Inside MarkLogic Server

Its data model, indexing system, update model, and operational behaviors

April 28, 2013

Jason Hunter, MarkLogic Corporation

April 28, 2013

Inside MarkLogic Server

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

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

The paper is organized into sections. The first section provides a high-level overview of MarkLogic Server. The next few sections explain MarkLogic's core indexes. The sections after that explain the transactional storage system, multi-host clustering, and the various connection options. At this point there's a natural stopping point for the casual reader, but those who read on will find sections covering advanced indexing features as well as topics like replication and failover. The final section discusses the ecosystem built up around MarkLogic.

This major update to the original version adds discussion on features introduced in MarkLogic 5 and MarkLogic 6: Database replication, journal archiving with point-in-time recovery, multi-statement transactions, XA transactions, Hadoop integration, tiered storage, large binary support, compartment security, SQL/ODBC access for BI tool integration, REST API access, user-defined functions, JSON support, document filters, path range indexes, application packaging, and system performance monitoring. It also adds coverage for several older features: text value matching, phrase handling, stop words, lexicons, and document compression.

Table of Contents

What Is MarkLogic Server?

MarkLogic Server is an Enterprise NoSQL Database[1]. It fuses together database internals, search-style indexing, and application server behaviors into a unified system. It uses XML documents as its data model, and stores the documents within a transactional repository. It indexes the words and values from each of the loaded documents, as well as the document structure. And, because of its unique Universal Index, MarkLogic doesn't require advance knowledge of the document structure (its "schema") nor complete adherence to a particular schema. Through its application server capabilities, it's programmable and extensible.

MarkLogic Server (referred to from here on as just "MarkLogic") clusters on commodity hardware using a shared-nothing architecture and differentiates itself in the market by supporting massive scale and fantastic performance — customer deployments have scaled to hundreds of terabytes of source data while maintaining sub-second query response time.

> MarkLogic Server is a document-centric, transactional, search-centric, structure-aware, schema-agnostic, programmatic, high performance, clustered, database server.

Let's look at all of this in more detail.

Document-Centric

MarkLogic uses documents, often written in XML, as its core data model. 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. In some cases the data might start formatted as XML documents (for example, Microsoft Office 2007 documents or financial products written in FpML), but if not, it can be transformed to XML documents during ingestion. Relational databases, in contrast, with their table-centric data models, can't represent data like this as naturally and so either have to spread the data out across many tables (adding complexity and hurting performance) or keep this data as unindexed BLOBs or CLOBs

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

In addition to XML, MarkLogic can store JSON, text, and binary documents. JSON documents are internally transformed to XML for purposes of indexing. Text documents are indexed as if each was an XML text node without a parent. Binary documents are by default unindexed, with the option to index their metadata and extracted contents.

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 purpose-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 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-Centric

When people think of MarkLogic they often think of its text search capabilities. The founding team has a deep background in search: Chris 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, 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, 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 11th and April 13th where the message contains both "wedding cake" and "empire state building" (case and punctuation insensitive) where the phrases have to be within 15 words of each other but the message 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 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. Or consider some other examples.

Find hidden financial exposure:

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

Review images:

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

Find a person's phone number from their emails:

> From a large corpus of emails find those sent by a particular user, sort them reverse chronological, and locate the last email they sent which had a footer block containing a phone number. Return the phone number.[2]

Schema-Agnostic

MarkLogic indexes the XML 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 for 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 efficiently index and query 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.

---

[2] How do you identify footers and phone numbers? You can do it via heuristics, with the markup added during ingestion. You can mark footer blocks as a `<footer>` element and a phone number entity as a `<phone>` element. Then it's easy to query for phone numbers within footers limited by sender name or address. MarkLogic includes built-in entity enrichment or you can use third-party software.

Of course, MarkLogic also works fine with data that does fully adhere to a schema. You can even use MarkLogic to enforce a schema, if you'd like.[3]

Programmatic

To interact with and program MarkLogic Server at the lowest level you have your choice between two W3C-standard programming languages, XQuery and XSLT. 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. Each language has its advantages; you don't have to pick. You can mix and match between the languages: XSLT can make in-process calls to XQuery and vice-versa. MarkLogic also exposes a REST API and a SQL interface over ODBC.

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

HTTP and HTTPS Web Protocols

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

XDBC 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 libraries in other languages. XDBC and the XCC client libraries make it easy to integrate MarkLogic into an existing application stack.

REST Protocol

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

SQL/ODBC Protocol

> 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 exposed as columns.

