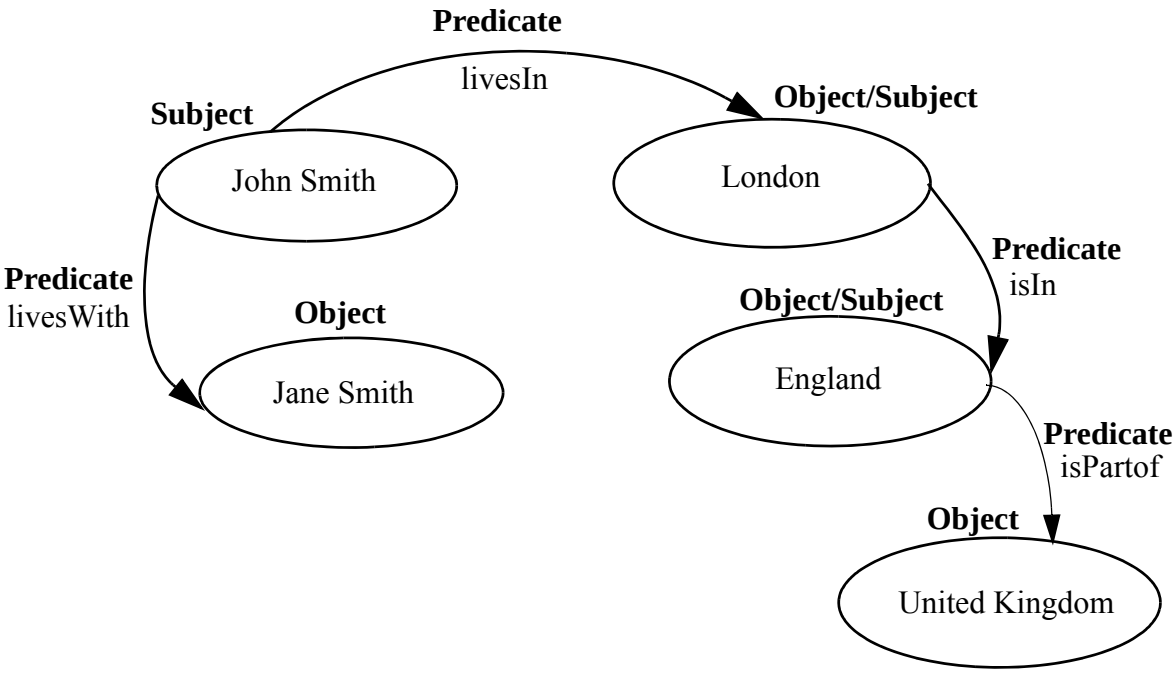
- Triple Index Basics
- Triple Data and Value Caches
- Triple Values and Type Information
- Triple Positions
- Index Files
- Permutations

3.2.4.1 Triple Index Basics

A triple index consists of the following three identifiers:

- Subject — Identifies a resource such as a person or an entity.
- Predicate — Identifies a property or characteristics of the subject or of the relationship between the subject and the object.
- Object — A node that identifies a property value, which in turn may be the subject in another triple.

For example:



The validity of `sem:triple` elements is determined by checking elements and attributes in the documents against the `sem:triple` schema (`/MarkLogic/Config/semantics.xsd`). If the `sem:triple` element is valid, an entry is created in the triple index, otherwise the element is skipped.

Unlike range indexes, triple indexes do not have to fit in memory, so there is little up-front memory allocation. For more about triple indexes, see Triple Index Overview in the *Semantics Developer's Guide*.

3.2.4.2 Triple Data and Value Caches

Internally, MarkLogic stores triples in two ways: *triple values* and *triple data*. The *triple values* are the individual values from every triple, including all typed literal, IRIs, and blank nodes. The *triple data* holds the triples in different permutations, along with a document ID and position information. The triple data refer to the triple values by ID, making for very efficient lookup. Triple data is stored compressed on disk, and triple values are stored in a separate compressed value store. Both the triple index and the value store are stored in compressed four-kilobyte (4k) blocks.

When triple data is needed (for example during a lookup), the relevant block is cached in either the triple cache or the triple value cache. Unlike other MarkLogic caches, the triple cache and triple value cache shrinks and grows, only taking up memory when it needs to add to the caches.

> **Note:** You can configure the size of the triple cache and the triple value cache for the host of your triple store.

3.2.4.3 Triple Values and Type Information

Values are stored in a separate value store on disk in “value equality” sorted order, so that in a given stand, the value ID order is equivalent to value equality order.

Strings in the values are stored in the range index string storage. Anything that is not relevant to value equality is removed from the stored values, for example timezone and derived type information.

Since type information is stored separately, triples can be returned directly from the triple index. This information is also used for RDF-specific "sameTerm" comparison required by SPARQL simple entailment.

3.2.4.4 Triple Positions

The triple positions index is used to accurately resolve queries that use `cts:triple-range-query` and the `item-frequency` option of `cts:triples`. The triple positions index is also used to accurately resolve searches that use the `cts:near-query` and `cts:element-query` constructors. The triple positions index stores locations within a fragment of the relative positions of triples within that fragment (typically, a fragment is a document). Enabling the triple positions index makes index sizes larger and will make document loads a little slower, but increases the accuracy of queries that need those positions.

For example:

```
xquery version "1.0-ml";

cts:search(doc(),
   cts:near-query((
   cts:triple-range-query(sem:iri("http://www.rdfabout.com/rdf/
usgov/sec/id/cik0001075285"), (), ()),

   cts:triple-range-query(sem:iri("http://www.rdfabout.com/rdf/
   usgov/sec/id/cik0001317036"), (), ())
   ),11), "unfiltered")
```

The `cts:near-query` returns a sequence of queries to match, where the matches occur within the specified distance from each other. The distance specified is in the number of words between any two matching queries.

The unfiltered search selects fragments from the indexes that are candidates to satisfy the specified `cts:query` and returns the document.

3.2.4.5 Index Files

To efficiently make use of memory, the index files for triple and value stores are directly mapped into memory. The type store is entirely mapped into memory.

Both the triple and value stores have index files consisting of 64-byte segments. The first segment in each is a header containing checksums, version number, and counts (of triples or values). This is followed by:

- Triples index - After the header segment, the triples index contains an index of the first two values and permutation of the first triple in each block, arranged into 64-byte segments. This is used to find the blocks needed to answer a given lookup based on values from the triple. Currently triples are not accessed by ordinal, so an ordinal index is not required.
- Values Index - After the header segment, the values index contains an index of the first value in each block, arranged into 64-byte segments. The values index is used to find the blocks needed to answer a given lookup based on value. This is followed by an index of the starting ordinal for each block, which is used to find the block needed to answer a given lookup based on a value ID.

The type store has an index file that stores the offset into the type data file for each stored type. This is also mapped into memory.

3.2.4.6 Permutations

The permutation enumeration details the role each value plays in the original triple. Three permutations are stored in order to provide access to different sort orders, and to be able to efficiently look up different parts of the triple. The permutations are acronyms made up from the initials of the three RDF elements (subject, predicate, and object), for example: { `SOP, PSO, OPS` }.

Use the `cts:triples` function to specify one of these sort orders in the options:

- `"order-pso"` - Returns results ordered by predicate, then subject, then object
- `"order-sop"` - Returns results ordered by subject, then object, then predicate
- `"order-ops"` - Returns results ordered by object, then predicate, then subject

3.3 Index Size

Many of the index options described in this chapter are either set automatically (with no option to disable) or are enabled by default. Out of the box, with the default set of indexes enabled, the on-disk size is often smaller than the size of the source XML. MarkLogic compresses the loaded XML and the indexes are often smaller than the space saved by the compression. With more indexes enabled, the index size can be two or three times the size of the XML source.

3.4 Fields

MarkLogic supports “fields” as a way to enable different indexing capabilities on different parts of the document. For example, a paper's title and abstract may need wildcard indexes, but the full text may not. For more information on fields, see Fields Database Settings in the *Administrator's Guide*.

3.5 Reindexing

Changes to the MarkLogic index settings require MarkLogic Server to reindex the database content. MarkLogic manages reindexing in the background while simultaneously handling queries and updates. If you change or add a new index setting, it will not be available to support requests until the reindexing has completely finished. If you remove an index setting, it stops being used right away.

You can manually initiate reindexing and monitor reindexing through the Admin Interface. You can also use the Admin Interface to set the "reindexer throttle" from 5 (most eager) to 1 (least eager), or set “reindexer enable” to disable to temporarily turn off reindexing when you don't want any background work happening.

3.6 Relevance

When performing a full text query it's not enough to just find documents matching the given constraint. The results have to be returned in relevance order. Relevance is a mathematical construct whose idea is simple. Documents with more matches are more relevant. Shorter documents that have matches are more relevant than longer documents having the same number of matches. If the search includes multiple words, some common and some rare, appearances of the rare terms are more relevant than appearances of the common terms. The math behind relevance can be very complex, but with MarkLogic you never have to do it yourself; it's done for you when you choose to order by relevance. You also get many control knobs to adjust relevance calculations when preparing and issuing a query.

3.7 Indexing Document Metadata

So far the discussions in this chapter have focused on how MarkLogic uses term lists for indexing text and structure. MarkLogic also makes use of term lists to index other things, such as collections, directories, and security rules. All of these indexes combined are referred to as the Universal Index.

This section covers the following topics:

- Collection Indexes
- Directory Indexes
- Security Indexes
- Properties Indexes

3.7.1 Collection Indexes

Collections in MarkLogic are a way to tag documents as belonging to a named group (a taxonomic designation, publication year, origin, whether the document is draft or published, etc). Each document can be tagged as belonging to any number of collections. A query constraint can limit the scope of a search to a certain collection or a set of collections. Collection constraints like that are implemented internally as just another term list to be intersected. There's a term list for each collection listing the documents within that collection. If you limit a query to a collection, it intersects that collection's term list with the other constraints in the query. If you limit a query to two collections, the two term lists are unioned into one before being intersected.

3.7.2 Directory Indexes

MarkLogic includes the notion of database “directories.” They're similar to collections but are hierarchical and non-overlapping. Directories inside MarkLogic behave a lot like filesystem directories: each contains an arbitrary number of documents as well as subdirectories. Queries often impose directory constraints, limiting the query to a specific directory or its subdirectories.

MarkLogic indexes directories a lot like collections. There's a term list for each directory listing the documents in that directory. There's also a term list listing the documents held in that directory or lower. That makes it a simple matter of term list intersection to limit a view based on directory paths.

3.7.3 Security Indexes

MarkLogic's security model leverages the intersecting term list system. Each query you perform has an implicit constraint based on your user's security settings. MarkLogic uses a role-based security model where each user is assigned any number of roles, and these roles have permissions and privileges. One permission concerns document visibility, who can see what. As part of each query that's issued, MarkLogic combines the user's explicit constraints with the implicit visibility constraints of the invoking user. If the user account has three roles, MarkLogic gathers the term lists for each role, and unions those together to create that user's universe of documents. It intersects this list with any ad hoc query the user runs, to make sure the results only display documents in that user's visibility domain. Implicitly and highly efficiently, every user's worldview is shaped based on their security settings, all using term lists.

3.7.4 Properties Indexes

Each document within MarkLogic has optional XML-based properties that are stored alongside the document in the database. Properties are a convenient place for holding metadata about binary or text documents which otherwise wouldn't have a place for an XML description. They're also useful for adding XML metadata to an XML document whose schema can't be altered. MarkLogic out of the box uses properties for tracking each document's last modified time.

Properties are represented as regular XML documents, held under the same URI (Uniform Resource Identifier, like a document name) as the document they describe but only available via specific calls. Properties are indexed and can be queried just like regular XML documents. If a query declares constraints on both the main document and its properties (like finding documents matching a query that were updated within the last hour), MarkLogic uses indexes to independently find the matching properties and main document, and does a hash join (based on the URI they share) to determine the final set of matches.

3.8 Fragmentation of XML Documents

So far in this chapter, each unit of content in a MarkLogic database has been described as a document, which is a bit of a simplification. MarkLogic actually indexes, retrieves, and stores *fragments*. The default fragment size is the document, and that's how most people leave it. However it is also possible to break XML documents into sub-document fragments through configured "fragment root" or "fragment parent" database settings controlled using the Admin Interface. Though rarely necessary, breaking an XML document into fragments can be helpful when handling a large documents where the unit of indexing, retrieval, storage, and relevance scoring should be something smaller than a document. You specify a QName (a technical term for an XML element name) as a fragment root, and the system automatically splits the document internally at that breakpoint. Or you can specify a QName as a fragment parent to make each of its child elements into a fragment root.

> **Note:** The maximum size of a fragment is 512 MB for 64-bit machines. Fragmentation can only be done on XML docuemnts. JSON documents are always stored as a single fragment.

For example, you might want to use fragmentation on a book, which may be too large to index, retrieve, and update as a single document (fragment). If you are doing chapter-based search and chapter-based display, it would be better to have `<chapter>` as the fragment root. With that change, each book document then becomes made up of a series of fragments, one fragment for the `<book>` root element holding the metadata about the book, and a series of distinct fragments for each `<chapter>`. The book still exists as a document, with a single URI, and it can be stored and retrieved as a single item, but internally it is broken into fragments.

Every fragment acts as its own self-contained unit. Each is the unit of indexing. A term list doesn't truly reference document ids; it references fragment ids. The filtering and retrieval process doesn't actually load documents; it loads fragments.

If no fragementation is enabled, each fragment in the database is a document. The noticeable difference between a fragmented document and a document split into individual documents is that a query pulling data from two fragments residing in the same document can perform slightly more efficiently than a query pulling data from two documents. See the documentation for the `cts:document-fragment-query` function for more details. Even with this advantage, fragmentation is not something to enable unless you are sure you need it.

4.0 Data Management

MarkLogic has the unique ability to bring multiple heterogenous data-sources (by structure and function) into a single platform architecture and allow for homogenous data access across disparate data-sources. Data sources do not have to be shredded or normalized to present a consistent view of the information. MarkLogic supports multiple mechanisms to present information to end-consumers in the language of their choice.

This chapter describes how MarkLogic manages data on disk and handles concurrent reads and writes. The main topics are:

- What's on Disk
- Ingesting Data
- Modifying Data
- Multi-Version Concurrency Control
- Point-in-time Queries
- Locking
- Updates
- Isolating an update
- Documents are Like Rows
- MarkLogic Data Loading Mechanisms
- Content Processing Framework (CPF)
- Organizing Documents
- Database Rebalancing
- Bitemporal Documents
- Managing Semantic Triples

4.1 What's on Disk

This section describes how data is managed on disks. The topics are:

- Databases, Forests, and Stands
- Tiered Storage
- Super Databases and Super Clusters
- Partitions, Partition Keys, and Partition Ranges

4.1.1 Databases, Forests, and Stands

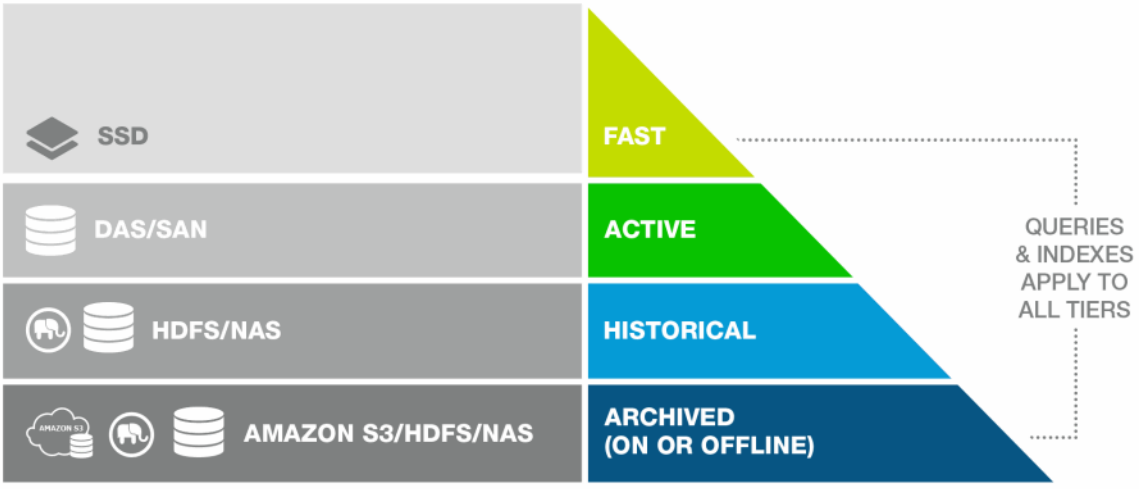
A database consists of one or more *forests*. A forest is a collection of documents that is implemented as a physical directory on disk. Each forest holds a set of documents and all of their indexes. A single machine may manage several forests, or in a cluster (when acting as an E-node) it might manage none. Forests can be queried in parallel, so placing more forests on a multi-core server can help with concurrency. A rule of thumb is to have one forest for every 2 cores on a box, with each forest holding millions or tens of millions of documents. In a clustered environment, you can have a set of servers, each managing their own set of forests, all unified into a single database.

Each forest holds zero or more *stands*. A stand (like a stand of trees) holds a subset of the forest data and exists as a physical subdirectory under the forest directory. Each stand contains the actual compressed document data (in TreeData) and indexes (in IndexData).

A forest might contain a single stand, but it is more common to have multiple stands because stands help MarkLogic ingest data more efficently and improve concurrency.

