MarkLogic®

Inside MARKLOGIC SERVER

Jason Hunter & Mike Wooldridge

Covers MarkLogic 8 · April 2016

PREFACE

This book describes the MarkLogic Server internals including its data model, indexing system, update model, and operational behaviors. It's intended for a technical audience such as someone new to MarkLogic who wants to understand its capabilities, or someone already familiar with MarkLogic who wants to understand what's going on under the hood.

This book is not an introduction to using MarkLogic Server. For that you can read the official product documentation. Instead, this book explains the principles upon which MarkLogic is built. The goal isn't to teach you to write code, but rather to help you understand what's going on behind your code, and thus help you to write better and more robust applications.

Chapter 1 provides a high-level overview of MarkLogic Server. Chapter 2 explains MarkLogic's core indexes, transactional storage system, multi- host clustering, and various coding options. This is a natural stopping point for the casual reader. Chapter 3 covers advanced indexing as well as topics like bitemporal, semantics, rebalancing, tiered storage, Hadoop integration, failover, and replication. It also discusses the ecosystem of tools, libraries, and plug-ins (many of them open source) built up around MarkLogic.

This third edition of the book adds discussions of features introduced in MarkLogic 7 and 8 including JSON and JavaScript support, semantics, bitemporal, rebalancing and forest assignment policies, tiered storage and super-databases, incremental backup, query-based flexible replication, the Java and Node.js Client APIs, custom tokenization, relevance superboosting, monitoring history, and a new distribute timestamps option. It also adds coverage for a few older features (such as "mild not" and wildcard matching) and expanded coverage on search relevance.

CODE EXAMPLES

What follows is a conceptual exploration of MarkLogic's capabilities—not a book about programming. However, the book does include code examples written in XQuery or JavaScript to explain certain ideas that are best relayed through code. A complete version of the code examples is available for download on GitHub.

When referring to MarkLogic built-in functions, we'll reference the XQuery versions of the functions, e.g., xdmp:document-load(). In most cases, there are equivalent JavaScript versions. To access the function in JavaScript, replace the `":"` with a `"."` and change the hyphenation to camel-cased text, e.g., xdmp.documentLoad().[1]

1 Tip: For fast access to documentation for a function, go to http://docs.marklogic.com/function-name. You don't even need the namespace.

You can find the full set of API documentation, as well as in-depth guides and tutorials, at the MarkLogic Product Documentation (http://docs.marklogic.com). The site includes a robust search feature that's built, naturally, on MarkLogic.

ABOUT THE AUTHORS

Jason Hunter is MarkLogic's CTO Asia-Pacific and one of the company's first employees. He works across sales, consulting, partnerships, and engineering (he led development on MarkMail.org). Jason is probably best known as the author of the book *Java Servlet Programming* (O'Reilly Media) and the creator of the JDOM open source project for Java-optimized XML manipulation. He's an Apache Software Foundation member and former vice president as well as an original contributor to Apache Tomcat and Apache Ant. He's also a frequent speaker.

Mike Wooldridge is a Senior Software Engineer at MarkLogic focusing on front-end application development. He has built some of the software tools that ship with MarkLogic, including Monitoring History, and has authored open-source projects such as MLPHP. He has also written dozens of books for Wiley on graphics software and web design, including *Teach Yourself Visually Photoshop CC* and *Teach Yourself Visually HTML5*. He's a frequent contributor to the MarkLogic Developer Blog and has spoken at MarkLogic World conferences.

Contents

CHAPTER 1

WHAT IS MARKLOGIC SERVER?

MarkLogic Server is the Enterprise NoSQL Database[1] that combines database internals, search-style indexing, and application server behaviors into a unified system. It is a multi-model database management system (DBMS) that combines a document model with a semantic triple model. It stores all of its data within a fully ACID-compliant transactional repository. It indexes the words and values from each of the loaded documents, as well as the document structure. Because of its unique Universal Index, MarkLogic doesn't require advance knowledge of the document structure (its "schema") nor complete adherence to a particular schema. And through its application server capabilities, it's programmable and extensible.

MarkLogic Server (referred to from here on as "MarkLogic") clusters on commodity hardware using a shared-nothing architecture and differentiates itself in the market by supporting massive scale and exceptional performance, all while maintaining the enterprise capabilities required for mission-critical applications.

Let's look at these features in more detail.

> “MarkLogic is a document-centric, multi-model, transactional, search-enabled, structure-aware, schema-agnostic, programmable, high-performance, clustered database server.”

1 NoSQL originally meant "No SQL" as a descriptor for non-relational databases that didn't rely on SQL. Now, because many non-relational systems including MarkLogic provide SQL interfaces for certain purposes, it has transmogrified into "Not Only SQL."

DOCUMENT-CENTRIC

MarkLogic uses documents, often written in XML or JSON, as one of its core data models. Because it uses a non-relational data model and doesn't rely on SQL as its primary means of connectivity, MarkLogic is considered a "NoSQL database." 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, and emails are all naturally modeled as documents. Relational databases, in contrast, with their table-centric data models, can't represent such data as naturally, and so they have to either spread the data out across many tables (adding complexity and hurting performance) or keep the data as unindexed BLOBs or CLOBs.

In addition to XML and JSON, MarkLogic can store text documents, binary documents, and semantic (RDF) triples. Text documents are indexed as if each were an XML text node without a parent. Binary documents are by default unindexed, with the option to index their metadata and extracted contents. Behind the scenes, MarkLogic turns RDF triples into an XML representation and then stores them as XML documents.

MULTI-MODEL

Data models determine how information is stored, documenting real-life people, things, and interactions in an organization and how they relate to one another. With MarkLogic's document-centric model, you can leverage the XML and JSON formats to represent records in ways that are richer than what is possible with relational rows. MarkLogic also stores semantic data, ideal for representing facts about the world, as subject-predicate-object structures known as RDF triples. Triples describe relationships among data entities—for example, a specific person attended a certain university, which is in a certain city, which is in a certain country, and of a certain classification. Triples can be stored side by side with database documents or embedded within documents. By offering both the document model and the semantic model, MarkLogic becomes a multi-model database.

TRANSACTIONAL

MarkLogic stores documents within its own transactional repository. The repository wasn't built on a relational database or any other third-party technology; it was built with a focus on maximum performance.

Because of the transactional repository, you can insert or update a set of documents as an atomic unit and have the very next query be 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 should have the same identifier), *Isolation* (uncompleted transactions are not otherwise visible), and *Durability* (once a commit is made it will not be lost).

ACID transactions are considered commonplace for relational databases, but they're a game changer for document-centric databases and search-style queries.

SEARCH-ENABLED

When people think of MarkLogic, they often think of its text search capabilities. The founding team has a deep background in search. Christopher Lindblad was the architect of Ultraseek Server, while Paul Pedersen was the VP of Enterprise Search at Google. MarkLogic supports numerous search features including word and phrase search, Boolean search, proximity and "mild not," wildcarding, stemming, tokenization, decompounding, case-sensitivity options, punctuation-sensitivity options, diacritic-sensitivity options, document quality settings, numerous relevance algorithms, individual term weighting, topic clustering, faceted navigation, custom-indexed fields, geospatial search, and more.

STRUCTURE-AWARE

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 11 and April 13 where the message contains both "wedding cake" and "empire state building" (case and punctuation insensitive) and where the phrases have to be within 15 words of each other, but the message can't 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 or JSON documents to represent each message and the structure-aware indexing to understand what's an IP, what's a date, what's a subject, and which text is quoted and which isn't, a query like this is actually easy to write and highly performant in MarkLogic. Let's consider some other examples.

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 five times more relevant than body text, and appearance in an abstract is two 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 in reverse chronological order, and locate the last email they sent that had a footer block containing a phone number. Return the phone number.

Retrieve information tracked on two time axes (so-called bitemporal data):
From a database of intelligence, where was our person of interest located on January 1, and how is our knowledge of that person's location different today than it was on January 15?

SCHEMA-AGNOSTIC

MarkLogic indexes the XML or JSON structure it sees during ingestion, whatever that structure might be. It doesn't 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 text as fundamentally the same. At an index level, matching the XPath expression `/a/b/c` can be performed similarly to matching the phrase `"a b c"`. That's the heart of the Universal Index.

Being able to index and query efficiently, without prior knowledge of a schema, provides real benefits with unstructured or semi-structured data where:

1. A schema exists but is either poorly defined or defined but not followed.
2. A schema exists and is enforced at a moment in time but keeps changing over time and may not always be kept current.
3. A schema may not be fully knowable, such as intelligence information being gathered about people of interest where anything and everything might turn out to be important.

It also benefits you when source data is highly structured and where source schemas exist by:

1. Isolating applications from schemas that change over time, even with little or no notice.
2. Making it easy to integrate data from a variety of sources without having to design a master schema that can accommodate all data from all sources.
3. Allowing new data to be added to an existing database without having to redesign the schema for existing data or rebuild the database.

These characteristics make MarkLogic the ideal technology for complex or large-scale data integration projects where data that exists in silos is brought together for operational use.

Of course, MarkLogic also works well with data that does fully adhere to a schema.

You can even use MarkLogic to enforce a schema.[2]

PROGRAMMABLE

To interact with and program MarkLogic Server at the lowest level, you can choose between four W3C-standard programming languages: XQuery, XSLT, JavaScript, and SPARQL. XQuery is an XML-centric functional language designed to query, retrieve, and manipulate XML. XSLT is a style-sheet language that makes it easy to transform content during ingestion and output. JavaScript is a dynamic, object-based language that excels at working with JSON. SPARQL is a SQL-like query language for retrieving semantic data.

MarkLogic lets you mix and match between the languages. XSLT can make in-process calls to XQuery and vice versa, JavaScript modules can import XQuery libraries and access the functions and variables as if they were JavaScript and built-in functions in XQuery and JavaScript let you execute SPARQL queries. MarkLogic also exposes a REST API and a SQL interface over ODBC.

MarkLogic operates as a single process per host, and it opens various socket ports for external communication. When configuring new socket ports for your application to use, you can pick from a variety of protocols:

HTTP and HTTPS Web Protocols

MarkLogic natively speaks HTTP and HTTPS. Incoming web calls can run XQuery, XSLT, or JavaScript programs 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 Wire Protocol

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

REST API

MarkLogic exposes a set of core services as an HTTP-based REST API. Behind the scenes the REST services are written in XQuery and placed on an HTTP or HTTPS port, but they're provided out of the box so users of the REST API don't need to see

2 See http://www.kellblog.com/2010/05/11/the-information-continuum-and-the-three-types-of-subtly-semi-structured-information/ for a deeper discussion of why so much structured information is really semi-structured information.

the XQuery. They provide services for document insertion, retrieval, and deletion; query execution with paging, snippeting, and highlighting; facet calculations; semantic queries using SPARQL; and server administration.

MarkLogic also supports a Java Client API and Node.js Client API. These libraries are wrappers around the REST API and let Java and Node.js developers build applications using the languages with which they are familiar.

SQL/ODBC Access

MarkLogic provides a read-only SQL interface for integration with Business Intelligence tools. Each document acts like a row (or set of rows) with internal values represented as columns.

WebDAV File Protocol

WebDAV is a protocol that lets a MarkLogic repository look like a file system 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 file system. This can be useful for small projects; large projects usually create an ingestion pipeline and send data over XDBC or REST.

HIGH-PERFORMANCE

Speed and scale are an obsession for MarkLogic. They're not features you can add after the fact; they have to be part of the core design. And they are, from the highly optimized native C++ code to the algorithms we'll discuss later. For MarkLogic customers, it's not unreasonable to compose advanced queries across petabytes of data and billions of documents and get answers in less than a second.

> “We selected MarkLogic as the basis for our new Track and Trace system because the proof of concept showed such incredibly fast response times, even at peak loads. It was immediately apparent that the technology is fully scalable and response times will not be unduly affected as we grow to meet the rising demand for online shopping.”
>
> Ad Hermans, IT Director, DHL Parcel Benelux

CLUSTERED

To achieve speed and scale beyond the capabilities of one server, MarkLogic clusters across commodity hardware connected on a LAN. A commodity server can be anything from a laptop (where much development usually happens), to a simple virtualized

instance, all the way up to (in 2016) a high-end box with two CPUs—each with 12 cores, 512 gigabytes of RAM, and either a large local disk array or access to a SAN. A high-end box like this can store terabytes of data.

Every host in the cluster runs the same MarkLogic process, but there are two specialized roles. 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, the same host may act as both a D-node and an E-node. In a single-host environment, that's always the case.

Clustering enables high availability. In the event that an E-node should fail, there's no host-specific state to lose—just the in-process requests (which 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 file system (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).

DATABASE SERVER

At its core, MarkLogic is a database server, but one with a lot of features not usually associated with a DBMS. It has the flexibility to store structured, unstructured, or semi-structured information from a variety of sources together in one place. It can run database-style queries, search-style queries, and semantic queries (or a combination of all three) across all of this data as well as run highly analytical queries. It can also scale horizontally. Built from the ground up, it's a platform that makes it dramatically quicker and easier to integrate heterogeneous data from multiple silos, and to author and deploy real-time information applications or data services.

CHAPTER 2

CORE TOPICS

INDEXING TEXT AND STRUCTURE FOR FAST SEARCH

Now that we've covered what MarkLogic is, let's dig into how it works, starting with its unique indexing model. MarkLogic indexes the words, phrases, structural elements, and metadata in database documents so it can search those documents efficiently. Collectively, this indexed information is known as the Universal Index. (MarkLogic uses other types of indexes as well—for example, range indexes, reverse indexes, and triple indexes. We'll cover those later.)

INDEXING WORDS

Let's begin with a thought experiment. Imagine that I give you printouts of 10 documents. I tell you that I'm going to provide you with a word, and you'll tell me which documents contain the word. What will you do to prepare? If you think like a search engine, you'll create a list of all words that appear across all the documents, and for each word, make a list of which documents contain that word. This is called an *inverted index*; inverted because instead of documents having words, it's words having document identifiers. Each entry in the inverted index is called a *term list*. A term is just a technical name for something like a word. Regardless of which word I give you, you can quickly give me the associated documents by finding the right term list. This is how MarkLogic resolves simple word queries.

Now let's imagine that I ask you for documents that contain two different words. That's not hard, you can use the same data structure. Find all document IDs with the first word and all document IDs with the second, and intersect the lists (see Figure 1). That results in the set of documents with both words.

It works the same with negative queries. If I ask for documents containing one word but excluding those with another, you can use the indexes to find all document IDs with the first word and all document IDs with the second word, then do a list subtraction.

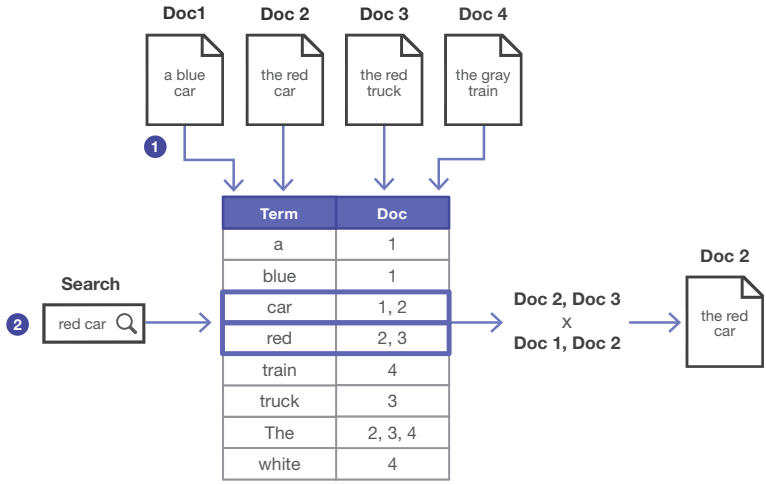

Figure 1: During ingestion, MarkLogic takes the terms in your documents and organizes them into an inverted index (**1**), which maps each term to the documents in which it appears. MarkLogic uses the index to resolve search queries (**2**).

INDEXING PHRASES

Now let's say that I ask you to find documents with a two-word phrase. There are three ways to resolve that query:

1. Use the same data structure you have already. Find documents that contain both words, and then look inside the candidate documents to determine if the two words appear together in a phrase.

2. Add word position information to each of your term list entries. 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.

3. Create new entries in your inverted index where instead of using a simple word as the lookup key, you use a two-word phrase. You make the two-word phrase the "term." When later given a two-word phrase, you can find the term list for that two-word phrase and immediately know which documents contain that phrase without further processing.

Which approach does MarkLogic take? It can use any of these approaches. The choice is made at runtime based on your database index settings. The MarkLogic Admin Interface lists index options that you can enable or disable. You can also configure index options via the REST-based Management API. These settings control what term lists are maintained and if position data should be held about each entry in a term list.

If you have the *fast phrase searches* option enabled, MarkLogic will incorporate two-word terms into its inverted index. With this index enabled, MarkLogic can use the third approach listed above. This makes phrase queries very efficient at the cost of slightly larger indexes on disk and slightly slower performance during document ingestion, since extra entries must be added to the indexes.

If you have the *word positions* option enabled, MarkLogic will use position information to resolve the phrase—the second approach listed above. This index resolution isn't as efficient as *fast phrase searches* because it requires position-matching work, but *word positions* can also support proximity queries that look for words near each other but not necessarily next to each other. *Word positions* also enables "mild not" where you can, for example, search for Mexico but not if it appears in the phrase New Mexico.

FAST PHRASE SEARCHES

Term	Doc
a	1
a blue	1
blue	1, 2
blue car	2, 3
…	…

WORD POSITIONS

Term	Doc:Pos
a	1:1
blue	1:2
car	1:3, 2:3
red	2:2, 3:2
…	…

Figure 2: With *fast phrase searches* turned on, MarkLogic also indexes word pairs. The *word positions* setting indexes information about a term's location in the document.

If neither the *fast phrase searches* nor *word positions* index is enabled, then MarkLogic will use the simple word indexes as best it can and rely on *filtering* the results. Filtering is what MarkLogic calls the act of opening a document and checking to see if it's a true match. Having the indexes off and relying on filtering keeps the indexes smaller but will slow queries proportional to how many candidate documents aren't actually matching documents—that is, how many have to be read as candidates but are thrown away during filtering.

It's important to note that the query results will include the same documents in each case; it's mostly just a matter of performance tradeoffs. Relevance order may differ slightly based on index settings because having more indexes available (as with *fast phrase searches*) will enable MarkLogic to do more accurate relevance calculations, because the calculations have to be made using indexes before pulling documents off disk. We'll talk more about relevance calculation later.

INDEXING LONGER PHRASES

What happens if instead of a two-word phrase I give you a three- or four-word phrase? Again, what would you do on paper? You can choose to rely solely on filtering, you can use position calculation, or you can create a term list entry for all three- and four-word phrases.

It goes beyond the point of diminishing returns to try to maintain a term list for all three- and four-word phrases, so that's not actually an option in MarkLogic. The *fast phrase searches* option only tracks two-word phrases. However, the two-word phrase can still be of some help with longer phrases though. You can use it to find documents that have the first and second words together, the second and third words together, and the third and fourth words together. Documents that satisfy those three constraints are more likely candidates than documents that just have all four words at unknown locations.

If *word positions* is enabled, will MarkLogic use *fast phrase searches* as well? Yes, because the position calculations require time and memory proportional to the number of terms being examined MarkLogic uses *fast phrase searches* to reduce the number of documents, and thus terms, that need to be processed. You'll notice that MarkLogic doesn't require monolithic indexes. Instead, it depends on lots of smaller indexes, turned on and off depending on your needs, cooperating to resolve queries. The usual routine is to look at the query, decide what indexes can help, and use the indexes in cooperation to narrow the results down to a set of candidate documents. MarkLogic can then optionally filter the documents to confirm that the documents actually match, eliminating false positives. The more index options enabled, the tighter the candidate result set can be. You may be able to tune your index settings so that false positives never occur (or occur so rarely that they're tolerable), in which case, you don't need filtering.

INDEXING STRUCTURE

Everything up to this point is pretty much standard search engine behavior. (Well, except that traditional search engines, because they don't hold the source data, can't actually do the filtering and always return the results unfiltered from their indexes.) Let's now look at how MarkLogic goes beyond simple search to index document structure.

Say I ask you to find XML documents that have a `<title>` element within them. What would you do? If you're MarkLogic, you'd create a term list for element `<title>`, the same way you do for a word. You'd have a term list for each element, and no matter what element name gets requested you can deliver a fast answer. (It works similarly for JSON documents. See "Indexing JSON" below for details.)

Of course, not many queries ask for documents containing just a named element. So let's say the question is more complex. Let's try to find XML 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. What would you do? With the simple element term list from above you can find documents having `<book>`, `<metadata>`, and `<title>` elements. That's good, but it doesn't respect the required hierarchical relationship. It's like a phrase query that looks for words without concern for placement.

MarkLogic uses a unique parent-child index to track element hierarchies. It's much like a *fast phrase searches* index except that instead of using adjoining words as the term key it uses parent-child names. There's a term list tracking documents that have a `book/metadata` relationship (that is, a `<book>` as a parent of a `<metadata>`) and another for `metadata/title`. There's even one tracking which documents have any particular root element. Intersecting these three lists produces an even better 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. The index is so useful that it's one MarkLogic always maintains; there's no configuration option to turn it off.

Note that even with the parent-child index, there's still the small potential for documents to be in the candidate set that aren't an actual match. Knowing that somewhere inside an XML 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.

Remember that the goal of index resolution is to make the candidate set so small and accurate that very little filtering is needed.

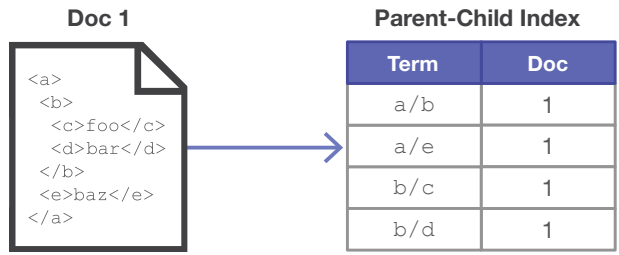

Figure 3: MarkLogic indexes the parent-child relationships of XML elements to enable search based on document structure. (It does the same for the properties in JSON documents.)

INDEXING VALUES

Now what if we want to search for element values? Let's imagine that I ask you for books published in a certain year. In XPath, that can be expressed as `/book[metadata/pubyear = 2016]`. How do we resolve this efficiently?

Thinking back to the paper-based approach, what you want to do is maintain a term list for each XML element value (or JSON property value). In other words, you can track a term list for documents having a `<pubyear>` equal to `2016`, as well as any other element name with any other value you find during indexing. Then, for any query asking for an element with a particular value, you can immediately resolve which documents have it directly from indexes. Intersect that value index with the parent-child structural indexes discussed above, and you've used several small indexes in cooperation to match a larger query. It works even when you don't know the schema or query in advance. This is how you build a database using the heart of a search engine.

Can an element-value index be stored efficiently? Yes, thanks to hashing. Instead of storing the full element name and value, you can hash the element name and value down to a succinct integer and use that as the term list lookup key. Then no matter how long the element name and value, it's actually a small entry in the index. MarkLogic uses hashes behind the scenes to store all term list keys, element-value or otherwise, for the sake of efficiency. The element-value index has proven to be so efficient and useful that it's always and automatically enabled within MarkLogic.

In the above example, `2016` is queried as an integer. Does MarkLogic actually store the value as an integer? By default, no—it's stored as the textual representation of the integer value, the same as it appeared in the document, and the above query executes the same as if `2016` were surrounded by string quotes. Often this type of fuzziness is sufficient. For cases where data type encoding matters, it's possible to use a range index, which is discussed later.

Indexing JSON

How does MarkLogic handle JSON documents when it comes to indexing? Very similarly to how it handles XML. Think of JSON properties as being like XML elements. For indexing purposes, the following JSON:

```
{"person": {"first": "John", "last": "Smith"}}
```

is equivalent in MarkLogic to the following XML:

```
<person><first>John</first><last>Smith</last></person>
```

MarkLogic stores both XML and JSON the same way under the covers—as a hierarchical tree of nodes—and it can index the parent-child relationships found in JSON the same way it indexes those relationships in XML. So enabling fast element word searches and other element-oriented indexes will also index the structure of your JSON documents. As a result, you can use the XPath `/person/last` to retrieve "Smith" from the JSON just as you can the XML.

But there are differences. In XML, all element and attribute values are stored as text, whereas JSON property leaf values can be text, numbers, Booleans, and nulls. MarkLogic indexes JSON numbers, Booleans, and null values in separate type-specific indexes. This has important implications for search. For example, a value search for `"12"` in a JSON property will match the number `"12.00"`, since the values are numeric equivalents. A similar search won't match when the data is stored as XML since, when treated as text, the values `"12"` and `"12.00"` are not equal. (To check for numeric equality in searches of XML, you can use range indexes.)

Another difference is that JSON properties can have an array as a value, for example:

```
{"fruits": ["apple", "banana"]}
```

For indexing purposes, each member of a JSON array is considered a value of the property and is referenced as a distinct text node. As a result, separate searches for those array members—for example, `json-property-value-query("fruits", "apple")` and `json-property-value-query("fruits", "banana")`—will both match. Also, the XPath `/fruits/text()` will match both the `"apple"` and `"banana"` text nodes.

INDEXING TEXT WITH STRUCTURE

Imagine we have a document in our database with the title "Good Will Hunting." What if we're not looking for a title with a specific value but simply one containing a word or phrase, like "Good Will"? The element-value indexes above aren't of any use because they only match full values. We need an index for knowing what words or phrases appear within what elements. That index option is called *fast element word searches*. When enabled, it adds a term list for tracking element names along with the individual words within the element. In other words, for XML documents, it adds a term list entry when it sees a `<title>` with the word "Good," another for a `<title>` with the word "Will," and another for a `<title>` with the word "Hunting." If it's a JSON document, it creates entries for the words associated with the `title` property. Using these indexes, we can additionally confirm that the words "Good" and "Will" are both present in the title of a document, which should improve our index resolution.

The indexes don't know if they're together in a phrase yet. If we want indexes to resolve at that level of granularity, we can use MarkLogic's *fast element phrase searches* and *element word positions* index options. When enabled, the *fast element phrase searches* option maintains a term list for every element and pair of words within the element. It will have a term list for times when a title element or property has the phrase "Good Will" 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 that the words are together in a phrase under a document's title. Whether those indexes provide much benefit depends on how often "Good Will" (or any other queried phrase) appears in a place other than the 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 `xdmp:plan()` to see the constraints for any particular XPath or `cts:search()` expression.