---

[3] 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.

WebDAV File Protocol

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

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 product in its core design. And they are, from the highly-optimized native C++ code to the algorithms we'll discuss later. For MarkLogic customers it's routine to compose advanced queries across terabytes of data that make up many millions of documents and get answers in less than a second. The largest live deployments now exceed 100 terabytes and tens of billions of documents.

> In the words of Flatirons Solutions, an integration partner: "It's fast. It's faster than anybody else. It's way, way faster. It blows you away it's so fast. It's actually so fast that it... makes it possible to do real-time queries against large XML databases; makes it possible to do large-scale personalization from XML data; makes it possible to think about classic problems in an entirely new way."

Clustered

To achieve speed and scale beyond the capabilities of one server, MarkLogic clusters across commodity hardware connected on a LAN. A commodity server in 2013 might be a box with 2 CPUs, each 8 cores, 128 gigabytes of RAM, and either a large local disk or access to a SAN. On a box such as this a rule of thumb is you can store roughly 1 to 2 terabytes of data, sometimes more and sometimes less, depending on your use case.

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 an E-node should fail, there's no host-specific state to lose, just the in-process requests (that can be retried), and the load balancer can route traffic to the remaining E-nodes. Should a D-node fail, that subset of the data needs to be brought online by another D-node. You can do this by using either a clustered filesystem (allowing another D-node to directly access the failed D-node's

storage and replay its journals) or intra-cluster data replication (replicating updates across multiple D-node disks, providing in essence a live backup).

Database Server

At its core you can think of MarkLogic as a database server — but one with a lot of features not usually associated with a database. It has the flexibility to store structured, unstructured, or semi-structured information. It can run both database-style queries and search-style queries, or a combination of both. It can run highly analytical queries too. It can scale horizontally. It's a platform, purpose-built from the ground up, that makes it dramatically easier to author and deploy today's information applications.

Core Topics

Indexing Text and Structure

Now that we've covered what MarkLogic is, let's dig into how it works, starting with its unique indexing model.

Indexing Words

Let's begin with a thought experiment. Imagine I give you ten documents printed out. I tell you I'm going to provide you with a word and you'll tell me which documents have 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 keep a list of which documents have 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 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 I'm going to 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, all document ids with the second, and intersect the lists. That results in the set of documents with both words. If you sort the term lists in advance and use a "skip list" implementation, the intersection can be done efficiently, and in logarithmic time. This is how MarkLogic resolves simple boolean queries.

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, all document ids with the second word, and do a list subtraction.

Indexing Phrases

Now let's say I'm going to ask you to find documents with a two-word phrase. You have three choices in how to resolve that query.

First, you can use the same data structure you have already: find documents that contain both words, and then actually look inside the candidate documents to determine if the two words appear together in a phrase.

Second, you can 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.

Third, you can create new entries in your inverted index where instead of a simple word as the lookup key you instead put 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 has the ability to take each of these approaches. The choice is made at runtime based on your database index settings. The *Database->Configuration* web page contains a list of index options that you can enable or disable via radio buttons. These settings control what term lists are maintained and what, if any, position data should be held about each entry in a term list.

If you have the *fast phrase searches* option enabled (the switch set to on), 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 ingestion performance to maintain the extra indexes.

If you have the *word positions* option enabled (another switch), 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.

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 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 thrown away during filtering.

It's important to note, 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 (such as *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.

Indexing Longer Phrases

Now 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 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. The two-word phrase can still be 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, so MarkLogic uses *fast phrase searches* to reduce the number of documents, and thus terms, that need to be processed. You'll notice MarkLogic doesn't require monolithic indexes. Instead, it uses lots of smaller indexes cooperating to resolve queries. After index resolution, MarkLogic filters the candidate documents to check if they're actual matches. That's the usual routine: Look at the query, decide what indexes can help, use the indexes in cooperation to narrow the results down to a set of candidate documents, then filter the documents to confirm the match. The more index options enabled, the tighter the candidate result set can be.

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'm going to ask you to find me 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.[4]

---

[4] We still call it a *term list* although technically it's more of a *fact list* because it's an inverted index tracking facts about a document not just terms within a document. The elements present in the document are one such fact.

Not many queries ask for documents containing just a named element, of course. So let's say the question is more complex. Let's try to find documents matching the XPath `/book/metadata/title` and for each return the title node. That is, we want documents having a `<book>` root element, with a `<metadata>` child, and a `<title>` subchild. 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 know their hierarchical relationship. It's like a phrase query that looks just for words.