4.1.2 Tiered Storage

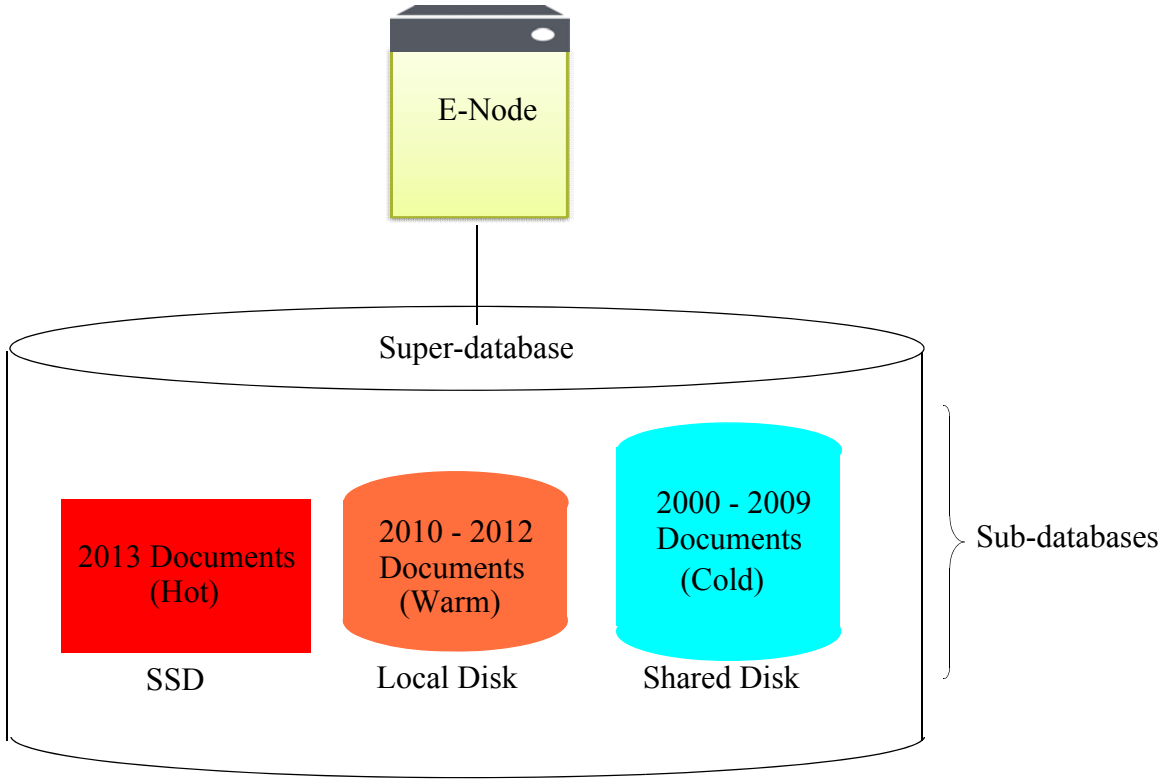
MarkLogic allows you to manage your data at different *tiers* of storage and computation environments, with the top-most tier providing the fastest access to your most critical data and the lowest tier providing the slowest access to your least critical data. Infrastructures, such as Hadoop and public clouds, make it economically feasible to scale storage to accommodate massive amounts of data in the lower tiers. Segregating data among different storage tiers allows you to optimize trade-offs among cost, performance, availability, and flexibility.



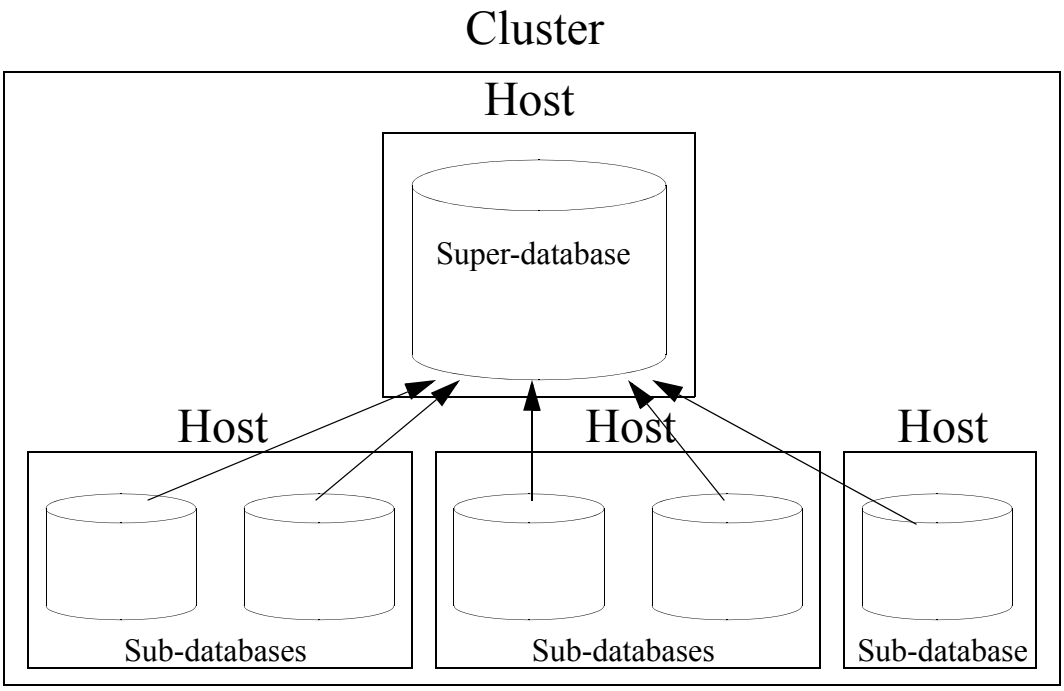
For more detail on tiered storage, see the Tiered Storage chapter in the *Administrator's Guide*.

4.1.3 Super Databases and Super Clusters

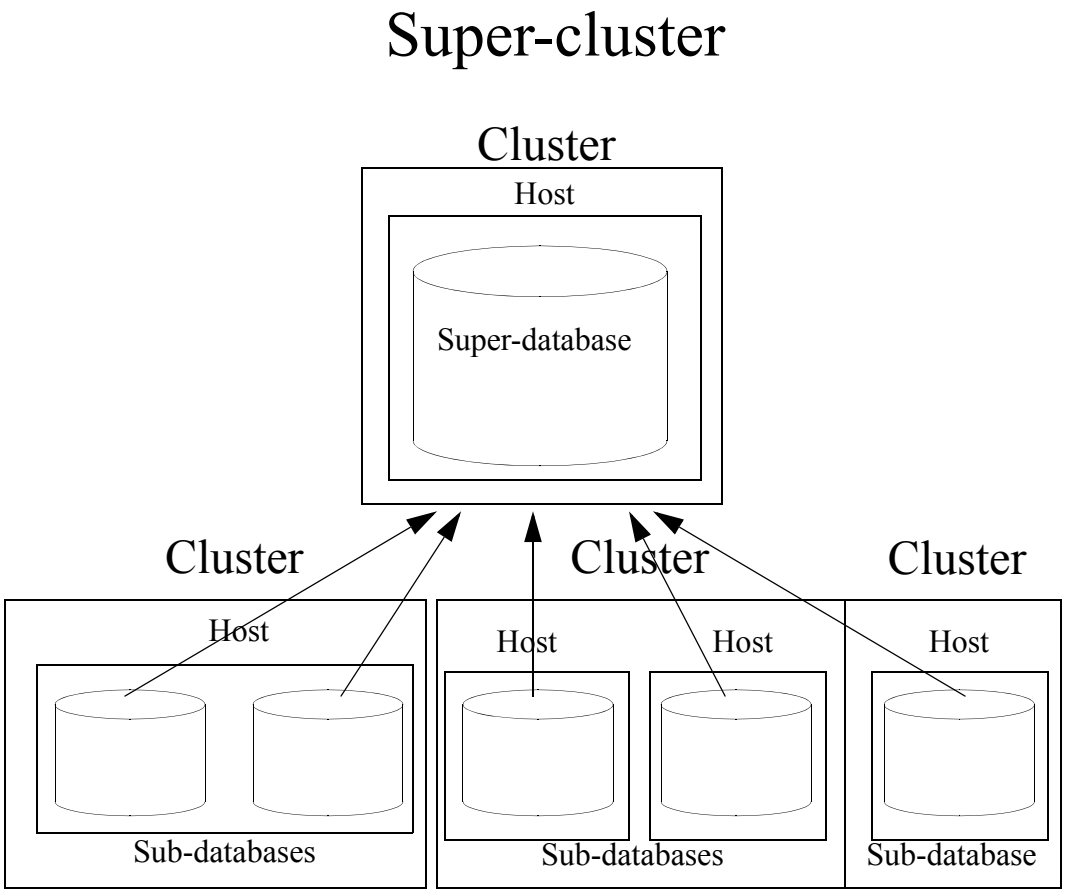
Multiple databases, even those that serve on different storage tiers, can be grouped into a *super-database* in order to allow a single query to be done across multiple tiers of data. Databases that belong to a super-database are referred to as *sub-databases*. A single sub-database can belong to multiple super-databases.



Sub-databases can be distributed on different storage tiers and on different clusters (called *super-clusters*). Updates are made on the sub-databases and they made visible for read in the super-database. Below is an illustration of a super-database and its sub-databases configured on a single cluster.



Below is a super-database configured with sub-databases on different foreign clusters. The cluster hosting the super-database must be coupled with the foreign clusters hosting the sub-databases.



For more detail on super-databases and sub-databases, see the Super Databases and Clusters chapter in the *Administrator's Guide*.

4.1.4 Partitions, Partition Keys, and Partition Ranges

MarkLogic Server tiered storage manages data in *partitions*. Each partition consists of a group of database forests that share the same name prefix and the same *partition range*.

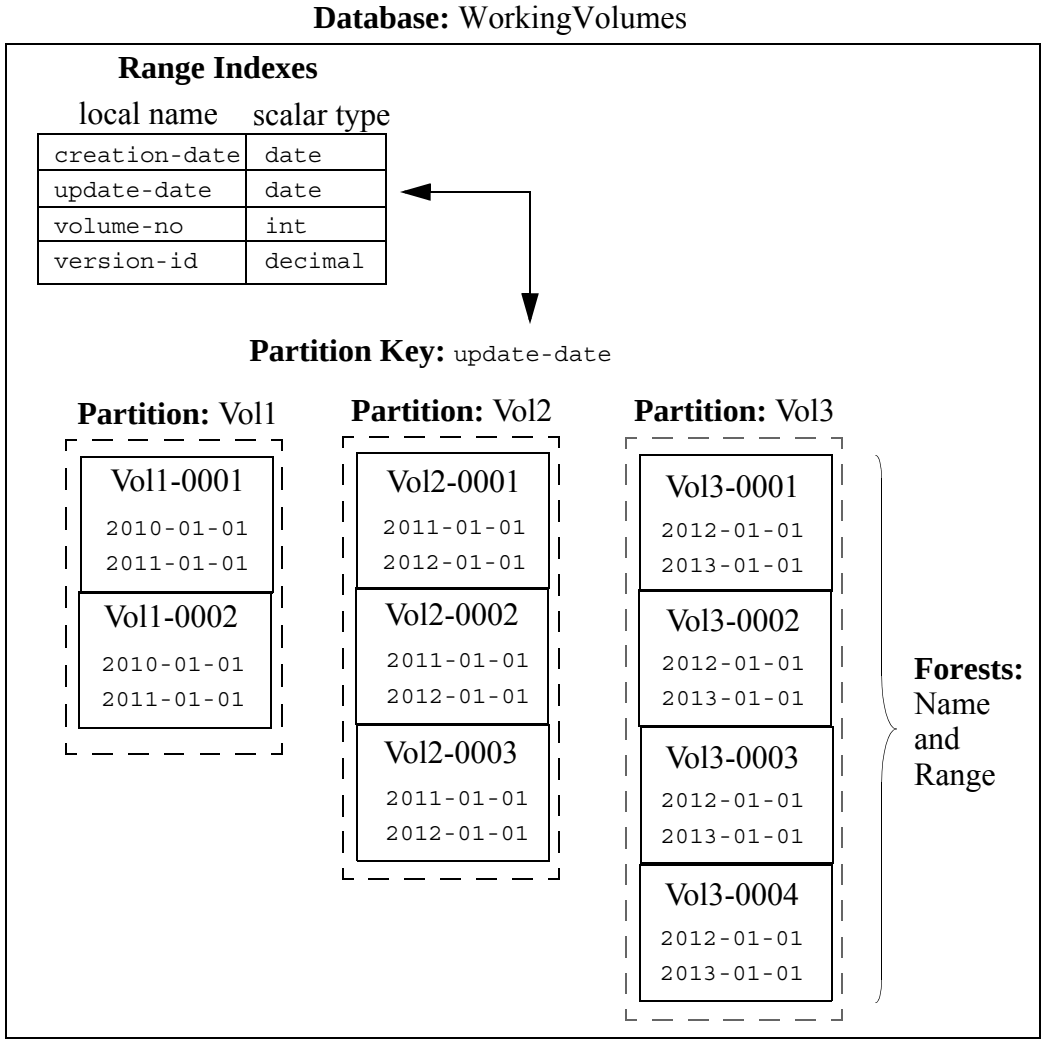
The range of a partition defines the scope of element or attribute values for the documents to be stored in the partition. This element or attribute is called the *partition key*. The partition key is based on a range index, collection lexicon, or field set on the database. The partition key is set on the database and the partition range is set on the partition, so you can have several partitions in a database with different ranges.

For example, you have a database, named `WorkingVolumes`, that contains nine forests that are grouped into three partitions. Among the range indexes in the `WorkingVolumes` database is an element range index for the `update-date` element with a type of `date`. The `WorkingVolumes` database has its partition key set on the `update-date` range index. Each forest in the `WorkingVolumes` database contains a lower bound and upper bound range value of type `date` that defines which documents are to be stored in which forests, as shown in the following table:

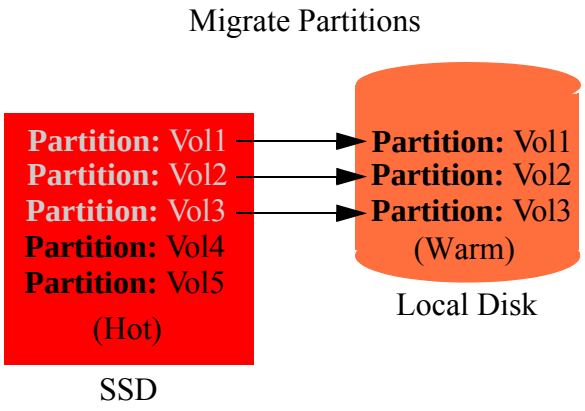
Partition Name	Forest Name (*prefix-name*)	Partition Range Lower Bound	Partition Range Upper Bound	Lower Bound Included
Vol1	Vol1-0001 Vol1-0002	2010-01-01	2011-01-01	false
Vol2	Vol2-0001 Vol2-0002 Vol2-0003	2011-01-01	2012-01-01	false
Vol3	Vol3-0001 Vol3-0002 Vol3-0003 Vol3-0004	2012-01-01	2013-01-01	false

In this example, a document with an `update-date` element value of `2011-05-22` would be stored in one of the forests in the `Vol2` partition. Should the `update-date` element value in the document get updated to `2012-01-02` or later, the document will be automatically moved to the `Vol3` partition. How the documents are redistributed among the partitions is handled by the database rebalancer, as described in “Range Assignment Policy” on page 204.

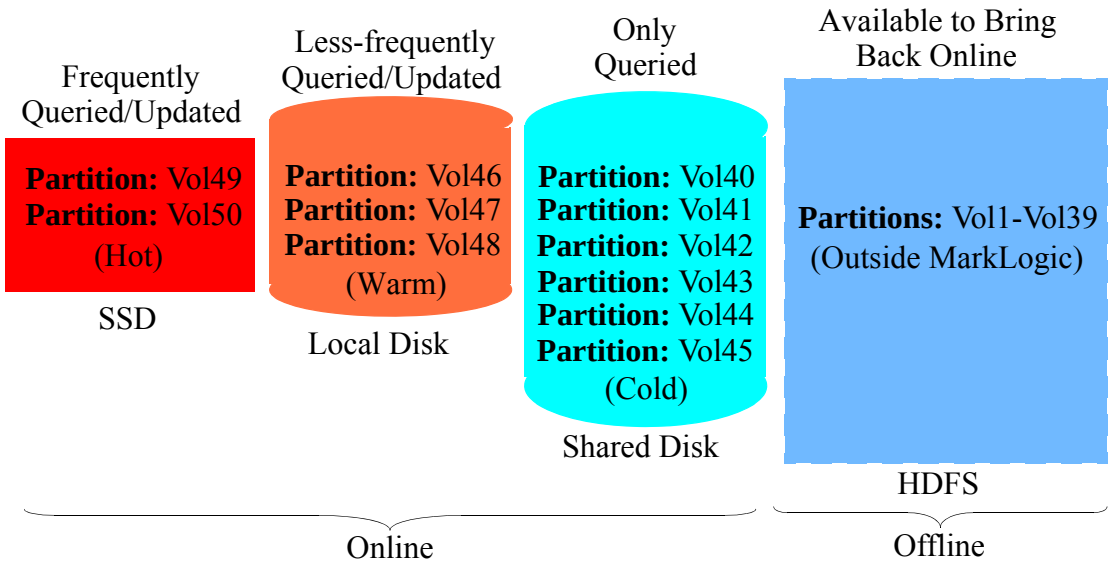
Following is an illustration of the `WorkingVolumes` database, showing its range indexes, partition key, and its partitions and forests.



In a few months, the volumes of documents grow to 5 and there is no longer enough space on the fast SSD device to hold all of them. Instead, the oldest and least queried volumes (Vol1-Vol3) are migrated to a local disk drive, which represents a slower storage tier.



After years of data growth, the volumes of documents grow to 50. After migrating between storage tiers, the partitions are eventually distributed among the storage tiers, as shown below.



4.2 Ingesting Data

To see how MarkLogic ingests data, imagine an empty database having a single forest that (because it has no documents) has no stands. When a new document is loaded into MarkLogic, MarkLogic puts this document into an in-memory stand and writes the action to an on-disk journal to maintain transactional integrity in case of system failure.

As new documents are loaded, they are also placed in the in-memory stand. A query request at this point will see all of the data on disk (nothing yet), as well as everything in the in-memory stand (our small set of documents). The query request cannot tell where the data is located, but will see the full view of data loaded at this point in time.

After enough documents are loaded, the in-memory stand will fill up and be flushed to disk, written out as an on-disk stand. Each new stand gets its own subdirectory under the forest directory, with names that are monotonically-increasing hexadecimal numbers. The first stand is named 00000000. That on-disk stand contains all the data and indexes for the documents loaded so far. The stand data is written from memory out to disk as a sequential write for maximum efficiency. Once written to disk, the in-memory stand's allocated memory is freed.