INDEX SIZE

With so many index options, how big are MarkLogic's indexes on disk? Out of the box, with the default set of indexes enabled, the on-disk size can often be smaller than the size of the source XML. That's because MarkLogic compresses the loaded XML, and often the indexes are smaller than the space saved by the compression. With more indexes enabled, the index size might be two or three times bigger than the XML source. Indexes to support wildcarding can expand the index size more than threefold.

MarkLogic supports "fields" as a way to enable different indexing capabilities on different parts of the document. A paper's title and abstract may need wildcard indexes, for example, but the full text may not.

REINDEXING

What happens if you decide to change your index settings after loading content? Just make the changes, and MarkLogic manages the update in the background. It's a key advantage of having all of the data in a transactional repository. When the background reindexing runs, it doesn't stop the system from handling queries or updates; the system just background reloads and reindexes all of the data affected by the change.

If you add a new index option, that index feature is not available to support requests until the reindexing has completely finished because otherwise you'd get an inconsistent view of your data. If you remove an index option, it stops being used right away.

You can monitor the reindexing via the Admin Interface. You can also control the "Reindexer Throttle" from 5 (most eager) to 1 (least eager) or just turn it off temporarily. Pro tip: If you turn off the reindexer and turn off an index, queries won't use the index, but the data relating to the index will be maintained. It's an easy way to test the performance effects with and without a specific index.

RELEVANCE

When performing a full text query, it's not enough to find documents that match 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 important 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. Relevance is covered in more depth later in this book in the "Advanced Text Handling" section.

LIFECYCLE OF A QUERY

Now let's walk through the step-by-step lifecycle of a real query to bring home the indexing topics we've discussed so far. We'll perform a text-centric search that finds the top 10 documents that are most relevant to a set of criteria:

- Document is an article
- Published in the year 2010
- Description contains the phrase "pet grooming"
- The phrases "cat" and "puppy dog" appear within 10 words of each other
- The keyword section must not contain the word "fish"

By default, filtering is turned on. For each result, we'll return a simple HTML paragraph holding the article's title, date, and calculated relevance score. Here's that program expressed in XQuery code:

```
for $result in cts:search(
   /article[@year = 2010],
   cts:and-query((
      cts:element-word-query(
         xs:QName("description"),
         cts:word-query("pet grooming")
   ),
   cts:near-query(
      (cts:word-query("cat"), cts:word-query("puppy dog")), 10
   ),
   cts:not-query(
      cts:element-word-query(
         xs:QName("keyword"), cts:word-query("fish")
      )
   )
)))[1 to 10]
return
<p>{
   <b>{ string($result/title) }</b>
   <i>{ string($result/@date }</i>
   (<small>{ cts:score($result) }</small>)
}</p>
```

MarkLogic uses term lists to perform the search. Exactly what term lists it uses depends on the indexes you've enabled on your database. For simplicity, let's assume that all possibly useful indexes are enabled. In that case, this search uses the following term lists:

A. Root element of `<article>`
B. Element `<article>` with an attribute `year` equal to `2010`
C. Element `<description>` containing `"pet grooming"`
D. Word `"cat"`
E. Phrase `"puppy dog"`
F. Element `<keyword>` containing `"fish"`

MarkLogic performs set arithmetic over the term lists:

(A intersect B intersect C intersect D intersect E) subtract F

This produces a set of document IDs. No documents outside this set could possibly be matches, and all documents within this set are likely matches.

The search has a positional constraint as well. MarkLogic applies it against the candidate documents using the positional indexes. That limits the candidate document set even further to those documents where `"cat"` and `"puppy dog"` appear within 10 words of each other.

MarkLogic then sorts the document IDs based on relevance score, calculated using term frequency data held in the term lists and term frequency statistics. This produces a score-sorted set of candidate document IDs.

At this point MarkLogic starts iterating over the sorted document IDs. Because this is a filtered search, MarkLogic opens the highest-scoring document and filters it by looking inside it to make sure it's a match. If it's not a match, it gets skipped. If it is, it continues on to the return clause.[1]

As each document makes it to the return clause, MarkLogic extracts a few nodes from the document and constructs a new HTML node in memory for the result sequence. After 10 hits (because of the `[1 to 10]` predicate), the search expression finishes and MarkLogic stops iterating.

Now, what if all indexes aren't enabled? As always, MarkLogic does the best it can with the indexes you've enabled. For example, if you don't have any positional indexes, MarkLogic applies the positional constraints during the filter phase. If you don't have *fast element phrase searches* to help with the `<description>` phrase match, MarkLogic uses *fast element word searches* and *element word positions* instead. If you don't have those, it keeps falling back to other indexes like *fast phrase searches*. The less specific the indexes, the more candidate documents to consider and the more documents that might have to be discarded during the filter phase. The end result after filtering, however, is always the same.

INDEXING DOCUMENT METADATA

By now you should have a clear understanding of how MarkLogic uses term lists for indexing both text and structure. MarkLogic doesn't stop there; it turns out that term lists are useful for indexing many other things such as collections, directories, and security rules. That's why it's named the Universal Index.

COLLECTION INDEXES

Collections in MarkLogic are a way to tag documents as belonging to a named group (a taxonomic designation, origin, whether it's draft or published, etc.) without modifying the document itself. 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 set of collections.

Collection constraints are implemented internally as just another term list to be intersected. There's a term list for each collection tracking 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.

1 Previously read term lists and documents get cached for efficiency, a topic discussed later.

DIRECTORY INDEXES

MarkLogic includes the notion of database "directories." They're similar to collections but are hierarchical and non-overlapping and based on the unique document names, which are technically URIs (Uniform Resource Identifiers). 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 a view 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 for each directory 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.

SECURITY INDEXES

MarkLogic's security model also leverages the intersecting term list system. MarkLogic employs a role-based security model where each user is assigned any number of roles, and these roles have associated permissions and privileges. The permissions control what documents the user can read, insert, and update, and the privileges control what actions the user can perform (e.g., restarting the server). The implementation of most security checks is simple: just make sure that the user has the necessary credentials before granting the requested action. So where does the intersecting term list system come in?

MarkLogic uses term lists to manage document reads, because reads support queries and queries have to operate at scale. You don't want to check document read permissions one at a time. Each role has a term list for what documents that role is allowed to see. As part of each query that's issued, MarkLogic combines the user's explicit constraints with the implicit visibility constraints of the invoking user's roles. 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.

MarkLogic can also provide "compartment security." In this system it's not enough that a user have a role that can view a document the user needs to have *all* necessary roles to view a document. Imagine a document that involves several top-secret projects. You can't read it unless you have visibility to all the top-secret projects. Internally, with security, the term lists for role visibility are intersected (ANDed) instead of unioned (ORed).[2]

2 MarkLogic security has been evaluated and validated in accordance with the provisions of the National Information Assurance Partnership (NIAP) Common Criteria Evaluation and Validation Scheme (CCEVS) for IT Security. It's the only NoSQL database to have done so.

PROPERTIES INDEXES

Each document within MarkLogic has an optional XML-based properties sheet. Properties are a convenient place for holding metadata about JSON, binary, or text documents that 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. Depending on configuration settings, MarkLogic can use properties for auto-tracking each document's last modified time.

Properties are represented as regular XML documents, held under the same URI (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 the properties sheet (like finding documents matching a query that were updated within the last hour), MarkLogic uses indexes to independently find the properties matches and the main document matches and does a hash join (based on the URI that they share) to determine the final set of matches. It's not quite as efficient as if the properties values were held within the main document itself, but it's close.

FRAGMENTATION

So far we've use the word *document* to represent each unit of content. That's a bit of a simplification. MarkLogic actually indexes, retrieves, and stores things called *fragments*. The default fragment size is the document, and that's how most people leave it (and should leave it). However, with XML documents, it's also possible to break documents into sub-document fragments through configured *fragment root* or *fragment parent* settings controlled via the Admin Interface or the APIs. This can be helpful when handling a large document 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 a fragment root. You cannot break JSON documents into sub-document fragments.

FRAGMENT VS. DOCUMENT

You can picture using fragmentation on a book. A full book may be the right unit of index, retrieval, and update, but perhaps it's too large. Perhaps in a world where you're 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 a series of fragments—one 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's broken into pieces.

Every fragment acts as its own self-contained unit. It's 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.

There's actually very little difference between fragmenting a book at the chapter level and just splitting each chapter element into its own document as part of the load. That's why people generally avoid fragmentation and just keep each document as its own singular fragment. It's a slightly easier mental model.

In fact, if you see "fragment" in MarkLogic literature (including this book), you can substitute "document" and the statement will be correct for any databases where no fragmentation is enabled.

There's one noticeable difference between a fragmented document and a document split into individual documents: 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()` query construct for more details. Even with this advantage, fragmentation isn't something you should enable unless you're sure you need it.

ESTIMATE AND COUNT

You'll find that you really understand MarkLogic's indexing and fragmentation system when you understand the difference between the `xdmp:estimate()` and `fn:count()` functions, so let's look at them here. Both take an expression and return the number of items matching that expression.

The `xdmp:estimate()` call estimates the number of items using nothing but indexes. That's why it's so fast. It resolves the given expression using indexes and returns how many fragments the indexes see as satisfying all of the term list constraints.

The `fn:count()` returns a number of items based on the actual number of document fragments. This involves indexes and also filtering of the fragments to see which ones truly match the expression and how many times it matches per document. That filtering takes time (due mostly to disk I/O), which is why it's not always fast, even if it's always accurate.

It's interesting to note that the `xdmp:estimate()` call can return results both higher and lower than, as well as identical to, those of `fn:count()`—depending on the query, data schema, and index options. The estimate results are higher when the index system returns fragments that would be filtered away. For example, a case-sensitive search performed without benefit of a case-sensitive index will likely have some candidate

results turn up with the wrong case. But the results might be lower if there happens to be multiple hits within the same fragment. For example, a call to `xdmp:estimate(//para)` by definition returns the number of fragments with at least one `<para>` element rather than the full count of `<para>` elements. That's because the indexes don't track how many `<para>` elements exist within each fragment. There's no visibility into that using just indexes. The `fn:count()` call will actually look inside each document to provide an accurate count.

At scale, `xdmp:estimate()` is often the only tool that makes sense, and a common goal for MarkLogic experts is to craft a system where `xdmp:estimate()` returns answers that would match `fn:count()`. Achieving this requires good index modeling, writing queries that make the most of indexes, and a data model that's amenable to good indexing. When you've achieved that, you can have both fast and accurate counts. It also means that when issuing queries, MarkLogic won't be reading any documents off disk only to then filter them away.

UNFILTERED

You do have some manual control over filtering. The `cts:search()` function takes a large set of options flags, one of which is `"unfiltered"`. A normal `cts:search()` runs filtered and will confirm that each result is a true match before returning it as a result. The "unfiltered" flag tells MarkLogic to skip the filtering step. Skipping the filtering step is so common that the newer `search:search()` function runs unfiltered by default.

THE RANGE INDEX

Now let's take a look at some other index options that MarkLogic offers, starting with the range index. A range index enables you to do many things:

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

2. Perform data-type-aware equality queries. For example, you can compare decimal or date values based on their semantic value rather than their lexical serialized value. (Note that you can perform equality comparisons for numeric and Boolean data types in JSON documents without a range index. See "Indexing JSON" above for details.)

3. 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 how often each sender appears. These are often called *facets* and are displayed with search results to aid search navigation. You can also perform fast aggregates against the extracted values to calculate things like standard deviation and covariance.

4. Perform optimized `order by` calculations. For example, you can sort a large set of product results by price.

5. 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.

6. Perform complex date-time queries on bitemporal documents. Bitemporal documents include four range indexes that track when events occurred in the real world as well as when the events were stored in MarkLogic. Querying the four range indexes and merging the results is key to resolving bitemporal queries. See the "Bitemporal" section for details.

7. 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. This is a more advanced use case, so we won't cover it in this book.

To really understand the role of a range index, imagine that you're at a party. You want to find people born within a week of you. You can yell out each of the particular dates in turn and see who looks at you funny. That's the usual inverted index approach to finding documents containing a fact; you identify the fact and see which documents match. But it doesn't work well for ranges, and it doesn't work well if you want to do some "analytics" and get a list of all the birthdays of everyone at the party. You probably don't want to start yelling out random dates in history. You could walk around and ask everyone his or her birthday—that's the equivalent of a database pulling every stored document off disk and looking inside for its value. It works, but it takes a lot of time. What's more efficient is to keep track of people's birthdays as they enter and exit the party. That way you always have a full list of birthdays that's suitable for many uses. That's the idea behind the range index.

When configuring a range index, you provide three things:

1. The document part to which you want the range index to apply (it can be an element name, an element-attribute name, a JSON property, a path expressed in a form of simplified XPath or a field)

2. A data type (int, date, string, etc.)

3. For strings, a collation, which is a string comparison and 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 ID, get the document's range index value(s); for any range index value, get the document IDs that have that value.

Conceptually, you can think of a range index as implemented by two data structures, written to disk and then memory mapped for efficient access. One can be thought of as an array of structures holding document IDs and values, sorted by document IDs, and the other an array of structures holding values and document IDs, sorted by values. It's not actually this simple or wasteful with memory and disk (in reality, the values are only stored once), but it's a good mental model. With our party example, you'd have a list of birthdays mapped to people, sorted by birthday, and a list of people mapped to birthdays, sorted by person.

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 the values in the range has a document ID associated with it, and 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 partygoers with a birthday between January 1, 1980, and May 16, 1980, you'd 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 of someone at the party, and it's easy to get the people's names because every birthdate 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. Let's say that 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.

Programmers probably think they're using range indexes only via functions such as `cts:element-range-query()`. In fact, range indexes are also used to accelerate regular XPath and XQuery expressions. The XPath expression `/book[metadata/price > 19.99]` looks for books above a certain price and will leverage a decimal (or double or float) range index on `<price>` if it exists. What if there's no such range index? Then MarkLogic won't be able to use any index to assist with the price

constraint (term lists are of no use) and will examine all books with any `<price>` elements. The performance difference can be dramatic.

Figure 4: Conceptually, range indexes are implemented with two data structures. One is an array of values and document IDs, sorted by value. The other is an array of document IDs and values, sorted by document ID. You can use the structures to quickly look up documents associated with a value range (**1**) as well as individual values associated with one or more documents (**2**).

DATA-TYPE-AWARE EQUALITY QUERIES

The same "value to document ID" lookup array can support equality queries that have to be data-type aware. Imagine that you're tracking not just the birthday but the exact time when every person was born. The challenge is that there are numerous ways to serialize the same timestamp value, due to trailing time zone decorations. The timestamps `2013-04-03T00:14:25Z` and `2013-04-02T17:14:25-07:00` are semantically identical. It's the same with the numbers `1.5` and `1.50`. If all of your values are serialized the exact same way, you can use a term list index to match the string representation. If the serialization can vary, however, it's best to use a range index because those are based on the underlying data-type value (but instead of specifying a range to match, you specify a singular value).

To perform a data-type-aware equality query, you can use `cts:element-range-query()` with the `"="` operator, or you can use XPath and XQuery. Consider the XPath mentioned earlier, `/book[metadata/pubyear = 2013]`. Because `2013` is an integer value, if there's a range index on `<pubyear>` of a type castable as an integer, it will be used to resolve this query.

Note that for numeric and Boolean values in JSON documents, you can perform equality comparisons without a range index since those data types are indexed in type-specific indexes (see the section "Indexing JSON" for details). However, you still need range indexes for inequality comparisons.

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. It first uses inverted indexes (and maybe range indexes as above) to isolate the set of document IDs that match a query. It then uses the "document ID to value" lookup array to find the values associated with each document ID. It can be quick because it doesn't touch the disk. It can also count how often each value appears. [3]

The extracted values can be *bucketed* as well. With bucketing, instead of returning counts per value, MarkLogic returns counts for values falling within a range of values. You can, for example, get the counts for different price ranges against source data listing specific prices. You specify the buckets as part of the query. Then, when walking down the range index, MarkLogic keeps track of how many values occur in each bucket.

If you're back at your party and want to find the birthdays of all partygoers who live in Seattle, you first use your inverted index to find the (term) list of 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 partygoer birthdays by month, you can do that with bucketing.

OPTIMIZED "ORDER BY"

Optimized `order by` calculations in XQuery 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 document IDs from the search 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.

Let's say you want to sort Seattle partygoers by age, finding the 10 oldest or youngest. You'd limit your results first to partygoers from Seattle, extract their birthdays, sort by birthday, and finally return the set of people in ascending or descending order.

3 Astute readers will notice that the document IDs passed to the range index for lookup are, in essence, unfiltered. It's a purely index-based operation. For more on extracting values from range indexes see the documentation for `cts:element-values()`, `cts:element-attribute-values()`, and `cts:frequency()`.

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.

USING RANGE INDEXES FOR JOINS

Range indexes turn out to be useful for cross-document joins. Here's the technique: You 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 relatively efficient. (You can perform cross-document joins even more efficiently with RDF triples. See the "Semantics" section for details.)

Understanding this technique requires a code example. Imagine that 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's example XQuery 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() - xs: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 of 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:element-range-query()` constraint and an `author-id` range index for the `cts:element-values()` 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's vastly more efficient than looking up the individual term lists.

USING RANGE INDEXES ON PATHS AND FIELDS FOR EXTRA OPTIMIZATION

Traditional range indexes let you specify the name of an element or element-attribute against which to build a range index. Path range indexes let you be more specific. Instead of having to include all elements or element-attributes with the same name, you can limit inclusion to those in a certain path. This proves particularly useful if you have elements with the same name but slightly different meanings based on placement. For example, the DocBook standard has a `<title>` element, but that can represent a book title, a chapter title, a section title, as well as others. To handle this difference, you can define range index paths for `book/title`, `chapter/title`, and `section/title`. As another example, perhaps you have prices that differ by currency, and you want to maintain separate range indexes. They can be defined using predicates such as `product/price[@currency = "USD"]` and `product/price[currency = "SGD"]`. Path definitions are very flexible. They can be relative or absolute, can include wildcard steps (*), and can even include predicates (the things in square brackets).[4]

The core purpose of path range indexes is to give you more specific control over what goes into a range index. However, they also enable a deeper optimization of XPath. Earlier we looked at the expression `/book[metadata/pubyear > 2010]` and noted how a range index on `<pubyear>` can be used to resolve the query. If there's also a path range index that matches, then because it's more specific, it will be used instead. If there's an integer path range index on `/book/metadata/pubyear`, that range index alone can resolve the full XPath; the term lists aren't really necessary.

4 There are also costs associated with path indexes since it's more work to calculate them and filter with them compared to straight element indexes. If you have a lot of overlapping paths defined, you may see greater index sizes and query times. So if straight element indexes work for you, use them.

Fields are another way to pinpoint what parts of your documents get indexed. They enable you to include (or exclude) different XML elements or JSON properties as a single indexable unit. For example, you can combine `<first-name>` and `<last-name>` elements (but exclude `<middle-name>`) as a field. By adding a range index on the field, you can perform range operations on those combined values. Or maybe you have documents that define last names in different ways, with `<last-name>` and `<lastName>` elements. By creating a range-indexed field for those variations, you can order your search results across these documents by last name even though the markup varies. See the "Advanced Topics" chapter for more on fields.

LEXICONS

When configuring a database, you have the options to configure a "URI lexicon" and "collection lexicon." These are range indexes in disguise. The URI lexicon tracks the document URIs held in the system, making it possible to quickly extract the URIs matching a query without touching the disk. The collection lexicon tracks the collection URIs, letting you do the same with them. Internally, they're both just like any other range index, with the value being the URI. The lexicon retrieval calls can accept a passed-in `cts:query` object constraint, making it easy and efficient to find the distinct URIs or collections for documents matching a query.

Why do we need a range index on these things? Isn't this stuff in the indexes? Not by default. Remember that term list lookup keys are hashes, so while it's possible with the Universal Index to find all documents in a collection (hash the collection name to find the term list and look at the document IDs), it's not efficient to find all collections (hashes are one-way). The lexicons can calculate the document and collection URIs the same way regular range indexes extract values from within documents.

It's also possible to configure a "word lexicon" to track the individual words in a database (or limited to a certain element). For memory efficiency the word lexicon is kept as a flat list of words with no associated document IDs, lest a common word require a large number of sequential entries in the list. You can still pass a `cts:query` object constraint to the word lexicon retrieval call to retrieve only words in documents matching the query. To impose that constraint, MarkLogic pulls each word from the lexicon and checks it, via a quick term list lookup, to see if any document IDs from its term list are in the document IDs matching the query. If so, the word is returned; if not, it's on to the next word. Word lexicons can help with wildcard queries, which will be discussed later.

DATA MANAGEMENT

In the next section, we'll take a look at how MarkLogic manages data on disk and handles concurrent reads and writes.

DATABASES, FORESTS, AND STANDS

MarkLogic organizes data hierarchically, with the database as the logical unit at the top. A system can have multiple databases, and generally will, but each query or update request typically executes against a particular database. (The exception is with a super-database, which lets you query multiple databases as a single unit. That's covered later.) Installing MarkLogic creates several auxiliary databases by default: `App-Services`, `Documents`, `Extensions`, `Fab`, `Last-Login`, `Meters`, `Modules`, `Schemas`, `Security`, and `Triggers`.

A database consists of one or more *forests*.[5] A forest is a collection of documents and is implemented as a physical directory on disk. In addition to the documents, a forest stores the indexes for those documents and a journal that tracks transaction operations as they occur. A single machine may manage several forests, or when in a cluster (acting as an E-node, an evaluator), it might manage none. Forests can be queried in parallel, so placing more forests on a multi-core server will help with concurrency. The optimal number of forests depends on a variety of factors, not all of which are related to performance. On modern server hardware, six primary forests per server (with six replica forests for failover, as discussed later) is a good starting point. The size of each forest will vary depending on if it's a high-performance or high-capacity environment. For high performance (e.g., a user-facing search site with facets), each forest might contain 8 million documents and 100 gigabytes on disk. For high capacity (e.g., an internal analytical application), each forest might contain 150 million documents and 500 gigabytes on disk. In a clustered environment, you can have a set of servers, each managing its own set of forests, all unified into a single database.[6] For more about setting up forests, see Performance: Understanding System Resources.

Each forest is made up of stands. A stand (like a stand of trees) holds a subset of the forest data. Each forest has an in-memory stand and some number of on-disk stands that exist as physical subdirectories under the forest directory. Having multiple stands helps MarkLogic efficiently ingest data. A stand contains a set of compressed binary files with names like `TreeData`, `IndexData`, `Frequencies`, and `Qualities`. This is where the actual compressed XML data (in `TreeData`) and indexes (in `IndexData`) can be found.

5 Why are they called "forests"? Because the documents they contain are stored as tree structures.

6 Is a forest like a relational database partition? Yes and no. Yes because both hold a subset of data. No because you don't typically allocate data to a forest based on a particular aspect of the data. Forests aren't about pre-optimizing a certain query. They instead allow a request to be run in parallel across many forests, on many cores, and possibly many disks.

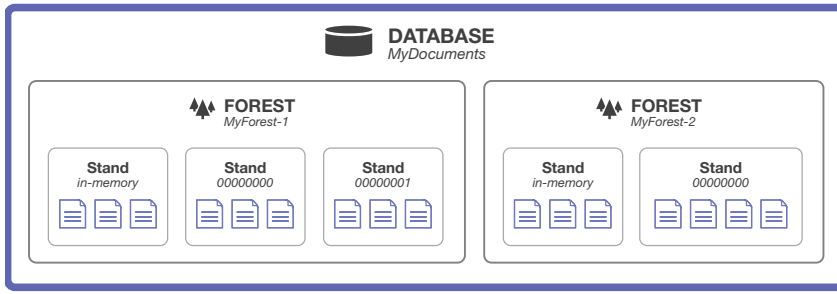

Figure 5: The hierarchical relationship between a database and its forests and stands in MarkLogic.

INGESTING DATA

To see how MarkLogic ingests data, let's start with an empty database having a single forest that (because it has no documents) has only an in-memory stand and no on-disk stands. At some point, a new document is loaded into MarkLogic, through an XCC, XQuery, JavaScript, REST, or WebDAV call. It doesn't matter, the effect is the same. MarkLogic puts this document in an in-memory stand and writes the action to the forest's on-disk journal to maintain durability and transactional integrity in case of system failure.

As new documents are loaded, they're also placed in the in-memory stand. A query request at this point will see all of the data on disk (technically, nothing yet) as well as everything in the in-memory stand (our small set of documents). The query request won't be able to tell where the data is, but it 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 checkpointed, which means it is 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 gets the lovely name `00000000`. That on-disk stand contains all of the data and indexes for the documents loaded thus far. It's written from the in-memory stand out to disk as a sequential write for maximum efficiency. Once it's written, the in-memory stand's allocated memory is freed, and the data in the journal is released.

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 gets written as a new on-disk stand, probably named `00000001` and about the same size as the first. Sometimes, under heavy load, you 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 of the data across all of the stands.

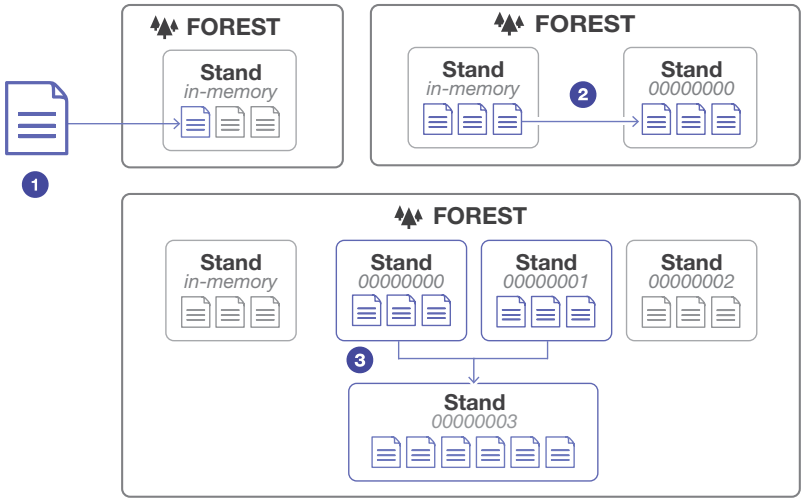

Figure 6: Newly ingested documents are put into an in-memory stand (**1**). When the in-memory stand reaches a certain size, the documents are written to an on-disk stand (**2**). On-disk stands accumulate until, for efficiency, MarkLogic combines them during a merge step (**3**).