Now imagine a parent-child index, which tracks parent-child element hierarchies. It's just like a *fast phrase searches* index except, instead of word relationships, it indexes element relationships. With it you can find documents having a `book/metadata` relationship (that is, a `<book>` as a parent of a `<metadata>`) and a `metadata/title` relationship also. That 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 parent-child index is so useful with XML indexing 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 a document there's a `<book>` parent of `<metadata>` and a `<metadata>` parent of `<title>` doesn't mean it's the same `<metadata>` between them. That's where filtering comes in: MarkLogic confirms each of the results by looking inside the document before the programmer sees them. While the document is open MarkLogic also extracts any nodes that match the query.

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

Indexing Values

Now what if we want to search for element values? Let's imagine I'm going to ask you for books published in a certain year. In XPath that can be expressed as `/book[metadata/pubyear = 2013]`. How do we resolve this efficiently?

What you want to do, thinking back to the paper-based approach, is maintain a term list for each element value. In other words, you can track a term list for documents having a `<pubyear>` equal to `2013`, 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.

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 the 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

behind the scenes uses hashes to store all term list keys, element-value or otherwise, for sake of efficiency. The element-value index has proven to be so efficient that it's always and automatically enabled within MarkLogic.

In the above example `2013` 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 2013 were surrounded by string quotes. Often this type fuzziness is sufficient. For other cases where data type encoding matters, it's possible to use a range index, discussed later.

Indexing Text Values

Now let's expand the query into a phrase match. Let's find books where the title equals the phrase "Good Will Hunting". You can express that as the XPath `/book[metadata/title = "Good Will Hunting"]`. As discussed above, MarkLogic maintains a term list for each element-value combination so there's a term list for <title> elements having the value "Good Will Hunting".

There's one wrinkle though. What version of string equality do we want? For example, the glyph of an A with an umlaut (**Ä**) can be written as one character (U+00C4) or as two characters (a regular A, U+0041, with a "combining character", U+0308). Should those two representations be seen as identical (because semantically and visually they are the same) or as different (because the character sequence is different)? There's no right answer, so you can specify the behavior you want with a "collation", a named algorithm for comparing strings. Unicode defines many different collation algorithms, giving each a URI name. You assign a collation as part of coding (in a header usually), and that collation determines how the XPath equality operates. Yes, text is complicated business.

The first option above (where the two representations mean the same thing) is called the "Root Collation" (and is the default) while the second (where the two representations are different) is the "Codepoint Collation". You can set a new default collation administratively, and you can also declare it locally at the top of your code files. There are hundreds of possible collations (such as whether you want whitespace and punctuation to matter or not), and the application server administration page includes a "collation builder" to help you craft the right collation URI.

MarkLogic maintains an element-value term list based on the exact sequence of characters. If you've configured Codepoint Collation as your collation, then that's how the XPath behaves, and then there's a singular term list available to do the precise element-value match.

What if you've configured a different collation, or are using the default Root Collation? There are too many combinatorial possibilities to have an element-value term list for each one, but MarkLogic has a fallback plan. MarkLogic maintains an additional element-value term list based on the *sequence of words in the value*, not just the *exact sequence of characters*. For this term list a new element value is calculated ignoring things like punctuation and whitespace. That's the value that gets hashed. So "Good Will Hunting"

and "Good Will: Hunting!" hash the same. It turns out this approach to creating a term list works well to support all the other kinds of collations. That's because, for example, if you want a punctuation sensitive match you can always start with a punctuation insensitive index and then filter.

As you're probably thinking right now, relying on collations to dictate string matching behavior is a bit convoluted. What you can do to avoid relying on collation is to use `cts:element-value-query()` and pass it a set of options to dictate the matching behavior desired for that query piece. Passing an option of "exact" uses the element-value term list based on character sequences, while other options ("case-insensitive", etc) use the term list based on the sequence of words.

Indexing Text with Structure

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 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, 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". Using these indexes we can additionally confirm out of indexes that the words "Good" and "Will" are both present directly under a `<title>` element, 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, MarkLogic has the *fast element phrase searches* and *element word positions* index options. The *fast element phrase searches* option, when enabled, maintains a term list for every element and pair of words within the element. It will have a term list for times when a `<title>` 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 the words are together in a phrase under the `<title>` element. 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.

Special Phrase Handling

If we query the following documents for the phrase "retract the landing gear", which XML should be considered a match?