As more documents are loaded, they go into a new in-memory stand. At some point this in-memory stand fills up as well, and the in-memory stand gets written as a new on-disk stand, probably named 00000001, and about the same size as the first on-disk stand. Sometimes under heavy load you may have two in-memory stands at once, when the first stand is still writing to disk as a new stand is created for additional documents. At all times an incoming query or update request can see all the data across the in-memory and on-disk stands.

As more documents are loaded, the process continues with in-memory stands filling up and writing to on-disk stands. As the total number of on-disk stands grows, an efficiency issue threatens to emerge. To read a single term list, MarkLogic must read the term list data from each individual stand and unify the results. To keep the number of stands to a manageable level where that unification is not a performance concern, MarkLogic merges some of the stands on disk into a new singular stand. The merge operation is done in the background, where the indexes and data are coalesced and optimized and previously deleted fragments are removed, as described below in “Modifying Data” on page 54. After the merge finishes and the new on-disk stand has been fully written, and after all the current requests using the old on-disk stands have completed, MarkLogic deletes the old on-disk stands.

MarkLogic uses an algorithm to determine when to merge, based on the size of each stand. In a normal server running under constant load you will usually see a few large stands, a few more mid-sized stands, and several more small stands. Over time, the smaller stands get merged with ever-larger stands. Merges tend to be CPU- and disk-intensive, so you have control over when merges can happen via system administration.

Each forest has its own in-memory stand and a set of on-disk stands. A new document gets assigned to a forest based on the rebalancer document assignment policy set on the database, as described in Rebalancer Document Assignment Policies in the *Administrator's Guide*. Loading and indexing content is a largely parallelizable activity, so splitting the loading effort across forests and potentially across machines in a cluster can help scale the ingestion work.

4.3 Modifying Data

If you delete a document, MarkLogic marks the document as deleted but does not immediately remove it from disk. The deleted document will be removed from query results based on its deletion markings, and the next merge of the stand holding the document will bypass the deleted document when writing the new stand.

If you change a document, MarkLogic marks the old version of the document as deleted in its current stand and creates a new version of the document in the in-memory stand. MarkLogic distinctly avoids modifying the document in place. Considering how many term lists a single document change might affect, updates in place would be inefficient. So, instead, MarkLogic treats any changed document like a new document, and treats the old version like a deleted document.

To keep the discussion simple, the delete and update operations have been described as being performed on documents. However, as described in “Fragmentation of XML Documents” on page 43, fragments (not documents) are the basic units of query, retrieval, and update. So if you have fragmentation rules enabled and make a change on a document that has fragments, MarkLogic will determine which fragments need to change and will mark them as deleted and create new fragments as necessary.

This approach is known as Multi-Version Concurrency Control (MVCC), which has several advantages, including the ability to run lock-free queries, as explained next in “Multi-Version Concurrency Control” on page 54.

4.4 Multi-Version Concurrency Control

In an MVCC system, changes are tracked with a timestamp number that increments for each transaction as the database changes. Each fragment gets its own creation-time (the timestamp at which it was created) and deletion-time (the timestamp at which it was marked as deleted, starting at infinity for fragments not yet deleted). On disk, you can see these timestamps in the Timestamps file, which is the only file in the stand directory that is not read-only.

For a request that does not modify data (called a query, as opposed to an update that might make changes), the system gets a performance boost by skipping the need for URI locking. The query is viewed as running at a certain timestamp, and throughout its life it sees a consistent view of the database at that timestamp, even as other (update) requests continue forward and change the data.

MarkLogic does this by adding to the normal term list constraints two extra constraints: first, that any fragments returned have to have been “created at or before the request timestamp” and second, that they have to have been “deleted after the request timestamp.” It is easy to create from these two primitives what is in essence a new implicit term list of documents in existence at a certain timestamp. This timestamp-based term list is implicitly added to every query as a high-performance substitute for locks.

4.5 Point-in-time Queries

Normally a query acquires its timestamp marker automatically based on the time the query started. However, it is also possible for a query to request data at a specific previous timestamp. MarkLogic calls this feature *point-in-time queries*. Point-in-time queries let you query the database as it used to be at any arbitrary point in the past, as efficiently as querying at present time. One popular use of point-in-time queries is to lock the public data at a certain timestamp while new data is loaded and tested. Only when the new data is approved does the public timestamp jump to be current. And, of course, if the data is not approved, you can undo all the changes back to a past timestamp (this is referred to as “database rollback”).

When doing point-in-time queries, you have to consider merging, which normally removes deleted documents. If you want to travel into the past to see deleted documents, you need to administratively adjust the merge setting to indicate a timestamp before which documents can be reclaimed and after which they can't. This timestamp becomes the point furthest in the past to which you can query a document.

4.6 Locking

An update request must use read/write locks to maintain system integrity while making changes. This lock behavior is implicit and not under the control of the user. Read-locks block for write-locks and write-locks block for both read- and write-locks. An update has to obtain a read-lock before reading a document and a write-lock before changing (adding, deleting, modifying) a document. Lock acquisition is ordered, first-come first-served, and locks are released automatically at the end of the update request.

In any lock-based system you have to worry about deadlocks, where two or more updates are stalled waiting on locks held by the other. In MarkLogic deadlocks are automatically detected with a background thread. When the deadlock happens on the same host in a cluster, the update farthest along (the one with the most locks) wins and the other update gets restarted. When it happens on different hosts, both updates start over.

MarkLogic differentiates queries from updates using static analysis. Before running a request, it looks at the code to determine if it includes any calls to update functions. If so, the request is an update. If not, the request is a query.

4.7 Updates

Locks are acquired during the update execution, yet the actual commit work only happens after the update has successfully finished. If the update exits with an error, all pending changes that were part of that update are discarded. Each statement is its own autocommit transaction.

During the update request, the executing code can't see the changes it is making. This is because an update function does not immediately change the data, but rather adds a “work order” to the queue of things to do should the update end successfully.

Code cannot see the changes it is making because XQuery is a functional language that allows different code blocks to be run in parallel if the blocks do not depend on each other. If the code blocks are to be run in parallel, one code block should not depend on updates from another code block to have already happened at any point.

Any batch of parallel updates has to be non-conflicting. The easiest definition of non-conflicting is that they could be run in any order with the same result. You can't, for example, add a child to a node in one code block and delete the node in another, because if the execution were the inverse it wouldn't make sense. You can however make numerous changes to the same document in the same update, as well as to many other documents, all as part of the same atomic commit.

4.8 Isolating an update

When a request potentially touches millions of documents (such as sorting a large data set to find the most recent items), a query request that runs lock-free will outperform an update request that needs to acquire read-locks and write-locks. In some cases you can speed up the query work by isolating the update work to its own transactional context.

This technique only works if the update does not have a dependency on the query, which is a common case. For example, you want to execute a content search and record the user's search string to the database for tracking purposes. The database update does not need to be in the same transactional context as the search itself, and would slow the transaction down if it were. In this case it is better to run the search in one context (read-only and lock-free) and the update in a different context.

See the documentation for the `xdmp:eval` and `xdmp:invoke` functions for details on how to invoke a request from within another request and manage the transactional contexts between the two.

4.9 Documents are Like Rows

When modeling data for MarkLogic, think of documents more like rows than tables. In other words, if you have a thousand items, model them as a thousand separate documents not as a single document holding a thousand child elements. This is for two reasons:

- Locks are managed at the document level. A separate document for each item avoids lock contention.
- All index, retrieval, and update actions happen at the fragment level. When finding an item, retrieving an item, or updating an item, it is best to have each item in its own fragment. The easiest way to accomplish that is to put them in separate documents.

Of course MarkLogic documents can be more complex than simple relational rows because XML and JSON are more expressive data formats. One document can often describe an entity (a manifest, a legal contract, an email) completely.

4.10 MarkLogic Data Loading Mechanisms

MarkLogic Server provides many ways to load content into a database. Choosing the appropriate method for a specific use case depends on many factors, including the characteristics of your content, the source of the content, the frequency of loading, and whether the content needs to be repaired or modified during loading. In addition, environmental and operational factors such as workflow integration, development resources, performance considerations, and developer expertise often need to be considered in choosing the best tools and mechanisms.

The following table summarizes content loading interfaces and their benefits.

Interface/Tool	Description	Benefits
MarkLogic Content Pump (mlcp)	A command line tool.	Ease of workflow integration, can leverage Hadoop processing, bulk loading of billions of local files, split and load aggregate XML or delimited text files.
Java Client API	A set of Java classes for supporting document manipulation and search operations.	Leverage existing Java programming skills.
Node.js Client API	A low-level scripting environment that allows developers to build network and I/O services with JavaScript.	Leverage existing Node.js programming skills.
REST Client API	A set of REST services hosted on an HTTP application server and associated with a content database that enable developers to build applications on top of MarkLogic Server without writing XQuery.	Leverage existing REST programming skills.
XCC	XML Contentbase Connector (XCC) is an interface to communicate with MarkLogic Server from a Java middleware application layer.	Create multi-tier applications with MarkLogic Server as the underlying content repository.
XQuery API	An extensive set of XQuery functions that provides maximum control.	Flexibility and expanded capabilities.

Interface/Tool	Description	Benefits
WebDAV client	A WebDAV client, such as Windows Explorer connected to a MarkLogic WebDAV server.	Allows drag and drop from Windows.
MarkLogic Connector for Hadoop	A set of Java classes that enables loading content from HDFS into MarkLogic Server.	Distributed processing of large amounts of data.

For details on loading content into a MarkLogic database, see the *Loading Content Into MarkLogic Server Guide*.

4.11 Content Processing Framework (CPF)

The Content Processing Framework (CPF) is an automated system for transforming documents from one file format type to another, one schema to another, or breaking documents into pieces. CPF uses properties sheet entries to track document states and uses triggers and background processing to move documents through their states. CPF is highly customizable and you can plug in your own set of processing steps (called a *pipeline*) to control document processing.

MarkLogic includes a "Default Conversion Option" pipeline that takes Microsoft Office, Adobe PDF, and HTML documents and converts them into XHTML and simplified DocBook documents. There are many steps in the conversion process, and all of the steps are designed to execute automatically, based on the outcome of other steps in the process.

For more information on CPF, see the *Content Processing Framework Guide*.

4.12 Organizing Documents

There are a number of ways documents can be organized in a MarkLogic database. This section describes the following organization mechanisms:

- Directories
- Collections

4.12.1 Directories

Documents are located in *directories* in a MarkLogic database. Directories are hierarchical in structure (like a filesystem directory structure) and are described by a URI path. Because directories are hierarchical, a directory URI must contain any parent directories. For example, a directory named `http://marklogic.com/a/b/c/d/e/` (where `http://marklogic.com/` is the root) requires the existence of the parent directories `d`, `c`, `b`, and `a`.

Directories are required for WebDAV clients to see documents. In other words, to see a document with URI `/a/b/hello/goodbye` in a WebDAV server with `/a/b/` as the root, directories with the following URIs must exist in the database:

```
/a/b/
/a/b/hello/
```

4.12.2 Collections

A collection is a named group of documents. The key differences in using collections to organize documents versus using directories are:

- Collections do not require member documents to conform to any URI patterns. They are not hierarchical; directories are. Any document can belong to any collection, and any document can also belong to multiple collections.
- You can delete all documents in a collection with the `xdmp:collection-delete` function. Similarly, you can delete all documents in a directory (as well as all recursive subdirectories and any documents in those directories) with the `xdmp:directory-delete` function.
- You cannot set properties on a collection; you can on a directory.

A document is assigned to a collection when it is inserted or updated. A document can also be assigned to a collection after it is loaded. A document can also be put into a collection implicitly when the document is loaded, based on the default collections assigned to the user's role(s). A document can belong to multiple collections.

MarkLogic supports the following types of collection:

- Unprotected Collections
- Protected Collections

4.12.3 Unprotected Collections

An unprotected collection is created implicitly when inserting or updating a document and specifying a collection URI that has not previously been used. An unprotected collection is not stored in the security database.

Any user with insert or update permissions on a document can add the document to or remove the document from an unprotected collection.

You can convert an unprotected collection to a protected collection, and vice versa.

4.12.4 Protected Collections

Protected collections enable you to control who can add documents to a collection. A protected collection does not affect access to documents in the collection. Use document permissions to control document access.

Only users with insert or update permissions on the collection can add documents to the collection. A user with update access to a document in a protected collection can update or delete the document whether or not they have any collection permissions. Such a user can also remove the document from the collection by re-inserting the document with a different set of collections. A user with read access to a document in a protected collection can read and search the document, whether or not they have any collection permissions.

A protected collection must be explicitly created using the Admin Interface, the Admin API, or the REST Management API. MarkLogic stores protected collection configuration information in the security database.

You can convert a protected collection to an unprotected collection, and vice versa.

4.13 Database Rebalancing

As your needs for data in a database expand and contract, the more evenly the content is distributed among the database forests, the better its performance and the more efficient its use of storage resources. MarkLogic includes a database rebalancing mechanism that enables it to evenly distribute content among the database forests.

A database rebalancer consists of two parts: an *assignment policy* for data insert and rebalancing and a *rebalancer* for data movement. The rebalancer can be configured with one of several assignment policies, which define what is considered “balanced” for a database. You choose the appropriate policy for a database. The rebalancer runs on each forest and consults the database's assignment policy to determine which documents do not “belong to” this forest and then pushes them to the correct forests. The assignment policies are described in Rebalancer Document Assignment Policies in the *Administrator's Guide*.

> **Note:** Document loads and inserts into the database follow the same document assignment policy used by the rebalancer, regardless of whether the rebalancer is enabled or disabled.

When you add a new forest to a database configured with a rebalancer, the database will automatically redistribute the documents among the new forest and existing forests. You can also *retire* a forest in a database to remove all of the documents from that forest and redistribute them among all of the remaining forests in the database.

In addition to enabling and disabling on the database level, the rebalancer can also be enabled or disabled at the forest level. For the rebalancer to run on a forest, it must be enabled on both the database and the forest.

For more information on database rebalancing, see the Database Rebalancing chapter in the *Administrator's Guide*.

4.14 Bitemporal Documents

Bitemporal data is data that is associated with two time values:

- Valid time - The actual time at which an event was known to occur.
- System time - The time at which the event is recorded in the database.

Bitemporal data is commonly used in financial applications to answer questions, such as:

- What were my customer's credit ratings last Monday as I knew it last Friday?
- What did we think our first quarter profit was when we gave guidance?
- What did we think the high day was when our trading strategy kicked in?

There are two aspects to bitemporal documents in MarkLogic Server:

- Bitemporal Data Management
- Bitemporal Queries

4.14.1 Bitemporal Data Management

As bitemporal data management addresses audit and regulatory requirements, no record in bitemporal database can be deleted. This rule drives the semantics of insert, delete and update. Bitemporality is defined on a protected collection. The following describes the insert, update, and delete operations on bitemporal data:

- Insert: Each insert must set system begin and end times to the element that represent the system time period. On insert, the system end time is set to the farthest possible time. The document being inserted should already include one and only one occurrence of valid time.
- Delete: No document can be deleted from a bitemp oral collection without admin privilege. A logically deleted document remains in the database with system end time set to the time of deletion.
- Update: Update is a delete followed by one or more insert(s). The end time is marked as the time when update is performed and new version(s) of the document are inserted.

It is possible that documents will come in random order of system time, meaning later documents can either have bigger or smaller timestamp than earlier documents. The eventual state of the database will be the same as long as the exact same documents are ingested regardless of order.

For more detail on managing bitemporal documents in MarkLogic, see Managing Temporal Documents in the *Temporal Developer's Guide*.

4.14.2 Bitemporal Queries

Bitemporal queries are basically some interval operations on time period such as, period equalities, containment and overlaps. Allen's interval algebra provides the most comprehensive set of these operations to the best of our knowledge. SQL 2011 also provides similar operators. However all the SQL operators can be expressed using Allen's Algebra.

For more detail on querying bitemporal documents in MarkLogic, see Searching Temporal Documents in the *Temporal Developer's Guide*.

4.15 Managing Semantic Triples

You can use SPARQL Update with MarkLogic to manage both your managed RDF triple data and the RDF graphs containing the triples. With SPARQL Update you can delete, insert, and delete/insert (or "update") RDF triples and graphs. It uses a syntax derived from the SPARQL Query Language for RDF to perform update operations on a collection of graphs.

SPARQL Update is a formal W3C recommendation for managing triples described in the SPARQL Update 1.1 specification:

http://www.w3.org/TR/2013/REC-sparql11-update-20130321/

For details on MarkLogic support of SPARQL Update and RDF triples, see SPARQL Update in the *Semantics Developer's Guide*.

5.0 Searching in MarkLogic Server

MarkLogic includes rich full-text search features. All of the search features are implemented as extension functions available in XQuery, and most of them are also available through the REST and Java interfaces. This section provides a brief overview some of the main search features in MarkLogic and includes the following parts:

- High Performance Full Text Search
- Search APIs
- Support for Multiple Query Styles
- Full XPath Search Support in XQuery
- Lexicon and Range Index-Based APIs
- Alerting API and Built-Ins
- Semantic Searches
- Template Driven Extraction (TDE)
- Where to Find Additional Search Information

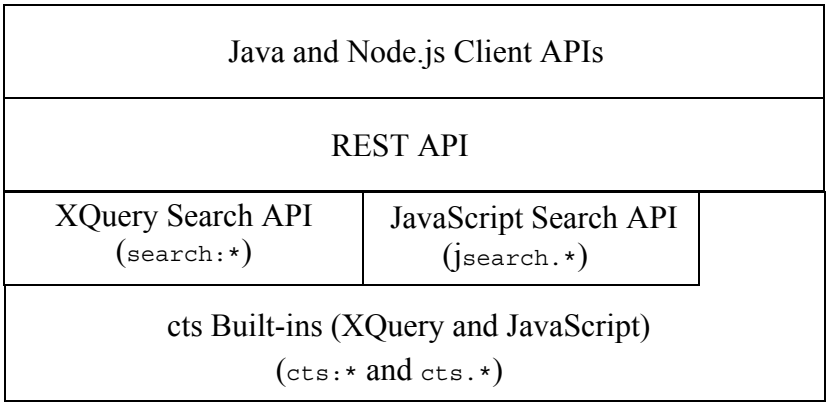
5.1 High Performance Full Text Search

MarkLogic is designed to scale to extremely large databases (100s of terabytes or more). All search functionality operates directly against the database, no matter what the database size. As part of loading a document, full-text indexes are created making arbitrary searches fast. Searches automatically use the indexes. Features such as the `xdmp:estimate` XQuery function and the `unfiltered` search option allow you to return results directly out of the MarkLogic indexes.