MERGING STANDS

The mechanism 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 stand and unify the results. To keep the number of stands to a manageable level where that unification isn't a performance concern, MarkLogic runs *merges* in the background. A merge takes some of the stands on disk and creates a new stand out of them, coalescing and optimizing the indexes and data, as well as removing any previously deleted fragments, which is a topic we'll discuss shortly. After the merge finishes and the new on-disk stand has been fully written, and after all of 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 and an awareness of how much data within each stand is still active versus how much has been deleted. Over time, smaller stands get merged together to produce larger stands, up to a configurable "merge max size" which defaults to 32 gigabytes. Merging won't ever produce a stand larger than that limit, which prevents merge activities from needing a large amount of scratch disk space.[7] Over time, forests will typically accumulate several stands around the merge max size as well as several smaller

7 The "merge max size" behavior described here is new in MarkLogic 7. In older server versions, merging could potentially create a new singular stand of size equal to all other stands put together, which required a lot of free disk space to enable the merge. The new "merge max size" caps the size of any single stand and thus limits how much free disk space is required. A large forest now needs only 50% extra: 0.5 TB of free disk capacity for every 1 TB of on-disk data stored.

stands. Merges tend to be CPU- and disk-intensive, and for this reason, you have control over when merges can happen via system administration.

Each forest has its own in-memory stand and set of on-disk stands. 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.

Document Compression

The `TreeData` file stores XML document data in a highly efficient manner. The tree structure of the document gets saved using a compact binary encoding. The text nodes are saved using a dictionary-based compression scheme. In this scheme, the text is tokenized (into words, white space, and punctuation) and each document constructs its own token dictionary, mapping numeric token IDs to token values. Instead of storing strings as sequences of characters, each string is stored as a sequence of numeric token IDs. The original string can be reconstructed using the dictionary as a lookup table. The tokens in the dictionary are placed in frequency order, whereby the most frequently occurring tokens will have the smallest token IDs. That's important because all numbers in the representation of the tree structure and the strings of text are encoded using a variable-length encoding, so smaller numbers take fewer bits than larger numbers. The numeric representation of a token is a unary nibble count followed by nibbles of data. (A nibble is half a byte.) The most frequently occurring token (usually a space) gets token ID 0, which takes only a single bit to represent in a string (the single bit 0). Tokens 1-16 take 6 bits (two bits for the count (10) and a single 4-bit nibble). Tokens 17-272 take 11 bits (three bits of count (110) and two 4-bit nibbles). And so on. Each compact token ID represents an arbitrarily long token. The end result is a highly compact serialization of XML, much smaller than the XML you see in a regular file.

MODIFYING DATA

What happens if you delete or change a document? 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, effectively removing it from disk.

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. If you consider how many term lists a single document change might affect, updates in place are an entirely inefficient proposition. So, instead, MarkLogic treats any changed document like a new document and treats the old version like a deleted document.

We simplified things a bit here. If you remember, fragments (not documents) are the basic unit of query, retrieval, and update. So if you have fragmentation rules enabled and make a change in a document that has fragments, MarkLogic will determine

which fragments need to change, mark them as deleted, and create new fragments as necessary. Also, MarkLogic has settings that enable it to retain deleted fragments for some guaranteed time period to allow fast database rollback and point-in-time recovery of data.

This approach is known in database circles as MVCC, which stands for *Multi-Version Concurrency Control*. It has several advantages, including the ability to run lock-free queries, as we'll explain.

MULTI-VERSION CONCURRENCY CONTROL (MVCC)

In an MVCC system, changes are tracked with a timestamp number that increments as transactions occur within a cluster. Each fragment gets its own creation-time (the timestamp at which it was created, starting at infinity for fragments not yet committed) and deletion-time (the timestamp at which it was marked as deleted, starting at infinity for fragments not yet deleted). On disk, you'll see these timestamps in the `Timestamps` file. Trivia buffs will note that it's the only file in the stand directory that's not read-only.

For a request that doesn't 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 any 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 two extra contraints to the normal term list constraints. First, that any fragments returned must have been "created at or before the request timestamp" and second, that they must have been "deleted after the request timestamp." It's easy to create from these two primitives what is in essence a new implicit term list of documents in existence at a certain timestamp. That timestamp-based term list is implicitly added to every query. It's a high-performance substitute for having to acquire locks.

POINT-IN-TIME QUERY

Normally a query acquires its timestamp marker automatically based on the time the query started. However, it's also possible for it to perform a point-in-time query at a specific previous timestamp. This lets you query the database as it used to be at any arbitrary point in the past, as efficiently as querying at the present time. One popular use of the feature is to lock the public world at a certain timestamp while new data is loaded and tested. Only when it's approved does the public world timestamp jump to be current again. And of course if it's not approved, you can undo all of the changes back to a past timestamp (what's called a "database rollback").

When doing point-in-time queries, you have to consider merging. Merging normally removes deleted documents. If you want to query in the past to see deleted documents, you need to administratively adjust the *merge timestamp* setting indicating a timestamp before which documents can be reclaimed and after which they can't. That timestamp becomes the point furthest in the past for which you can perform a point-in-time query.

LOCKING

An update request, because it isn't read-only, has to lock affected documents to maintain system integrity while making changes. This lock behavior is implicit and not under the control of the user.[8] Read locks block other operations from trying to write; write locks block both read and write operations. An update has to obtain a read lock before reading a document and a write lock before changing (adding, deleting, or modifying) a document. Lock acquisition is ordered, first-come first-served, and a request waiting on a write lock will block any newly requested read locks for the same resource (otherwise it might never actually get the write lock). Locks are released automatically at the end of a 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, the update farthest along (based on the number of write locks and past retries) wins and the other update gets restarted.

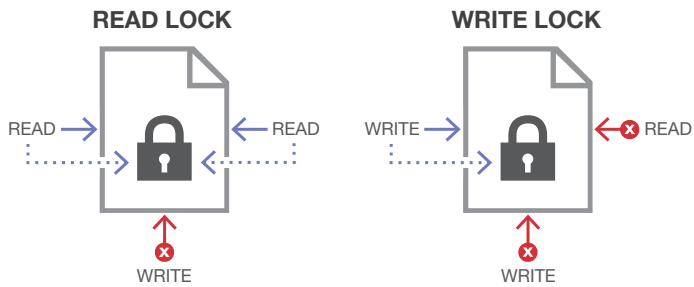

Figure 7: A read lock allows other reads to occur but blocks writes. A write lock blocks both reads and writes. In a system with simultaneous requests, some requests may have to wait their turn to complete.

8 For the most part, it's not under the control of the user. The one exception is an `xdmp:lock-for-update($uri)` call that requests a write-lock on a document URI, without actually having to issue a write and in fact, without the URI even having to exist. Why bother? By explicitly getting a write lock for a document before performing expensive calculations that will be written into that document, you ensure that the calculations won't have to be repeated should deadlock detection have to restart the statement. You can also use this call to serialize execution between statements by having the statements start by getting write locks on the same URI. This can be used to take predicate locks and avoid phantom reads (see "Isolation Levels and MarkLogic").

In XQuery, MarkLogic 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, it's an update. If not, it's a query. Even if at execution time the update doesn't actually invoke the updating function, it still runs as an update. (Advanced tip: There's an `xdmp:update` prolog statement to force a request to be seen as an update. This is useful when your request performs an update from evaluated or invoked code that can't be seen by the static analyzer.) In JavaScript, the system doesn't use static analysis, so any code that performs updates needs to state its intentions with a `declareUpdate()` function.

UPDATES

Locks are acquired during the update execution, yet the actual commit work happens only after the update finishes successfully. If the update exits with an error, all pending changes that were part of that update are discarded. By default, each statement is its own auto-commit transaction. We'll discuss multi-statement transactions later.

During the update request, the executing code can't see the changes it's making. Technically that's because they haven't taken place. Invoking an update function doesn't immediately change the data, it just adds a "work order" to the queue of things to do should the update end successfully.

Philosophically, code can't see the changes it's making because XQuery is a functional language, and functional languages allow the interesting optimization that different code blocks can potentially be run in parallel if the blocks don't depend on each other. If the code blocks can potentially be run in parallel, you shouldn't depend on updates (which are essentially side effects) to have already happened at any point.

Any batch of updates within a statement has to be non-conflicting. The easiest (but slightly simplistic) 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 and then delete the node, 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.

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 doesn't have a dependency on the outer query, but that turns out to be a common case. For example, let's say that you want to execute a content search and record the user's search string to the database for tracking purposes. The database update doesn't need to be in the same transactional context as the search itself, and would slow things down if it were. In this case, it's better to run the search in one context (read-only and lock-free) and the update in a different context.

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

DOCUMENTS ARE LIKE ROWS

When modeling data for MarkLogic, think of documents as 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. There are two reasons for this.

First, locks are managed at the document level. A separate document for each item avoids lock contention. Second, all index, retrieval, and update actions happen at the fragment level. This means that when you're finding, retrieving, or updating an item, it's 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 of the expressivity of the XML and JSON data formats. One document can often describe an entity (a manifest, a legal contract, an email) completely.

LIFECYCLE OF A DOCUMENT

Now, to bring all of this down to earth, let's track the lifecycle of a document from first load to deletion until the eventual removal from disk.

Let's assume that the document starts life with an XML document load request. The request acquires a write lock for the target URI as part of an `xdmp:document-load()` function call. If any other request is already doing a write to the same URI, our load will block for it and vice versa. At some point, when the full update request completes successfully (without any errors that would implicitly cause a rollback), the actual insertion work begins and the queue of update work orders is processed.

MarkLogic starts by parsing and indexing the document contents, converting the document from serialized XML to a compressed binary fragment representation of the XML data model (strictly, the XQuery Data Model[9]). The fragment is added to the in-memory stand. At this point, the fragment is considered a *nascent* fragment—a term you'll see sometimes on the Admin Interface status pages. Being nascent means it exists in a stand but hasn't been fully committed. (On a technical level, nascent fragments' creation and deletion timestamps are both set to infinity, so they can be managed by the system while not appearing in queries prematurely.) If you're doing a large transactional insert, you'll accumulate a lot of nascent fragments while the documents are being processed. They stay nascent until they've been committed.

At some point, our statement completes successfully, and the request is ready to commit. MarkLogic obtains the next timestamp value, journals its intent to commit the transaction, and then makes the fragment available by setting the creation timestamp for the new fragment to the transaction's timestamp. At this point, it's a durable transaction, replayable in the event of server failure, and it's available to any new queries that run at this timestamp or later, as well as any updates from this point forward (even those in progress). As the request terminates, the write lock gets released.

Our document lives for a time in the in-memory stand, fully queryable and durable, until at some point, the in-memory stand fills up and is written to disk. Our document is now in an on-disk stand.

Sometime later, based on merge algorithms, the on-disk stand will get merged with some other on-disk stands to produce a new on-disk stand. The fragment will be carried over, its tree data and indexes incorporated into the larger stand. This might happen several times.

At some point, a new request makes a change to the document, such as with an `xdmp:node-replace()` call. The request making the change obtains a read lock on the URI when it first accesses the document, then promotes the read lock to a write lock when executing the `xdmp:node-replace()` call. If another write lock were already present on the URI from another executing update, the read lock would have blocked until the other write lock released. If another read-lock were already present, the lock promotion to a write lock would have blocked.

Assuming that the update request finishes successfully, the work runs similar to before: parsing and indexing the document, writing it to the in-memory stand as a nascent fragment, acquiring a timestamp, journaling the work, and setting the creation

9 The data model is defined in exhaustive detail at XQuery and XPath Data Model and may differ from the serialized format in many ways. For example, when serialized in XML, an attribute value may be surrounded by single or double quotes. In the data model, that difference is not recorded.

timestamp to make the fragment live. Because it's an update, it has to mark the old fragment as deleted. It does that by setting the deletion timestamp of the original fragment to the transaction timestamp. This combination effectively replaces the old fragment with the new. When the request concludes, it releases its locks. Our document is now deleted, replaced by a new, shinier model.

It still exists on disk, of course. In fact, any query that was already in progress before the update incremented the timestamp, or any point-in-time query with an old timestamp, can still see it. Eventually, the on-disk stand holding the fragment will be merged again, and that will be the end of the line for this document. It won't be written into the new on-disk stand that is, unless the administration "merge timestamp" was set to preserve it. In that case, it will live on, sticking around for use with any subsequent point-in-time queries until finally a merge happens at a timestamp that is later than its deleted timestamp and it gets merged out.

MULTI-STATEMENT TRANSACTIONS

As mentioned above, each statement by default acts as its own auto-commit transaction. Sometimes it's useful to extend transactions across statement boundaries. Multi-statement transactions let you, for example, load a document and then process it, making it so that a failure in the processing will roll back the transaction and keep the failed document out of the database. Or, you can use a Java program to perform multiple sequential database updates, having the actions tied together as a singular unit. Multi-statement transactions aren't only for updates. A multi-statement read-only query transaction is an easy way to group multiple reads together at the same timestamp.

To execute a multi-statement transaction from Java, you call `session.setTransactionMode(Session.TransactionMode.UPDATE)` or `session.setTransactionMode(Session.TransactionMode.QUERY)`. The first puts you in read-write update mode (using locks) and the second puts you in read-only query mode (using a fixed timestamp). Call `session.commit()` or `xdmp:rollback()` to end the transaction. In XQuery, you control the transaction mode using the special `xdmp:set-transaction-mode prolog` or by passing options to `xdmp:eval()`, `xdmp:invoke()`, or `xdmp:spawn()`, and end it with `xdmp:commit()` or `xdmp:rollback()`.

An important aspect of multi-statement transactions is that each statement in the series sees the results of the previous statements, even though those changes aren't fully committed to the database and aren't visible to any other transaction contexts. MarkLogic manages this trick by keeping a set of "added" and "deleted" fragment IDs associated with each update transaction context. These act as overrides on the normal view of the database. When a statement in a multi-statement transaction adds a document, the document is placed in the database as a nascent fragment (having creation and deletion timestamps of infinity). Normal transactions won't see

a document like this, but the statement also adds the new fragment ID to its personal "added" list. This overrules the timestamps view. When a later statement in the same transaction looks at the database, it knows the fragments in the "added" list should be visible even if they're nascent. It's the same for deletion. When a statement in a multi-statement transaction deletes a document, the deletion timestamp isn't updated right away, but it gets added to the "deleted" list. These fragments are hidden from the transaction context even if the timestamp data says otherwise. What about document modification? With MVCC, that's the same as a combination delete/add. The logic is actually pretty simple: if a fragment is in the "deleted" list, then it's not visible; else if it's in the "added" list, then it is visible; else just follow the usual creation/deletion timestamp data for the fragment.[10]

Multi-statement transactions make deadlock handling a little more interesting. With single-statement, auto-commit transactions, the server can usually retry the statement automatically because it has the entire transaction available at the point of detection. However, the statements in a multi-statement transaction from a Java client may be interleaved with arbitrary application-specific code of which MarkLogic has no knowledge. In such cases, instead of automatically retrying, MarkLogic throws a `RetryableXQueryException`. The calling application is then responsible for retrying all of the requests in the transaction up to that point.

Should any server participating in the multi-statement transaction have to restart during the transaction, it's automatically rolled back.

XA TRANSACTIONS

An XA transaction is a special kind of multi-statement transaction involving multiple systems. XA stands for "eXtended Architecture" and is a standard from The Open Group for executing a "global transaction" that involves more than one back-end system. MarkLogic supports XA transactions, enabling transaction boundaries to cross between multiple MarkLogic databases or between MarkLogic databases and third-party databases.

MarkLogic implements XA through the Java JTA (Java Transaction API) specification. At a technical level, MarkLogic provides an open source `XAResource` that plugs into a JTA Transaction Manager (TM) and handles the MarkLogic-specific part of the XA process. The Transaction Manager (provided by your Java EE vendor, not by MarkLogic) runs a typical two-phase commit process. The TM instructs all participating databases to prepare to commit. Each participant responds with whether or not it is ready to commit. If all participants are prepared to commit, the TM instructs them to commit. If one or more participants are not prepared to commit, the TM instructs all participants to roll back. It's the point of view of the TM that decides

10 Multi-statement transactions were introduced in MarkLogic 6, but this is not new code. MarkLogic has had post-commit triggers for years that use the same system.

if the global commit happened or not. There are special handling abilities should the TM fail longer than temporarily at the critical moment, such that the commit state could be ambiguous.

STORAGE TYPES WITHIN A FOREST

In the next short section, we'll see how MarkLogic uses different storage types to maximize performance while minimizing storage costs within the same forest. The "Tiered Storage" section discusses moving whole forests between different storage tiers.

FAST DATA DIRECTORY ON SSDS

Solid-state drives (SSDs) perform dramatically better than spinning hard disks, at a price that's substantially higher. The price is high enough that only deployments against the smallest data sets would want to host all of their forest data on SSDs. To get some of the benefits of SSD performance without the cost, MarkLogic has a configurable "fast data directory" for each forest, which you set up to point to a directory built on a fast filesystem (such as one using SSDs). It's completely optional. If it's not present, then nothing special happens and all the data is placed in the regular "data directory." But if it is present, then each time a forest does a checkpoint or a merge, MarkLogic will attempt to write the new stand to the fast data directory. When it can't because there's no room, it will use the regular data directory. Of course, merging onto the "data directory" will then free up room on the fast data directory for future stands. The journals for the forest will also be placed on the fast data directory. As a result, all of the checkpoints and the smaller and more frequent merges will happen on SSD, improving utilization of disk I/O bandwidth. Note that frequently updated documents tend to reside in the smaller stands and thus are more likely to reside on the SSD.

Testing shows that making 5% to 10% of a system's storage capacity solid state provides good bang for the buck. Note that if you put solid-state storage in a system, it should be SLC flash (for its higher performance, reliability, and life span) and either be PCI-based or have a dedicated controller.

LARGE DATA DIRECTORY FOR BINARIES

Storing binary documents within MarkLogic offers several advantages; it makes them an intrinsic part of your queryable database. The binaries can participate in transactions, and they can be included in transactional database backups. Erroneous updates can be corrected with a fast point-in-time recovery. They can be part of the data replication actions needed to support failover and high availability, and can be secured under MarkLogic's security system.

However, if you remember the merge behavior described above, it's desirable that large binaries not be included in the merge process. It's inefficient and unnecessary to copy large binary documents from one stand to another.

This is why MarkLogic includes special handling for large binaries. Any binary document over a "large size threshold" (by default 1 megabyte, configurable from 32KB to 512MB) receives special handling and gets placed on disk into a separate directory rather than within the stands. By default, it's a directory named `Large` residing next to the stand directories. Under this `Large` directory, the binary documents are placed as regular files, with essentially random names, and handles are kept within the database to reference the separate files. From a programmer's perspective, the binary documents are as much a part of the database as any other. But as an optimization, MarkLogic has persisted them individually into the `Large` directory where they don't need to participate in merges. If the database document should be deleted, MarkLogic manages the deletion of both the handle within the database and the file on disk.

Storing large binaries doesn't require fast disks, so MarkLogic includes a configurable "large data directory" for each forest. It's the opposite of the "fast data directory." It can point to a filesystem that's large but cheaper per gigabyte, with possibly slow seek times and reduced I/O operations per second. Often it's hosted remotely on a NAS or in HDFS. With a "large data directory" configured, MarkLogic will save large binaries there instead of in the `Large` directory next to the stands.

MarkLogic also supports "external binary documents" that behave in many ways like large binary documents, with the difference that the separate files on disk are managed by the programmer. MarkLogic still manages a handle to the files, but the path to the physical file and its creation and deletion are left to the programmer. This is a good choice when deploying to Amazon Web Services and placing large binaries in S3. By hosting them as external binary documents, they can be pulled directly by clients from AWS servers without going through the MarkLogic nodes.

CLUSTERING AND CACHING

As your data size grows and your request load increases, you might hear one of two things from a software vendor. The vendor might tell you that they have a monolithic server design requiring you to buy ever larger boxes (each one exponentially more expensive than the last), or they might tell you they've architected their system to cluster—to take a group of commodity servers and have them operate together as a single system (thus keeping your costs down). MarkLogic? It's designed to cluster.

Clustering provides four key advantages:

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

MarkLogic servers placed in a cluster specialize in one of two roles. They can be Evaluators (E-nodes) or 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, as well as to 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 two 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 to the D-nodes any subexpression of the request involving data retrieval or storage. If the request needs, for example, the 10 most recent items matching 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 recent results for its portion of the data. The E-node coalesces the partial answers into a single unified answer: the most recent items across the full cluster. It's an intra-cluster back-and-forth that 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.

D-nodes, for the sake of efficiency, don't send full fragments across the wire unless they're truly needed by the E-node. When doing a relevance-based search, for example, each D-node forest returns an iterator, within which there's 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.

CLUSTER MANAGEMENT

The MarkLogic Server software installed on each server is always the same regardless of its role. If the server is configured to listen on a socket for incoming requests (HTTP, XDBC, WebDAV, ODBC, etc.), then it's an E-node. If it manages data (has one or more attached forests), then it's a D-node. In the MarkLogic administration pages, you can create named Groups of servers, each of which shares the same configuration, making it easy to have an "E Group" and "D Group." For a simple one-server deployment, such as on a laptop, the single MarkLogic instance does both E-node and D-node duties.

Setting up a cluster is simple. The first time you access the Admin Interface for a new MarkLogic instance, it asks if you want the instance to join a pre-existing cluster. If so, you give it the name of any other server in the cluster and what Group it should be part of, and the new system configures itself according to the settings of that Group. Nodes within a cluster communicate using a proprietary protocol called XDQP running on port 7999.

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'll bump up the caches related to request evaluation, at the expense of those related to data management. For a Data Manager, you'll do the opposite. Here are some of the key caches, what they do, and how they change in a clustered environment:

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 this size, while on an E-node you can set it to the minimum.

Compressed Tree Cache

This cache holds the document fragments after they've been read off disk. They're stored compressed to reduce space and improve I/O efficiency. Reading fragments off disk is solely a D-node task, so again you'll probably want to increase this cache size 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.

You'll want to raise the Expanded Tree Cache size on E-nodes and greatly reduce it on D-nodes. Why not reduce it to zero? D-nodes need their own Expanded Tree Cache as a workspace to support background reindexing. 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 looks first in its local Expanded Tree Cache. If it's not there, it asks the D-node to send it. The D-node looks first in its Compressed Tree Cache. Only if it's not there does the D-node read the fragment off disk to send 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. Also notice that, because of MVCC, the Expanded Tree Cache never needs invalidation in the face of updates.[11]

CACHING BINARY DOCUMENTS

Large and external binary documents receive special treatment regarding the tree caches. Large binaries (those above a configurable size threshold, and thus managed in a special way on disk by MarkLogic) and external binaries (those held externally with references kept within MarkLogic) go into the Compressed Tree Cache, but only in chunks. The chunking ensures that even a massive binary won't overwhelm the cache. These binaries don't ever go into the Expanded Tree Cache. There's no point, as the data is already in its native binary form in the Compress Tree Cache.

Note that external binaries are pulled in from the external source by the E-nodes. This means a system with external binaries should make sure that the E-node has a sufficiently large Compressed Tree Cache.

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. Here's how it works: Any thread making a change to a cache needs to acquire a write lock for the

11 If ever you receive an "XDMP-EXPNTREECACHEFULL: Expanded tree cache full" error, it doesn't necessarily mean you should increase the size of your Expanded Tree Cache. The error message means your query is bringing in so much data that it's filling the memory buffer, and the right fix is usually to rewrite the query to be more efficient.

cache in order to keep the update thread-safe. It's a short-lived lock, but it still has the effect of serializing write access. With only one partition, all threads need to serialize through that single write lock. With two partitions you get what's in essence two different caches and two different locks, and double the number of threads can make cache updates concurrent.

How does a thread know in which cache partition to store or retrieve an entry? It's deterministic based on the cache lookup key (the name of the item being looked up). A thread accessing a cache first determines the lookup key, determines which cache would have that key, and goes to that cache. There's no need to read-lock or write-lock any partition other than the one appropriate for the key.

You don't want to have 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 elect to remove the most stale entries from itself, even if there's a more stale entry in another partition.

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

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 making its caches stand-aware. Stands, if you recall, are the read-only building blocks of forests. Existing stand contents don't change when new documents are loaded, so there's no need for the performance-killing global cache invalidation when updates occur.

LOCKS AND TIMESTAMPS IN A CLUSTER

Managing locks and the coordinated transaction timestamp across a cluster seems like a task that could introduce a bottleneck and hurt performance. Luckily, that's not the case with MarkLogic.

MarkLogic manages locks in a decentralized way. Each D-node is responsible for managing the locks for the documents under its forest(s). It does this as part of its regular data access work. If, for example, 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. The D-nodes don't immediately inform the E-node about their lock actions; it's not necessary. In fact, a D-node doesn't have to check with any other hosts in the cluster to acquire locks on its subset of data. (This is why all fragments for a document are always placed in the same forest.)

In the regular heartbeat communication sent between the hosts in a cluster, each host reports on the transactions upon which it is waiting (due to locks) 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, 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 the latest timestamp it's aware of, and hosts keep track of the maximum across the cluster. Timestamps used to be monotonically increasing integers but now are values closely matching regular time (see `xdmp:timestamp-to-wallclock()` and `xdmp:wallclock-to-timestamp()`). This makes database rollback to a particular wall clock time easier. It also means that it's important for all machines in a MarkLogic cluster to have accurate clocks.

What about the case where two updates happen at about the same time and each set of D-nodes picks the same new timestamp value because they haven't seen the next heartbeat yet? That can happen, and it's actually OK. Both sets of updates will be marked as having happened at the same timestamp. This can only happen if the updates are wholly independent of each other (because otherwise a D-node participating in the update would know about the higher timestamp) and in that case there's no reason to serialize one to look like it happened before the other one. 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.