Example A:

```
<steps>
<step number="3">Wait for the automatic pilot control to retract.</step>
<step number="4">The landing gear should now be ready to deploy.</step>
</steps>
```

Example B:

```
<warning>While waiting for the flaps to fully retract <footnote>The landing gear
need to be
stowed or else the flaps will remain deployed</footnote>, ensure that the automatic
pilot
control is at hand.</warning>
```

Example C:

```
<p>It is vital that you retract <b>the landing gear</b>, but leave the flaps fully
deployed.</p>
```

Example D:

```
<step number = "5">Partially retract <footnote>It is not recommended to fully
retract at
this point.</footnote> the landing gear.</step>
```

The naïve answer (ignoring markup) is A, B, and C. The default MarkLogic answer (requiring phrases be in contiguous text blocks) is no matches at all. The right answer is C and D. As English speakers, we know that's the right answer because we understand that `<b>` means bold and `<footnote>` means an inline side note. You can administratively give MarkLogic some hints so it knows this as well and can do the right thing. You configure `<b>` (as well as `<i>`, `<span>`, etc) as "phrase-through" elements and `<footnote>` (as well as `<aside>`, etc) as "phrase-around" elements. After that change, the indexes adjust and will handle this XML correctly for purposes of phrasing and proximity. MarkLogic includes built-in understanding for XHTML markup.

Phrasing is one situation where the index model and data model don't have to match or have a one-to-one correlation. Fields are another, discussed later.

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 2x or 3x of the XML source.

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 the data in a transactional repository. When the background reindexing runs it doesn't even stop the system from handling queries or updates. The system just background reloads and reindexes all 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 administration console. You can also control the "Reindexer Throttle" from 5 (most eager) to 1 (least eager) or just turn it off temporarily.

Relevance

When performing a full text query it's not enough to just find documents matching the given constraint. The results have to be returned in relevance order. Relevance is a mathematical construct whose idea is simple. Documents with more matches are more relevant. Shorter documents that have matches are more relevant than longer documents having the same number of matches. If the search includes multiple words, some common and some rare, appearances of the rare terms are more 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 paper.

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 ten documents 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".

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 in our explanation here, let's assume 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 also. 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. 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.[5]

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 ten 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 though, after filtering, 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 term lists are useful for indexing many other things, like 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, publication year, 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 a 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.

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

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

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

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; the privileges control what actions the user can perform (e.g. restarting the server). The implementation of most security checks is simple: just check whether 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 an associated 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.

For government customers, MarkLogic also provides "compartment security", a system where it's not enough that a user has a role that can view a document, the user needs to have *all* necessary roles to view a document. Imagine a document involving several top secret projects. You can't read it unless you have visibility to all the top secret projects. Internally, with compartment security the term lists for role visibility are intersected (AND'd) instead of unioned (OR'd).[6]

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

---

[6] 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.

documents which otherwise wouldn't have a place for an XML description. They're also useful for adding XML metadata to an XML document whose schema can't be altered. MarkLogic out of the box uses properties for tracking each document's last modified time.

Properties are represented as regular XML documents, held under the same URI (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 both on 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 I've use the word *document* to represent each unit of content. That's a bit of a simplification. MarkLogic actually indexes, retrieves, and stores something called *fragments*. The default fragment size is the document, and that's how most people leave it. However it's also possible to break documents into sub-document fragments through configured *fragment root* or *fragment parent* settings controlled via administration. 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 into a fragment root.

Fragment vs. Document

You can picture using fragmentation on a book. Perhaps a book is 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 made up of a series of fragments, one fragment for the `<book>` root element holding the metadata about the book, and a series of distinct fragments for each `<chapter>`. The book still exists as a document, with a 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 the chapter elements each into their own document as part of the load.

Perhaps 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 paper) you can substitute "document" and the statement will be correct for any databases where there's no fragmentation 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 to enable unless you're sure you need it.

Estimate and Count

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

The `xdmp:estimate()` call uses 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 the term list constraints.

The `fn:count()` call also uses indexes but then filters the fragments returned by the indexes to check which truly match the expression and how many times it matches per document. That filtering takes time (due mostly to disk IO), 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, as well as identical, from those of `fn:count()` — depending on the query, data schema, and index options. The estimate results are higher when there are fragments returned by the index system 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. The results might be lower, on the other hand, if there happen 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, and

furthermore it means 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 each result is a true match before returning it as a result. The `"unfiltered"` flag tells MarkLogic to skip the filtering step. It's a sharp tool, but very useful in certain cases, such as when you want to jump to the millionth search result. It's not practical to filter a million documents off disk; however, you can easily jump to the millionth document based on the index results, and unfiltered lets you do just that. If `xdmp:estimate()` matches `fn:count()` for the query, it means you'll be able to find the precise millionth result accurately.