5.2 Search APIs

MarkLogic provides search features through a set of layered APIs. The core text search foundations in MarkLogic are the XQuery `cts:*` and JavaScript `cts.*` APIs, which are built-in functions that perform full-text search. The XQuery `search:*`, JavaScript `jsearch.*`, and REST APIs above this foundation provide a higher level of abstraction that enable rapid development of search applications. For example, the XQuery `search:*` API is built using `cts:*` features such as `cts:search`, `cts:word-query`, and `cts:element-value-query`. On top of the REST API are the Java and Node.js Client APIs that enable users familiar with those interfaces access to the MarkLogic search features.

The following diagram illustrates the layering of the Java, Node.js, REST, XQuery (`search` and `cts`), and JavaScript APIs.



The XQuery `search:*`, JavaScript `jsearch.*`, REST, Java or Node.js APIs are sufficient for most applications. Use the `cts` APIs for advanced application features, such as using reverse queries to create alerting applications and creating content classifiers. The higher-level APIs offer benefits such as the following:

- Abstraction of queries from the constraints and indexes that support them.
- Built in support for search result snippeting, highlighting, and performance analysis.
- An extensible simple string query grammar.
- Easy-to-use syntax for query composition.
- Built in best practices that optimize performance.

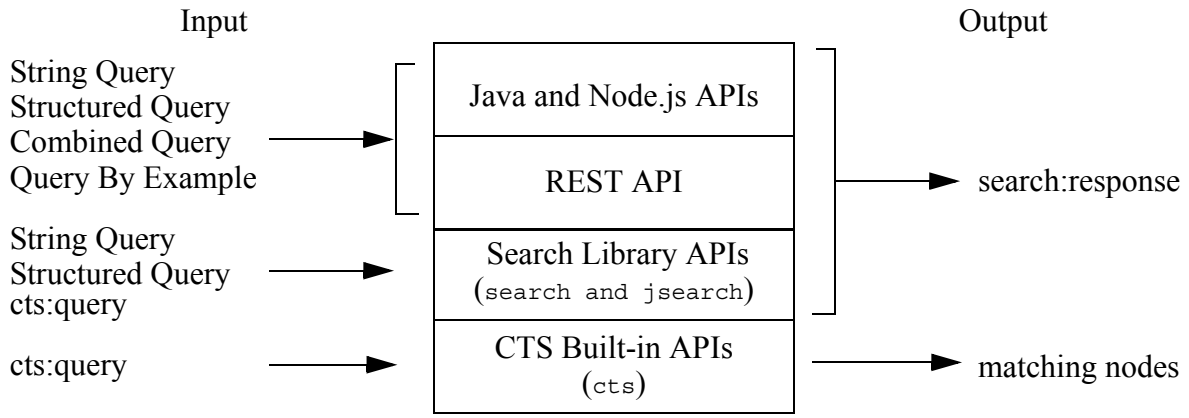
You can use more than one of these APIs in an application. For example, a Java application can include an XQuery extension to perform custom search result transformations on the server. Similarly, an XQuery application can call both `search:*` and `cts:*` functions.

5.3 Support for Multiple Query Styles

Each of the APIs described in "Search APIs" on page 66 supports one or more input query styles for searching content and metadata, from simple string queries (`cat OR dog`) to XML or JSON representations of complex queries. Search results are returned in either raw or report form. The supported query styles and result format vary by API.

For example, the primary search function for the `cts:*` API, `cts:search`, accepts input in the form of a `cts:query`, which is a composable query style that allows you to perform fine-grained searches. The `cts:search` function returns raw results as a sequence of matching nodes. The `search:*`, `jsearch.*`, REST, Java, and Node.js APIs accept more abstract query styles such as string and structured queries, and return results in report form, such as a `search:response` XML element. This customizable report can include details such as snippets with highlighting of matching terms and query metrics. The REST, Java, and Node.js APIs can also return the results report as a JSON map with keys that closely correspond to a `search:response` element.

The following diagram summarizes the query styles and results formats each API provides for searching content and metadata:



The following table provides a brief description of each query style. The level of complexity of query construction increases as you read down the table.

Query Style	Supporting APIs	Description
String Query	• Java • Node.js • REST • search • jsearch	Construct queries as text strings using a simple grammar of terms, phrases, and operators such as as AND, OR, and NEAR. String queries are easily composable by end users typing into a search text box. For details, see Searching Using String Queries in the *Search Developer's Guide*.

Query Style	Supporting APIs	Description
Query By Example	• Java • Node.js • REST	Construct queries in XML or JSON using syntax that resembles your document structure. Conceptually, Query By Example enables developers to quickly search for "documents that look like this". For details, see Searching Using Query By Example in the *Search Developer's Guide*.
Structured Query	• Java • Node.js • REST • search • jsearch	Construct queries in JSON or XML using an Abstract Syntax Tree (AST) representation, while still taking advantage of Search API based abstractions and options. Useful for tweaking or adding to a query originally expressed as a string query. For details, see Searching Using Structured Queries in the *Search Developer's Guide*.
Combined Query	• Java • Node.js • REST	Search using XML or JSON structures that bundle a string and/or structured query with query options. This enables searching without pre-defining query options as is otherwise required by the REST and Java APIs. For details, see Specifying Dynamic Query Options with Combined Query in *REST Application Developer's Guide* or Apply Dynamic Query Options to Document Searches in *Java Application Developer's Guide*
`cts:query` `cts.query`	• search • jsearch • CTS	Construct queries in XML from low level `cts:query` elements such as `cts:and-query` and `cts:not-query`. This representation is tree structured like Structured Query, but much more complicated to work with. For details, see Composing cts:query Expressions in the *Search Developer's Guide*.

5.4 Full XPath Search Support in XQuery

MarkLogic Server implements the XQuery language, which includes XPath 2.0. XPath expressions are searches which can search across the entire database. For example, consider the following XPath expression:

```
/my-node/my-child[fn:contains(., "hello")]
```

This expression searches across the entire database returning `my-child` nodes that match the expression. XPath expressions take full advantage of the indexes in the database and are designed to be fast. XPath can search both XML and JSON documents.

5.5 Lexicon and Range Index-Based APIs

MarkLogic Server has range indexes which index XML and JSON structures such as elements, element attributes, XPath expressions, and JSON keys. There are also range indexes over geospatial values. Each of these range indexes has lexicon APIs associated with them. The lexicon APIs allow you to return values directly from the indexes. Lexicons are very useful in constructing facets and in finding fast counts of element or attribute values. The Search, Java, and REST APIs makes extensive use of the lexicon features. For details about lexicons, see Browsing With Lexicons in the *Search Developer's Guide*.

5.6 Alerting API and Built-Ins

You can create applications that notify users when new content is available that matches a predefined query. There is an API to help build these applications as well as a built-in `cts:query` constructor (`cts:reverse-query`) and indexing support to build large and scalable alerting applications. For details on alerting applications, see Creating Alerting Applications in the *Search Developer's Guide*.

5.7 Semantic Searches

MarkLogic allows you use SPARQL (SPARQL Protocol and RDF Query Language) to do semantic searches on the Triple Index, described in "Triple Index" on page 37. SPARQL is a query language specification for querying over RDF (Resource Description Framework) triples.

It is a formal W3C recommendation from the RDF Data Access Working Group, described in the SPARQL Query Language for RDF recommendation:

http://www.w3.org/TR/rdf-sparql-query/

MarkLogic supports SPARQL 1.1. SPARQL queries are executed natively in MarkLogic to query either in-memory triples or triples stored in a database. When querying triples stored in a database, SPARQL queries execute entirely against the triple index.

For details on MarkLogic support of SPARQL and RDF triples, see Semantic Queries in the *Semantics Developer's Guide*.

5.8 Template Driven Extraction (TDE)

Template Driven Extraction (TDE) enables you to define a relational lens over your document data, so you can query parts of your data using SQL or the Optic API. Templates let you specify which parts of documents make up rows in a view. You can also use templates to define a semantic lens, specifying which values from a document make up triples in the triple index.

TDE enables you to generate rows and triples from ingested documents based on predefined templates that describe the following:

- The input data to match
- The data transformations that apply to the matched data
- The final data projections that are translated into indexed data.

TDE enables you to access the data in your documents in several ways, without changing the documents themselves. A relational lens is useful when you want to let SQL-savvy users access your data and when users want to create reports and visualizations using tools that communicate using SQL. It is also useful when you want to join entities and perform aggregates across documents. A semantic lens is useful when your documents contain some data that is naturally represented and queried as triples, using SPARQL.

TDE is applied during indexing at ingestion time and serves the following purposes:

- SQL/Relation indexing. TDE allows the user to map parts of an XML or JSON document into SQL rows. With a TDE template instance, users can create different rows and describe how each column in a row is constructed using the extracted data from a document. For details, see Creating Template Views in the *SQL Data Modeling Guide*.
- Custom Embedded Triple Extraction. TDE enables users to ingest triples that do not follow the `sem:triple` schema. A user can define many triple projections in a single template, where each projection specifies the different parts of a document that are mapped to subjects, predicates or objects. For details, see Using a Template to Identify Triples in a Document in the *Semantics Developer's Guide*.
- Entity Services Data Models. For details, see Creating and Managing Models in the *Entity Services Developer's Guide*.

TDE data is also used by the Optic API, as described in "Optic API" on page 93.

For details on TDE, see Template Driven Extraction (TDE) in the *Application Developer's Guide*.

5.9 Where to Find Additional Search Information

The *MarkLogic XQuery and XSLT Function Reference* and *Java Client API Documentation* describe the XQuery and JavaScript function signatures and descriptions, as well as many code examples. The *Search Developer's Guide* contains descriptions and technical details about the search features in MarkLogic Server, including:

- Search API: Understanding and Using
- Composing cts:query Expressions
- Relevance Scores: Understanding and Customizing
- Browsing With Lexicons
- Using Range Queries in cts:query Expressions
- Highlighting Search Term Matches
- Geospatial Search Applications
- Entity Extraction and Enrichment
- Creating Alerting Applications
- Using fn:count vs. xdmp:estimate
- Understanding and Using Stemmed Searches
- Understanding and Using Wildcard Searches
- Collections
- Using the Thesaurus Functions
- Using the Spelling Correction Functions
- Language Support in MarkLogic Server
- Encodings and Collations

For information on search using the REST API, see the Using and Configuring Query Features chapter in the *REST Application Developer's Guide*.

For information on search using the Java API, see the Searching chapter in the *Java Application Developer's Guide*.

For information on search using Node.js, see the Querying Documents and Metadata chapter in the *Node.js Application Developer's Guide*

For other information about developing applications in MarkLogic Server, see the *Application Developer's Guide*.

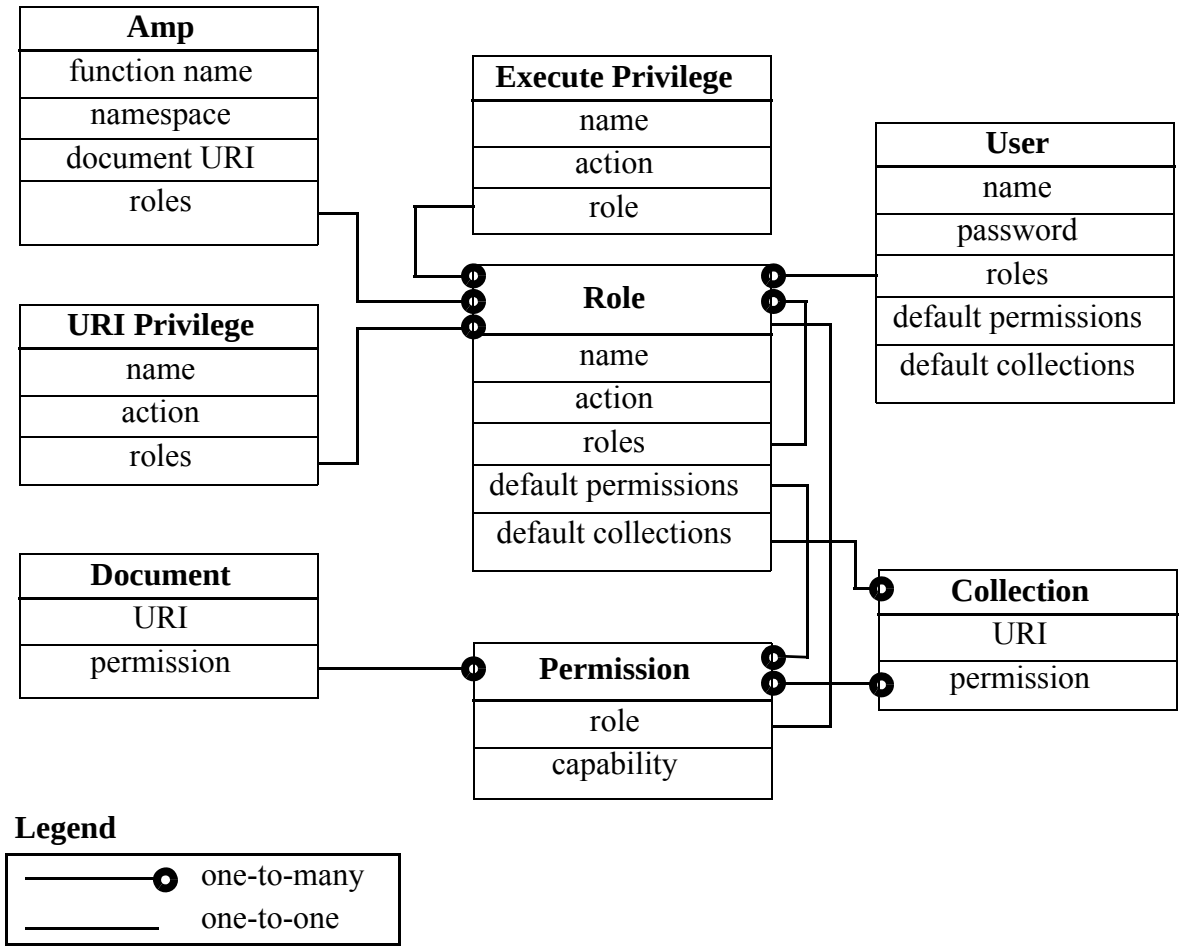
6.0 MarkLogic Server Security Model

MarkLogic Server includes a powerful and flexible role-based security model to protect your data according to your application security requirements. This chapter introduces the MarkLogic Server security model and includes the following sections:

- Role-Based Security Model (Authorization)
- The Security Database
- Security Administration
- SSL and TLS Support
- External Authentication (LDAP and Kerberos)
- Secure Credentials
- Encryption at Rest
- Element Level Security
- Other Security Tools

6.1 Role-Based Security Model (Authorization)

Roles are the central point of authorization in the MarkLogic Server security model. Privileges, users, other roles, and document permissions all relate directly to roles. The following diagram illustrates the relationships between the different entities in the MarkLogic Server security model. Note how each of these entities point to one or more roles.



Privileges are the primitives used to grant users the authority to access assets in MarkLogic Server. There are two types of privileges: *execute privileges* and *URI privileges*. Execute privileges provide the authority to perform protected actions. Examples of protected actions are the ability to execute a specific user-defined function, the ability to execute a built-in function (for example, `xdmp:document-insert`), and so on. URI privileges provide the authority to create documents within particular base URIs. When a URI privilege exists for a base URI, only users assigned to roles that have the URI privilege can create documents with URIs starting with the base string.

Privileges are assigned to zero or more roles, roles are assigned to zero or more other roles, and users are assigned to zero or more roles. A privilege is like a door and, when the door is locked, you need to have the key to the door in order to open it. The keys to the doors are distributed to users through roles.

Amps provide users with the additional privileges to execute specific protected actions by temporarily giving users additional roles for the duration of the protected action.

Permissions are used to protect documents. Permissions are assigned to documents either at load time or when a document is updated. Each permission is a combination of a role and a capability (`read`, `insert`, `update`, `execute`).

Users assigned the role corresponding to the permission have the ability to perform the capability. You can set any number of permissions on a document.