What about the case where one request performs a write and then very quickly, within the one-second gap in heartbeat communications, another request performs a read-only query to another host that wasn't involved in the first write? In this case the second host may not know about the increased timestamp and the second request could read stale data. (It's important to note, however, that the second request couldn't make an erroneous update based on the stale read because this scenario only happens if the second request is purely read-only.) If this is a concern, you can tweak the *distribute timestamps* app server configuration option. It defaults to "fast," where updates return as quickly as possible and no special timestamp notifications are broadcast to hosts not involved in a transaction. When it's set to "strict," all updates immediately broadcast timestamp notification messages to every other host in the group, and updates do not return until their timestamp has been distributed.

LIFECYCLE OF A QUERY IN A CLUSTER

Earlier we covered the lifecycle of a query. Let's do that again but with an eye toward how the E-nodes and D-nodes communicate and where the caches fit in.

Let's assume that we're running the same `cts:search()` as before—the one that liked cats and puppy dogs but didn't like fish. In order to gather term lists, the E-node pushes a representation of the search expression to each forest in the database in parallel. Each forest on a D-node returns an iterator to its score-sorted results. All the term list selection and intersection happen on the D-node. Locks would happen on the D-node side as well, but since this is a read-only query, it runs lock-free.

The E-node selects from the iterators holding the highest scores and pulls from them to locate the most relevant results from across the database. Over the wire, it receives fragment IDs along with score data. When the E-node retrieves the actual fragment associated with the fragment ID, it looks first to its own Expanded Tree Cache. If it can't find it, it requests the fragment from the D-node from which the fragment ID came. The D-node looks in its own Compressed Tree Cache for the fragment, and if it's not in that cache either, it pulls it off disk.

At this point, the E-node may filter the fragment. It looks at its tree structure and checks if it truly matches the search constraints. Only if it survives the filtering does it proceed to the return clause and become a generated result. If the query is running as unfiltered, no checking occurs.

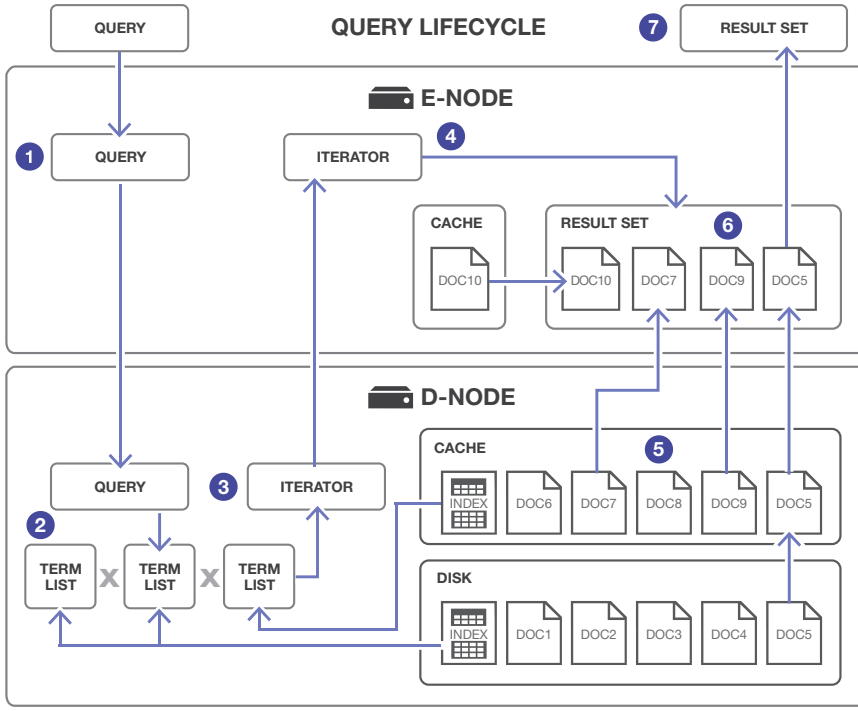

Figure 8: The query is submitted to the E-node and a representation of the query is pushed to the forests in the D-node(s) (**1**). Term list selection and intersection happen in the D-node (**2**). Term lists are retrieved from the cache or from disk. Each D-node returns an iterator pointing to its score-sorted results (**3**). The E-node uses the iterators to retrieve the most relevant results from the D-node(s) (**4**). The D-node retrieves the result fragments from its cache or from disk (**5**). (Fragments retrieved from disk are stored in the cache for future access.) When a document fragment is returned, the E-node can perform filtering to confirm that the fragment is indeed a match for the query (**6**). The result set is returned with results that successfully passed any filtering (**7**).

LIFECYCLE OF AN UPDATE IN A CLUSTER

Earlier we also covered the lifecycle of a document being updated. Let's revisit that scenario to see what happens in more detail and in a clustered environment. This will demonstrate how MarkLogic is ACID compliant, meaning MarkLogic transactions are:

- **Atomic:** Either all of the data modifications in a transaction are performed or none of them are performed.
- **Consistent:** When completed, a transaction leaves all data in a consistent state.
- **Isolated:** Transactions are protected from the impact of other transactions running concurrently.
- **Durable:** After a transaction is committed, its effects remain even in cases of unexpected interruptions. A durable database has mechanisms to recover from system failure.

At the point of the insert call, the request needs to acquire a write-lock. To get the lock, it issues a request to all forests to see if the document already exists. If so, that forest will obtain the write lock (and the new version of the document will be placed back into that forest). If the URI is new, the E-node picks a forest in which it will place the document, and that forest obtains a write lock. By default, the E-node picks the forest in a deterministic way based on how its URI maps to a set of abstract "buckets" (see the "Rebalancing" section for details) so as to ensure that any other concurrent requests look to the same D-node for the write lock. (Different ways of choosing a forest are also possible. See the "Rebalancing" and "Locking with Forest Placement" sections for details.)

By obtaining locks during writes, MarkLogic ensures that updates are *isolated*. Transactions that want to update the same document must wait their turn. Locks also help ensure *consistency*. Without locks, transactions making simultaneous updates to the same documents could leave values in an inconsistent state.

When the update request code completes successfully, the E-node starts the commit work. It parses and indexes the document and sends the compressed document to the D-node forest selected earlier. The D-node updates the in-memory stand, obtains a timestamp, journals the change on disk, and changes the timestamp of the document to make it live, then releases the write lock.

The fact that MarkLogic writes the update to an on-disk journal before making the updated document live helps to ensure *durability*. If a system failure were to wipe out changes that had been made in memory, the data could be reconstituted by replaying what's in the journal.

What happens with an insert or update that involves multiple documents? And what happens when that transaction crosses multiple forests in the same transaction? MarkLogic uses a classic two-phase commit between the D-nodes to ensure that the update either happens *atomically* or doesn't happen at all. See Figure 9 for details.

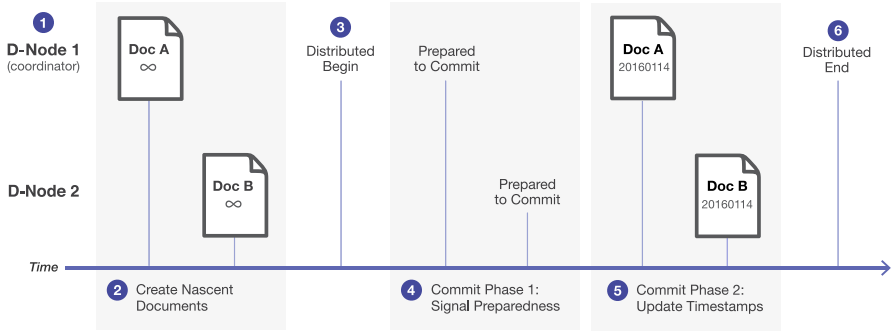

Figure 9: During a commit involving multiple D-nodes, one D-node acts as the coordinator (**1**). First, the system creates the nascent documents (with timestamps set to infinity) (**2**) and the coordinator writes a *distributed begin* entry to its journal (**3**). During the first phase of the commit, participating nodes signal that their updates have occurred and they are prepared to commit (**4**). Only after all hosts confirm does the second phase occur, when timestamps are updated and documents go live (**5**). If any nodes signal that they are unprepared, MarkLogic rolls back the transaction. Finally, the coordinator writes a *distributed end* entry to the journal and the commit is complete (**6**).

Locking with Forest Placement

By default, MarkLogic chooses where to place new documents according to a selected assignment policy. Insertion calls can also request that a document be placed in a specific forest. There are two ways to do this. First, a document insertion call can include forest placement keys, specified as forest IDs (as parameters to the insertion call), to target the document into a particular forest. In this event, locks are managed the same as if there were no placement keys, by the forest selected by the deterministic hash. This ensures that two concurrent forest placements against two different forests will respect the same lock. Second, a document insertion call can execute within an "in-forest eval" (using an undocumented parameter to the `eval()` function) whereby all executed code runs only within the context of the selected forest. With this more advanced technique only the selected forest manages the lock. This executes a bit faster, but MarkLogic doesn't check for duplicates of the URI in other forests. That's left to the application. The Hadoop connector sometimes uses this advanced technique.

ISOLATION LEVELS AND MARKLOGIC

Isolation is the ACID property that lets developers act as if their code were the only code executing against the database. One way a database could achieve this would be to make all transactions serialized, so each transaction waits until the previous transaction completes. However, the performance of such a database is usually intolerable. That's why MarkLogic, like other high-performance databases, runs transactions in parallel. The extent to which one transaction can see another transaction's activities is called the database's "isolation level." There are three potential ways that one transaction could see the activities of another:

Dirty reads. When one transaction is allowed to read a document that another transaction has created or changed but not yet committed. If the second transaction fails and performs a rollback, then the first transaction would have read a document that shouldn't exist.

Non-repeatable reads. When a document is read multiple times during a transaction and shows different values due to concurrent updates made by other transactions.

Phantom reads. When, during the course of a transaction, two identical search queries are executed but the set of documents returned is different, due to another concurrent transaction inserting, updating, or deleting data matching the query.

ANSI SQL defines four *isolation levels* that describe how well a database management system contends with the above issues:

- **Read Uncommitted**: All three phenomena are possible. This is basically no isolation at all.
- **Read Committed**: Dirty reads are not possible, but non-repeatable and phantom reads are.
- **Repeatable Read**: Dirty and non-repeatable reads are not possible, but phantom reads are.
- **Serializable**: None of the three phenomena are possible.

How does MarkLogic fare? For read-only transactions, MarkLogic provides Serializable isolation thanks to MVCC. A read-only transaction in MarkLogic is associated with a timestamp that determines which documents the transaction can see. If a concurrent transaction updates a document as the reads occur, the updated document is assigned a later timestamp. This means that the updated version of the document cannot be seen by the reads.

For update transactions, MarkLogic is Repeatable Read with a coding pattern enabling Serializable. Write locks keep transactions from reading any uncommitted data (preventing dirty reads); read locks keep documents from being changed by

other transactions as the documents are being read (preventing non-repeatable reads). MarkLogic prevents some types of phantom reads by taking read locks on the documents in a search result. This prevents one transaction from removing or updating a document that another transaction has previously searched. However, this does not prevent a transaction from *inserting* a document into a results set that another transaction previously searched. That's the one loophole.

How can a system guard against every possible phantom read? Usually it's not necessary. But when two transactions have to be fully Serializable, they can each programmatically get a lock on a sentinel document to ensure serialized execution. You can take read locks on a synthetic URI with `fn:doc()` and write locks with `xdmp:lock-for-update()`.

CODING AND CONNECTING TO MARKLOGIC

Now that we've covered MarkLogic's data model, indexing system, update model, and operational behaviors, let's look at the various options you have for programming and interacting with MarkLogic. First we'll look at the server-side language options (XQuery, XSLT, and JavaScript) that enable you to develop single-tier applications on MarkLogic. Then we'll look at multi-tier options that are based on communicating with MarkLogic's REST API. These include the Java and Node.js Client APIs, which enable you to build standard three-tier applications on MarkLogic.

XQUERY AND XSLT

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.

The server actually speaks three dialects of XQuery:

1.0-ml

The most common language choice. It's a full implementation of XQuery 1.0 with MarkLogic-specific extensions to support search, update, try/catch error handling, and other features that aren't in the XQuery 1.0 language.

1.0

Also called "strict." It's an implementation of XQuery 1.0 without any extensions, provided for compatibility with other XQuery 1.0 processors. You can still use MarkLogic-specific functions with 1.0 but you have to declare the function namespaces yourself, and of course those calls won't be portable. Language extensions such as the try/catch error handling aren't available.

0.9-ml

Included for backward compatibility. It's based on the May 2003 XQuery pre-release specification, which MarkLogic implemented in the years prior to XQuery 1.0.

At the top of each XQuery file, you have the option to declare the dialect in which the file is written. Without that declaration, the application server configuration determines the default dialect. It's perfectly fine to mix and match dialects in the same program. In fact, it's very convenient. It lets new programs leverage old libraries, and old programs use newly written libraries.

In addition to XQuery, you have the option to use XSLT. You also 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.

For more information see the Application Developer's Guide and the XQuery and XSLT API Documentation.

JAVASCRIPT

MarkLogic also includes server support for JavaScript, an increasingly popular language that started as the programming language of web browsers. In recent years, JavaScript has expanded its footprint to server-side platforms such as Node.js. MarkLogic's native JavaScript support expands that familiarity one level deeper into the database.

To support JavaScript, MarkLogic integrates the Google V8 JavaScript engine, a high-performance, open source C++ implementation of the language. JavaScript programs written in MarkLogic have access to:

- Built-in MarkLogic JavaScript functions for managing data and performing server administration
- Custom JavaScript and XQuery libraries stored in MarkLogic
- MarkLogic's XQuery libraries

You import XQuery and JavaScript libraries into your programs using the `require()` function, which follows the CommonJS standard of module imports.

One important note: Most JavaScript functions that read data from the database return an instance of a ValueIterator, rather than a full instantiation of the data. This allows the evaluation environment to lazily and asynchronously load data as it's required. The ValueIterator interface implements the ECMAScript 6 `Iterator` interface. As with any iterator, you can loop through a ValueIterator using a `for...of` loop. This lets you efficiently stream search results in an application.

For more information about JavaScript programming in MarkLogic, see the JavaScript Reference Guide.

MODULES AND DEPLOYMENT

XQuery, XSLT, and JavaScript 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` or `.sjs` scripts, or `.xslt` templates) under a filesystem directory and you're done. Putting code in a database, on the other hand, gives you some deployment conveniences. In a clustered environment, it's easier to make sure that every E-node is using the same codebase, because each file exists just 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 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, XSLT, or JavaScript code. MarkLogic does, however, maintain a "module cache" to optimize repeated execution of the same code.

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.
- JSON, the JavaScript Object Notation format common in Ajax applications. It's easy to translate between XML and JSON. MarkLogic includes built-in translators.
- 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 PDFs.
- 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 recommend you use MarkLogic's Office Toolkits.
- Adobe InDesign and QuarkXPress. Like Microsoft Office, these publishing formats use native XML formats.

SINGLE-TIER WEB DEPLOYMENT

MarkLogic includes a native web server with built-in SSL support. Incoming web requests can invoke XQuery, XSLT, or JavaScript code the same way other servers invoke PHP, JSP, or ASP.NET scripts.

MarkLogic includes a set of built-in functions that scripts use to handle common web tasks: fetching parameters, processing file uploads, reading request headers, writing response headers, and writing a response body, as well as niceties like tracking user sessions, rewriting URLs, and scripting error pages.

MarkLogic's built-in HTTP/HTTPS server gives you the option to write a full web site (or set of data-centric REST endpoints) as a single-tier application. Is that a good idea? It has some advantages:

- It simplifies the architecture. There are fewer moving parts.
- There's no impedance mismatch. When your back end holds structured markup and you need to produce structured markup for the front end, it's terrific to have a language in between that's built for processing structured markup. Web programming gets a lot easier if you don't have to model tables into objects, then turn around and immediately model objects as markup.
- The code runs closer to the data. A script generating a web response can make frequent small requests for back-end data without the usual cross-system communication overhead.
- It's easy to create well-formed, escaped HTML output. Because XQuery and XSLT speak XML natively, they see web output as a structured tree, not just a string, and that gives you some real advantages. Everything is naturally well formed and properly escaped. In contrast, with PHP, you have to manage your own start and end tag placement and call the escape function `htmlspecialchars($str)` any time you output a user-provided string in HTML body text. Any time you forget, it opens the door to Cross-Site Scripting vulnerabilities. There's nothing to forget with XQuery or XSLT.

For more information, see Application Programming in XQuery and XSLT and Server-Side JavaScript in MarkLogic. Single-tier architecture isn't necessary, of course. Maybe you want to fit MarkLogic into a multi-tier architecture.

REST API FOR MULTI-TIER DEVELOPMENT

What if you don't want to program XQuery, XSLT, or JavaScript on the server? For that, MarkLogic provides a standard REST web service interface. The REST API exposes the core functionality required by applications connecting to MarkLogic: document insertion, retrieval, and deletion; query execution with paging, snippeting, and highlighting; facet calculations; and server administration.

You construct a REST API endpoint the same as one for HTTP or XDBC. You then communicate with it by making remote web calls from any client language (or program) you like. MarkLogic provides and supports libraries for Java and Node.js. The libraries are wrappers around the REST API, hiding the underlying network calls and data marshaling from the developer. Other languages can communicate directly to REST or use one of several community-developed open source libraries.

An advantage to the REST API is that it lets you keep your favorite IDE, web framework, unit testing tools, and all the rest. If you want features beyond what the REST API provides, it includes an extensibility framework. For that, you (or someone) will need to write a bit of custom XQuery or JavaScript.

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

JAVA CLIENT API

The Java Client API provides out-of-the-box data management, query, aggregation, and alerting for MarkLogic. It encapsulates performance and scalability best practices into a comprehensive API that is familiar and natural to an enterprise Java developer. The API has hooks for tapping into Java's rich I/O ecosystem to work efficiently with native JSON and XML in MarkLogic, or with plain old Java objects (POJOs) in Java.

For applications that require the extra performance benefits of running server-side code, the Java Client API provides an RMI-like extensibility mechanism and APIs to manage server-side libraries from Java. This hook allows developers to package JavaScript or XQuery code that runs in the database, close to the data.

The Java API co-exists with Java XCC, described below, which provides a lower-level interface for running remote or ad hoc XQuery or JavaScript calls.

Use of the Java Client API is demonstrated in the MarkLogic Samplestack project. Samplestack is a three-tier reference architecture that uses Java as a middle tier (the project also includes a middle tier based on the Node.js Client API).

The Java Client API is being developed by MarkLogic in the open at GitHub. You can find more information in the Java Application Developer's Guide.

NODE.JS CLIENT API

Node.js is an open source runtime environment for developing middle-tier applications in JavaScript. Node.js uses an event-driven, non-blocking I/O model that makes it useful for writing scalable web applications.

Like the Java client, the Node.js Client API is a wrapper around the MarkLogic REST API. It follows best practices on both the Node and MarkLogic ends—for example, enforcing entirely asynchronous I/O and providing conveniences for building complex queries that leverage MarkLogic's rich indexes. The Node.js client can be integrated into a project using npm, the standard method for importing external Node.js libraries.

Below is a typical Node.js client operation, one that reads a document from the MarkLogic database and writes the content to the console:

```
db.documents.read('/example.json').result(
   function(documents) {
      documents.forEach(function(document) {
         console.log( JSON.stringify(document) );
      });
   },
   function(error) {
      console.log( JSON.stringify(error) );
   }
);
```

Node.js's asynchronous programming model means that, instead of a program blocking execution while it waits for a response from the database, a Node.js program defines a callback function (passed as the first argument to `result()` above) to be executed when the response returns. This allows program execution to continue and for many database operations to be executed efficiently in parallel.

Benefits to using the Node.js Client:

- It's useful to developers familiar with Node.js who want to integrate MarkLogic into multi-tier JavaScript web applications or interact with MarkLogic via the Node.js command line.
- It encapsulates the complexity associated with HTTP REST calls so developers can focus on MarkLogic functionality. This includes support for digest authentication and connection pooling.
- It includes built-in support for promises, object streams, and chunked streams.
- It offers automatic serialization and deserialization between JSON documents and JavaScript objects.

The Node.js Client API is also being developed in the open at GitHub. You can find more information in the Node.js Application Developer's Guide.

SEARCH AND JSEARCH APIS

The Search API is a code-level XQuery library designed to make creating search applications easier. Used by the REST API, it combines searching, search parsing, an extensible/customizable search grammar, faceting, snippeting, search term completion, and other search application features into a single API. Even those who are expert in `cts:query` constructs can appreciate its help in transforming user-entered strings into `cts:query` hierarchies. For more information, see the Search API documentation.

The JSearch API is a JavaScript library that makes it straightforward for JavaScript developers to execute complex searches in MarkLogic. JSearch includes many of the same features available through the Search API and `cts:` functions but also returns results as JavaScript objects, lets you chain your method calls, and offers convenience methods for snippet generation and faceting. For more information, see Creating JavaScript Search Applications.

SPARQL FOR QUERYING TRIPLES

You can access semantic data in MarkLogic with SPARQL, which is the standard language for querying for RDF triples. (For more information about semantic features, see the "Semantics" section.) SPARQL enables you to describe RDF triple patterns and then return the triples that match those patterns. Its syntax is similar to that of SQL. For example, the following SPARQL query returns a list of people born in Paris:

```
SELECT ?s
WHERE {
   ?s <http://dbpedia.org/ontology/birthPlace/>
   <http://dbpedia.org/resource/Paris>
}
```

In MarkLogic, you can query using SPARQL in various ways:

- Query Console can evaluate SPARQL directly, just like it can evaluate XQuery and JavaScript.
- You can use server-side XQuery and JavaScript semantic functions, which accept SPARQL queries as arguments.
- You can submit SPARQL queries to semantic endpoints to access triple data via the REST API.

MarkLogic also supports SPARQL Update operations. SPARQL Update is a separate standard that lets you manage collections of semantic data (semantic graphs) by inserting and deleting RDF triples.

For more information about SPARQL and MarkLogic, see the Semantics Developer's Guide.

XDBC/XCC FOR JAVA ACCESS

The XDBC wire protocol provides programmatic access to MarkLogic from Java using an open source Java client library named XCC. The XCC client library provides you with Java objects to manage connections and sessions, run code invocations, load content, pull result streams, and generally interact with MarkLogic. It will feel pretty natural to those who are familiar with JDBC or ODBC.

The XDBC wire protocol by default is not encrypted, but you can layer SSL on top of XDBC to secure the protocol across untrusted networks.

MarkLogic 7 introduced the ability to run XDBC over HTTP by turning on a special `xcc.httpcompliant` configuration option in the client. This lets XDBC traffic flow through standard web load balancers. The load balancers can monitor the E-nodes for health as well as perform session affinity (whereby repeated requests from the same client go to the same E-node—an important requirement for multi-statement transactions where all requests in the transaction have to use the same E-node). The singular downside to running XDBC over HTTP is that HTTP provides no way for the server to call back to the client to resolve XML external entities.

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

WEBDAV: REMOTE FILESYSTEM ACCESS

WebDAV provides yet another 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 of the major operating systems include built-in WebDAV clients, though third-party clients are often the more robust. Note that WebDAV can manage thousands of documents well, because that's what WebDAV clients expect, but larger data sets will cause them problems.

Some developers use WebDAV for managing XQuery, XSLT, or JavaScript 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 MarkLogic system using a local editor.

WebDAV doesn't include a mechanism to execute XQuery, XSLT, or JavaScript code; it's just for file transport. For more information, see WebDAV Servers.

SQL/ODBC ACCESS FOR BUSINESS INTELLIGENCE

You might be surprised that MarkLogic offers a SQL/ODBC interface. After all, the "S" in SQL stands for Structured, so how does that work against a document-oriented database? And why do it, anyway?

The purpose of the SQL system is to provide a read-only view of the database suitable for driving a Business Intelligence (BI) tool—such as IBM Cognos, Tableau, or MicroStrategy—that expects databases to adhere to the relational model. The "tables" accessed in MarkLogic are fully synthetic; everything remains as documents. The tables are constructed as "views" placed atop range indexes. If, for example, your database stores financial trades with configured range indexes on the date, counterparty, and value elements, then you can have those values as your columns. There's a special column (whose name matches the name of the view) that represents the full document data. It's possible to use that special column to specify constraints against the full document data via the SQL MATCH operator. This feature looks small but has big implications because it lets you limit the retrieved rows based on any part of the backing document, even parts that aren't exposed in the SQL view.

Everything executes out of memory. The system is similar in some ways to columnar databases in that columns are stored together rather than rows, and the tuples are created as part of the query using an n-way co-occurrence. The SQL version is SQL92 as implemented in SQLite with the addition of SET, SHOW, and DESCRIBE statements.

MarkLogic exposes SQL via ODBC, a standard C-based API for accessing relational databases. (For testing you can use the XQuery `xdmp:sql()` function or an MLSQL command-line tool.) The ODBC driver is based on the open source PostgreSQL ODBC driver.

Doing data modeling for a relational view on top of a document back-end requires some consideration. If a document has multiple values for a single range index (which isn't possible in normal relational modeling), it has to be represented as a cross-product across multiple rows. If a range index value is missing or invalid, there's a configuration option for how to handle that. If the view defines the column as "nullable," then the given column value is shown as null. If it's not "nullable," the document doesn't match the required constraints and produces no rows.

For more information, see the SQL Data Modeling Guide.

QUERY CONSOLE FOR REMOTE CODING

Not actually a protocol unto itself, but still widely used by programmers wanting raw access to MarkLogic, is the Query Console web-based code execution environment. A web application included with the server, it's fundamentally just a set of XQuery and

JavaScript scripts, that when accessed (password-protected, of course) enables you to run ad hoc code from a text area in your web browser. You can execute scripts written in JavaScript, SPARQL, SQL, and XQuery. It's a great administration and coding tool.

Query Console includes syntax highlighting, multiple open queries that are groupable into workspaces, history tracking, state saved to the server, beautified error messages, and the ability to switch between any databases on the server, and it has output options for XML, HTML, or plain text. It also includes a profiler—a web front end on MarkLogic's profiler API—that helps you identify slow spots in your code and an Explorer to view files in a database. You'll find it here:
`http://yourserver:8000/qconsole`

For more information, see the Query Console User Guide.