The Range Index

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

1. Perform fast range queries. For example, you can provide a query constraint for fragments 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 not their lexical serialized value.
3. Quickly extract specific values from the entries in a result set. For example, you can get a distinct list of message senders from fragments in a result set, as well as the frequency of how often each sender appears. These are often called *facets* and displayed with search results as an aid in search navigation. 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. 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 paper.

To really understand the role of a range index, imagine 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 go ask everyone his or her birthday. That's the equivalent in a database of 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, suitable for many uses. That's the range index.

To configure a range index, you use the admin console to provide the document portion on which you want the range index to apply (it can be an element name, an element-attribute name, or even a path expressed in a form of simplified XPath); a data type (int, date, string, etc); and (for strings) a collation, which is in essence 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 fragment id get the fragment's range index value(s), or, for any range index value get the fragment 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 fragment ids and values, sorted by fragment ids; the other an array of structures holding values and fragment 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 fragment id" lookup array. Because the lookup array is sorted by value, there's a specific subsequence in the array that holds the values between the two user-defined endpoints. Each of those values in the range has a fragment id associated with it, and those fragment ids can be quickly gathered and used like a synthetic term list to limit the search result to fragments 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 for someone at the party, and it's easy to get the people's names because every birthday date has the person listed right next to it. If multiple people have the same birthday, you'll have multiple entries in the array with the same value but a different corresponding name. In MarkLogic, instead of people's names, the system tracks fragment ids.

Range queries can be combined with any other types of queries in MarkLogic. Say you want to limit results to those within a date range as above but also having a certain metadata tag. MarkLogic uses the range index to get the set of fragment ids in the range, uses a term list to get the set of fragment 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 it 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.[7]

Data-Type Aware Equality Queries

The same "value to fragment id" lookup array can support equality queries that have to be data-type aware. Imagine you're tracking not just the birthday but the exact time when every person was born. The challenge is that there's 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 your values are serialized the exact same way, you can use a term list index to match the string representation, but if the serialization can vary, it's best to use a range index because range indexes are based on the underlying data-type value. Only 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.

Extracting Values

To quickly extract specific values from the documents in a result set, MarkLogic uses the data structure that maps fragment ids to values. After first using inverted indexes to resolve the fragment ids that match a query, MarkLogic then uses the "fragment id to value" lookup array to find the values associated with each fragment id, quickly and without touching the disk. It can also count how often each value appears.[8]

---

[7] This is one reason to prefer the `cts:query` constructors over XPath. It's not possible to express anything with `cts:query` that isn't optimized.

[8] Astute readers will notice that the fragment 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()`.

The counts can be *bucketed* as well. With bucketing, instead of returning counts per value, MarkLogic returns counts falling within a range of values. You can, for example, get the counts per day, week, or month against source data listing specific dates. You specify the buckets as part of the query. Then, when walking down the range index, MarkLogic keeps count of how many entries 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 simple bucketing comparisons.

Optimized "Order By"

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

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

To perform optimized `order by` calculations, MarkLogic again uses the "fragment id to value" lookup array. For any given result set, MarkLogic takes the fragment 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 ten oldest or youngest. You'd limit your list first to partygoers from Seattle, extract their birthdays, then sort by birthday and finally return the list of people in ascending or descending order as you wish.

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

For more information on optimized order by expressions and the exact rules for applying them, see the *Query and Performance Tuning* guide (available at *http://docs.marklogic.com*).

Using Range Indexes for Joins

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

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

Here's example code:

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

The first block of code finds all the author ids for people who've been on Twitter for at least a year. It uses the `signup-date` range index to resolve the `cts: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 fragments associated with that author, with a potential disk seek per author. With a range index, MarkLogic can map author ids to fragment 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 Path Range Indexes 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 it 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).

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 a path range index that matches also, 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.

Lexicons

When configuring a database there are options to configure a "URI Lexicon" and a "Collection Lexicon". These are both 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.

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), it's not efficient to find all collections (so that's what the Collection Lexicon is for). The lexicons can calculate the document and collection URIs the same way regular range indexes extract values from within documents.[9]

---

[9] The URI Lexicon is identical to a range index with data type `anyUri`, namespace `http://marklogic.com/xdmp`, and localname `document`; while the Collection Lexicon is identical to a range index with data type `anyUri`, namespace `http://marklogic.com/xdmp`, and localname `collection`. Usually you access these lexicons with the `cts:uris()` and `cts:collections()` functions, but if you know the secret you can access them as regular range indexes: `cts:element-`

Data Management

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

What's on Disk: Databases, Forests, and Stands

A *database* is the maximum query unit within MarkLogic. A system can have multiple databases, and generally will, but each query or update request executes against a particular database.