Capabilities represent actions that can be performed. There are four capabilities in MarkLogic Server:

- `read`
- `insert`
- `update`
- `execute`

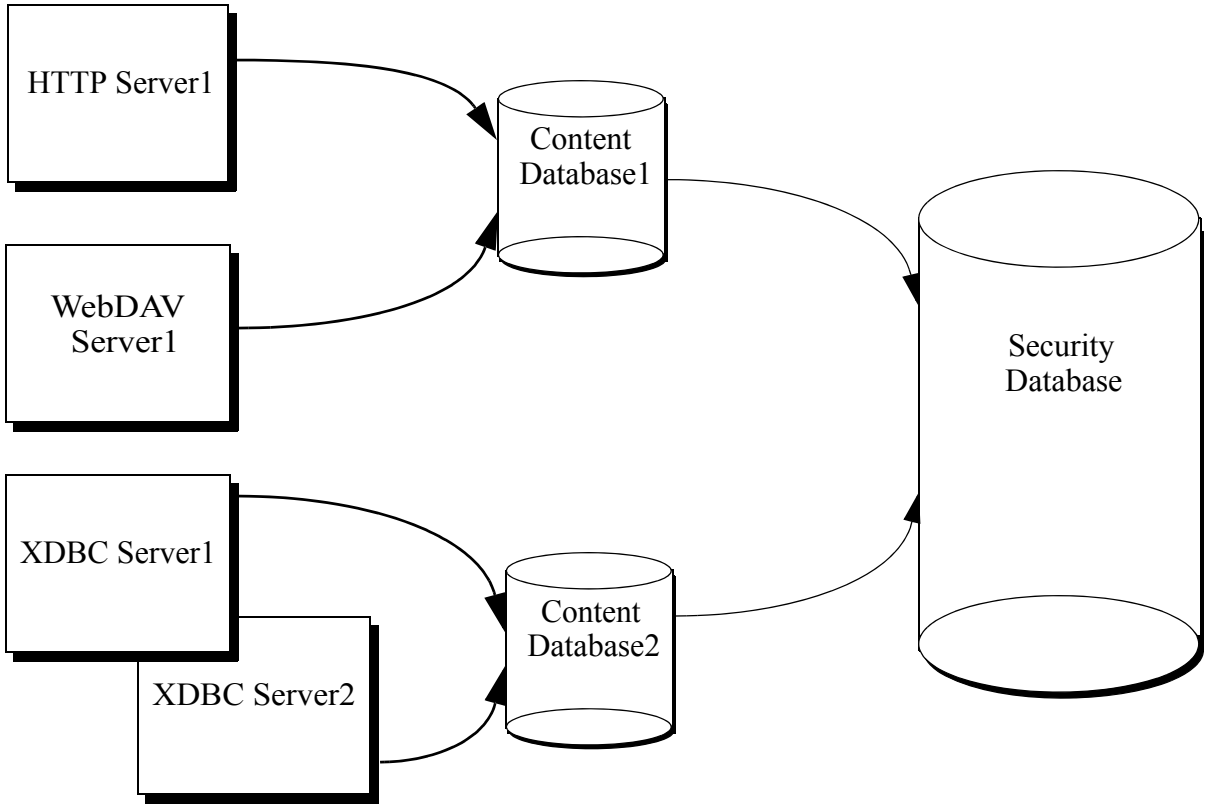
Users inherit the sum of the privileges and permissions from their roles.

For more details on how roles, privileges and permissions work in MarkLogic Server, see the *Security Guide*.

6.2 The Security Database

Authentication in MarkLogic Server occurs via the *security database*. The security database contains security objects, such as privileges, roles, and users. A security database is associated with each HTTP, XDBC, ODBC, or WebDAV server. Typically, a single security database services all of the servers configured in a system. Actions against the server are authorized based on the security database. There is always a single security database associated with each HTTP, XDBC, ODBC, or WebDAV server.

The configuration that associates the security database with the database and servers is at the database level. App servers each access a single content database, and each content database in turn accesses a single security database. Multiple content databases can access the same security database. The following figure shows many servers accessing some shared and some different content databases, all of which access the same security database.



Sharing the security database across multiple servers provides a common security configuration. You can set up different privileges for different databases if that makes sense, but they are all stored in a common security database.

In addition to storing users, roles, and privileges that you create, the security database also stores pre-defined privileges and pre-defined roles. These objects control access to privileged activities in MarkLogic Server. Examples of privileged activities include loading data and accessing URIs. The security database is initialized during the installation process.

The security database also contains the configuration data for collections, as well as the certificate authorities and templates used to enable SSL-based security on App Servers. For details on collections, see “Protected Collections” on page 80. For details on SSL, see “SSL and TLS Support” on page 77.

6.3 Security Administration

MarkLogic Server administrators are privileged users who have the authority to perform tasks such as creating,deleting, modifying users, roles, privileges, and so on. These tasks change or add data in the security database. Users who perform these tasks must have the `security` role, either explicitly or by inheriting it from another role (for example, from the `admin` role). Typically, users who perform these tasks have the `admin` role, which provides the authority to perform any tasks in the database. Use caution when assigning users to the `security` and/or `admin` roles; users who are assigned the `admin` role can perform any task on the system, including deleting data.

MarkLogic Server provides two ways to administer security:

- Admin Interface
- XQuery security administration functions in the admin and sec APIs.

For details on administering security using the Admin Interface, see the *Administrator's Guide*. For details on administering security using the admin and sec APIs, see the *Scripting Administrative Tasks Guide*.

6.4 SSL and TLS Support

SSL (Secure Sockets Layer) is a transaction security standard that provides encrypted protection between browsers and App Servers. When SSL is enabled for an App Server, browsers communicate with the App Server by means of an HTTPS connection, which is HTTP over an encrypted Secure Sockets Layer. HTTPS connections are widely used by banks and web vendors for secure transactions over the web.

A browser and App Server create a secure HTTPS connection by using a handshaking procedure. When browser connects to an SSL-enabled App Server, the App Server sends back its identification in the form of a digital certificate that contains the server name, the trusted certificate authority, and the server's public encryption key. The browser uses the server's public encryption key from the digital certificate to encrypt a random number and sends the result to the server. From the random number, both the browser and App Server generate a *session key.* The session key is used for the rest of the session to encrypt/decrypt all transmissions between the browser and App Server, enabling them to verify that the data didn't change in route.

The end result of the handshaking procedure described above is that only the server is authenticated. The client can trust the server, but the client remains unauthenticated. MarkLogic Server supports mutual authentication, in which the client also holds a digital certificate that it sends to the server. When mutual authentication is enabled, both the client and the server are authenticated and mutually trusted.

MarkLogic Server uses OpenSSL to implement the Secure Sockets Layer (SSL v3) and Transport Layer Security (TLS v1) protocols.

6.5 External Authentication (LDAP and Kerberos)

MarkLogic Server allows you to configure MarkLogic Server so that users are authenticated using an external authentication protocol, such as Lightweight Directory Access Protocol (LDAP) or Kerberos. These external agents serve as centralized points of authentication or repositories for user information from which authorization decisions can be made.

When a user attempts to access a MarkLogic App Server that is configured for external authentication, the requested App Server sends the username and password to the LDAP server or Kerberos for authentication. Once authenticated, the LDAP or Kerberos protocol is used to identify the user on MarkLogic Server.

Users can be authorized either internally by MarkLogic Server, externally by an LDAP server, or both. If internal authorization is used, the user needs to exist in the MarkLogic Security database where his or her "external name" matches the external user identity registered with either LDAP or Kerberos, depending on the selected authentication protocol.

If the App Server is configured for LDAP authorization, the user does not need to exist in MarkLogic Server. Instead, the external user is identified by a username with the LDAP server and the LDAP groups associated with the DN are mapped to MarkLogic roles. MarkLogic Server then creates a temporary user with a unique and deterministic id and those roles.

If the App Server is configured for both internal and LDAP authorization, users that exist in the MarkLogic Security database are authorized internally by MarkLogic Server. If a user is not a registered MarkLogic user, then the user must be registered on the LDAP server.

For details on compartment security see External Security in the *Security Guide*.

6.6 Secure Credentials

Secure credentials enable a security administrator to manage credentials, making them available to less privileged users for authentication to other systems without giving them access to the credentials themselves.

Secure credentials consist of a PEM encoded x509 certificate and private key and/or a username and password. Secure credentials are stored as secure documents in the Security database on MarkLogic Server, with passwords and private keys encrypted. A user references a credential by name and access is granted if the permissions stored within the credential document permit the access to the user. There is no way for a user to get access to the unencrypted credentials.

Secure credentials allow you to control which users have access to specific resources. A secure credential controls what URIs it may be used for, the type of authentication (e.g. digest), whether the credential can be used to sign other certificates, and the user role(s) needed to access the resource.

The security on a credential can be configured three different ways:

- Credentials that secure a resource by username and password.
- Credentials that secure a resource by a PEM encoded X509 certificate and a PEM encoded private key.
- Credentials that secure a resource by username and password, as well as a PEM encoded X509 certificate and a PEM encoded private key.

For details on compartment security see Secure Credentialsin the *Security Guide*.

6.7 Encryption at Rest

Increasing security risks and compliance requirements sometimes mandate the use of encryption at rest to prevent unauthorized access to data on disk. Encryption at rest protects your data on disk - "at rest" as opposed to when that data is being used in a process or "in motion." Encryption at rest allows data, configuration, and logs to be encrypted in a transparent mode; encrypted while at rest, decrypted (with the proper permissions) when in use. The administrator can choose which databases to encrypt, if configuration should be encrypted, and/or if logs should be encrypted.

Encryption works with keys generated by the internal MarkLogic Key Management System (KMS) or keys generated by an external third party KMS. The MarkLogic KMS (a PKCS #11 secured wallet) can function as an internal keystore for encryption, or you can employ a third party KMIP 1.2 compliant server as a keystore. For details about encryption at rest, see Encryption at Rest in the *Security Guide*.

6.8 Element Level Security

Element-level Security provides secure access control to protect XML elements or JSON properties within documents, in addition to the existing document-level security and compartment security provided by MarkLogic. This means that specific information inside a document may be hidden from a particular user based on his or her role, while still providing access to other information in the document the user is allowed to see.

Element level security allows you to specify more complex security rules on specific elements within documents. Permissions on an element or property are similar to permissions defined on a document. Elements or properties may contain all supported datatypes. Search results and update built-ins will honor the permissions defined at the element level. For more about element level security, see Element Level Security in the *Security Guide*.

6.9 Other Security Tools

This section describes two additional MarkLogic security tools:

- Protected Collections
- Compartments

6.9.1 Protected Collections

As described in "Collections" on page 59, collections group documents that are related and enables queries to target subsets of documents within a database. A document can belong to any number of collections simultaneously. A collection exists in the system when a document in the system states that it is part of that collection. However, an associated collection object is not created and stored in the security database unless it is protected. A collection created through the Admin Interface is a protected collection and is stored in the security database.

Read, Insert, Update, and Execute capabilities apply for permissions on a collection. A user requires insert or update permissions on the collection to add documents to a protected collection. Collection permissions have no effect on document permissions, so a user with sufficient document permissions can read, update, or delete a document in a protected collection, whether or not the user has any collection permissions.

6.9.2 Compartments

MarkLogic Server includes an extension to the security model called compartment security. Compartment security allows you to specify more complex security rules on documents.

A *compartment* is a name associated with a role. You specify that a role is part of a compartment by adding the compartment name to each role in the compartment. When a role is *compartmented*, the compartment name is used as an additional check when determining a user's authority to access or create documents in a database. Compartments have no effect on execute privileges. Without compartment security, permissions are checked using OR semantics.

For example, if a document has `read` permission for `role1` and `read` permission for `role2`, a user who possesses *either* `role1` or `role2` can read that document. If those roles have different compartments associated with them (for example, `compartment1` and `compartment2`, respectively), then the permissions are checked using AND semantics for each compartment, as well as OR semantics for each non-compartmented role. To access the document if `role1` and `role2` are in different compartments, a user must possess both `role1` and `role2` to access the document, as well as a non-compartmented role that has a corresponding permission on the document.

If any permission on a document has a compartment, then the user must have that compartment in order to access any of the capabilities, even if the capability is not the one with the compartment.

Access to a document requires a permission in each compartment for which there is a permission on the document, regardless of the capability of the permission. So if there is a `read` permission for a role in `compartment1`, there must also be an `update` permission for some role in `compartment1` (but not necessarily the same role). If you try to add `read`, `insert`, or `execute` permissions that reference a compartmented role to a document for which there is no `update` permission with the corresponding compartment, an exception is thrown.

For details on compartment security see Compartment Security in the *Security Guide*.

7.0 Clustering and Caching

You can combine multiple instances of MarkLogic to run as a *cluster*. The cluster has multiple machines (*hosts*), each running an instance of MarkLogic Server. Each host in a cluster is sometimes called a *node*, and each node in the cluster has its own copy of all of the configuration information for the entire cluster.

When deployed as a cluster, MarkLogic implements a *shared-nothing* architecture. There is no single host in charge; each host communicates with every other host, and each node in the cluster maintains its own copy of the configuration. The security database, as well as all of the other databases in the cluster, are available to each node in the cluster. This shared-nothing architecture has great advantages when it comes to scalability and availability. As your scalability needs grow, you simply add more nodes.

Clustering MarkLogic provides four key advantages:

- The ability to use commodity servers, bought for reasonable prices.
- The ability to incrementally add (or remove) new servers as need demands.
- The ability to maximize cache locality, by having different servers optimized for different roles and managing different parts of the data.
- The ability to include failover capabilities to handle server failures.

Cluster data can be replicated, as described in "Failover and Database Replication" on page 104.

This chapter has the following sections:

- MarkLogic E-nodes and D-nodes
- Communication Between Nodes
- Communication Between Clusters
- Cluster Management
- Caching
- Cache Partitions
- No Need for Global Cache Invalidation
- Locks and Timestamps in a Cluster

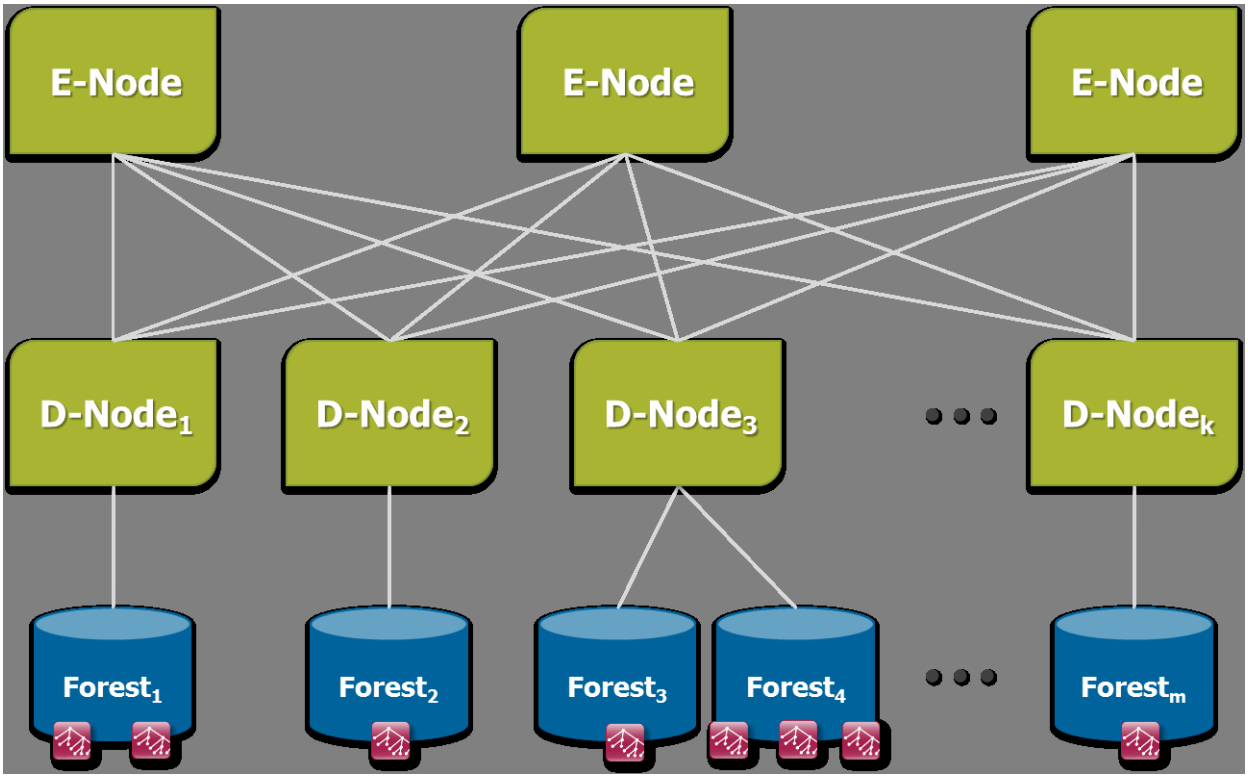
7.1 MarkLogic E-nodes and D-nodes

MarkLogic servers placed in a cluster are typically configured for one of two types of operations. They can be Evaluators (*E-nodes*) or they can be Data Managers (*D-nodes*). E-nodes listen on a socket, parse requests, and generate responses. D-nodes hold data along with its associated indexes, and support E-nodes by providing them with the data they need to satisfy requests and process updates.

As your user load grows, you can add more E-nodes. As your data size grows, you can add more D-nodes. A mid-sized cluster might have 2 E-nodes and 10 D-nodes, with each D-node responsible for about 10% of the total data.

A load balancer usually spreads incoming requests across the E-nodes. An E-node processes the request and delegates out to the D-nodes for any subexpression of the request involving data retrieval or storage. For example, if a request needs the top ten paragraphs most relevant to a query, the E-node sends the query constraint to every D-node with a forest in the database, and each D-node responds with the most relevant results for its portion of the data. The E-node coalesces the partial answers into a single unified answer: the most relevant paragraphs across the full cluster. An intra-cluster back-and-forth happens frequently as part of every request. It happens any time the request includes a subexpression requiring index work, document locking, fragment retrieval, or fragment storage.

For efficiency, D-nodes do not send full fragments across the wire unless they are truly needed by the E-node. For example, when doing a relevance-based search, each D-node forest returns an iterator containing an ordered series of fragment ids and scores extracted from indexes. The E-node pulls entries from each of the iterators returned by the D-nodes and decides which fragments to process, based on the highest reported scores. When the E-node wants to process a particular result, it fetches the fragment from the appropriate D-node.



7.2 Communication Between Nodes

Each node in a cluster communicates with all of the other nodes in the cluster at periodic intervals. This periodic communication, known as a *heartbeat*, circulates key information about host status and availability between the nodes in a cluster. Through this mechanism, the cluster determines which nodes are available and communicates configuration changes with other nodes in the cluster. If a node goes down for some reason, it stops sending heartbeats to the other nodes in the cluster.

The cluster uses the heartbeat to determine if a node in the cluster is down. A heartbeat from a given node communicates its view of the cluster at the moment of the heartbeat. This determination is based on a vote from each node in the cluster, based on each node's view of the current state of the cluster. To vote a node out of the cluster, there must be a *quorum* of nodes voting to remove a node. A quorum occurs more than 50% of the *total* number of nodes in the cluster (including any nodes that are down) vote the same way. Therefore, you need at least 3 nodes in the cluster to reach a quorum. The voting that each host performs is done based on how long it has been since it last had a heartbeat from the other node. If at half or more of the nodes in the cluster determine that a node is down, then that node is disconnected from the cluster. The wait time for a host to be disconnected from the cluster is typically considerably longer than the time for restarting a host, so restarts should not cause hosts to be disconnected from the cluster (and therefore they should not cause forests to fail over). There are configuration parameters to determine how long to wait before removing a node (for details, see "XDQP Timeout, Host Timeout, and Host Initial Timeout Parameters" on page 59).