CHAPTER 3

ADVANCED TOPICS

ADVANCED TEXT HANDLING

At the start of this book, we introduced MarkLogic's Universal Index and explained how MarkLogic uses term lists to index words and phrases as well as structure. In that section, we only scratched the surface of what MarkLogic can do regarding text indexing. In this section, we'll dig a little deeper.

Note that these indexes work the same as the ones you've already learned about. Each new index option just tells MarkLogic to track a new type of term list, making index resolution more efficient and `xdmp:estimate()` calls more precise.

TEXT SENSITIVITY OPTIONS

Sometimes when querying text, you'll need to specify whether you desire a case-sensitive match. For example, "Polish" and "polish" mean different things.[1] It's easy to specify this as part of your query; just pass a "case-sensitive" or "case-insensitive" option to each query term.[2] The question is, how does MarkLogic resolve these queries?

By default, MarkLogic maintains only case-insensitive term list entries (think of them as having every term lowercased). If you conduct a query with a case-sensitive term, MarkLogic will rely on indexes to find case-insensitive matches and filtering to identify the true case-sensitive matches. That's fine when case-sensitive searches are rare, but when they're more common, you can improve efficiency by turning on the *fast case sensitive searches* index option. This tells MarkLogic to maintain case-sensitive term list entries along with case-insensitive. With the index enabled, case-insensitive terms will use the case-insensitive term list entries, case-sensitive terms will use the case-sensitive term list entries, and all results will resolve quickly and accurately out of indexes.

1 Words like this that have different meanings when capitalized are called "capitonyms." Another example: "March" and "march." Or "Josh" and "josh."

2 If you don't specify "case-sensitive" or "case-insensitive," MarkLogic does something interesting: it looks at the case of your query term. If it's all lowercase, MarkLogic assumes that case doesn't matter to you and treats it as case-insensitive. If the query term includes any uppercase characters, MarkLogic assumes that case does matter and treats it as case-sensitive.

The *fast diacritic sensitive searches* option works the same way, but for diacritic sensitivity. (Diacritics are ancillary glyphs added to a letter, like an umlaut or accent mark.) By default, if you search for "resume," you'll see matches for "résumé" as well. That's usually appropriate, but not in every case. By turning on the diacritic-sensitive index, you tell MarkLogic to maintain both diacritic-insensitive and diacritic-sensitive term list entries, so you can resolve diacritic-sensitive matches out of indexes.[3]

STEMMED INDEXES

Stemming is another situation where MarkLogic provides optimized indexing options. Stemming is the process of reducing inflected (or sometimes derived) words to their root form. It allows a query for "run" to additionally match "runs," "running," and "ran" because they all have the same stem root of "run." Performing a stemmed search increases recall by expanding the set of results that can come back. Often that's desirable, but sometimes not, such as when you're searching for proper nouns or precise metadata tags.

MarkLogic gives you the choice of whether to maintain stemmed or unstemmed indexes, or both. These indexes act somewhat like master index settings in that they impact all of the other text indexes. For example, the *fast phrase searches* option looks at the master index settings to determine if phrases should be indexed as word pairs or stem pairs, or both.

In versions 8 and earlier, MarkLogic by default enables the *stemmed searches* option and leaves *word searches* disabled. Usually that's fine, but it can lead to surprises—e.g., if you're searching for a term in a metadata field where you don't want stemming. For such cases, you can enable the *word searches* index and pass `"unstemmed"` as an option while constructing the search query constraint. In version 9, the settings are reversed, and MarkLogic by default disables *stemmed searches* and enables *word searches*.

MarkLogic uses a language-specific stemming library to identify the stem (or sometimes stems) for each word. It has to be language-specific because words like "chat" have different meanings in English and French and thus the roots are different. It's also possible to create language-specific custom dictionaries to modify the stemming behavior. For details, see the Search Developer's Guide.[4]

3 MarkLogic follows the Unicode definition of what is and isn't a diacritic, which can be important when indexing certain characters. For example, ł (U+0142, "LATIN SMALL LETTER L WITH STROKE") does not have a decomposition mapping in Unicode. Consequently, that character is not considered to have a diacritic.

4 Domain-specific technical jargon often isn't included in the default stemming library. For example, the word "servlets" should stem to "servlet" but doesn't by default.

MarkLogic provides basic language support for hundreds of languages and advanced support—stemming, tokenization (breaking text into words), and collation (sorting)—for 14 languages including English, French, Italian, German, Spanish, Arabic, Persian/Farsi, Chinese (simplified and traditional), Japanese, Korean, Russian, Dutch, Portuguese, and Norwegian. Because it supports Unicode Level 5.2-0, MarkLogic can process, store, and run unstemmed searches against any language represented by Unicode. It identifies text language using `xml:lang` attributes, charset inference, character sequence heuristics, and database default settings in that order.

The stemmed index uses a simple trick to get maximum performance: it bakes the stemming into the Universal Index. Within the stemmed index, MarkLogic treats all stems of a word as if they were the stem root.[5] Any word appearing in text that wasn't a root gets simplified to its root before indexing. That means there's a single term list for both "run" and "ran," based on the stem root "run." At query time, a stemmed search for "ran" also gets simplified to its root of "run," and that root is used as the lookup key to find the right term list. The term list has pre-computed the list of documents with any version of that stem. It's an efficient way for a word to match any other stemmed form, because the index only deals in stem roots.[6]

If both *word searches* and *stemmed searches* are enabled, there will be two differently encoded entries added for each word—one as it actually appears and one for its stem root. The search query option controls which term list to use.

Stemming Options

There are actually a few stemming options. "Basic" indexes the shortest stem of each word, "advanced" indexes all stems of each word, and "decompounding" indexes all stems along with smaller component words of large compound words. Each successive level of stemming improves the recall of word searches but expands the index size.

The "advanced" option is occasionally useful in English, such as for the word "running," which can be used as a verb (stem "run") or a noun (stem "running"). It is more useful for other languages where polysemy (which is when a word has multiple meanings) is more common. For instance, in French, "bois" is a noun that means "woods" (stem "bois") as well as a verb that means "drinks" (stem "boire").

The "decompounding" option only applies to certain languages (such as German, Dutch, Norwegian, Japanese, and Swedish), but for those languages, you really need it. They create new nouns out of sequences of existing nouns, all running together, and you can get poor recall in searches if you don't turn it on.

5 MarkLogic performs inflectional stemming, which keeps the stemmed version as the same part of speech, instead of derivational stemming, which might not. For example, "functioning" (verb) is stored as the stem "function" (verb) but "functional" (adjective) is stored unchanged.

6 Use `cts:stem("running", "en")` to check a word's stem root(s). The second argument is the language code.

TOKENIZATION

MarkLogic sees text not only as a sequence of characters but also as a sequence of tokens. Each token gets a type classification, such as "word," "space," or "punctuation." You can examine the tokenization logic using `cts:tokenize()`:

```
xdmp:describe(cts:tokenize("+1 650-655-2300"), 100)
```

Produces:

```
(cts:punctuation("+"), cts:word("1"), cts:space(" "),
cts:word("650"), cts:punctuation("-"), cts:word("655"),
cts:punctuation("-"), cts:word("2300"))
```

The search functions in the `cts:` namespace operate (as humans do) against tokens rather than characters. This ensures that a search for "230" won't match "2300," nor will a search for "foo" match "food." Space and punctuation tokens are used to split words, but they're ignored in text indexing. That's why a search for the email address "foo@bar.com" gets resolved from indexes the same as a phrase search for "foo bar com." In this case, the space and punctuation tokens come into play during filtering if the query specifies white-space and/or punctuation sensitivity. Similarly, "don't" will not match "dont" even with punctuation-insensitive searches, since the first is three tokens and the second is one token.[7]

Tokenization is language-aware and language-dependent. As English speakers, we split word tokens on space and punctuation, but other languages can be much more complex. Japanese, for example, often has no white space between words. Yet MarkLogic can convert Japanese text to a proper sequence of `cts:word` tokens thanks to a Japanese-aware tokenizer. It can also stem each `cts:word` token thanks to a Japanese-aware stemmer. A major differentiator of MarkLogic is this deep understanding of text.

CUSTOM TOKENIZATION

Beginning with MarkLogic 7, you can override the tokenization of specific characters to achieve different behaviors. Any character can be force-tokenized as a "word," "space," "punctuation," "remove," or "symbol" character.

When marked as "remove," the character is ignored during tokenization and it's read as if the character never appeared in the text at all. For a phone number, you can "remove" any hyphen, space, parenthesis, or plus sign. That way, all phone numbers, no matter how they're written in the document, tokenize down to a simple, easier-to-match singular `cts:word` token. Of course, you'd be crazy to have a character as common

7 Tokenization applies to value queries, too, as does stemming. With stemming turned on, the following returns true: `cts:contains(<text>I do run</text>, cts:element-value-query(xs:QName("text"),"me done ran")`

as a space removed from all text, so when configuring custom tokenization you should have it apply only to certain parts of a document, like a phone number field. Fields are discussed in the "Fields" section.

Now what does "symbol" do? Being a symbol makes the character something special. It's not simply punctuation, because punctuation doesn't appear in the text index. Symbols do. Symbols act like single-character words. They're available in the index, but they're still separate from the words nearby.

To understand the value of symbols, let's look at how Twitter data is processed. Twitter hashtag topic classifications are prefixed with # (the hash sign). Without custom tokenization rules, the hash sign is treated as punctuation and ignored for indexing. That's OK, but it does mean that if you search for "#nasa," MarkLogic will use the indexes to find occurrences of "nasa" and then filter the results to find "#nasa" where the hash sign is present. That may not be efficient. What if we make # into a word character? Then "#nasa" resolves perfectly, exactly as we wanted, but has the unfortunate side effect that a simple search for "nasa" won't match the hashtag. Word characters become an intrinsic part of the word they're modifying, and that's not quite right in this case.

What we really want is # as a symbol. Then the text "#nasa" tokenizes to `(cts:symbol("#"), cts:word("nasa"))`.[8] A search for "nasa" will match, and a search for "#nasa" will resolve perfectly out of indexes.

For more information and examples, see the "Custom Tokenization" section of the Search Developer's Guide.

WILDCARDING

MarkLogic lets you specify text constraints below the level of a token using wildcard characters. An asterisk (*) means that any number of characters will match; a question mark (?) matches exactly one character. For example, a `cts:word-query("he*")` will match any word starting with "he," while `cts:word-query("he?")` will only match three-letter words. Wildcard characters can appear at any location with no restrictions. The search pattern `*ark??g*c` is perfectly legitimate.

Several indexes can speed the performance of wildcard queries. The simplest is the *trailing wildcard searches* index. When enabled, it creates a term list for every sequence of three or more characters at the beginning of words. If "MarkLogic" appears in the source text, this index creates term list entries for "mar*", "mark*", "markl*", etc. (Whether it does this as case-insensitive or also adds case-sensitive entries depends on the separate case-sensitivity index setting.) This trailing wildcard index gives perfect and

8 The `cts:tokenize()` function accepts a language as a second argument and a field name as the third, if you want to experiment.

quick answers for the common case where a search pattern has a single trailing asterisk. Related index settings are the *trailing wildcard word positions* and *fast element trailing wildcard searches*, which track the position and element location of the matched text.

More general-purpose is the *three character searches* index (and its companions, *three character word positions* and *fast element character searches*). These indexes create term list entries for any three-character sequences that appear in source text and thus speed up search patterns containing a sequence of three or more characters. The wildcard query above, `*ark??g*c`, has a three-character sequence ("ark") that can be used to isolate the results. The index also tracks the three-character sequences appearing at the start and end of every word and value. So `cts:word-query("book*ark")` can isolate just documents where "boo" appears at the start of a word, "ark" appears at the end, and "ook" appears somewhere. It can also use the `book*` trailing wildcard index, if it exists. And if positions are enabled, they can be used to ensure that the matching sequences are within the same word.

Is that the best we can do? No. The word lexicon provides an elegant approach to resolving wildcard queries. Remember, the word lexicon tracks all words that appear in the database. By comparing the wildcard pattern to the list of known words in the word lexicon,[9] MarkLogic can build an or-query enumerating all of the words in the database that match the pattern. Full lexicon expansion provides accurate results from indexes (useful if you want accurate facets) but can produce a long list of terms and require a lot of term list lookups to resolve.

Instead of full expansion, the lexicon can be used to deduce the list of three-character prefixes and postfixes for all words in the database that match the pattern, and an or-query can be built out of those patterns. Since most matching words will tend to share prefix and postfix character sequences, this can be more efficient, if though slightly less accurate, than full expansion. This approach requires that the *three character searches* index be enabled.

By default, MarkLogic uses heuristics based on index settings and data statistics to decide how to resolve each wildcarded word query: full lexicon expansion, prefix-postfix expansion, or no expansion (just use character sequences). You can optionally give the server a hint as to how you'd like the wildcard resolved in the word query construction.[10]

9 It's best to configure the word lexicon with the codepoint collation because wildcarding operates against codepoints (i.e., individual characters).

10 To inspect how a wildcard search was performed under the hood, check the query expression with `xdmp:plan()`. Since the heuristics depend on data size, it can be useful to see why your wildcard query was processed in a particular way, and whether that is OK for your application.

If instead of matching documents that contain a pattern you want to simply extract words or values in the database that match the pattern, like in the expansion above, you can do it yourself using functions like `cts:word-match()` and `cts:value-match()`. These operate directly against the lexicon values.

For more information, see the section "Understanding and Using Wildcard Searches" in the Search Developer's Guide.

RELEVANCE SCORING

We mentioned relevance scoring of results earlier but didn't actually talk about how text-based relevancy works. Let's do that now. The mathematical expression of the relevance algorithm is the formula:

> log(term frequency) * (inverse document frequency)

The *term frequency* factor indicates what percentage of words in the document match the target word. A higher term frequency increases relevance, but only logarithmically, so the effect is reduced as the frequency increases. It's multiplied by the *inverse document frequency* (the same as dividing by the document frequency), which normalizes for how commonly the word appears in the full database so that rare words get a boost.

MarkLogic calls this algorithm `"score-logtfidf"`, and it's the default option to a cts:search() expression. Other options are `"score-logtf"`, which does not do the inverse document frequency calculation; `"score-simple"`, which simply counts the number of matching terms without regard to frequency; `"score-random"`, which generates a random ordering and can be useful when doing sampling; and `"score-zero"`, which does no score calculation and thus returns results a bit faster when order doesn't matter.

MarkLogic maintains term frequency information in its indexes and uses that data during searches to quickly calculate relevance scores. You can picture each term list as having a list of document IDs, possibly locations (for things like proximity queries and phrase searches), and also term frequency data. All of the numerical values are written using delta coding to keep them as small as possible on disk. While intersecting term lists, MarkLogic is also performing a bit of math to calculate which results have the highest score, based on the term frequency data and the derived document frequency data.

All leaf-node `cts:query` objects include a weight parameter for indicating the importance of that part of the query. A higher weight means that it matters more; a zero weight means that it doesn't matter for scoring purposes, although it still has to be satisfied. Using weights provides a way for certain terms to be weighted over other terms, or certain placements (such as within a title) to be weighted over other placements (such as within body text).

Term weights are specified as `xs:double` values ranging from -64.0 to +64.0, with a default of +1.0. Their effect is exponential, so small changes have a big impact. Weights beyond +/-16.0 are considered "superweighted" because their effect is so strong it dominates the search results. This can be useful if you want to force results matching that portion of the query toward the top. Negative term weights boost documents that don't contain that term by lowering the score of those that do. You may want to use negative term weights on words or phrases associated with spammy or irrelevant results.

The cts:query Object

MarkLogic uses `cts:query` objects to represent search constraints. These objects are built via constructor functions with names like `cts:word-query`, `cts:element-attribute-value-query`, and `cts:element-range-query`. Some special query types are aggregators, designed to place other queries into a hierarchy with names like `cts:and-query`, `cts:or-query`, `cts:and-not-query`, and `cts:boost-query`. You'll notice that the leaf-node (i.e., non-aggregating) `cts:query` types match up one-to-one with the internal indexes. That makes it straightforward for MarkLogic to leverage its indexes to execute `cts:query` hierarchies.

Of course, users don't think in terms of `cts:query` objects. Users like to click GUIs and type flat search strings. It's an application requirement to take the GUI settings and search strings and convert them to a `cts:query` hierarchy for actual execution. Tools like the Search API make this easy and configurable.

Even a simple one-word user query might be expanded to a `cts:or-query` internally in order to list out the locations where matches are acceptable and that might be implicitly placed into an outer `cts:and-query` to control the viewable scope based on GUI settings. Hierarchies that expand to thousands of leaf-nodes are typical with many customers and execute efficiently.

By making the `cts:query` structured and programmable (as opposed to being built up out of awkward string concatenation as with SQL), MarkLogic gains immense flexibility and power. For example, the thesaurus function `thsr:expand()` works by walking the provided `cts:query` hierarchy, finding all leaf-node word queries needing expansion, and replacing each with a `cts:or-query` holding the synonym word queries.

Documents in MarkLogic have an intrinsic quality, akin to a Google PageRank. The quality of each document gets added to the calculated score for the document. Higher-quality documents rise to the top of search results; lower-quality documents get suppressed. Quality is programmatically set and can be based on anything you like. MarkMail.org, for example, gives code check-in messages significant negative quality so check-in messages only appear in results when no other regular messages match.

Searches include a quality-weight parameter dictating how much importance to give to the quality. It's a floating-point number multiplied by the quality during calculation. A value of "0" means to ignore the quality values completely for that query. It's good practice to give documents a wide gamut of quality scores (say, up to 1,000) and use a fractional quality-weight to tune it down until you get the right mix of scores based on quality and terms. By having a wide gamut, you keep granularity.

You can watch as the server does the scoring math if you use the Admin Interface to turn on the diagnostic flag `"relevance"`, which dumps the math to the server error log file. For programmatic access, you can pass `"relevance-trace"` as an option to `cts:search()` and use `cts:relevance-info()` to extract an XML-formatted trace log. Tracing has a performance impact, so it's not recommended for production.

STOP WORDS

A stop word is a word that is so common and with so little meaning by itself that it can be ignored for purposes of search indexing. MarkLogic does not require or use stop words. This means it's possible to search for phrases such as "to be or not to be" which consist solely of stop words, or phrases like "Vitamin A" where one of the words is a stop word.

However, MarkLogic does limit how large the positions list can be for any particular term. A positions list tracks the locations in documents where the term appears and supports proximity queries and long phrase resolution. The maximum size is a configurable database setting called *positions list max size* and is usually several hundred megabytes. Once a term has appeared so many times that its positions list exceeds this limit, the positional data for that term is removed and no longer used in query resolution (because it would be too inefficient to do so). You can detect whether any terms have exceeded this limit by looking for a file named `StopKeySet` having non-zero length in one of the on-disk stands.

FIELDS

MarkLogic allows administrators to turn on and off various indexes. The list of indexes is long: stemmed searches, word searches, fast phrase searches, fast case-sensitive searches, fast diacritic-sensitive searches, fast element word searches, element word positions, fast element phrase searches, trailing wildcard searches, trailing wildcard word positions, three-character searches, three-character word positions, two-character searches, and one-character searches. Each index improves performance for certain types of queries at the expense of increased disk consumption and longer load times.

In some situations, it makes sense to enable certain indexes for parts of documents but not for other parts. For example, the wildcard indexes may make sense (i.e., justify their overhead) for titles, authors, and abstracts but not for the longer full body text.

Fields let you define different index settings for different subsets of your documents. Each field gets a unique name, a list of elements or attributes to include, and another list to exclude. For example, you can include titles, authors, and abstracts in a field but exclude any footnotes within the abstract. Or, maybe you want to include all document content in a field but remove just `<encoded-blob>` elements from the field's index. That's possible, too. At request time, you use field-aware functions like `cts:field-word-query()` to express a query constraint against a field.

Beyond more efficient indexing, fields give you the option to move the definition of which parts of a document should be searched from something coded in XQuery or JavaScript to something declared by an administrator. Let's say that you're querying Atom and RSS feeds with their different schemas. You might write this code to query across schema types:

```
let $elts := (xs:QName("atom:entry"), xs:QName("item"))
let $query := cts:element-word-query($elts, $text) ...
```

It's simple enough, but if there's a new schema tomorrow, it requires a code change. As an alternative, you could have a field define the set of elements to be considered as feed items and query against that field. The field definition can change, but the code remains the same:

```
let $query := cts:field-word-query("feeditem", $text)
```

As part of defining a field, you can also apply weights to each contributing element or attribute, making each more or less relevant in a search match. This provides another performance advantage. Normally if you want to weight titles more than authors and authors more than abstracts, you provide the weights at query time. For example:

```
cts:or-query((
   cts:element-word-query(xs:QName("author"), $text, (), 3.0),
   cts:element-word-query(xs:QName("title"), $text, (), 2.0),
   cts:element-word-query(xs:QName("abstract"), $text, (),
   0.5)
))
```

You can include these weights in a field definition, and they'll be baked into the index:

```
cts:field-word-query("metadata", $text)
```

That means less math computation at execution time. The downside is that adjusting the weighting of a field requires an administrator change and a background content reindexing to bake the new values into the index. It's often best to experiment with ad hoc weightings until you're satisfied that you have the weightings correct, then bake the weightings into the field definition.

MORE WITH FIELDS

Fields also provide a way to create singular indexed values out of complex XML. Imagine that you have XML structured like this:

```
<name>
   <fname>John</fname>
   <mname>Fitzgerald</mname>
   <lname>Kennedy</lname>
</name>
```

You can create a field named "fullname" against `<name>` defined to contain all of its child elements. Its singular value will be the person's full name as a string. You can create another field named "simplename" against `<name>` but excluding `<mname>`. Its singular value will be just the first and last names as a string. Using `cts:field-value-query()` you can do optimized queries against either of these computed values. It's much faster than computing values as part of the query. Within the Universal Index, each field simply adds term lists, one for each value.

As mentioned previously, you can even create a range index against a field. Then you can sort by, constrain by, and extract values not even directly present in the documents.

REGISTERED QUERIES

Registered queries are another performance optimization. They allow you to register a `cts:query` as something you plan to use repeatedly and whose results you'd like MarkLogic to remember for later use.

Let's say that you're going to generate a report on the number of Firefox browsers per day that hit any *index.html* page on your site. You may want to register the portion of the query that doesn't include the day, then use it repeatedly by intersecting it with the day constraint.

For the code below, let's assume that we have a set of documents in the `fact` namespace that records data about visiting browsers: their name, version, and various attributes, and the session ID in which they appear. You can think of those documents as the XML equivalent of a fact table. Let's assume that we also have a set of documents in the `dim` namespace (the XML equivalent of a dimension table) that records data about particular page hits: the page they hit, the date on which they were seen, and other aspects of the request, such as its recorded performance characteristics.

To run this query efficiently, we first need to use a *shotgun or* join the two document sets based on session IDs. Then, because that's a not insignificant operation, we register the query.

```
(: Register a query that includes all index.html page views from
Firefox users :)

let $ff := cts:element-value-query(xs:QName("fact:brow"),
"Firefox")

let $ff-sessions := cts:element-values(xs:QName("fact:s"), "",
(), $ff)

let $registered :=
   cts:registered-query(cts:register(cts:and-query((
      cts:element-word-query(
         xs:QName("dim:url"), "index.html"),
         cts:element-range-query(xs:QName("dim:s"),
         "=",
         $ff-sessions)
      )
   )
), "unfiltered")

(: Now for each day, count the hits that match the query. :)

for $day in ("2010-06-22", "2010-06-23", "2010-06-24")

let $query := cts:and-query((
   $registered,
   cts:element-value-query(
      xs:QName("dim:date"),
      xs:string($day)
   )
))
return concat(
   $day,
   ": ",
   xdmp:estimate(cts:search(/entry, $query))
)
```

This query runs quickly because for each day MarkLogic only has to intersect the date term list against the cached results from the registered query. The registration persists between queries as well, so a second execution, perhaps in a similar query limiting by something other than a date, also sees a benefit.

The cts:register() call returns an xs:long identifier for the registration. The cts:registered-query() call turns that xs:long into a live cts:query object. The code above takes advantage of the fact that if you register a query that's exactly the same as one the server's seen registered before, it returns the same xs:long identifier. That saves us from having to remember the xs:long value between queries.

Registered queries are tracked in a memory cache, and if the cache grows too big, some registered queries might be aged out of the cache. Also, if MarkLogic stops or restarts, any queries that were registered are lost and must be re-registered. By registering (or possibly re-registering) immediately before use, as in our example here, we avoid that issue.

Registered queries have many use cases. As a classic example, imagine that you want to impose a visibility constraint on a user, where the definition of the visibility constraint is either defined externally (such as in an LDAP system) or it changes so often that it doesn't make sense to use MarkLogic's built-in security model to enforce the rules. When the user logs in, the application can declare the user's visibility as a cts:query and registers that cts:query for repeated optimized use. As the user interacts with the data, all of the user's actions are intersected with the registered query to produce the visibility-limited view. Unlike with built-in security, you're free to alter the visibility rules on the fly. Also unlike built-in security, there's an initial cost for the first execution.

The first time you execute a registered query, it takes as long to resolve as if it weren't registered. The benefit comes with later executions because the results (the set of fragment IDs returned by the cts:query) are internally cached.

Registered queries behave like synthetic term lists. A complex cts:query might require intersections and unions with hundreds or thousands of term lists and processing against numerous range indexes. By registering the query, MarkLogic captures the result of all that set arithmetic and range index processing and creates a simple cached synthetic term list (the results are actually placed into the List Cache). You always have to pass "unfiltered" to the registration call to acknowledge that this synthetic term list will operate unfiltered.

Somewhat amazingly, as documents are added or deleted, the cache is updated so you always get a transactionally consistent view. How does that work?

At the lowest level, for each registered query, there's a synthetic term list maintained for every stand inside every forest. That's the secret to MarkLogic's ability to keep the cache current in the face of updates.

Inside an on-disk stand, the synthetic term list gets generated the first time the registered query is used. If there's a document update or delete, the only thing that can happen to any of the fragments in an on-disk stand is that they can be marked as

deleted (one of the perks of MVCC). That means the synthetic term list doesn't have to change. Even after a delete it can report the same list values. It can be left to the timestamp-based term lists to remove the deleted fragments.