A database consists of one or more *forests*. A forest is a collection of documents (mostly XML trees, thus the name), implemented as a physical directory on disk. Each forest holds a set of documents and all their indexes. A single machine may manage several forests, or in a cluster (when 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 can help with concurrency. Depending on the workload, you might have one forest corresponding to every physical core on a box, or sometimes a forest for every two cores, with each forest holding millions or tens of millions of documents and perhaps a hundred gigabytes. In a clustered environment, you can have a set of servers, each managing their own set of forests, all unified into a single database.[10]

Each forest holds zero or more *stands*. A stand (like a stand of trees) holds a subset of the forest data and exists as a physical subdirectory under the forest directory. It contains a set of compressed binary files with names like `TreeData`, `IndexData`, `Frequencies`, `Qualities`, and such. This is where the actual compressed XML data (in `TreeData`) and indexes (in `IndexData`) can be found.

A forest might contain a single stand, but it's more common to have multiple stands because stands help MarkLogic ingest data.

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 no stands. At some point a new document is loaded into MarkLogic, through an XCC, XQuery, REST, or WebDAV call; it doesn't matter, the effect is the same. MarkLogic puts this document in an *in-memory stand* and

---

`values(xs:QName("xdmp:document"))` and `cts:element-values(xs:QName("xdmp:collection"))`. This might sound like trivia, but you need to know this in order to use them in co-occurrence calculations.

[10] 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.

writes the action to an on-disk journal to maintain transactional integrity in case of system failure.

As new documents are loaded, the same happens to them; they're placed in the in-memory stand. A query request at this point will see all 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 will see the full view of data loaded at this point in time.

After enough documents are loaded, the in-memory stand will fill up and be flushed to disk, written out as an *on-disk stand*. Each new stand gets its own subdirectory under the forest directory, with names that are monotonically-increasing hexadecimal numbers. The first stand gets the lovely name `00000000`. That on-disk stand contains all 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 also 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 the in-memory stand 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 the data across all the 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 individual 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 singular stand out of them, coalescing and optimizing the indexes and data, as well as removing any previously deleted fragments, a topic we'll discuss shortly. After the merge finishes and the new on-disk stand has been fully written, and after all the current requests using the old on-disk stands have completed, MarkLogic deletes the old on-disk stands.

MarkLogic uses an algorithm to determine when to merge, based on the size of each stand. In a normal server running under constant load you'll usually see a few large stands, a few more mid-sized stands, and several more small stands. Over time the smaller stands get merged with ever-larger stands. Merges tend to be CPU- and disk-intensive, 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. A new document gets assigned to a forest basically at random unless you override the selection as part of the load call. 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 get saved using a dictionary-based compression scheme. In this scheme, the text gets tokenized (into words, whitespace, 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 gets 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 one 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.

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 from earlier, 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 and will mark them as deleted and create new fragments as necessary.

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 I'll explain.

Multi-Version Concurrency Control (MVCC)

In an MVCC system, changes are tracked with a timestamp number which increments as transactions occur within a cluster. Each fragment gets its own creation-time (the timestamp at which it was created) and deletion-time (the timestamp at which it was marked as deleted, starting at infinity for fragments not yet deleted). On disk you'll see these timestamps in the `Timestamps` file. Trivia buffs will note 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 to the normal term list constraints two extra constraints: first, that any fragments returned have to have been "created at or before the request timestamp" and second, that they have to have been "deleted after the request timestamp". It'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.

Time Travel

Normally a query acquires its timestamp marker automatically based on the time the query started. However, it's also possible for it to request a specific previous timestamp. MarkLogic calls this feature *time travel*. It lets you query the database as it used to be at any arbitrary point in the past, as efficiently as querying at present time. One popular use of 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 the changes back to a past timestamp (what's called "database rollback").

When doing time travel you have to consider merging. Merging normally removes deleted documents. If you want to travel into 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 to which you can time travel.

Locking

An update request, because it isn't read-only, has to use read/write locks to maintain system integrity while making changes. This lock behavior is implicit and not under the control of the user.[11] Read-locks block for write-locks; write-locks block for both read- and write-locks. An update has to obtain a read-lock before reading a document and a write-lock before changing (adding, deleting, modifying) a document. Lock acquisition is ordered, first-come first-served, and 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 number of locks and past retries) wins and the other update gets restarted.