Each node in the cluster continues listening for the heartbeat from the disconnected node to see if it has come back up, and if a quorum of nodes in the cluster are getting heartbeats from the node, then it automatically rejoins the cluster.

The heartbeat mechanism enables the cluster to recover gracefully from things like hardware failures or other events that might make a host unresponsive. This occurs automatically, without human intervention; machines can go down and automatically come back up without requiring intervention from an administrator. If the node that goes down hosts content in a forest, then the database to which that forest belongs goes offline until the forest either comes back up or is detached from the database. If you have failover enabled and configured for that forest, it attempts to fail over the forest to a secondary host (that is, one of the secondary hosts will attempt to mount the forest). Once that occurs, the database will come back online. For details on failover, see High Availability of Data Nodes With Failover in the *Scalability, Availability, and Failover Guide*.

7.3 Communication Between Clusters

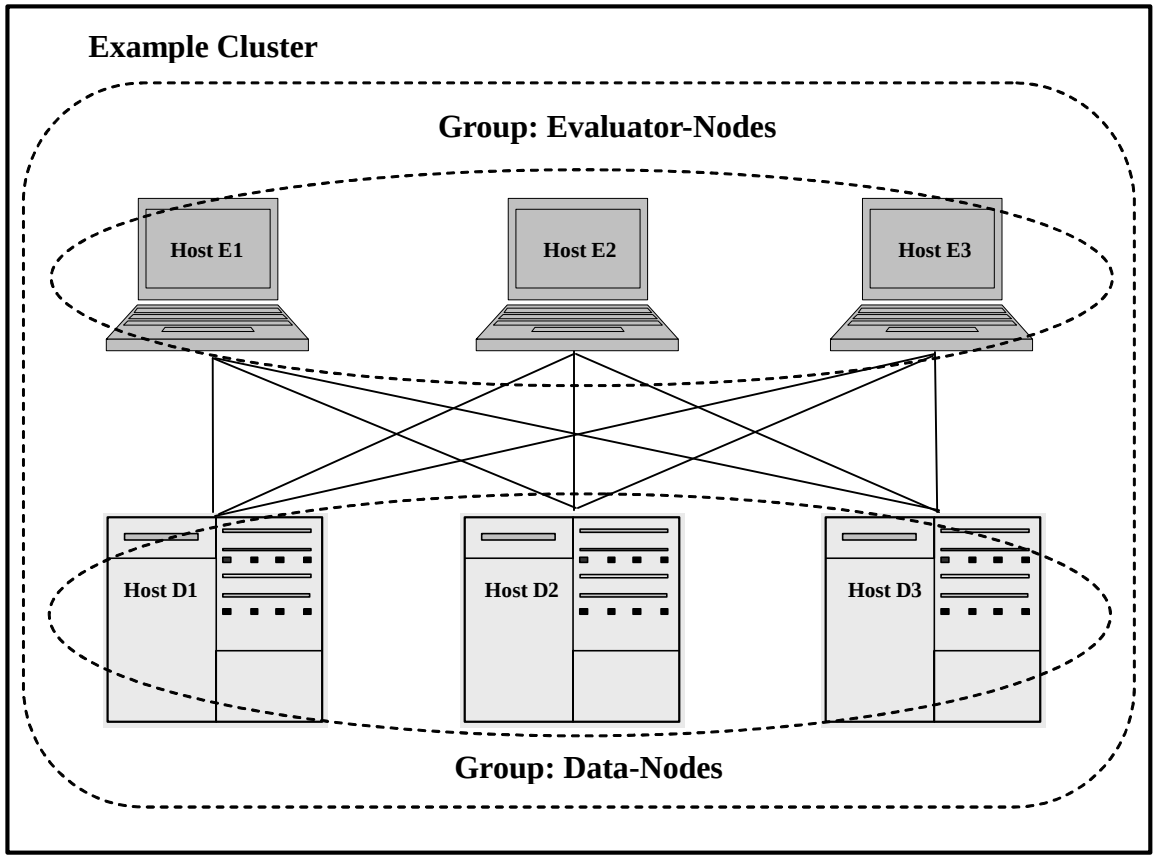
Database replication uses inter-cluster communication. Communication between clusters uses the XDQP protocol. Before you can configure database replication, each cluster in the replication scheme must be aware of the configuration of the other clusters. This is accomplished by coupling the local cluster to the foreign cluster. For more information, see Inter-cluster Communication and Configuring Database Replication in the *Database Replication Guide* and Clusters in the *Administrator's Guide*.

7.4 Cluster Management

The MarkLogic Server software installed on each server is always the same regardless of its role in the cluster. If the server is configured to listen on a socket for incoming requests (HTTP, XDBC, WebDAV, etc), then it is an E-node. If it manages data (has one or more attached forests), then it is a D-node. In the MarkLogic administration pages you can create named groups of servers, each of which share the same configuration, making it easy to have an "E Group" and a "D Group."

The first time you access the administration screens for a new MarkLogic instance, it asks if you want the instance to join a preexisting cluster. If so, you give it the name of any other server host in the cluster and what group it should be part of. The new system configures itself according to the settings of that group.

The following diagram illustrates how the concepts of clusters, hosts and groups are implemented in an example multi-host distributed architecture. The diagram shows a single cluster involving six hosts that are segmented into two groups:



7.5 Caching

On database creation, MarkLogic assigns default cache sizes optimized for your hardware, using the assumption that the server will be acting as both E-node and D-node. You can improve performance in a clustered environment by optimizing each group's cache sizes. With an E-node group, you can increase the size of the caches related to request evaluation at the expense of those related to data management. For a D-node, you can do the opposite. The table below describes some of the types of caches and how they should be changed in a clustered environment:

Cache Type	Description
List Cache	This cache holds term lists after they've been read off disk. Index resolution only happens on D-nodes, so in a D-node group you'll probably want to increase the size of the List Cache. On an E-node, you can set it to the minimum (currently 16 Megs).
Compressed Tree Cache	This cache holds the XML fragments after they've been read off disk. The fragments are stored compressed to reduce space and improve IO efficiency. Reading fragments off disk is solely a D-node task, so you will probably want to increase the size of this cache for D-nodes and set it to the minimum for E-nodes.
Expanded Tree Cache	Each time a D-node sends an E-node a fragment over the wire, it sends it in the same compressed format in which it was stored. The E-node then expands the fragment into a usable data structure. This cache stores the expanded tree instances. For binary documents it holds the raw binary data. See the discussion below for how to manage this cache on D-nodes and E-nodes.

You should increase the size of the Expanded Tree Cache on E-nodes and greatly reduce it on D-nodes. D-nodes require some Expanded Tree Cache as a workspace to support background reindexing, so the cache should never be reduced to zero. Also, if the D-node group includes an admin port on 8001, which is a good idea in case you need to administer the box directly should it leave the cluster, it needs to have enough Expanded Tree Cache to support the administration work. A good rule of thumb: set the Expanded Tree Cache to 128 Megabytes on a D-node.

When an E-node needs a fragment, it first looks in its local Expanded Tree Cache. If the fragment is not there, the E-node asks the D-node to send it. The D-node first looks in its Compressed Tree Cache. If the fragment is not there, the D-node reads the fragment off disk and sends it over the wire to the E-node. Notice the cache locality benefits gained because each D-node maintains the List Cache and Compressed Tree Cache for its particular subset of data.

7.6 Cache Partitions

Along with setting each cache size, you can also set a cache partition count. Each cache defaults to one or two or sometimes four partitions, depending on your memory size. Increasing the count can improve cache concurrency at the cost of efficiency.

Cache partitioning works as follows: Before a thread can make a change to a cache, it needs to acquire a write lock for the cache in order to keep the update thread-safe. The lock is short-lived, but it still has the effect of serializing write access. If the cache had only one partition, all of the threads would need to serialize through a single write lock. With two partitions, there is effectively two different caches and two different locks, and double the number of threads can make cache updates concurrently.

A thread uses a cache lookup key to determine which cache partition to store or retrieve an entry. A thread accessing a cache first determines the lookup key, determines which cache has that key, and goes to that cache. There is no need to read-lock or write-lock any partition other than the one appropriate for the key.

You want to avoid having an excessive number of partitions because it reduces the efficiency of the cache. Each cache partition has to manage its own aging out of entries, and can only select to remove the most stale from itself, even if there is a more stale entry in another partition.

For more information on cache tuning, see the Query Performance and Tuning guide.

7.7 No Need for Global Cache Invalidation

A typical search engine forces its administrator to make a tradeoff between update frequency and cache performance because it maintains a global cache and any document change invalidates the cache. MarkLogic avoids this problem by managing its List Cache and Compressed Tree Cache at the stand level. As described in “What's on Disk” on page 45, stands are the read-only building blocks of forests. Stand contents do not change when new documents are loaded, only when merges occur, so there is no need for the performance-killing global cache invalidation when updates occur.

7.8 Locks and Timestamps in a Cluster

MarkLogic manages locks in a decentralized way. Each D-node has the responsibility for managing the locks for the documents under its forest(s). It does this as part of its regular data access work. For example, if an E-node running an update request needs to read a set of documents, the D-nodes with those documents will acquire the necessary read-locks before returning the data. It is not necessary for the D-nodes to immediately inform the E-node about their lock actions. Each D-node does not have to check with any other hosts in the cluster to acquire locks on its subset of data. (This is the reason all fragments for a document always get placed in the same forest.)

In the regular heartbeat communication sent between the hosts in a cluster, each host reports on which locks it is holding in order to support the background deadlock detection thread.

The transaction timestamp is also managed in a decentralized way. As the very last part of committing an update, the D-node or D-nodes making the change look at the latest timestamp from their point of view, increase it by one, and use that timestamp for the new data. Getting a timestamp doesn't require cluster-wide coordination. Other hosts see the new timestamp as part of the heartbeat communication sent by each host. Each host broadcasts its latest timestamp, and hosts keep track of the maximum across the cluster.

In the special cases where you absolutely need serialization between a set of independent updates, you can have the updates acquire the same URI write-lock and thus naturally serialize their transactions into different numbered timestamps.

8.0 Application Development on MarkLogic Server

This chapter describes the various ways application developers can interact with MarkLogic Server. The main topics are as follows:

- Server-side XQuery and XSLT APIs
- Server-side JavaScript API
- REST API
- Java and Node.js Client APIs
- XML Contentbase Connector (XCC)
- SQL Support
- Optic API
- HTTP Functions to Access Internal and External Web Services
- Output Options
- Remote Filesystem Access
- Query Console for Remote Coding
- MarkLogic Connector for Hadoop

8.1 Server-side XQuery and XSLT APIs

MarkLogic includes support for XQuery 1.0 and XSLT 2.0. These are W3C-standard XML-centric languages designed for processing, querying, and transforming XML.

In addition to XQuery you have the option to use XSLT, and you have the option to use them both together. You can invoke XQuery from XSLT, and XSLT from XQuery. This means you can always use the best language for any particular task, and get maximum reuse out of supporting libraries.

XQuery includes the notion of *main modules* and *library modules*. Main modules are those you invoke directly (via either HTTP or XDBC). Library modules assist main modules by providing support functions and sometimes variables. With XSLT there is no formal separation. Every template file can be invoked directly, but templates often import one another.

XQuery and XSLT code files can reside either on the filesystem or inside a database. Putting code on a filesystem has the advantage of simplicity. You just place the code (as `.xqy` scripts or `.xslt` templates) under a filesystem directory. Putting code in a database, on the other hand, gives you some deployment conveniences: In a clustered environment it is easier to make sure every E-node is using the same codebase, because each file exists once in the database and doesn't have to be replicated across E-nodes or hosted on a network filesystem. You also have the ability to roll out a big multi-file change as an atomic update. With a filesystem deployment some requests might see the code update in a half-written state. Also, with a database you can use MarkLogic's security rules to determine who can make code updates, and you can expose (via WebDAV) remote secure access without a shell account.

There's never a need for the programmer to explicitly compile XQuery or XSLT code. MarkLogic does however maintain a "module cache" to optimize repeated execution of the same code.

You can find the full set of XQuery and XSLT API documentation at http://docs.marklogic.com. The documentation is built on MarkLogic.

8.2 Server-side JavaScript API

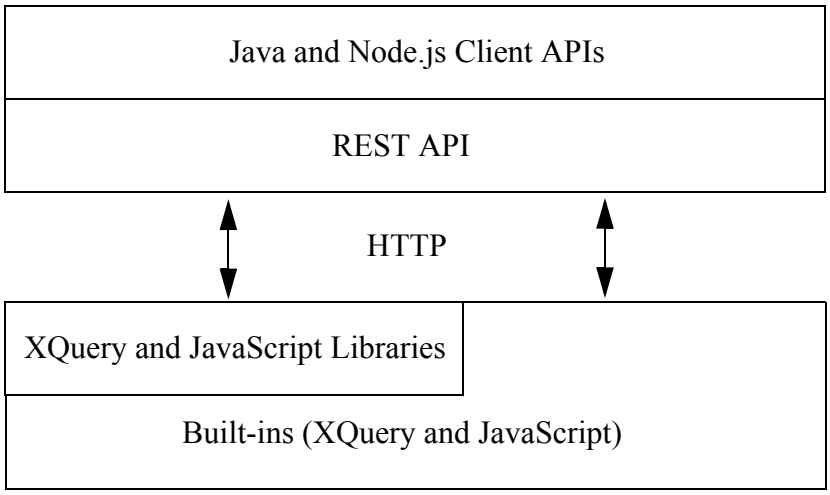
MarkLogic provides a native JavaScript API for core MarkLogic Server capabilities, such as search, lexicons, document management, App Servers, and so on.

For more information on JavaScript, see the *JavaScript Reference Guide*.

8.3 REST API

MarkLogic provides a *REST* Library. REST stands for *Representational State Transfer*, which is an architecture style that makes use of HTTP to make calls between applications and MarkLogic Server. The REST API consists of *resource addresses* that take the form of URLs. These URLs invoke XQuery *endpoint* modules in MarkLogic Server.

The REST Library provides a foundation to support other langages, such as Java and Node.js. The following diagram illustrates the layering of the Java, Node.js, REST, and XQuery and JavaScript APIs. The REST API is extensible, as described in Extending the REST API in the *REST Application Developer's Guide*, and works in a large number of applications.



Note: Some tiers may not expose all the features of adjacent tiers.

For more information on the REST API, see the *REST Application Developer's Guide*.

8.4 Java and Node.js Client APIs

MarkLogic's Java and Node.js Client APIs are built on top of the MarkLogic REST API described in "REST API" on page 91.

The Java API enables programmers to use MarkLogic without having to learn XQuery, and can easily take advantage of MarkLogic's advanced capabilities for persistence and search of unstructured documents.

In terms of performance, the Java API is very similar to MarkLogic's Java XCC, with only about a 5% difference at most on compatible queries. However, because the Java API is REST-based, to maximize performance, the network distance between the Java client and MarkLogic Server should be kept to a minimum.

When working with the Java API, you first create a manager for the type of document or operation you want to perform on the database (for instance, a `JSONDocumentManager` to write and read JSON documents or a `QueryManager` to search the database). To write or read the content for a database operation, you use standard Java APIs such as InputStream, DOM, StAX, JAXB, and Transformer as well as Open Source APIs such as JDOM and Jackson.

The Java API provides a handle (a kind of adapter) as a uniform interface for content representation. As a result, you can use APIs as different as InputStream and DOM to provide content for one `read` or `write` method. In addition, you can extend the Java API so you can use the existing `read` or `write` methods with new APIs that provide useful representations for your content.

To stream, you supply an InputStream or Reader for the data source not only when reading from the database but also when writing to the database. This approach allows for efficient write operations that do not buffer the data in memory. You can also use an OutputWriter to generate data as the API is writing the data to the database.

MarkLogic also supports JavaScript middleware in the form of a Node.js process for network programming and server-side request/response processing. Node.js is a low-level scripting environment that allows developers to build network and I/O services with JavaScript. Node.js is designed around non-blocking I/O and asynchronous events, using an event loop to manage concurrency. Node.js applications (and many of its core libraries) are written in JavaScript and run single-threaded, although Node.js uses multiple threads for file and network events.

For more information on Java, see the *Java Application Developer's Guide*. For more information on Node.js, see the *Node.js Application Developer's Guide*.

8.5 XML Contentbase Connector (XCC)

The XML Contentbase Connector (XCC) is an interface to communicate with MarkLogic Server from a Java middleware application layer.

XCC has a set of client libraries that you use to build applications that communicate with MarkLogic Server. XCC requires that an XDBC server is configured in MarkLogic Server.

An XDBC server responds to XDBC and XCC requests. XDBC and XCC use the same wire protocol to communicate with MarkLogic Server. You can write applications either as standalone applications or ones that run in an application server environment. Your XCC-enabled application connects to a specified port on a system that is running MarkLogic Server, and communicates with MarkLogic Server by submitting requests (for example, XQuery statements) and processing the results returned by those programs. These XQuery programs can incorporate calls to XQuery functions stored and accessible by MarkLogic Server, and accessible from any XDBC-enabled application. The XQuery programs can perform the full suite of XQuery functionality, including loading, querying, updating and deleting content.

For more information on XCC, see the *XCC Developer's Guide*.

8.6 SQL Support

The SQL supported by the core SQL engine is SQL92, with the addition of SET, SHOW, and DESCRIBE statements. MarkLogic SQL enables you to connect Bussiness Intelligence (BI) tools, such as Tableau and Qlik, to analyze your data, as described in Connecting Tableau to MarkLogic Server and Connecting Qlik to MarkLogic Server in the *SQL Data Modeling Guide*.

For more information on MarkLogic SQL, see the *SQL Data Modeling Guide*.