For an in-memory stand, deletions are handled the same way as on-disk stands, but fragments can also be inserted via updates. The server deals with this by invalidating the synthetic term lists specific to the in-memory stand whenever a fragment gets inserted into it. Later, if a registered query is used after a fragment insert, the synthetic term list is regenerated. Since in-memory stands are very small compared to on-disk stands, regenerating the synthetic term lists is fast.

After an in-memory stand is written to disk, or after a background stand merge completes, there will be a new on-disk stand without the synthetic term list. Its synthetic term list will be regenerated (lazily) as part of the next query execution. There's no overhead to having registered queries that aren't used.

For information on using registered queries, see the Search Developer's Guide.

THE GEOSPATIAL INDEX

MarkLogic's geospatial indexes let you add query constraints based on geographic locations mentioned in documents. The geographic points can be explicitly referenced in the XML document (with convenience functions for those using the GML, KML, GeoRSS/Simple, or Metacarta markup standards) or they can be added automatically via entity identification, where place names are identified and geocoded automatically according to text analysis heuristics, using third-party tools.

MarkLogic's geospatial indexes let you match by point (that is, an exact latitude/longitude match), against a point-radius (a circle), against a latitude/longitude box (a Mercator "rectangle"), or against an ad hoc polygon (efficient up to tens of thousands of vertices, useful for drawing features like city boundaries or the terrain within some distance of a road). It also lets you match against complex polygons (ones with interior polygon gaps, such as the boundary of Italy minus the Vatican).

The geospatial indexes fully support the polar regions and the anti-meridian longitude boundary near the International Date Line and take into account the non-spherical ellipsoid shape of the earth. They're also fully composable with all the other indexes, so you can find documents most relevant to a search term, written within a certain date range, and sourced within a place inside or outside an abstract polygon. You can also generate frequency counts based on geospatial bucketing to, for example, efficiently count how many documents matching a query appear within geographic bounding boxes. These counts can then be used to generate heatmaps.

At a technical level, MarkLogic's geospatial indexes work like a range index with points as data values. Every entry in the geospatial range index holds not just a single scalar

value but a latitude and longitude pair. Picture a long array of structures holding lat/long values and associated document IDs, sorted by latitude major and longitude minor, held in a memory-mapped file. (Latitude major and longitude minor means they're sorted first by latitude, then by longitude for points with the same latitude.)

Point queries can be easily resolved by finding the matching points within the pre-sorted index and extracting the corresponding document ID or IDs. Box queries (looking for matches between two latitude values and two longitude values) can be resolved by first finding the subsection of the geospatial index within the latitude bounds, then finding the sections within that range that also reside within the longitude bounds.[11]

For circle and polygon constraints, MarkLogic employs a high-speed comparator to determine if a given point in the range index resides inside or outside the circle or polygon constraint. The geospatial indexes use this comparator where a string-based range index would use a string collation comparator. The comparator can compare 1 million to 10 million points per second per core, allowing for a fast scan through the range index. The trick is to look northward or southward from any particular point, counting arc intersections with the bounding shape: an even number of intersections means the point is outside, odd means it's inside.

As an accelerator for circles and polygons, MarkLogic uses a set of first-pass bounding boxes (a box or series of boxes that fully contain the circle or polygon) to limit the number of points that have to be run through the detailed comparator. A circle constraint thus doesn't require comparing every point, only those within the bounding box around the circle.

Searches on certain parts of the globe complicate matters. The poles represent singularities where all longitude lines converge, so MarkLogic uses special trigonometry to help resolve searches there. For geospatial shapes that cross the anti-meridian (where longitude values switch from negative to positive), the server generates multiple smaller regions that don't cross this special boundary and unions the results.

This subject is described more in Geospatial Search Applications.

11 The worst-case performance on bounding boxes? A thin vertical slice.

An Advanced Trick

If you use the coordinate system "raw" in the API calls, MarkLogic treats the world as flat (as well as infinite) and compares points simplistically without the complex great circle ellipsoid math. This comes in handy if you ever want a two-value range index to represent something other than geography, such as a medical patient's height and weight. Assume height represented as longitude (x-axis) and weight as latitude (y-axis). If the data were laid out as a chart, the results from a large set of patients would look like a scatter plot. Using MarkLogic's geospatial calls you can draw arbitrary polygons around the portions of the chart that are meaningful—such as which patients should be considered normal, overweight, or underweight according to different statistical measures—and use those regions as query constraints. The trick works for any data series with two values where it makes sense to think of the data as a scatter plot and limit results by point, box, circle, or polygon.

THE REVERSE INDEX

All of the indexing strategies we've discussed 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 of the matching queries (the set of stored queries that if executed would match this document).

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

MarkLogic executes reverse queries efficiently and at scale by turning on the "fast reverse searches" index. 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. We'll examine how that works later, but first let's look at some situations where reverse queries are helpful.

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 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. How about when an earthquake occurs in a certain city? Define the city as a geospatial query in the reverse index and check for matching latitude/longitude data when earthquakes occur.

Without reverse query indexing, for each new document or set of documents, you'd have to loop over all of your queries to see which ones 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 book, 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 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.

A REVERSE QUERY CARPOOL MATCH

Let's use the carpool example to make the idea concrete. You have a driver, a non-smoking woman driving from San Ramon to San Carlos, leaving at 8:00am, 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 Drive, 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 their preferences match you. The combination of rules is defined by the `cts:and-query`. 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/*),
      cts:reverse-query($me)
   ))
)
return base-uri($match)
```

Again, the preferences of 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.

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 that would 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. In turn, the second phase examines each of these serialized query documents and determines which in fact fully match the source document, based on all of the other constraints present. Basically, it quickly finds documents that have any matches, then isolates down to those that have all matches.

To support the first phase, MarkLogic maintains a custom index in which it gathers all of the distinct *leaf node* query terms (that is, the simple query terms like words or values, not compound query terms like `cts:and-query`) across all of 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 (for negative queries). It's a term list of query terms.

Later, when 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 and finds all of the serialized query document IDs having a query with

at least one term match in the source document. For a simple, one-word query, this produces the final answer. 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 of the required constraints are true, short-circuiting whenever possible. If all of 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 example, 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.

It's complicated, but it works well, and it works quickly.

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 that 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:

	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 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.

BITEMPORAL

MarkLogic's bitemporal feature allows you to track database documents along two time axes simultaneously. It lets you keep track of the time when a fact was true (the *valid time*) as well as the time when the database knew that truth (the *system time*). Tracking data on two time axes is critical in industries such as finance, insurance, and intelligence. It enables organizations to justify why they made decisions based on what they knew about the world at a given time.

How does MarkLogic manage bitemporal data? Each temporal document has four timestamp values (valid start, valid end, system start, and system end) that define the effective valid and system times for the document data. When you insert temporal documents for new time periods, MarkLogic keeps track of the system timestamp values and makes sure the overall version history is filled in appropriately. By default, the versions that make up the bitemporal history of a document never go away (although their timestamps may change).

You can query bitemporal data by specifying ranges for the valid and system times. To resolve bitemporal queries, MarkLogic uses four range indexes, one for each timestamp, to quickly isolate the matching documents.

A BITEMPORAL EXAMPLE

Imagine a simple example where we're tracking a person's location over time with temporal documents. The valid time indicates when the person truly was at a location while the system time tracks when we knew that information (by default, this is based on when the document was saved in the database). Each time the person's location changes, we ingest a new temporal document representing that knowledge. Behind the scenes, MarkLogic performs housekeeping on the existing temporal documents to keep the timestamps up-to-date so we have a history of "what we knew" and "when we knew it."

Let's say that a person moved to the city of Alameda on January 1. We learn about this and add the data to our database on January 5. Since our document is a temporal document, it includes four timestamps—a valid start, valid end, system start, and system end. At this point, we've never known the person to be anywhere else, so only the start times are set (the end times stay set to the default, infinity). The valid start time tells us when the person was in Alameda, and the system start time records when we knew it:

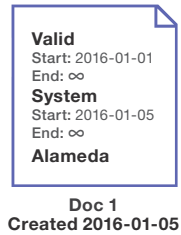

Figure 10: A document representing a person's location at a given time is added to a bitemporal database.

Now let's say that the person moved to Berkeley on January 10, and we learn this on January 12. We insert this knowledge as a new temporal document (Doc 3). To go with this new document, MarkLogic creates a document to reflect the person's time in Alameda as we know it now (Doc 2). MarkLogic also updates our first document to give it a system end date. The first document is now a historical record of what we knew about the person up to January 12:

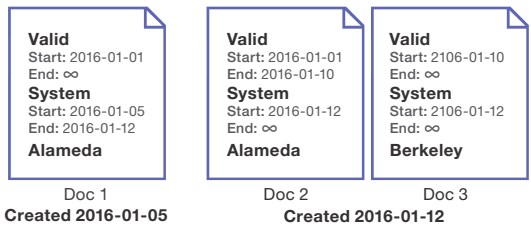

Figure 11: A new person document (Doc 3) reflects the new location information. MarkLogic adds a document (Doc 2) and updates the original version (Doc 1) to fill in the historical record.

On January 15, we discover a mistake: the person actually moved to Berkeley on January 8! No problem. We insert a revised document noting the new valid times (Doc 5). Behind the scenes, MarkLogic creates a document (Doc 4) and edits system end times in existing documents (Doc 2, Doc 3) to maintain an accurate historical record:

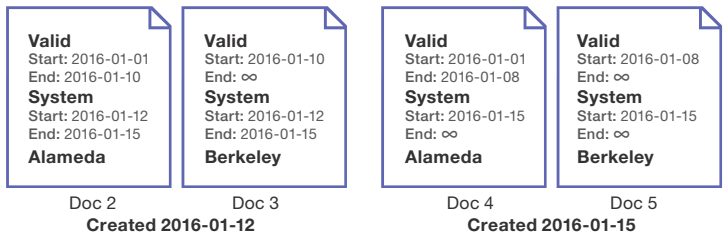

Figure 12: The person's location information changes again (Doc 5). Again, MarkLogic adds a document (Doc 4) and updates times to fill in the historical record.

On January 18, we find out that the person we are tracking isn't who we thought they were. The previous information is now invalid, so we perform a bitemporal delete. Behind the scenes, MarkLogic updates the end values in the latest set of documents (rather than actually deleting documents from the database). Even though the facts ended up being wrong, the knowledge trail we followed has been fully recorded:

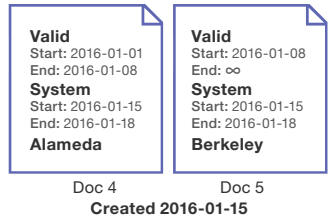

Figure 13: Performing a bitemporal delete does not actually delete documents. MarkLogic updates the timestamps of the document versions to keep an accurate record of what was known over time.

We can visualize bitemporal data as a two-dimensional chart with the system time on the horizontal axis and the valid time on the vertical axis. Below, the shaded boxes represent the five document versions in the example.

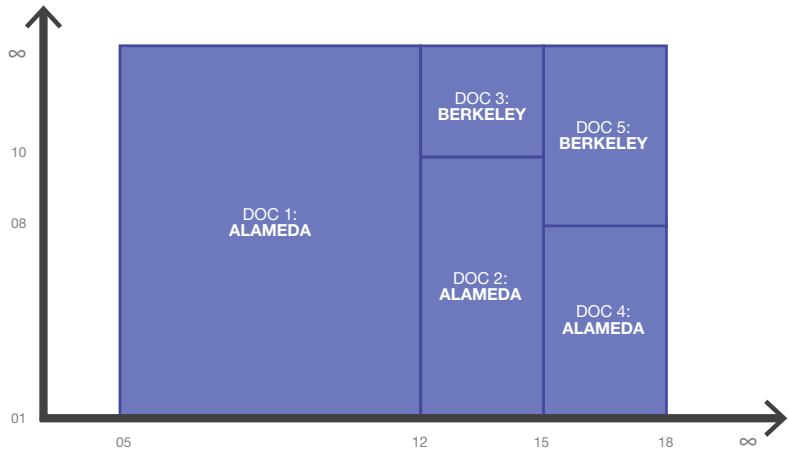

Figure 14: A two-dimensional chart representing the data recorded in a bitemporal database over time.

As shown in the example, MarkLogic keeps the timestamps for the document versions consistent and the version history filled in as new information is added. What's interesting is that inserting data for a time period often results in updates to more than one document. For example, if the inserted data relates to a valid time range fully contained by a previously inserted temporal document, that document will be split into two to cover the range before and after the new insert, with the new document covering the range in the middle.

BITEMPORAL QUERIES

Querying on bitemporal data makes use of the range indexes on each of the four timestamps. Consider the following query: "What were the locations of our person of interest between January 8 and January 9 as we knew them between January 13 and January 14?" We express this query using cts:period constraints for defining time ranges:

```
cts:search(fn:doc(), cts:and-query((
   cts:period-range-query(
      "valid",
      "ALN_CONTAINED_BY",
      cts:period(xs:dateTime("2016-01-08T00:00:00"),
                 xs:dateTime("2016-01-09T23:59:59.99Z"))),
      cts:period-range-query(
      "system",
      "ALN_CONTAINED_BY",
      cts:period(xs:dateTime("2016-01-13T13:00:00"),
      xs:dateTime("2016-01-14T23:59:59.99Z"))
   )
)))
```

The code uses shortcuts for defining time intervals based on the relations of Allen's Interval Algebra.[12] These define all of the possible relationships between two time intervals—for example, preceding, overlapping, containing, meeting, finishing, and more. Internally, MarkLogic can break this query into the following inequalities:

```
Valid Start <= 2016-01-09
Valid End > 2016-01-08
System Start <= 2016-01-14
System End > 2016-01-13
```

Consulting the range indexes gives us the following:

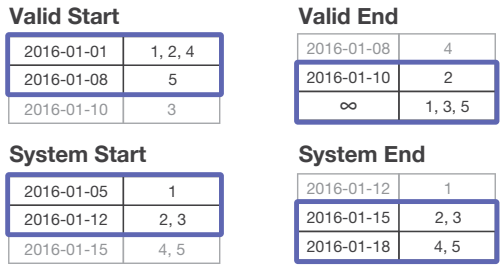

Figure 15: To resolve a bitemporal query, MarkLogic can perform inequality operations on the range indexes for the documents.

12 MarkLogic also lets you define time relations using a set of SQL 2011 operators, which are similar to Allen's operators but less restrictive.

Performing an intersection on the four result sets gives us a single document, #2. This tells us that the person was in Alameda during those time ranges.

MarkLogic also organizes temporal documents under three collections. Since bitemporal data can coexist with regular data in a database, MarkLogic adds all temporal documents into a *temporal collection* (named "temporalCollection").[13] The temporal collection is special in that the only way to add to it or delete from it is through the temporal functions. (When creating a temporal collection, you pass an option indicating if the admin should have the ability to manipulate temporal documents using non-temporal functions; if not, then no user, not even the admin, can change or delete temporal documents in a non-temporal manner!) Additionally, all of the versions of a temporal document that get created over time are assigned to a *URI collection*. This logically ties the documents together even though they all have different physical URIs. In our location-tracking example, if our original document had the URI "person.json", each version of that document would be assigned to a "person.json" collection. If we inserted a second document with the URI "person2.json", that document would be assigned to the "person2.json" collection. Finally, there is a collection for the most recent active versions of the documents—i.e., documents with system end times set to infinity (named "latest"). When a new version of a document is inserted, it is put into the latest collection and the document it is updating is removed. Documents that have been deleted in a bitemporal collection will not have a latest version.

For more information, see the Temporal Developer's Guide.

SEMANTICS

With MarkLogic semantics, you have yet another way to represent your data: as triples. A triple describes a relationship among data elements and consists of a subject, predicate, and object (similar to human language):

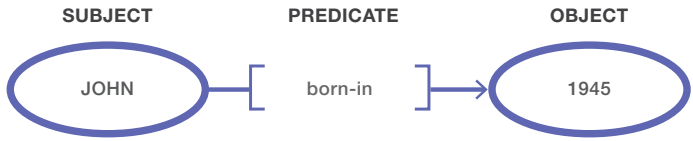

Figure 16: A semantic triple represents a fact about the world as a subject-predicate-object relationship.

13 Note that documents aren't required to be in a temporal collection for you to use the temporal queries and comparison operators. The documents just need to have start and end timestamps, with range indexes applied to those timestamps and an axis that ties the timestamps together. For such documents, MarkLogic doesn't automatically update the timestamps or create additional documents to fill in the historical record when data changes (like it does for temporal documents).

A key aspect of semantic data is not just that it describes relationships (that let you answer the question, "When was John born?"), but that these relationships can be interlinked. You can add more triples describing John, and also triples describing Mary, and some of these facts will intersect. This results in a connected web of information that can be represented as a graph:

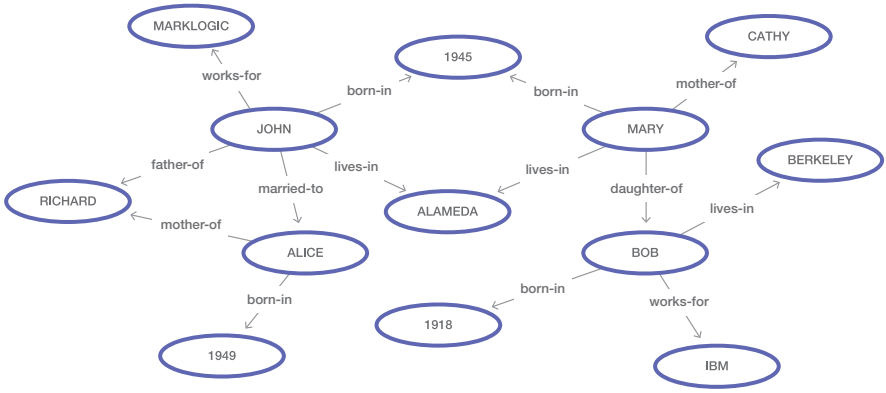

Figure 17: The relationships between many semantic triples can form a web of information.

MarkLogic applications can query large collections of triples (or semantic graphs) to tell us interesting things. By traversing the triples in a graph, an application might tell us that John not only was born in the same year as Mary, but also lived on the same block and went to the same school. With the help of semantic data, we can infer that John might have known Mary.

STORING TRIPLES

Resource Description Framework (RDF) is the W3C standard that describes how to represent facts using subject-predicate-object syntax, which is why triples in MarkLogic are called RDF triples. In RDF, subjects and predicates are represented by internationalized resource identifiers (IRIs).[14] Objects can be IRIs but also can be typed literal values such as strings, numbers, and dates.

When MarkLogic ingests a raw RDF triple, it turns the triple into an XML representation and then loads it as an XML document. So under the hood, MarkLogic represents semantic data the same way it represents regular XML documents. Consequently, triples can be managed with all of the features and conveniences associated with other MarkLogic content. They can be protected with role-based security, organized in collections and directories, and updated within ACID-compliant transactions. Triples can also be added to traditional XML or JSON content when

14 IRIs are similar to URIs but have broader support for international characters. For details, see Wikipedia.

they're enclosed in `sem:triple` tags (for XML) or in `triple` properties (for JSON). This enables you to enhance existing documents with subject-predicate-object facts and then search (or join) the content using both regular and semantic search strategies.

INDEXES TO SUPPORT SEMANTICS

There are three special indexes that support semantic search: a triple index, triple values index, and triple type index.

Triple Index

In the triple index, there are columns for the subject, predicate, and object of each triple. There is also a column mapping the triple to the document where it's found (just like in term indexes) and a column for position information (if triple positions is turned on). Each triple is stored as three different permutations in the triple index, in sorted order: subject-object-predicate, predicate-subject-object, and object-predicate-subject. (An extra column in the triple index specifies the permutation.)

Being able to choose among these three permutations helps MarkLogic resolve queries as fast as possible. If a query requires locating triples with a given subject, there will exist a sequence of triples sorted by subject with all matching triples in a contiguous block. It's the same for matching triples with a specific predicate or object. If a query requires locating triples with a given subject and predicate, then there's a sequence of triples sorted primarily by predicate and secondarily by subject, which will put all matching triples together in a contiguous block. With the three permutations, it's possible to have a contiguous block for any two parts of a triple. And if a query requires triples with a specific subject/predicate/object value, that sequence is available as well.

The triple index behaves in many ways like a special range index, but while a range index is always fully memory mapped, the triple index is designed to be much larger than physical memory and to page large blocks of triples in and out of memory as needed.

Triple Values Index

The triple index doesn't actually store the values that occur in each triple. Instead, the index represents them by integer IDs for faster lookups and smaller space consumption. The IDs are mapped to actual values in the triple values index. When triples are retrieved from the triple index for a search result, MarkLogic reconstructs their values by referencing the triple values index. This means that to return results for a semantic query, MarkLogic only has to access the triple indexes and not the original documents.

Triple Type Index

Object values in triples can be typed literals (strings, dates, numbers, etc.). MarkLogic takes this type information, and any other information not necessary for value ordering (e.g., date time-zone information), and stores it in a triple type index. The triple type index is usually relatively small and is memory mapped.

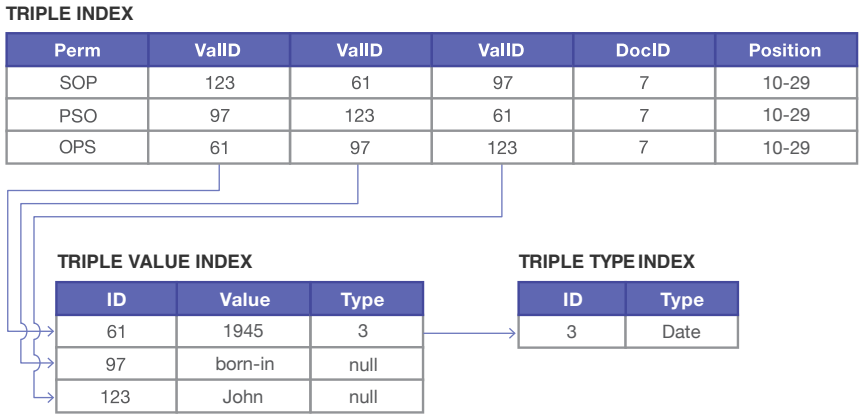

Figure 18: MarkLogic supports semantic queries with three indexes: a triple index, triple values index, and triple type index. To support efficient lookups, each triple is stored in three permutations: subject-object-predicate, predicate-subject-object, and object-predicate-subject.

QUERYING TRIPLES

You can search for triples in your MarkLogic database using SPARQL, a query language for retrieving semantic data. It's a lot like using SQL but for triples. To construct a SPARQL SELECT query, the most common type, you describe one or more triple patterns using IRIs, literal values, and variables. For example, let's say that you have semantic data about university mascots:

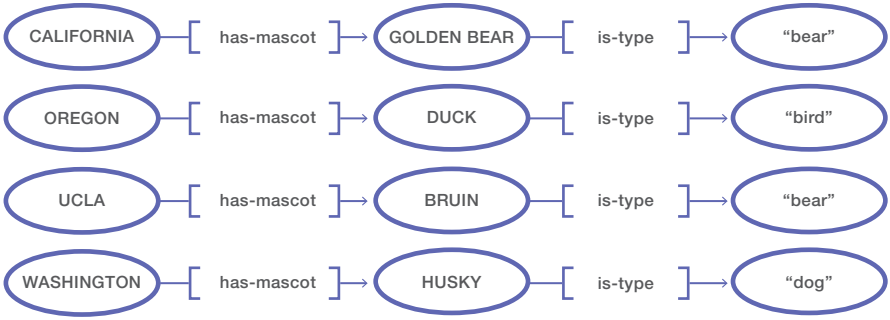

Figure 19: A set of triples representing schools and their mascots.

Each line above represents a pair of triples. The first triple describes the school's mascot, and the second describes the mascot's type. The following SPARQL query returns schools that have mascots that are bears:

```
SELECT ?school
   WHERE {
      ?school <http://example.org/has-mascot> ?mascot .
      ?mascot <http://example.org/is-type> "bear"
}
```

MarkLogic processes the two triple patterns in the WHERE clause and finds the matches in the triple index. The first pattern matches everything and returns four triples. For the second pattern, two triples are returned—the ones with `"bear"` as the object. To get the final result, MarkLogic joins the two sets of matches based on a shared variable—`?mascot`. This gives us two joined results, the IRIs for California and UCLA. This is a simple example, but it demonstrates the two key steps in resolving a semantic query: retrieving triples corresponding to each pattern and then joining those sets of triples to get a final result.

SPARQL queries can also be combined with the other types of MarkLogic search queries. Combining such queries is straightforward, since triples are stored as documents just like all other content in the MarkLogic database. When a SPARQL query returns a set of triples, those triples are associated with a set of document IDs. MarkLogic can intersect these document IDs with those returned from any other type of query. This lets us restrict our SPARQL search results to those triples associated with, for example, a certain collection, directory, or security role.

You can read more about SPARQL in the Semantics Developer's Guide.

FINDING THE RIGHT QUERY PLAN

MarkLogic can solve a complex SPARQL query in any of a thousand different ways, depending on how it chooses to retrieve and combine the many sets of triples involved. MarkLogic can choose from different join algorithms when merging sets of triples (hash join, scatter join, bloom join, or merge join). It can execute the joins in different orders. It can select different permutations when retrieving triples from the triple index. The way MarkLogic structures the operations to solve a SPARQL query is called the *query plan*. The more efficient a query plan it can devise, the faster it will be able to return the results.

What MarkLogic *doesn't* do is attempt to figure out the very best query plan using a fixed set of rules. Instead, it uses an algorithm called *simulated annealing* that takes advantage of computer-based natural selection to find the best plan in a fixed amount of time. MarkLogic starts with a default plan based on how the SPARQL query is written, then

estimates the efficiency of that plan using cost analysis. The analysis takes into account the cardinality of the triples, the memory used by the joins, and whether intermediate results come back ordered or unordered, among other things. (MarkLogic gathers some of the statistics used for cost analysis, such as the cardinality information, as it loads and indexes the triples.)

Then, to improve the plan, MarkLogic:

1. Mutates the operations in the plan to create a new version that will still return the same result. For example, it could use different join strategies or change the order of the joins.
2. Estimates the efficiency of the new plan using cost analysis.
3. Compares the efficiency of the new plan with the efficiency of the current plan and selects one or the other.

Next, if time is up, it stops and uses the plan it has chosen to execute the SPARQL query. Otherwise, it repeats the steps.

During the selection process, MarkLogic will always choose the new plan over the current plan when the new plan is more efficient. But there's also a chance it will choose the new plan even if it's less efficient. This is because query plans are related to one another in unpredictable ways, and discovering a highly optimal plan may require initially selecting an intermediate plan that is less efficient. As time goes by, the probability that MarkLogic will choose a new query plan that is less efficient goes down. By the end of the process, MarkLogic will have found its way to a plan that's highly efficient (but not necessarily optimal).

This strategy for choosing a query plan offers an advantage: you can specify the amount of time MarkLogic spends optimizing by passing in a setting in with your SPARQL query. You can allow more time for finding efficient strategies for complex queries and less time for simpler ones. You can also allow more time for queries that are run repeatedly for long periods of time. This is because MarkLogic will optimize such queries the first time and then store the optimized query plan in its cache.

TRIPLES FOR CROSS-DOCUMENT JOINS

Previously, we discussed using range indexes for cross-document joins with tweet data. You can use RDF triples for the same purpose. You define relationships between documents by their URIs, then you join those documents at query time with SPARQL. It's similar to joining rows from different tables in a relational system except that we're leveraging the multi-model (document and semantic) capabilities of MarkLogic.

For our tweet data, we can use RDF triples to relate our tweets to their authors and also relate authors to one another to define who follows whom. As JSON, the triples look like this:

```
"triple": {
   "subject": tweet-uri,
   "predicate": "http://example.org/has-author",
   "object": author-uri
}

"triple": {
   "subject": author-uri,
   "predicate": "http://example.org/follows",
   "object": author-uri
}
```

Let's that say we want to find the author of the most recent tweet mentioning "healthcare" and retrieve all of that author's followers. Here's the code in JavaScript:

```
var tweet = fn.head(cts.search(
   "healthcare", cts.indexOrder(
      cts.jsonPropertyReference("created"),
      "descending"
   )
));

var params = {"id": tweet.toObject()['tweet-id']};

var followers = sem.sparql(' \
   SELECT ?follower \
   WHERE { \
      ?author <http://example.org/tweeted> $id. \
      ?follower <http://example.org/follows> ?author \
   }',
   params
);
```

The first block retrieves the tweets that contain our term and sorts them in descending order of when they were created. It selects the most recent tweet from the results and then puts the ID for that tweet into an object that can be passed to the SPARQL query. The second block joins the tweet ID to the tweet's author and then the author to the followers. At the end, we're left with a set of follower URIs with which we can access the associated author documents.

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 can be initiated via the administrative web console (as a push-button action or as a scheduled job), a server-side XQuery or JavaScript script, or the REST API. You specify a database to back up and a target location. Backing up a database saves copies of its configuration files, all the forests in the database, and the corresponding security and schema databases. It's particularly important to backup the security database because MarkLogic tracks role identifiers by numeric IDs rather than string names, and the backup forest data can't be read if the corresponding numeric role IDs don't exist in the security database.

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

TYPICAL BACKUP

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 briefly to write out a fully consistent view for the backup, flushing everything from memory to disk.

A database backup always creates a new subdirectory for its data in the target backup directory. A forest backup writes to the same backup directory each time. 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 won't copy any files that already exist and are identical in the target. This offers a nice performance boost.

FLASH BACKUP

MarkLogic also supports *flash backups*. Some filesystems support taking a *snapshot* of files as they exist at a certain point in time. The filesystem basically tracks the disk blocks in use and makes them read-only for the lifetime of the snapshot. Any changes to those files go to different disk blocks. This has the benefit of being pretty quick and doesn't require duplication of any disk blocks that aren't undergoing change. It's a lot like what MarkLogic does with MVCC updates.

To do a flash backup against MarkLogic data, you have to tell MarkLogic to put the forest being backed up into a fully consistent on-disk state, with everything flushed from memory and no in-progress disk writes. Each forest has an "Updates Allowed" setting. The settings are:

`all`
The default, indicating the forest is fully read-write.

`delete-only`
Indicates that the forest can't accept new fragments but can delete existing fragments. Normal document inserts will avoid loading new data in this forest.

`read-only`
Indicates that the forest can't accept any changes. Modifying or deleting a fragment held in the forest generates an error.

`flash-backup`
Similar to the read-only setting except that any request to modify or delete a fragment in the forest gets retried until the "Retry Timeout" limit (default of 120 seconds) instead of immediately generating an error. The idea is that for a few seconds you put your forests in the flash-backup state, take the snapshot, and then put it back in the normal "all" state.

For more information on backing up and restoring a database, see the Administrator's Guide.

JOURNAL ARCHIVING AND POINT-IN-TIME RECOVERY

When performing a database backup, MarkLogic asks if you want *journal archiving*. Answering "yes" causes MarkLogic to write an ever-growing journal next to the backup, recording all of the database transactions committed after the backup. Note that this is a separate journal from the main rotating journal kept with the database.

Having the journal in the backup provides two major advantages. First, it reduces how much data you can lose if a catastrophe occurs in the primary database filesystem. A normal backup can only recover data as it existed at the time of the backup event, but a backup with journal archiving lets MarkLogic replay everything that happened after the backup out of the journal, restoring the database to very much like it was right before the catastrophe.

Second, it enables a recovery to any point in time between the backup and the current time, called a *point-in-time recovery*. Many enterprise systems need resilience not only to disk failure but also to human error. What if a person accidently or maliciously, or as a

result of a code bug, modifies data inappropriately? With journal archiving MarkLogic can recover the database to any point in the past by starting at the backup checkpoint and then replaying the journal up to the desired point in time.

Does this mean you never need to do another backup after the first one? No, because it takes time to replay the journals forward. The journal replay runs faster than the initial execution did because locks and regular disk syncs aren't required, but it still takes time. There might even be merges that happen during a long replay. The backup also grows unbounded because deleted data can never go away. On a busy system or one with lots of data churn, you should still do a backup periodically to checkpoint a new starting time. Note that you can only journal-archive to one database backup at a time. If you archive to a new one, it halts writing to the old (once the new finishes).

A journal archive introduces the possibility that writes to the database could slow down. There's a configurable "lag time" indicating how far behind the journal archive can get from the database, by default 15 seconds, after which writes to the database stall until the journal archive catches up. This lag is designed to enable a burst of writes to go forward on the database even if the journal archive is on a slower filesystem.

Interestingly, when doing a backup with journal archiving, the journal starts recording transactions nearly immediately even as the writing of the main forest data proceeds. This is because the backup could take hours, so journal archiving needs to get started on all updates after the timestamp against which the backup is happening.

You always configure journal archiving as part of a database backup; however, the journaling itself actually happens per forest. This means that a full restore after a primary database failure has the potential to be "jagged"—with some forests having journal frames referencing transactions beyond that which other forest journals (which were a bit behind in their writes) know anything about. By default, a restore will recover all of the data from the backup, even if jagged. That means you get all of the data, but some transactions might be only partially present. For some use cases, this is proper; for others, it's not. The journal archives track how current they are by doing a write at least once per second, and the *Administrator's Guide* includes an XQuery script you can execute to put things back into the last "safe" timestamp, reverting any jagged transactions.

INCREMENTAL BACKUP

You can also bolster your backup strategy by turning on incremental backups, and MarkLogic will periodically save any new data since the last full backup (or previous incremental backup). An organization might set full backups to occur every week, with daily incremental backups running in between, and run journal archiving to minimize potential data loss. MarkLogic determines what to save during an incremental backup by scanning the `Timestamps` files. Incremental backups are much faster to execute than full backups since you're saving a fraction of your data. Incremental backups also

provide quicker recovery of data compared to replaying frames from a journal archive. Users can implement different combinations of full backups, incremental backups, and journal archiving to balance storage-space requirements with needs for recovery granularity.

When an incident occurs and you need to restore data, MarkLogic will look for the most recent incremental backup. If journal archiving is turned on, it can recover any data updates since the last incremental backup by replaying journal frames. If you are recovering to a point in time, it will take the nearest incremental backup after that time (if possible), and roll back. This is because rolling back is typically faster than replaying journal frames to move forward from an earlier backup.

Incremental backups also offer point-in-time recovery, similar to what journal archiving offers. You enable this feature by turning on the "retain until backup" setting under the database's merge policy in the Admin Interface (or by calling `admin:database-set-retain-until-backup()`). This will keep deleted documents in stands until those documents have been saved in a backup. By including the deleted documents along with the rest of the data, an incremental backup can reconstitute a database to its state at any point in time since the previous backup. Point-in-time recovery using incremental backup allows you to discard journal archives for those time periods, reducing your storage footprint.

FAILOVER AND REPLICATION

Failover allows a MarkLogic cluster to continue uninterrupted in the face of prolonged host failure.

One of the reasons hosts in a cluster exchange heartbeat information once per second is to determine when a failure has occurred. If a host has not received a heartbeat from another host for a configurable timeout (the default is 30 seconds), and also has not received any heartbeat information indicating that any other host has received a heartbeat from that host, the uncommunicative host is disconnected from the cluster.

If the disconnected host is an E-node and has no forests mounted locally, then all other hosts in the cluster can continue as normal; only requests initiated against the disconnected host will fail, and the load balancer can detect and route around the failure by sending requests to other E-nodes. The cluster simply needs enough warm or hot standby E-nodes in the cluster to handle the redistributed load.

Should the disconnected host be a D-node, there needs to be a way to get the data hosted by that D-node back online, complete with all of its recently committed transactions. To do this, you have your choice of two approaches: Shared-Disk Failover

and Local-Disk Failover. Failover will only happen if the host taking over is part of a quorum of greater than 50% of the hosts in the cluster.[15] (For administrative details see the Scalability, Availability, and Failover Guide.)

SHARED-DISK FAILOVER

Shared-Disk Failover requires that multiple hosts in a cluster have access to the same forest data. This can be done with a clustered filesystem: Veritas VxFS, Red Hat GFS, or Red Hat GFS2. Every D-node stores its forest data on a Storage Area Network (SAN) that's potentially accessible by other servers in the cluster at the same path. Should one D-node server fail, it will be removed from the cluster and other servers in the cluster with access to the SAN will "remotely mount" each of its forests. This can also be done with a Network File System (NFS), where servers connect to the forests via the network. Note that each forest in a MarkLogic cluster is only ever mounted by one host at a time.

With Shared-Disk Failover, the failover D-nodes 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. All failover hosts need sufficient spare operating capacity to handle their own forests as well as any possible remotely mounted forests.

When the failed D-node comes back online, it doesn't automatically remount the forests that were remotely mounted by other hosts. This avoids having the forests "ping pong" between hosts in the event that the primary host has a recurring problem that takes some time to solve. The administrator should "restart" each forest when it's appropriate for the forest to be mounted again by the primary.

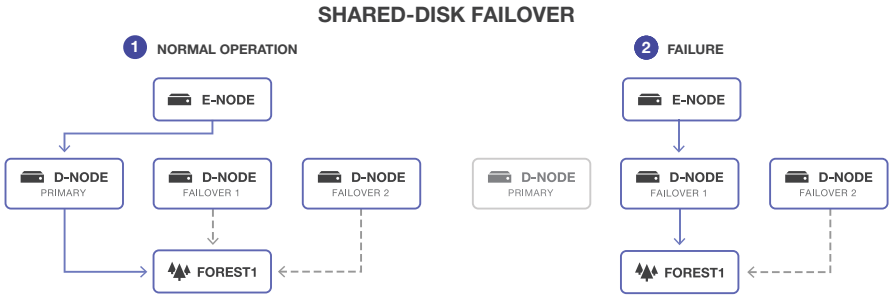

Figure 20: With Shared-Disk Failover, forest data is stored centrally on a SAN or NFS (**1**). In the case of failure of one D-node, other D-nodes can take over by remotely mounting the forest (**2**).

15 Notice that the need for a failover host to be part of a majority requires a minimum of three hosts in a cluster to support failover. If a majority wasn't required, a cluster could suffer split-brain syndrome in which a network severing between halves could result in each half cluster thinking it was the surviving half.

LOCAL-DISK FAILOVER

Local-Disk Failover uses intra-cluster forest replication. With forest replication, all updates to one forest are automatically replicated to another forest or set of forests, with each forest physically held on a different set of disks (generally cheap local disks) for redundancy.

Should the server managing the primary copy of the forest go offline, another server managing a replica forest (with a complete copy of the data) can continue forward as the new primary host. When the failed host comes back online, any updated data in the replica forest will be resynchronized back to the primary to get them in sync again. As with Shared-Disk Failover, the failed D-node won't automatically become the primary again until the data has been resynchronized and the replica forest administratively restarted.

MarkLogic starts forest replication by performing fast bulk synchronization for initial "zero day" synchronization (in the admin screens you'll see its status as *async replicating*). It also does this if a forest has been offline for an extended period. During this phase the forest is not yet caught up and therefore can't step in during a failover situation. Once the forests are in sync, MarkLogic continually sends journal frames from the primary forest to the replica forest(s) (the admin status here is *sync replicating*). During this time, the replica forests are ready to step in should there be a problem with the primary. The replay produces an equivalent result in each forest, but the forests are not "byte for byte" identical. (Imagine that one forest has decided to merge while the other hasn't.) Commits across replicated forests are synchronous and transactional, so a commit to the primary is a commit to the replica.[16]

A D-node usually hosts six primary forests, and it's good practice to "stripe" its forest replicas across at least two other D-nodes in an arrangement where every D-node has six replica forests from a variety of hosts. This ensures that if the primary fails, the work of managing its forests is spread across as many machines as possible to minimize the performance impact while the host is failed.

Each type of intra-cluster 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 disks, and doesn't require a clustered filesystem or fencing software. Local-Disk Failover doubles the ingest load because both forests index and merge independently. Local-Disk scales automatically as hosts are added by adding more controllers and spindles, while Shared-Disk would just carve the bandwidth of a SAN or NAS into smaller and smaller pieces of the pie. Shared-Disk may also be competing with other applications for

16 While async replicating, MarkLogic throttles writes to the primary forest to a cap of 50% as much data as is being sent over the network to the replica, to ensure that replication will eventually catch up.

bandwidth. Local-Disk is faster when recovering because the replica forest is ready to go at failover. Shared-Disk behaves like a host restart and must replay the journal before becoming available.

When configuring failover, don't forget to configure it for the auxiliary databases as well: Security, Modules, Triggers, Meters, and Schemas.

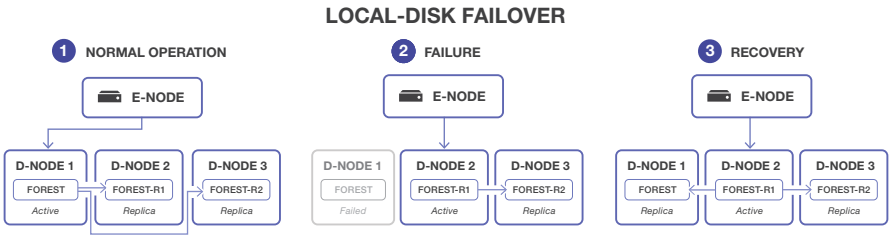

Figure 21: With Local-Disk Failover, forest data is automatically replicated to other D-nodes (**1**). In the case of failure of one D-node, a replica can take over duties as the primary host (**2**). When the failed D-node comes back online, it can be resynchronized from the replica (**3**).

DATABASE REPLICATION

What about when the whole cluster fails, such as with a data center power outage? Or, what if you just want multiple clusters geographically separated for efficiency? To enable inter-cluster replication like this, MarkLogic offers Database Replication.

Database Replication acts in many ways like Local-Disk Failover, except with the replica forests hosted in another cluster. In the administration screens or via the APIs, you first "couple" two clusters together. Each cluster offers one or more "bootstrap" hosts to initiate communication. Communication runs over the same XDQP protocol that is used for intra-cluster communication, but it runs on port 7998 instead of 7999. Database Replication is configured at the database level but, despite the name, actually happens on a per-forest basis. Normally, forests in the primary and replica database match up by name, but this can be manually overridden if necessary. The physical replication activity uses the same techniques discussed in Local-Disk Failover: bulk synchronization at first and then a continual sending of journal frames.

A few notable differences: First, with Local-Disk Failover, the primary and replica forests are kept in lock-step with each other. In Database Replication, there's a configurable lag time, by default 15 seconds, indicating how far behind a replica is allowed to be before the primary stalls new transaction commits. That's needed because between clusters there's usually a significant latency and often a less-reliable network. If the replica cluster goes fully down, the primary keeps going. (The idea is to not double the points of failure.)

Second, with Local-Disk Failover, the replica doesn't get used except in cases of failover. With database replication, the replica database can be put online to support read-only queries. This trick can help reduce the load on the primary cluster.

Finally, Database Replication crosses configuration boundaries, as each cluster has its own independent administration. It's up to the global administrator to keep the database index settings aligned between the primary and the replica. The Security database contents need to be aligned too, for user access to work correctly. Of course, that's easy—just run Database Replication on the Security database.

A primary forest can replicate to multiple replicas across multiple clusters. Clusters can even replicate to each other, but each database has one primary cluster at a time. The rest are always read-only.

What if you want to share data across geographies? Imagine that you want to replicate bi-directionally between locations A and B. Accounts near location A should be primary on Cluster A, accounts near location B should be primary on Cluster B, and all data should be replicated in case either cluster fails. You can do this, but you need two databases.

The act of failing over a database from one cluster to another requires external coordination because the view from within each cluster isn't sufficient to get an accurate reading on the world, and there are often other components in the overall architecture that need to fail over at the same time. An external monitoring system or a human has to detect the disaster and make a decision to initiate disaster recovery, using external tools like DNS or load balancers to route traffic accordingly and administering MarkLogic to change the Database Replication configuration to make the replica the new primary.[17]

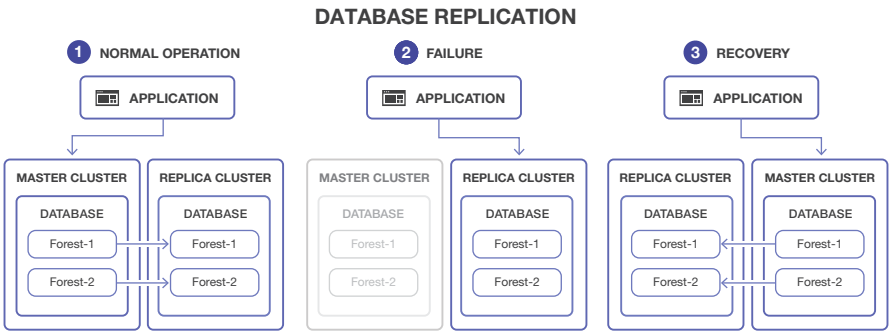

Figure 22: Database replication couples multiple clusters together and copies forest data between them (**1**). In the case of failure of the master cluster, applications can switch to using a replica (**2**). When the original master comes back online, it can assume duties as a replica (**3**).

17 Fun fact: When clusters couple their timestamps will synchronize.

Contemporaneous vs. Non-Blocking

Each application server has a configuration option called "multi-version concurrency control" to control how the latest timestamp gets chosen for lock-free read-only queries. When set to "contemporaneous" (the default), MarkLogic chooses the latest timestamp for which any transaction is known to have committed, even though there still may be other transactions at that same timestamp that have not yet fully committed. Queries will see more timely results, but may block waiting for contemporaneous transactions to fully commit. When set to "nonblocking," MarkLogic chooses the latest timestamp for which all transactions are known to have committed, even though there may be a slightly later timestamp for which another transaction has committed. Queries won't block waiting for transactions, but they may see less timely results.

Under Database Replication with a "hot" replica, you may want to configure the application servers running against the replica database as "nonblocking" to ensure that it doesn't have to wait excessively as transactions stream in from the primary forests, or possibly wait forever if contact with the primary gets severed with data in a jagged state. This is not the default, so it's an important item for an administrator to be aware of. For an application server running against a primary database it's usually best to stay "contemporaneous."

FLEXIBLE REPLICATION

While Database Replication strives to keep an exact and transactionally consistent copy of the primary data in another data center by transmitting journal frames, Flexible Replication enables customizable information sharing between systems by transmitting documents.

Technically, Flexible Replication is an asynchronous, non-transactional, trigger-based, document-level, inter-cluster replication system built on top of the Content Processing Framework (CPF). 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 the HTTP 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 primary into multiple documents on the replica. Or it can simply filter the documents, deciding which documents to replicate 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 Database Replication. 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 functions documentation.

QUERY-BASED FLEXIBLE REPLICATION

Flexible Replication can also be combined with MarkLogic's alerting if you need fine-grained control over what documents get replicated to what nodes as well as who gets to see what based on permissions. This feature is known as Query-Based Flexible Replication (QBFR). QBFR is useful in the case of heterogeneous networks where large central servers are mixed with smaller systems, such as laptop computers out in the field. All of the systems run MarkLogic server, but the laptops only require a subset of the documents that are stored on the main servers. Use cases for QBFR include military deployments, where field units only need information specific to their location, and scientific applications, where remote researchers only need data related to their field of study.

As described previously, alerting lets you define queries that specify which documents in a data set a user is interested in. When used in QBFR, the queries define the documents to be replicated from the master clusters to the replica clusters (which could be simple one-node clusters as in the laptop example). For example, queries in the military example might define geospatial coordinates, while queries in the scientific example might define technical keywords. Each replica cluster in the field has a query associated with it; when a document is inserted or updated on the central master cluster, a reverse query is run to determine which of the stored queries match the document. The matching queries determine to which clusters the document is replicated. Each stored query is also associated with a user, allowing role-based control of replication.

Pull-based replication is typically the preferred method of operation with QBFR. This allows field clusters that might have times of limited or no connectivity to retrieve documents when they have network access. Replication can also be bi-directional. A field cluster can act as a master when it connects and transfer any new data it has collected back to the home base. For example, military field units might send back reconnaissance data, with the data's distribution controlled by additional queries.

REBALANCING

Rebalancing in MarkLogic redistributes content in a database so that the forests that make up the database each have a similar number of documents. Spreading documents evenly across forests lets you take better advantage of concurrency among hosts. A

database allowed to have most of its documents in one forest would do most of the work on that forest's host, whereas redistributing the documents will spread the work more evenly across all of the hosts. In this way, rebalancing results in more efficient use of hardware and improved cluster performance.

Rebalancing is triggered by any reconfiguration of a database, such as when forests are added or retired.[18] If you plan to detach and delete a forest, you'll want to retire it first. This ensures that any content in that forest first gets redistributed among the remaining forests. If you just detach and delete a forest without retiring it, any content in that deleted forest is lost.

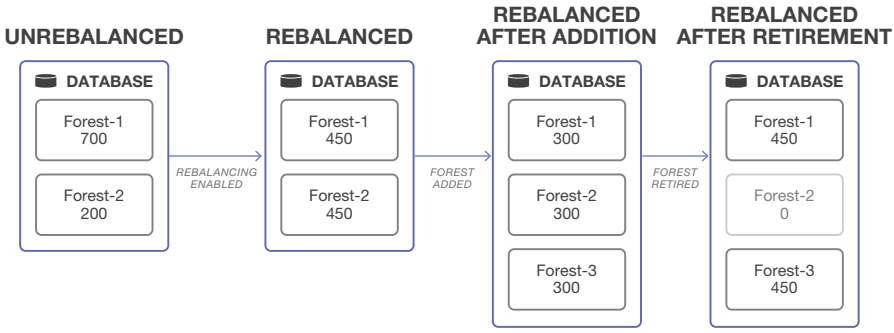

Figure 23: Rebalancing keeps documents evenly distributed among a database's forests.

ASSIGNMENT POLICIES

Rebalancing works based on an *assignment policy*, which is a set of rules that determine what document goes into what forest. A database's assignment policy applies to rebalancing as well as how documents are distributed to forests in the first place during ingest.

An assignment policy defines how data can be distributed horizontally across multiple hosts to improve performance. This is what MarkLogic is doing when it allocates documents in a database across multiple forests and those forests exist on different hosts. But even if you're running MarkLogic on a single host, spreading documents across more than one forest can improve performance because it takes advantage of parallel processing on a host's multiple cores.

There are four assignment policies: bucket, legacy, statistical, and range.

The *bucket* policy (the default) uses an algorithm to map a document's URI to one of 16,000 "buckets," with each bucket being associated with a forest. (A table mapping

18 You can control how aggressively rebalancing occurs by setting a throttle value, which establishes the rebalancer's priority when it comes to the use of system resources.

buckets to forests is stored in memory for fast assignment.) Employing such a large number of buckets—many more than the number of forests in a cluster—helps keep the amount of content transferred during redistribution to a minimum.[19]

The *legacy* policy uses the same algorithm that older MarkLogic releases used when assigning documents during ingestion. It distributes documents individually based on a hash of its URI. The legacy policy is less efficient than bucket because it moves more content than the bucket policy (the addition of one new forest greatly changes almost every document's assignment), but it is included for backward compatibility. The bucket and legacy policies are deterministic since a document will always end up in the same place given the same set of forests.

The *statistical* policy assigns documents to the forest that currently has the least number of documents of all the forests in the database. The statistical policy requires even less content to be moved compared with the bucket policy, but the statistical policy is non-deterministic—where a document ends up will be based on the content distribution in the forests at a given time. For this reason, the statistical policy can only be used with strict locking.[20]

The *range* policy is used in connection with tiered storage and assigns documents based on a range index. This means that documents in a database can be distributed based on things like their creation or update time. This allows recent documents or those that are more frequently accessed to be stored on faster solid-state media and older documents to be stored on slower disk-based media. For more information, see the "Tiered Storage" section. With the range policy, there will typically be multiple forests within each range; documents are assigned to forests within a range based on the algorithm that is used with the statistical policy.

MOVING THE DATA

Under the covers, rebalancing involves steps that are similar to how you move documents between forests programmatically. During a rebalancing event, documents are deleted from an existing forest and inserted into a new forest, with both steps happening in a single transaction to maintain database consistency.

For greater efficiency, the documents are transferred not one at a time but in batches, with the sizes of the batches varying with the assignment policy (and how resource-intensive the policy's operations are). Furthermore, for the statistical policy, a difference

19 Suppose there are M buckets and M is sufficiently large. Also suppose that a new forest is added to a database that already has N forests. To again get to a balanced state, the bucket policy requires the movement of N x (M/N − M/(N+1)) x 1/M = 1/(N+1) of the data. This is almost ideal. However, the larger the value of M is the more costly the management of the mapping (from bucket to forest).

20 A consequence of strict locking is that MarkLogic Content Pump (MLCP) can't use the fastload option.

threshold needs to be reached among the forests to trigger rebalancing. This keeps rebalancing from occurring unnecessarily when there is a small difference in forest counts.[21]

HADOOP