MarkLogic differentiates queries from updates using static analysis. Before running a request, it looks at the code to determine if it includes any calls to update functions. If so, 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; useful when your request performs an update from evaluated or invoked code that can't be seen by the static analyzer.)

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

---

[11] For the most part it's not under the control of the user. The one exception is there's 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 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.

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

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. When finding an item, retrieving an item, or updating an item, that means 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 XML is a more expressive data format. One document can often describe an entity (a manifest, a legal contract, an email) completely.

Lifecycle of a Document

Now, to bring all 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 the document starts life with a document load request inside XQuery or XSLT. The request acquires a write-lock for the target URI as part of the `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, processing the queue of update work orders.

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[12]). The fragment gets added to the in-memory stand. At this point the fragment is considered a *nascent* fragment, a term you'll see sometimes on the administration console status pages. Being nascent means it exists in a stand but hasn't been fully committed. (On a technical level, nascent fragments have creation and deletion timestamps 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.

Once the fragment is placed into the in-memory stand, since our statement is complete, the request is ready to commit. It 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 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 gets 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,

---

[12] The data model is defined in exhaustive detail at http://www.w3.org/TR/xpath-datamodel/ 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.

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 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 timestamp to make the fragment live. Because it's an update, it has to mark the old fragment as deleted also, and 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 query doing time travel 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 allow deep time travel. In that case it will live on, sticking around in case any new queries want to time travel to see old fragments.

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 rollback 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 `session.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 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 gets 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 just 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.[13]

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 re-trying all 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.

---

[13] Multi-statement transactions were introduced in MarkLogic 6 but this is not new code. MarkLogic has had post-commit triggers for years which use the same system.

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 report prepared to commit, the TM instructs all participants to commit. If one or more participants is not prepared to commit, the TM instructs all participants to rollback. It's the point of view of the TM that decides 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.

Tiered Storage

In the next short section we'll see how MarkLogic uses different storage tiers to maximize performance while minimizing storage costs.

Fast Data Directory on SSDs

Solid-state drives (SSDs) have performance that's 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 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 setup to point to a directory built on a fast filesystem, such as one using SSDs. It's completely optional; if 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 merge (saving an in-memory stand included), MarkLogic will attempt to merge onto 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 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 the best 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 otherwise 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, they can be included in database backups, they can be part of the data replication actions

needed to support failover and high-availability, and they 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 needless to copy large binary documents from one stand to another. (For smaller binaries it doesn't really matter.)

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. 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" which 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.

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 need demands (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 for one of two roles. They can be Evaluators (E-nodes) or they can be Data Managers (D-nodes). E-nodes listen on a socket, parse requests, and generate responses. D-nodes hold data along with its associated indexes, and support E-nodes by providing them with the data they need to satisfy requests, 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 2 E-nodes and 10 D-nodes, with each D-node responsible for about 10% of the total data.

A load balancer usually spreads incoming requests across the E-nodes. An E-node processes the request and delegates out to the D-nodes for any subexpression of the request involving data retrieval or storage. If the request needs, for example, the ten 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 share 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 surprisingly simple. The first time you access the administration screens 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 (currently 16 Megs).

Compressed Tree Cache

> This cache holds the XML fragments after they've been read off disk. They're stored compressed, to reduce space and improve IO 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.[14]

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 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 the E-node has sufficiently a 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 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 concurrently.

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 select to remove the most stale entries from itself, even if there's a more stale entry in another partition.

---

[14] If ever you receive an "XDMP-EXPNTREECACHEFULL: Expanded tree cache full" error it doesn't mean you should increase the size of your expanded tree cache. That's like getting a faster CPU to deal with an infinite loop. 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.

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 managing its List Cache and Compressed Tree Cache at the stand level. 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 actually the case with MarkLogic.

MarkLogic manages locks in a decentralized way. Each D-node has the responsibility for managing the locks for the documents under its forest(s). It does this as part of its regular data access work. 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. Each D-node in fact doesn't have to check with any other hosts in the cluster to acquire locks on its subset of data. (This is the reason all fragments for a document always get placed in the same forest.)

In the regular high-frequency heartbeat communication sent between the hosts in a cluster, they report on which locks they're holding, in order to support the background deadlock detection thread.