8.7 Optic API

The MarkLogic Optic API makes it possible to perform relational operations on indexed values and documents. The Optic API is not a single API, but rather a set of APIs exposed within the XQuery, JavaScript, and Java languages.

The Optic API can read any indexed value, whether the value is in a range index, the triple index, or rows extracted by a template. The extraction templates, such as those used to create template views described in Creating Template Views in the *SQL Data Modeling Guide*, are a simple, powerful way to specify a relational lens over documents, making parts of your document data accessible via SQL. Optic gives you access to the same relational operations, such as joins and aggregates, over rows. The Optic API also enables document search to match rows projected from documents, joined documents as columns within rows, and dynamic document structures – all performed efficiently within the database and accessed programmatically from your application.

The Optic API allows you to use your data as-is and makes it possible to make use of MarkLogic document and search features using JavaScript or XQuery syntax, incorporating common SQL concepts, regardless of the structure of your data. Unlike SQL, Optic is well suited for building applications and accessing the full range of MarkLogic NoSQL capabilities. Because Optic is integrated into common application lauguages, it can perform queries within the context of broader applications that perform updates to data and process results for presentation to end users.

For more information on the Optic API, see Optic API for Multi-Model Data Access in the *Application Developer's Guide*.

8.8 HTTP Functions to Access Internal and External Web Services

You can access web services, both within an intranet and anywhere across the internet, with the XQuery-level HTTP functions built into MarkLogic Server. The HTTP functions allow you to perform HTTP operations such as GET, PUT, POST, and DELETE. You can access these functions directly through XQuery, thus allowing you to post or get content from any HTTP server, including the ability to communicate with web services. The web services that you communicate with can perform external processing on your content, such as entity extraction, language translation, or some other custom processing. Combined with the conversion and HTML Tidy functions, the HTTP functions make it very easy to process any content you can get to on the web within MarkLogic Server.

The XQuery-level HTTP functions can also be used directly with `xdmp:document-load`, `xdmp:document-get`, and all of the conversion functions. You can then, for example, directly process content extracted via HTTP from the web and process it with HTML Tidy (`xdmp:tidy`), load it into the database, or do anything you need to do with any content available via HTTP.

8.9 Output Options

With MarkLogic you can generate output in many different formats:

- XML, of course. You can output one node or a series of nodes.
- HTML. You can output HTML as the XML-centric xhtml or as traditional HTML.
- RSS and Atom. They're just XML formats.
- PDF. There's an XML format named XSL-FO designed for generating PDF.
- Microsoft Office. Office files use XML as a native format beginning with Microsoft Office 2007. You can read and write the XML files directly, but to make the complex formats more approachable we'd recommend you use MarkLogic's open source Office Toolkits.
- Adobe InDesign and QuarkXPress. Like Microsoft Office, these publishing formats use native XML formats.
- JSON, the JavaScript Object Notation format common in Ajax applications. It's easy to translate between XML and JSON. MarkLogic includes built-in translators.

8.10 Remote Filesystem Access

WebDAV provides a third option for interfacing with MarkLogic. WebDAV is a widely used wire protocol for file reading and writing. It's a bit like Microsoft's SMB (implemented by Samba) but it's an open standard. By opening a WebDAV port on MarkLogic and connecting to it with a WebDAV client, you can view and interact with a MarkLogic database like a filesystem, pulling and pushing files.

WebDAV works well for drag-and-drop document loading, or for bulk copying content out of MarkLogic. All the major operating systems include built-in WebDAV clients, though third-party clients are often more robust. WebDAV doesn't include a mechanism to execute XQuery or XSLT code. It's just for file transport.

Some developers use WebDAV for managing XQuery or XSLT code files deployed out of a database. Many code editors have the ability to speak WebDAV and by mounting the database holding the code it's easy to author code hosted on a remote system with a local editor.

8.11 Query Console for Remote Coding

Not actually a protocol into itself, but still widely used by programmers wanting raw access MarkLogic, is the Query Console web-based code execution environment. Query Console enables you to run ad hoc JavaScript, SPARQL, SQL, or XQuery code from a text area in your web browser. It's a great administration tool.

It includes multiple buffers, history tracking, beautified error messages, the ability to switch between any database on the server, and has output options for XML, HTML, or plain text. Query Console also allows you to list and open the files in any database. It also includes a profiler — a web front-end on MarkLogic's profiler API — that helps you identify slow spots in your code.

8.12 MarkLogic Connector for Hadoop

Hadoop MapReduce Connector provides an interface for using a MarkLogic Server instance as a MapReduce input source and/or a MapReduce output destination.

This section provides a high level overview of the features of The MarkLogic Connector for Hadoop. The MarkLogic Connector for Hadoop manages sessions with MarkLogic Server and builds and executes queries for fetching data from and storing data in MarkLogic Server. You only need to configure the job and provide map and reduce functions to perform the desired analysis.

The MarkLogic Connector for Hadoop API provides tools for building MapReduce jobs that use MarkLogic Server, such as the following:

- `InputFormat` subclasses for retrieving data from MarkLogic Server and supplying it to the map function as documents, nodes, and user-defined types.
- `OutputFormat` subclasses for saving data to MarkLogic Server as documents, nodes and properties.

- Classes supporting key and value types specific to MarkLogic Server content, such as nodes and documents.
- Job configuration properties specific to MarkLogic Server, including properties for selecting input content, controlling input splits, and specifying output destination and document quality.

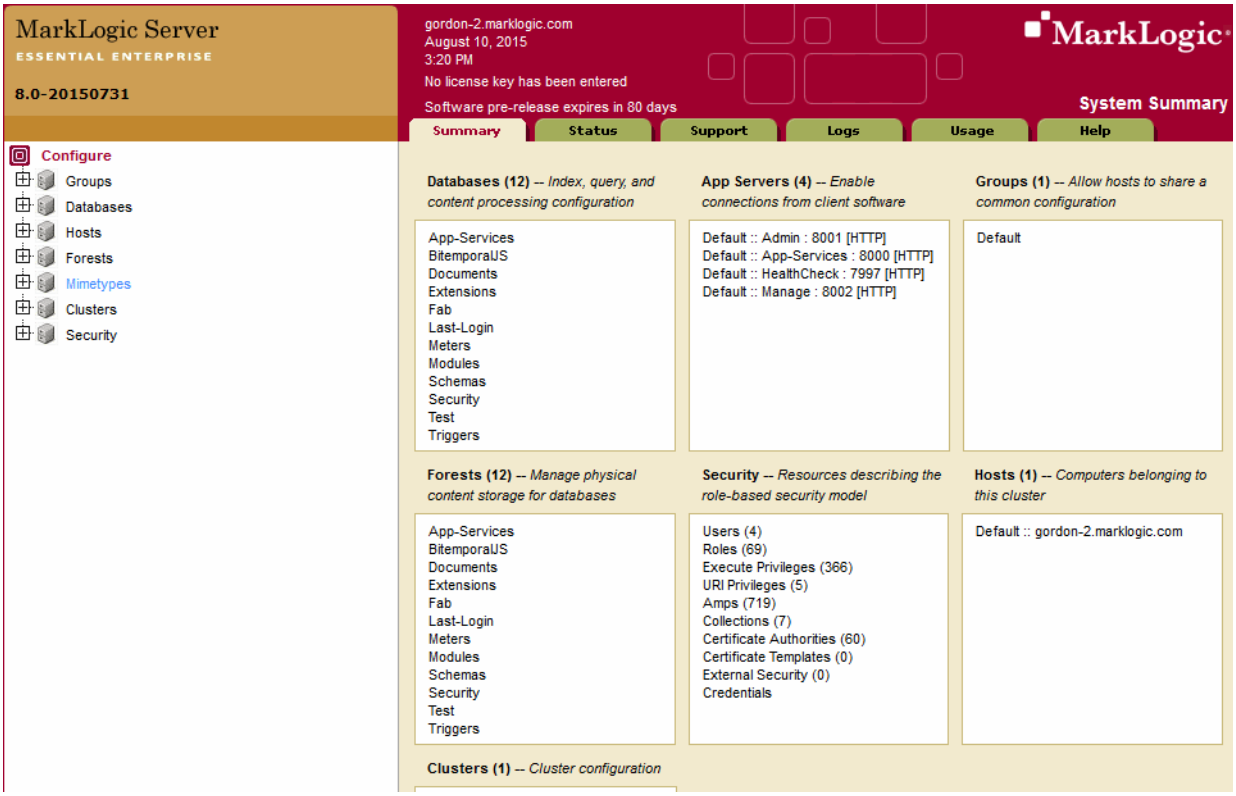
9.0 Administrating and Monitoring MarkLogic Server

This chapter describes the tools available for configuring and monitoring a MarkLogic cluster. The topics are as followings:

- Administration Interface
- Administration APIs
- Configuration Management
- Monitoring Tools
- Telemetry

9.1 Administration Interface

The Administraton Interface (Admin Interface) is graphic user interface for managing MarkLogic Server and is implemented as a web application that runs in your browser.



With the Admin Interface, you can complete any of the following tasks:

- Manage basic software configuration
- Create and configure groups
- Create and manage databases

- Create and manage new forests
- Back up and restore forest content
- Create and manage new web server and Java-language access paths
- Create and manage security configurations
- Tune system performance
- Configure namespaces and schemas
- Check the status of resources on your systems

Users with the `admin` role, known as authorized administrators, are trusted personnel and are assumed to be non-hostile, appropriately trained, and follow proper administrative procedures.

For detailed procedures on using the Admin Interface, see the *Administrator's Guide*.

9.2 Administration APIs

The MarkLogic Admin APIs provide a flexible toolkit for creating new and managing existing configurations of MarkLogic Server. The Admin APIs take the following forms:

- XQuery Functions
- Server-side JavaScript Functions
- REST Resources

The common use cases of the Admin APIs include.

- Configuring multiple, identical instances of MarkLogic Server (across a cluster or multiple clusters and/or to maintain consistency between develpment, certification and production environments).
- Automating Server Maintenance. For example, backups based on criteria other than a schedule.
- Managing server resources. For example, delete-only forest rotation.
- Making Bulk Updates to Server Configuration. For example, changing roles across a large subgroup of users.
- Generating notifications (log and/or email) for specific server events.

The *Scripting Administrative Tasks Guide* provides code samples that demonstrate the various uses of the MarkLogic Server Administration APIs.

9.3 Configuration Management

The MarkLogic Server Configuration Manager allows you to export and import the configuration settings for MarkLogic Server resources. A *resource* is a MarkLogic Server object, such as a database, forest, App Server, group or host.

The Configuration Manager provides the following capabilities:

- Allow non-admin users read-only access to configuration settings for databases, forests, groups, hosts, and App Servers.
- Easily search for resources and configuration settings.
- Safely review settings in read-only mode, then jump directly to the resource in the Admin Interface to modify the settings. (Administrative privileges are required to modify settings).
- Export all or part of a MarkLogic Server configuration to a zip folder.
- Import a configuration. Importing a configuration allows you to compare the imported configuration against your current server configuration, modify the imported configuration, and apply the imported configuration to your server.
- Rollback an imported configuration to restore the original configuration.

For details on the Configuration Manager, see Using the Configuration Manager in the *Administrator's Guide*.

9.4 Monitoring Tools

MarkLogic provides a rich set of monitoring features that include pre-configured monitoring and monitoring history dashboards, plugins that allow you to monitor MarkLogic with a Management API that allows you to integrate MarkLogic with existing monitoring applications or create your own custom monitoring applications.

In general, you will use a monitoring tool for the following:

- To keep track of the day-to-day operations of your MarkLogic Server environment in realtime.
- Capture historical performance metrics.
- For initial capacity planning and fine-tuning your MarkLogic Server environment. For details on how to configure your MarkLogic Server cluster, see the *Scalability, Availability, and Failover Guide*.
- To troubleshoot application performance problems. For details on how to troubleshoot and resolve performance issues, see the *Query Performance and Tuning Guide*.
- To troubleshoot application errors and failures.

MarkLogic includes the following monitoring tools:

- A Monitoring dashboard that monitors MarkLogic Server. This dashboard is pre-configured to monitor specific MarkLogic Server metrics. For details, see Using the MarkLogic Server Monitoring Dashboard in the *Monitoring MarkLogic Guide*.
- A Monitoring History dashboard to capture and make use of historical performance data for a MarkLogic cluster. For details, see MarkLogic Server Monitoring History in the *Monitoring MarkLogic Guide*.
- A RESTful Management API that you can use to integrate MarkLogic Server with existing monitoring application or create your own custom monitoring applications. For details, see Using the Management API in the *Monitoring MarkLogic Guide*.

9.5 Telemetry

MarkLogic telemetry provides faster, more complete communication with MarkLogic Support to facilitate the resolution of issues. When telemetry is enabled, it collects, encrypts, and sends diagnostic and anonymized system-level information about a MarkLogic cluster to a secure MarkLogic destination. Telemetry collects only system-level information, and sends it to a protected and secure location, where it can only be accessed by the MarkLogic technical teams to facilitate troubleshooting and monitor performance.

You can enable/disable each data type that you allow to be transmitted: Log data, Metering data, Configuration files, and Support Request data. Telemetry does not collect user data or application logs. At all times, information collected through telemetry is handled in accordance with the MarkLogic Privacy Policy available at http://www.marklogic.com/privacy-policy/. You can preview the monitoring data to be sent either by viewing it in a browser or saving it to a local file. For more information, see Telemetry in the *Monitoring MarkLogic Guide*.

10.0 High Availability and Disaster Recovery

This chapter describes the MarkLogic features that provide high availability and disaster recovery. The main topics are as follows:

- Managing Backups
- Failover and Database Replication

10.1 Managing Backups

MarkLogic supports online backups and restores, so you can protect and restore your data without bringing the system offline or halting queries or updates. Backups are initiated via administration calls, either via the web console or an XQuery script. You specify a database to backup and a target location. Backing up a database backs up its configuration files, all the forests in the database, as well as the corresponding security and schemas databases. It's particularly important to backup the security database because MarkLogic tracks role identifiers as xs:long values and the backup forest data can't be read without the corresponding roles existing in the security database.

You can also choose to selectively backup an individual forest instead of an entire database. That's a convenient option if only the data in one forest is changing.

The topics in this section are as follows:

- Typical Backup
- Backup with Journal Archiving
- Incremental Backup

10.1.1 Typical Backup

Throughout most of the time when a backup is running all queries and updates proceed as usual. MarkLogic simply copies stand data from the source directory to the backup target directory, file by file. Stands are read-only except for the small Timestamps file, so this bulk copy can proceed without needing to interrupt any requests. Only at the very end of the backup does MarkLogic have to halt incoming requests for a brief moment in order to write out a fully consistent view for the backup, flushing everything from memory to disk.

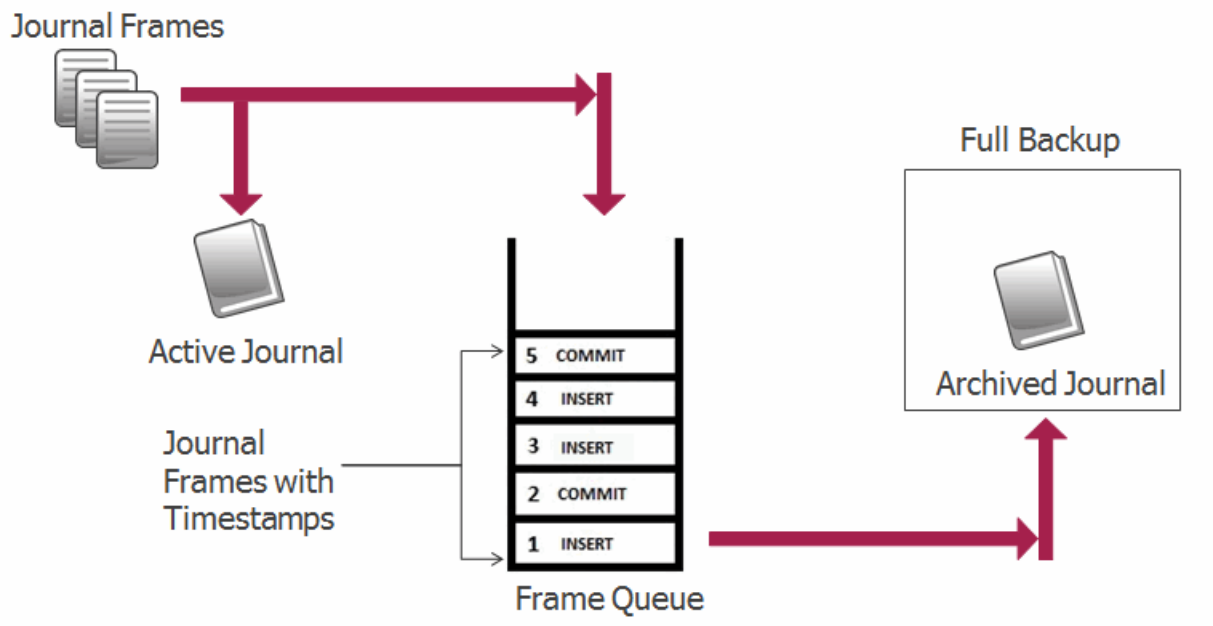
If the target backup directory already has data from a previous backup (as is the case when old stands haven't yet been merged into new stands), MarkLogic skips copying any files that already exist and are identical in the target. This isn't quite an incremental backup, but it's similar, and it gives a nice performance boost.

For more detail on backup and restore, see Backing Up a Database and Restoring a Database without Journal Archiving in the *Administrator's Guide*.

10.1.2 Backup with Journal Archiving

The backup/restore operations with journal archiving enabled provide a point-in-time recovery option that enables you to restore database changes to a specific point in time between full backups with the input of a wall clock time. When journal archiving is enabled, journal frames are written to backup directories by near synchronously streaming frames from the current active journal of each forest.

When journal archiving is enabled, you will experience longer restore times and slightly increased system load as a result of the streaming of journal frames.



Note: Journal archiving can only be enabled at the time of a full backup. If you restore a backup and want to reenable journal archiving, you must perform a full backup at that time.