MarkLogic leverages Apache Hadoop—specifically the MapReduce part of the Hadoop stack—to facilitate bulk processing of data. The Hadoop MapReduce engine has become a popular way to run Java-based, computationally intensive programs across a large number of nodes. It's called MapReduce because it breaks down all work into two tasks. This first is a coded "map" task that takes in key-value pairs and outputs a series of intermediate key-value pairs. The second is a "reduce" task that takes in each key from the "map" phase along with all earlier generated values for that key, then outputs a final series of values. This simple model makes it easy to parallelize the work, running the map and reduce tasks in parallel across machines, yet the model has proven to be robust enough to handle many complex workloads. MarkLogic uses MapReduce for bulk processing: for large-scale data ingestion, transformation, and export. (MarkLogic does not use Hadoop to run live queries or updates.)

At a technical level, MarkLogic provides and supports a bi-directional connector for Hadoop.[22] This connector is open source, written in Java, and available separately from the main MarkLogic Server package. The connector includes logic that enables coordinated activity between a MarkLogic cluster and a Hadoop cluster. It ensures that all connectivity happens directly between machines, node-to-node, with no machine becoming a bottleneck.

A MarkLogic Hadoop job typically follows one of three patterns: 1. Read data from an external source, such as a filesystem or HDFS (the Hadoop Distributed File System), and push it into MarkLogic. This is your classic ETL (extract-transform-load) job. The data can be transformed (standardized, denormalized, deduplicated, reshaped) as much as necessary within Hadoop as part of the work. 2. Read data from MarkLogic and output to an external destination, such as a filesystem or HDFS. This is your classic database export. 3. Read data from MarkLogic and write the results back into MarkLogic. This is an efficient way to run a bulk transformation. For example, the MarkMail.org project wanted to "geo-tag" every email message with a latitude-longitude location based on IP addresses in the mail headers. The logic to determine the location based on IP was written in Java and executed in parallel against all messages with a Hadoop job running on a small Hadoop cluster.

21 Suppose the average fragment count of the forests is AVG and the number of forests is N. A threshold is calculated as max(AVG/(20*N),10000). So, data is moved between forests A and B only when A's fragment count is greater than AVG + threshold and B's fragment count is smaller than AVG − threshold.

22 MarkLogic supports only specific versions of Hadoop, and only certain operating systems. See the documentation for specifics.

A MapReduce job reading data from MarkLogic employs some tricks to make the reading more efficient. It gets from each forest a manifest of what documents are present in the forest (optionally limiting it to documents matching an ad hoc query constraint, or to subdocument sections), divides up the documents into "splits," and assigns those splits to map jobs that can run in parallel across Hadoop nodes. Communication always happens between the Hadoop node running the map job and the machine hosting the forest. (Side note: In order for the Hadoop node to communicate to the MarkLogic node hosting the forest, the MarkLogic node has to have an open XDBC port, meaning it needs to be a combination E-node/D-node.)

The connector code also internally uses the little-known "unordered" feature to pull the documents in the order they're stored on disk. The `fn:unordered($sequence)` function provides a hint to the optimizer that the order of the items in the sequence does not matter. MarkLogic uses that liberty to order the results in the same order they're stored on disk (by increasing internal fragment ID), providing better performance on disks that optimize sequential reads.

A MapReduce job writing data to MarkLogic also employs a trick to gain performance. If the reduce step inserts a document into the database and the connector deems it safe, the insert will use in-forest placement so that the communication only needs to happen directly to the host with the forest.

As an administrator, whether you want to place a Hadoop process onto each MarkLogic node or keep them separate depends on your workload. Co-location reduces network traffic between MarkLogic and the MapReduce tasks but places a heavier computational and memory burden on the host.

The MarkLogic Content Pump (MLCP), discussed later, is a command-line program built on Hadoop for managing bulk import, export, and data copy tasks. MarkLogic can also leverage the Hadoop Distributed File System (HDFS) for storing database content. This is useful for tiered storage deployments, discussed next.

A World with No Hadoop

What about a world with no Hadoop? You could still do the same work, and people have, but usually for simplicity's sake, people have loaded from one machine, driven a bulk transformation from one machine, or pulled down an export to one machine. This limits performance to that of the one client. Hadoop lets an arbitrarily large cluster of machines act as the client, talking in parallel to all of the MarkLogic nodes, and that speeds up data flow into and out of MarkLogic.

TIERED STORAGE

All storage media are not created equal. Fast but expensive storage is great for high-value documents that are accessed often. Slower but cheaper storage, including storage in the cloud, can be a good fit for older data that is rarely accessed but needs to be kept for historical or auditing purposes.

MarkLogic's tiered storage lets you automatically store database documents on the media that's most appropriate. Documents can be saved to different locations based on range-index properties—for example, the date the documents were created or last modified. This is in contrast to how documents are typically assigned, which is with algorithms that ensure that documents are spread evenly across all forests. By storing database documents depending on access needs, tiered storage can help users get better performance at lower costs.

You set up tiered storage by organizing a database's forests into *partitions*. Each partition is associated with a start and end value from the range index as well as a storage location. Imagine a tiered storage database that uses the last-modified date as the index. (You can have MarkLogic automatically track this value for documents with the "maintain last modified" database setting.) You could define a "Newest" partition for the most recently modified data, setting the range to the years 2014 to 2016 and the location to a directory on an SSD. An "Intermediate" partition might handle data from 2010 to 2013 and be set to a regular directory on a local disk. An "Archival" partition could handle data from 2000 to 2009 and be set to cloud storage.

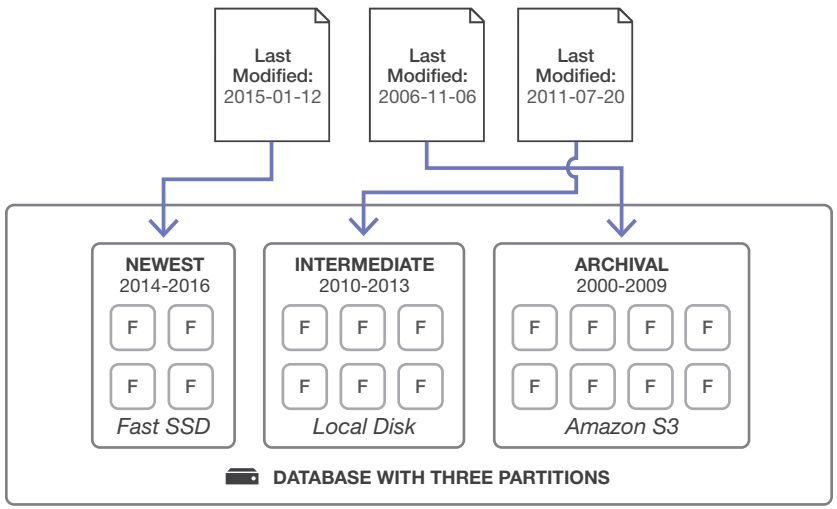

Figure 24: With tiered storage, you can organize a database's forests into partitions and then insert documents into the partitions based on a range index value.

When you perform a document insertion, the document is mapped to a partition based on its range-index value. In our example, a document last modified on January 12, 2015, would be assigned to a forest in the "newest" partition and saved to the SSD. If the partition includes multiple forests, the document is assigned to the forest in the partition that has the fewest documents.

Tiered storage offers various operations to maintain your data over time:

- As documents age and you want to move them to lower-cost tiers, you can **migrate partitions** to different storage locations. Built-in functions and REST endpoints make this easy to do, even between local and shared storage locations.

- As the database grows, you can **add forests** to partitions and MarkLogic will rebalance the data within the forests to keep them performing efficiently. (For more details, see the "Rebalancing" section.)

- Similarly, you can **retire forests** from partitions if your storage needs decline. Documents in retired forests are redistributed to other forests in the partition.

- You can **redefine ranges** for your partitions. This results in rebalancing of the documents within and between partitions.

- You can take partitions **online and offline**. Offline forests are excluded from queries, updates, and most other operations. You can take a partition offline to archive data and save RAM, CPU, and network resources but bring it back online quickly if you need to query it again.

Note that when moving documents to different tiers, such as when you age out documents to less-expensive storage, migrating is more efficient than rebalancing. MarkLogic's migration operations copy forests in a partition all at once; updates that involve rebalancing (redefining a partition range, for instance) move documents in much smaller batches, which is computationally more expensive.

Also note that partitioning documents with a range index is different from setting a Fast Data Directory for a database, which is another way to leverage fast but expensive storage such as SSDs. Defining a Fast Data Directory is optional when setting up your database; defining one allows MarkLogic to leverage fast storage for critical operations and to improve overall system performance. See "Storage Types within a Forest" for details.

SUPER-DATABASES

Another part of MarkLogic's tiered storage strategy is the super-database. A super-database lets you organize your documents into separate databases stored on different

storage media but still query those documents as a single unit. For queries to work, the super- and sub-databases must have the same configuration. Querying multiple data sources at once and then aggregating the results is also known as *federated search*.

For example, let's say that you have a Database A of active documents on fast, expensive storage and a Database B of archival documents on slower, lower-cost storage. You can create a super-database and associate it with Database A and Database B. (The super-database itself can also contain forests of documents, although that is not recommended since those forests cannot be queried independently of the associated sub-databases.) With this arrangement, you can query just the high-value documents in Database A, just the archival documents in Database B, or the entire corpus by querying the super-database.

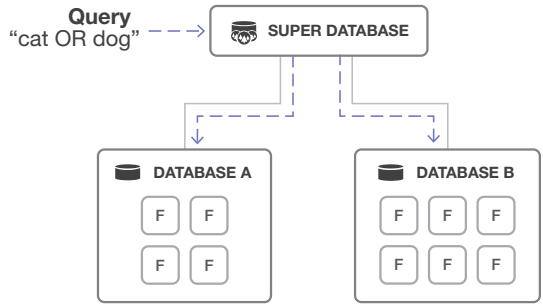

Figure 25: A super-database lets you query multiple databases as a single unit.

There are limitations on super-databases. You cannot perform updates on documents that reside in a sub-database via a super-database, and you must perform the updates directly on the sub-database where the documents reside. You also cannot have a super-database with more than one level of sub-databases beneath it. This is to avoid circular references—i.e., two databases referencing one another.

HDFS AND AMAZON S3

A key to tiered storage is the ability to store less-active data on lower-cost storage media. Two low-cost options are Hadoop Distributed File System (HDFS) and Amazon S3 (Simple Storage Service).

HDFS is the storage part of Hadoop. Because HDFS is open source and runs on commodity hardware, it can store data less expensively than traditional shared storage solutions such as SAN (storage area network) and NAS (network-attached storage). Once you configure MarkLogic to use HDFS, you can specify HDFS as a storage directory for a database using the "hdfs:" prefix.

Amazon S3 is cloud-based storage service offered through Amazon Web Services. S3 customers can interact with documents stored on the service through various interfaces,

including REST. S3 storage is inexpensive because of its enormous scale. In 2013, Amazon reported that more than two trillion objects were being stored on S3. To set up S3 on a MarkLogic cluster, you submit your S3 credentials, which are stored in the Security database. Then you can specify an S3 location as a storage directory for a database using the "s3:" prefix.

Although using S3 for storage on MarkLogic can be cheap, there is a performance tradeoff. Data stored on S3 is "eventually consistent," meaning that after you update a document, there may be a latency period before you can read the document and see the changes. For this reason, S3 can't be relied upon for document updates or journaling, nor can it be used for Shared-Disk Failover. S3 is recommended for use with read-only forests, which don't require journaling. For tiered storage, this can be appropriate for archival partitions containing documents that are rarely accessed and never updated. S3 is also a good option for backup storage.

AGGREGATE FUNCTIONS AND UDFS IN C++

Aggregate functions compute singular values from a long sequence of data. Common aggregate functions are count, sum, mean, max, and min. Less common ones are median (the middle value), percentile, rank (a numeric placement in order), mode (the most frequently occurring), variance (quantifying how far a set of values is spread out), standard deviation (the square root of the variance), covariance (how much two random variables change together), correlation (quantifying how closely two sets of data increase or decrease together), and linear model calculation (calculating the relationship between a dependent variable and an independent variable).

All of the above aggregate functions and more are provided out of the box in MarkLogic. You can perform any of these functions against any sequence, usually a sequence of values held within range indexes. Sometimes the functions operate against one range index (as with count or standard deviation) and sometimes several (as with covariance or linear model calculations). When run against range indexes they can operate in parallel across D-nodes, across forests, and even across stands (to the extent the underlying algorithm allows).

What if you don't see your favorite aggregate function listed in the documentation? For that, MarkLogic provides a C++ interface for defining custom aggregate functions, called user-defined functions (UDFs). You build your UDF into a dynamically linked library, package it as a native plug-in, and install the plug-in in MarkLogic Server. You gain the same parallel execution that the native aggregate functions enjoy. In fact, the advanced aggregate functions listed above were implemented using the same UDF framework exposed to end-users. MarkLogic calls this execution model "In-Database MapReduce" because the work is mapped across forests and reduced down until the E-node gets the final result. When writing a UDF, you put your main logic in `map()` and `reduce()` functions.

LOW-LEVEL SYSTEM CONTROL

When scaling a system, there are a few administrative settings you can adjust that control how MarkLogic operates and can, if used judiciously, improve your loading and query performance.

1. **"Last Modified" and "Directory Last Modified" options:** These options track the last modified time of each document and each directory by updating its property sheet after each change. This is convenient information to have, but it incurs an extra fragment write for every document update. Turning off these options saves that bit of work. Starting with MarkLogic 7, these options are disabled by default, but in earlier versions and in systems upgraded from earlier versions they will likely be enabled.

2. **"Directory Creation" option:** When set to "automatic," MarkLogic creates directory entries automatically, so if you create a document with the path `/x/y/z.xml`, MarkLogic will create the directory entries for `/x/` and `/x/y/` if they don't already exist. When it's set to "manual," you have to create those directories yourself if you want them to exist. Directories are important when accessing a database via WebDAV, where the database has to look and behave like a filesystem, and they're important if you're doing directory-based queries, but they otherwise aren't necessary. Setting directory creation to "manual" saves the ingestion overhead of checking if the directories mentioned in the path already exist and the work of creating them if they don't. This option is by default disabled starting with MarkLogic 7.

3. **"Locking" options:** MarkLogic's locking strategy ensures that documents in a database have distinct URIs as they are ingested. What setting you need depends on your database's assignment policy, described in the "Rebalancing" section. The default bucket and legacy policies require "fast" locking, which means only the URIs for existing documents are checked for uniqueness. The statistical and range policies require "strict" locking, which checks the URIs for both new and existing documents. If you're absolutely certain that all of the documents you're loading have distinct URIs, as often happens with a bulk load, you can relax this restriction and set locking to "off."

4. **"Journaling" option:** MarkLogic supports three journaling options. The "strict" option considers a transaction committed only after it's been journaled and the journal has been fully flushed to disk. This protects against MarkLogic Server process failures, host operating system kernel failures, and host hardware failures. This option is always used with forests configured for Shared-Disk Failover. The "fast" option considers a transaction committed after it's been journaled, but the

operating system may not have fully written the journal to disk. The "fast" option protects against MarkLogic Server process failures but not against host operating system kernel failures or host hardware failures. The "off" option does not journal at all and does not protect against MarkLogic Server process failures, host operating system kernel failures, or host hardware failures. For some specialized cases where the data is recoverable, such as during a bulk load, it may be appropriate to set journaling "off" temporarily to reduce disk activity and process overhead.

5. **Linux Huge Pages and Transparent Huge Pages:** Linux Huge Pages are two-megabyte blocks of memory; normal pages are just four kilobytes. Besides being bigger, they also are locked in memory and cannot be paged out. Employing Huge Pages gives MarkLogic more efficient access to memory and also ensures that its in-memory data structures remain pegged in memory. MarkLogic recommends setting Linux Huge Pages to 3/8 the size of your physical memory and recommends against using Transparent Huge Pages, which while enabled by default on Red Hat 6 are still buggy. You'll see advice to this effect in the ErrorLog.txt should MarkLogic detect anything less.

6. **Swap space:** If the MarkLogic process uses all available memory, the OS will use swap space as a temporary memory store. Once this happens, things will get slow—very slow. Having swap space will allow your performance to gradually degrade over time instead of just crashing when the OS can't allocate more memory. If your hardware has less than 32 gigabytes RAM, then use 1x available RAM for swap. If your hardware has more than 32 gigabytes RAM, use 32 gigabytes RAM.

7. **Red Hat Linux deadline I/O scheduler:** The deadline I/O scheduler improves performance and ensures that operations won't starve for resources and possibly never complete. On Red Hat Linux platforms, the deadline I/O scheduler is required.

8. **Forests:** Forests are in most ways independent from each other, so with multi-core systems, you can often benefit by having more forests to get more parallel execution. Also, forests can be queried in parallel by different threads running on different cores. It's true that the stands within a forest can be queried in parallel by separate threads, but after a merge, there might be just one stand and so no chance for parallelization. This makes it important to configure a sufficient number of forests for your deployment.

For more on best practices for optimizing MarkLogic performance, see the white paper Performance: Understanding System Resources.

OUTSIDE THE CORE

That completes our coverage of MarkLogic internals, the core of the system. There are actually a lot of great and important technologies in the ecosystem around MarkLogic, some officially supported and some open source. We'll cover a few of them here in the last major section.

CONFIGURATION MANAGER

Configuration Manager offers a read-only view of system settings suitable for sharing with non-administrator users. It also includes a Packaging feature that makes it easy to export and import these settings in order to move application server and database configurations between machines.

MONITORING

A suite of monitoring tools let you track all kinds of performance statistics describing what's going on within a MarkLogic Server cluster. These include web-based interfaces for viewing real-time and historical information about server operation as well as plug-ins for Nagios, the popular open source monitoring tool. Performance data is stored in a `Meters` database and is also available via a REST API, allowing you to build your own custom monitoring applications.

MARKLOGIC CONTENT PUMP (MLCP)

The MarkLogic Content Pump (MLCP) is an open source, MarkLogic-supported, Java-driven, Hadoop-assisted command-line tool for handling import, export, and data copy tasks. It can run locally (on one machine) or distributed (on a Hadoop cluster). As data sizes grow, it can be advantageous to perform import/export/copy jobs in parallel, and Hadoop provides a standard way to run large Java parallel operations.

MLCP can read as input regular XML/text/binary documents, delimited text files, aggregate XML files (ones with repeating elements that should be broken apart), Hadoop sequence files, documents stored within ZIP/GZip archives, and a format specific to MLCP called a "database archive" that contains document data plus the MarkLogic-specific metadata for each document (collections, permissions, properties, and quality) in a compressed format. MLCP can output to regular documents, a series of compressed ZIP files, or a new database archive. It can also run direct copies from one database to another.

For performance reasons, when importing and copying MLCP by default runs 100 updates in a batch and 10 batches per transaction. This tends to be the most efficient way to bulk-stream data in or out. Why? Because each transaction involves a bit of overhead. So for maximum speed, you don't want each document in its own transaction; you'd rather they be grouped. But lock management has its own overhead, so you don't want too many locks in one transaction either. Doing 1,000 documents in

a transaction is a good middle ground. Each batch gets held in memory (by default, you can also stream), so dividing a transaction into smaller batches minimizes the memory overhead. MLCP can also perform direct forest placement (which it calls "fastload"), but only does this if requested, as direct forest placement doesn't do full duplicate URI checking. MLCP, as with vanilla Hadoop, always communicates directly to the nodes with the target forests.

Additionally, MLCP's Direct Access feature enables you to bypass MarkLogic and extract documents from a database by reading them directly from forests on disk. This is primarily intended for accessing data that is archived through tiered storage. Let's say that you have data that ages out over time and doesn't need to be available for real-time queries through MarkLogic. You can archive that data by taking the containing forests offline but still access the contents using Direct Access.

For most purposes, MLCP replaces the open source (but unsupported) RecordLoader and XQSync projects.

CONTENT PROCESSING FRAMEWORK

The Content Processing Framework (CPF) is another officially supported service included with the MarkLogic distribution. It's an automated system for managing document lifecycles: transforming documents from one file format type to another or one schema to another, or breaking documents into pieces.

Internally, CPF uses properties sheet entries to track document states and uses triggers and background processing to move documents through their states. It's 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 them are designed to execute automatically, based on the outcome of other steps in the process.

OFFICE TOOLKITS

MarkLogic offers Office Toolkits for Word, Excel, and PowerPoint. These toolkits make it easy for MarkLogic programs to read, write, and interact with documents in the native Microsoft Office file formats.

The toolkits also include a plug-in capability whereby you can add your own custom sidebar or control ribbon to the application, making it easy to, for example, search and select content from within MarkLogic and drag it into a Word document.

CONNECTOR FOR SHAREPOINT

Microsoft SharePoint is a popular system for document management. MarkLogic offers (and supports) a Connector for SharePoint that integrates with SharePoint, providing more advanced access to the documents held within the system. The connector lets you mirror the SharePoint documents in MarkLogic for search, assembly, and reuse, or it lets MarkLogic act as a node in a SharePoint workflow.

DOCUMENT FILTERS

Built into MarkLogic behind the unassuming xdmp:document-filter() function is a robust system for extracting metadata and text from binary documents that handles hundreds of document formats. You can filter office documents, emails, database dumps, movies, images, and other multimedia formats, and even archive files. The filter process doesn't attempt to convert these documents to a rich XML format, but instead extracts the standard metadata and whatever text is within the files. It's great for search, classification, or other text-processing needs. For richer extraction (such as feature identification in an image or transcribing a movie) there are third-party tools.

LIBRARY SERVICES API

Library Services offers an interface for managing documents, letting you do check-in/check-out and versioning. You can combine the Library Services features with role-based security and the Search API to build a content management system on top of MarkLogic.

COMMUNITY-SUPPORTED TOOLS, LIBRARIES, AND PLUG-INS

The MarkLogic Developer Site (http://developer.marklogic.com) also hosts or references a number of highly useful projects. Many are built collaboratively on GitHub (https://github.com/marklogic), where you can contribute to their development if you are so inclined.

Converter for MongoDB

A Java-based tool for importing data from MongoDB into MarkLogic. It reads JSON data from MongoDB's mongodump tool and loads data into MarkLogic using an XDBC Server.

Corb2

A Java-based tool designed for bulk content reprocessing of documents stored in MarkLogic. It works off of a list of database documents and performs operations against them. Operations can include generating a report across all documents, manipulating individual documents, or a combination thereof.

ML Gradle

A plug-in for the Gradle build automation system. ML Gradle can deploy MarkLogic applications and interact with other features such as MarkLogic Content Pump. ML Gradle works primarily via the MarkLogic REST API.

ml-lodlive

LodLive is an RDF browser that lets you visually explore semantic data across distributed endpoints. ml-lodlive makes it easier to use the LodLive browser with data stored in MarkLogic via the MarkLogic REST API.

MarkLogic Workflow

A project that aims to provide useful methods for defining CPF pipelines, including ways to model CPF pipelines and then generate pipelines from the models.

MarkMapper

A tool for modeling and persisting objects in Ruby to a MarkLogic database. MarkMapper is based on the MongoMapper ORM, which does the same for Ruby and MongoDB.

MLPHP

A PHP API that lets you store documents, manage document metadata, and execute search queries in MarkLogic on a web server running PHP. It communicates with MarkLogic via the MarkLogic REST API.

MLSAM

An XQuery library that allows easy access to relational database systems from the MarkLogic environment. MLSAM lets you execute arbitrary SQL commands against any relational database and captures the results as XML for processing in MarkLogic. MLSAM uses HTTP calls to send the SQL query to a servlet, and the servlet uses JDBC to pass the query to the relational database.

oXygen

An XML editing environment with MarkLogic integration. Support includes multiple server connections, XQuery execution and debugging, and resource management and editing through WebDAV.

Roxy

A hugely popular utility for configuring and deploying MarkLogic applications to multiple environments. With Roxy, you can define your app servers, databases, forests, groups, tasks, etc. in local configuration files. Roxy can then remotely create, update, and remove those settings from the command line. This project also provides a unit test framework and XQuery MVC application structure.

Sublime Text Plug-in

An add-on to the popular Sublime Text source-code editor. Features include syntax highlighting and code completion for MarkLogic's built-in functions, linting to check XQuery for errors, and integration with the MarkLogic online documentation.

xmlsh

A command-line shell for XML. xmlsh provides a familiar scripting environment that's tailored for automating XML processes. It includes an extension module for connecting to MarkLogic.

XQDT

A set of open source plug-ins for the Eclipse IDE. XQDT supports syntax highlighting and content-assisted editing of XQuery modules, as well as a framework for executing and debugging modules in a supported XQuery interpreter.

XQuery Commons

A MarkLogic umbrella project that encompasses small, self-contained components. These small "bits and bobs" do not constitute complete applications in their own right but can be re-used in many different contexts. XQuery components available include HTTP, Date, and Search utilities.

XQuery XML Memory Operations

An XQuery library for performing operations on XML in memory. By using XPath axes, node comparisons, and set operators, the library can make changes to XML while only reconstructing nodes within the direct path of the nodes being altered.

xray

A framework for writing XQuery unit tests on MarkLogic Server. It can output test results as HTML, XML, xUnit-compatible XML, JSON, and plain text, making it easy to integrate with many CI servers.

BUT WAIT, THERE'S MORE

There are far more contributed projects than can be mentioned in this book. You'll find a full list at MarkLogic Developer Community Tools. Of course, you're welcome to join in and contribute to any of these projects. Or, feel free to start your own!

Still have questions? Try searching the MarkLogic Product Documentation (http://docs.marklogic.com), ask them on MarkLogic's general development mailing list or post them to Stack Overflow. Good luck!

MARKLOGIC CORPORATION
999 Skyway Road, Suite 200 San Carlos, CA 94070
+1 650 655 2300 | +1 877 992 8885 | www.marklogic.com | sales@marklogic.com

999 Skyway Road, Suite 200 San Carlos, CA 94070
+1 650 655 2300 | +1 877 992 8885
www.marklogic.com | sales@marklogic.com