When journal archiving is enabled, you can set a lag limit value that specifies the amount of time (in seconds) in which frames being written to the forest's journal can differ from the frames being streamed to the backup journal. For example, if the lag limit is set to 30 seconds, the archived journal can lag behind a maximum of 30 seconds worth of transactions compared to the active journal. If the lag limit is exceeded, transactions are halted until the backup journal has caught up.

The active and backup journal are synchronized at least every 30 seconds. If the lag limit is less than 30 seconds, synchronization will be performed at least once in that period. If the lag limit is greater than 30 seconds, synchronization will be performed at least once every 30 seconds. The default lag limit is 15 seconds.

The decision on setting a lag limit time is determined by your Recovery Point Objective (RPO), which is the amount of data you can afford to lose in the event of a disaster. A low RPO means that you will restore the most data at the cost of performance, whereas a higher RPO means that you will potentially restore less data with the benefit of less impact to performance. In general, the lag limit you chose depends on the following factors:

A lower lag limit implies:

- Accurate synchronization between active and backup journals at the potential cost of system performance.
- Use when you have an archive location with high I/O bandwidth and your RPO objective is low.

A higher lag limit implies:

- Delayed synchronization between active and backup journals, but lesser impact on system performance.
- Higher server memory utilization due to pending frames being held in memory.
- Use when you have an archive location with low I/O bandwidth and your RPO objective is high.

For more detail on backup and restore with journal archiving, see Backing Up Databases with Journal Archiving and Restoring Databases with Journal Archiving in the *Administrator's Guide*.

10.1.3 Incremental Backup

An incremental backup stores only the data that has changed since the previous full or incremental backup. Typically a series of incremental backups are done between full backups. Incremental backups are more compact than archived journals and are faster to restore. It is possible to schedule frequent incremental backups (for example, by the hour or the minute) because an incremental backup takes less time to do than a full backup.

Full and incremental backups need to be scheduled separately. An example configuration might be:

- Full backups scheduled monthly
- Incremental backups scheduled daily

A full backup and a series of incremental backups can allow you to recover from a situation where a database has been lost. Incremental backup can be used with or without journal archiving. If you enable both incremental backup and journal archiving, you can replay the journal starting from the last incremental backup timestamp. See "Backup with Journal Archiving" on page 102 for more about journal archiving.

> **Note:** When you restore from an incremental backup, you need to do a full backup before you can continue with incremental backups.

Incremental backup and journal archiving both provide disaster recovery. Incremental backup uses less disk space than journal archiving, and incremental backup is faster than using journal archiving.

For recovery you only need to specify the timestamp for the recovery to start and the server will figure out which full backup and which incremental backup(s) to use. You only need to schedule the incremental backup; the server will link together (or chain) the sequence the incremental backups automatically.

For more detail on incremental backup and restore, see Incremental Backup and Restoring from an Incremental Backup with Journal Archiving in the *Administrator's Guide*.

10.2 Failover and Database Replication

MarkLogic provides a variety of mechanisms to ensure high availability and disaster recovery. The general mechanisms and their uses are:

- Failover to ensure high availability within a cluster.
- Replication to ensure high availability and disaster recovery between multiple clusters.

Failover for MarkLogic Server provides high availability for data nodes in the event of a data node or forest-level failure. Data node failures can include operating system crashes, MarkLogic Server restarts, power failures, or persistent system failures (hardware failures, for example). A forest-level failure is any disk I/O or other failure that results in an error state on the forest. With failover enabled and configured, a forest can go down and the MarkLogic Server cluster automatically and gracefully recovers from the outage, continuing to process queries without any immediate action needed by an administrator.

Database Replication is the process of maintaining copies of databases in multiple MarkLogic Server clusters. At a minimum, there will be a Master database and one Replica database. Should there be a failure involving the Master database, the Replica database is available to resume operation. A failure of a Master database may be the result of a host or database failure, or a failure of the entire cluster, such as power outage.

The topics in this section are as follows:

- Local- and Shared-Disk Failover
- Database Replication

10.2.1 Local- and Shared-Disk Failover

Databases in a MarkLogic cluster have forests that hold their content, and each forest is served by a single host in the cluster. To guard against a host going down and being disconnected from the cluster or forest-level failures, each forest allows you to set up one of two types of failover:

- Local-Disk Failover
- Shared-Disk Failover

Both types of failover are controlled and configured at the forest level. Each type of failover has its pros and cons. Shared-Disk Failover is more efficient with disk. Local-Disk Failover is easier to configure, can use cheaper local disk, and doesn't require a clustered filesystem or fencing software.

10.2.1.1 Local-Disk Failover

Local-Disk Failover uses the intra-cluster forest replication capability introduced with MarkLogic 4.2. With forest replication you can have all writes to one forest be automatically replicated to another forest or set of forests, with each forest held on a different set of disks, generally cheap local disks, for redundancy.

Should the server managing the master copy of the forest data go offline, another server managing a different forest with a copy of the data can continue forward as the new master. When the first forest comes back online, any updated data in the replica forest can be re-synchronized back to the master to get them in sync again.

MarkLogic starts forest replication by performing fast bulk synchronization for initial "zero day" synchronization. It also does this if a forest has been offline for an extended period. It then performs an incremental journal replay once the forests are in sync, sending journal frames from the master forest to the replica forest(s) as part of each commit. The replay produces an equivalent result in each forest, but the forests may not be "bit for bit" identical. (Imagine for example that one forest has been told to merge while the other hasn't.) Commits across replicated forests are synchronous and transactional, so a commit to the master is a commit to the replica.

For more information about local-disk failover see Local-Disk Failover in the *Scalability, Availability, and Failover Guide*.

10.2.1.2 Shared-Disk Failover

Shared-Disk Failover uses a clustered filesystem, such as Veritas or GFS. (The full list of supported clustered filesystems can be found in the Scalability, Availability, and Failover Guide.) Every Data Manager stores its forest data on a SAN that's potentially accessible by other servers in the cluster. Should one D-node server fail, it will be removed from the cluster and another server in the cluster with access to the SAN will take over for each of its forests. The failover Data Managers can read the same bytes on disk as the failed server, including the journal up to the

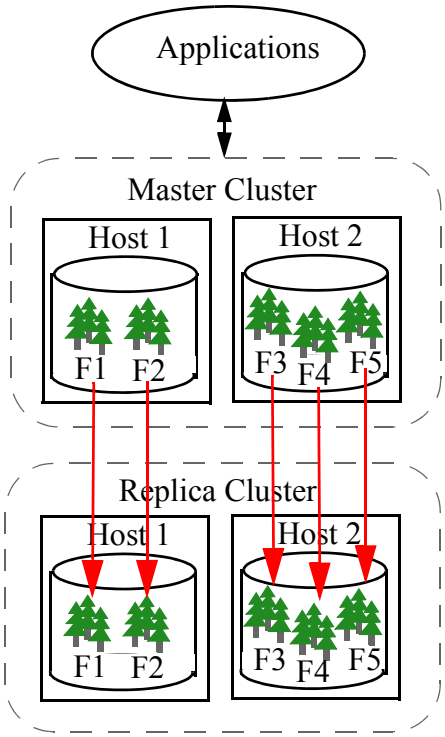
point of failure, with filesystem consistency between the servers guaranteed by the clustered filesystem. As part of configuring each forest, you configure its primary host as well as its failover hosts. It's perfectly legitimate to use an E-node as a backup for a D-node. Instances can switch roles on the fly.

For more information about shared-disk failover see Shared-Disk Failover in the *Scalability, Availability, and Failover Guide*.

10.2.2 Database Replication

Database replication operates at the forest level by copying journal frames from a forest in the Master database and replaying them on a corresponding forest in the foreign Replica database.

As shown in the illustration below, each host in the Master cluster connects to the remote hosts that are necessary to manage the corresponding Replica forests. Replica databases can be queried but cannot be updated by applications.



10.2.2.1 Bulk Replication

Any content existing in the Master databases before Database Replication is configured is bulk replicated into the Replica databases. Bulk replication is also used after the Master and foreign Replica have been detached for a sufficiently long period of time that journal replay is no longer possible. Once bulk replication has completed, journal replication will proceed.

The bulk replication process is as follows:

1. The indexing operation on the Master database maintains a catalog of the current state of each fragment. The Master sends this catalog to the Replica database.

2. The Replica compares the Master's catalog to its own and updates its fragments using the following logic:

    - If the Replica has the fragment, it updates the nascent/deleted timestamps, if they are wrong.
    - If the Replica has a fragment the Master doesn't have, it marks that fragment as deleted (it likely existed on the Master at some point in the past, but has been merged out of existence).
    - If the Replica does not have a fragment, it adds it to a list of missing fragments to be returned by the Master.

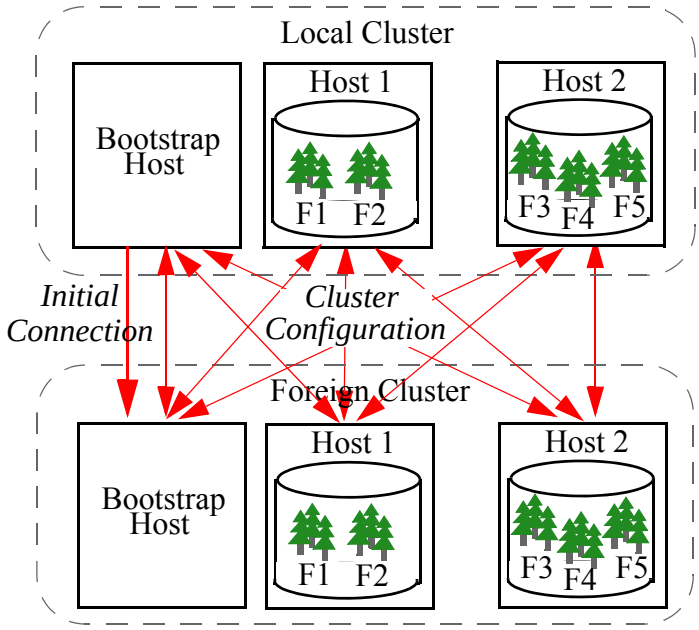
3. The Master iterates over the list of missing fragments returned from the Replica and sends each of them, along with their nascent/deleted timestamps, to the Replica where they are inserted.

For more information on fragments, see the Fragments chapter in the *Administrator's Guide*.

10.2.2.2 Bootstrap Hosts

Each cluster in a Database Replication scheme contains one or more bootstrap hosts that are used to establish an initial connection to foreign clusters it replicates to/from and to retrieve more complete configuration information once a connection has been established. When a host initially starts up and needs to communicate with a foreign cluster, it will bootstrap communications by establishing a connection to one or more of the bootstrap hosts on the foreign cluster. Once a connection to the foreign cluster is established, cluster configuration information is exchanged between all of the local hosts and foreign hosts.

For details on selecting the bootstrap hosts for your cluster, see Coupling Clusters in the *Administrator's Guide*.



10.2.2.3 Inter-cluster Communication

Communication between clusters is done using the intra-cluster XDQP protocol on the *foreign bind port*. A host will only listen on the foreign bind port if it is a bootstrap host or if it hosts a forest that is involved in inter-cluster replication. By default, the foreign bind port is port 7998, but it can be configured for each host, as described in Changing the Foreign Bind Port in the *Database Replication Guide*. When secure XDQP is desired, a single certificate / private-key pair is shared by all hosts in the cluster when communicating with foreign hosts.

XDQP connections to foreign hosts are opened when needed and closed when no longer in use. While the connections are open, foreign heartbeat packets are sent once per second. The foreign heartbeat contains information used to determine when the foreign cluster's configuration has changed so updated information can be retrieved by the local bootstrap host from the foreign bootstrap host.

10.2.2.4 Replication Lag

Queries on a Replica database must run at a timestamp that lags the current cluster commit timestamp due to replication lag. Each forest in a Replica database maintains a special timestamp, called a Non-blocking Timestamp, that indicates the most current time at which it has complete state to answer a query. As the Replica forest receives journal frames from its Master, it acknowledges receipt of each frame and advances its nonblocking timestamp to ensure that queries on the local Replica run at an appropriate timestamp. Replication lag is the difference between the current time on the Master and the time at which the oldest unacknowledged journal frame was queued to be sent to the Replica.

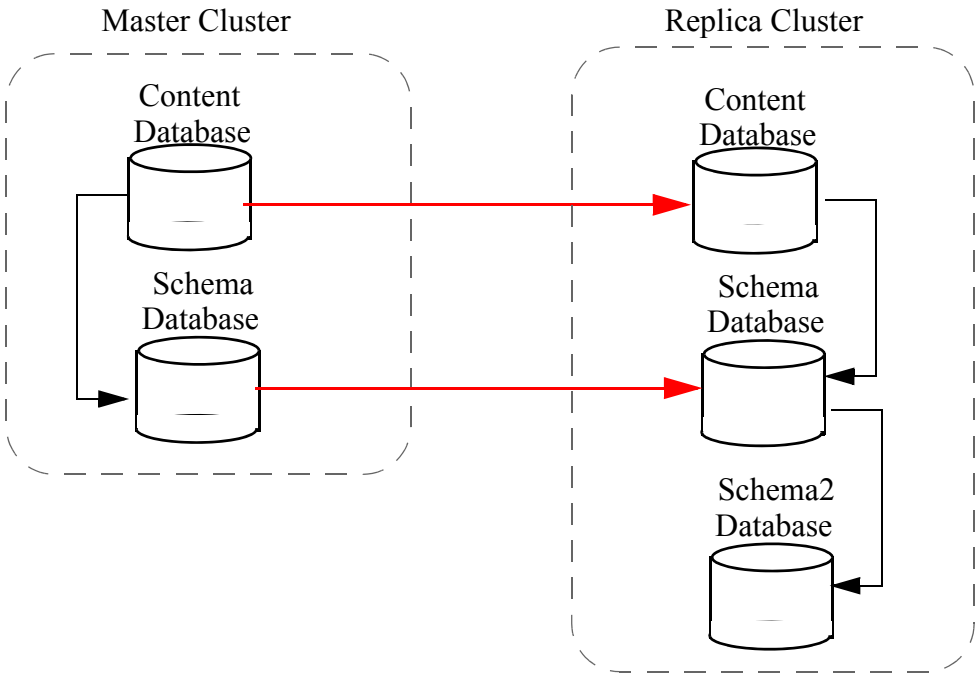
You can set a lag limit in your configuration that specifies, if the Master does not receive an acknowledgement from the Replica within the time frame specified by the lag limit, transactions on the Master are stalled. For the procedure to set the lag limit, see Configuring Database Replication in the *Database Replication Guide*.

10.2.2.5 Master and Replica Database Index Settings

Indexing information is not replicated by the Master database and is instead regenerated on the Replica system. If you want the option to switch over to the Replica database after a disaster, the index settings must be identical on the Master and Replica clusters.

If you need to update index settings after configuring Database Replication, update the settings on the Replica database before updating those on the Master database. This is because changes to the index settings on the Replica database only affect newly replicated documents and will not trigger reindexing on existing documents. Changes to the index settings on the Master database will trigger reindexing, after which the reindexed documents will be replicated to the Replica. When a Database Replication configuration is removed for the Replica database (such as after a disaster), the Replica database will reindex, if necessary.

The process of recreating index information on the Replica database requires information located in the schema database. You cannot replicate a Master database that acts as its own schema database. When replicating a Master schema database, create a second empty schema database for the Replica schema database on the Replica cluster. The diagram below shows the Schema2 database as the schema database from the Replica schema database.



10.3 Flexible Replication

Flexible Replication is an asynchronous (non-transactional), single-master, trigger-based, document-level, inter-cluster replication system built on top of the Content Processing Framework (CPF) described in “Content Processing Framework (CPF)” on page 58. With the Flexible Replication system active, any time a document changes it causes a trigger to fire, and the trigger code makes note of the document's change in the document's property sheet. Documents marked in their property sheets as having changed will be transferred by a background process to the replica cluster using an HTTP-friendly protocol. Documents can be pushed (to the replica) or pulled (by the replica), depending on your configuration choice.

Flexible Replication supports an optional plug-in filter module. This is where the flexibility comes from. The filter can modify the content, URI, properties, collections, permissions, or anything else about the document as it's being replicated. For example, it can split a single document on the master into multiple documents on the replica. Or it can simply filter the documents, deciding which documents to replicate and which not to, and which documents should have only pieces replicated. The filter can even wholly transform the content as part of the replication, using something like an XSLT stylesheet to automatically adjust from one schema to another.

Flexible Replication has more overhead than journal-based intra-cluster replication. It supports sending approximately 250 documents per second. You can keep the speed up by increasing task server threads (so more CPF work can be done concurrently), spreading the load on the target with a load balancer (so more E nodes can participate), and buying a bigger network pipe between clusters (speeding the delivery).

For more information about Flexible Replication, see the *Flexible Replication Guide* and the flexrep Module API documentation.

10.4 Query-Based Flexible Replication

Query-Based Flexible Replication (QBFR) combines Flexible Replication with Alerting to enable customized information sharing between hosts communicating across disconnected, intermittent, and latent networks.

The purpose of Alerting is to notify a user when new content is available that matches a predefined query associated with that user. Combining Alerting with Flexible Replication allows you to replicate documents to select replica targets, only if those documents contain content that matches that target’s predefined queries and users. When new content is inserted or updated in the Master database, it is replicated only to replica targets configured with the matching query/user Alerting criteria.

A user can have more than one alert, in which case they would receive documents that match any of their alerts. In addition to queries, the permissions for a user are taken into account. The user will only receive replicated content that they have permission to view in the database. If the permissions change, the replica will be updated accordingly. Most often QBFR is a pull configuration, but it can also be set up as a push configuration.

By setting up alerts, replication takes place any time content in the database matches that query. Any new or updated content within the domain scope will cause all matching rules or alerts to perform their corresponding action. QBFR can be used with filters to share specific documents or parts of documents. Filters can be set up in either a push or pull configuration.

For more information about Query-Based Flexible Replication, see Configuring Alerting With Flexible Replication in the *Flexible Replication Guide*.

11.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

12.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: November 18, 2019

COPYRIGHT

© 2019 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.