The transaction timestamp is also managed in a decentralized way. As the very last part of committing an update, the D-node or D-nodes making the change look at the latest timestamp from their point of view, increase it, 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.[15]

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

[15] 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 does mean it's important for all machines in a MarkLogic cluster to have fairly accurate clocks.

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.

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 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 happens on the D-node. Locks would happen on the D-node side as well, but since this is a 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 selects a fragment id for filtering, it first looks to its own Expanded Tree Cache to see if it can find the fragment there. If it can't, 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 filters 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.

Lifecycle of an Update in a Cluster

Earlier we also covered the lifecycle of a fragment being updated. Let's revisit that scenario also, to see what happens in more detail and in a clustered environment.

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 in 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. The E-node picks the forest in a deterministic way (based on a hash of the URI) so as to ensure any other concurrent requests look to the same D-node for the write-lock.

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

What happens with an insert or update crossing multiple forests in the same transaction? MarkLogic uses a classic two-phase commit between the D-nodes to ensure the update either happens atomically or doesn't happen at all.[16]

Locking With Forest Placement

In normal circumstances MarkLogic chooses where to place new documents. It's also possible for an insertion call to request a specific placement. There's two ways to do this. First, a document insertion call can include forest placement keys (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 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 duplicate copies of the URI in other forests. That's left to the application. The Hadoop connector sometimes makes use of this advanced technique.

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 choices you have for programming and interacting with MarkLogic.

---

[16] In case you're curious about how a two-phase commit works: The coordinator picks a timestamp and tells all participating forests to prepare. The participating forests write to their journal files and update their fragment timestamps, then report they're prepared. If all participants report being prepared, the coordinator journals the transaction as having completed. If not, it can still be undone. In a replay situation, such as after a system crash, it's the coordinator's point of view that determines if the transaction completed or not.

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

Modules and Deployment

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

XQuery and XSLT code files can reside either on the filesystem or inside a database. Putting code on a filesystem has the advantage of simplicity. You just place the code (as `.xqy` scripts or `.xslt` templates) under a filesystem directory 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 every E-node is using the same codebase,

because each file just exists once in the database and doesn't have to be replicated across E-nodes or hosted on a network filesystem. You also have the ability to roll out a big multi-file change as an atomic update. With a filesystem deployment some requests might see the code update in a half-written state. Also, with a database you can use MarkLogic's security rules to determine who can make updates, and can expose (via WebDAV) remote secure access without a shell account.

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

You can find the full set of API documentation at http://docs.marklogic.com. It includes a robust search feature that's built, naturally, on MarkLogic.[17]

Output Options

With MarkLogic you can generate output in many different formats:

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

Single-Tier Web Deployment

MarkLogic includes a native web server with built-in SSL support. Incoming web requests can invoke XQuery or XSLT scripts 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.

---

[17] Tip: For fast access to function documentation go to http://docs.marklogic.com/*function-name*. You don't even need the namespace.

MarkLogic's built-in HTTP/HTTPS server gives you the option to write a full web site or REST endpoint 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, well-escaped 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. Everything is naturally properly escaped. With PHP, in contrast, 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.

Single-tier architecture isn't necessary of course. Maybe you want to fit MarkLogic into a multi-tier architecture.

XDBC/XCC for Java and .NET Access

The XDBC wire protocol provides programmatic access to MarkLogic from a separate language environment. MarkLogic officially supports open source Java and .NET client libraries, named XCC. There are community-developed open source libraries in other languages.

The XCC client library provides you with Java or .NET objects to manage connections and sessions, run code invocations, load content, pull result streams, and generally interact with MarkLogic. It feels pretty natural to people familiar with JDBC or ODBC.

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

XDBC and the XCC client offer their own set of advantages:

- It makes it easy to integrate MarkLogic into an existing application stack. If you have a pre-existing Java EE or ASP.NET system, MarkLogic can act as a support service. If you have an enterprise service bus, MarkLogic can jump on as an endpoint. If you need MarkLogic to speak your proprietary protocol, it's easy to front it with Java or .NET to make the translation happen.

- It's great for bulk loading. A loading pipeline can run Java- or .NET-based ETL (Extract/Transform/Load) services and push data into MarkLogic. It's simpler and more robust than sending files over HTTP.
- It enables non-web deployments. Not every application of MarkLogic involves the web.

For detailed information on programming XCC see the *XCC Developer's Guide* and the XCC API documentation.

REST API

What if you don't want to learn XQuery? 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 an open source Java client library. The library hides 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.

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.

Using the REST API offers several advantages:

- Quick to get started, no knowledge of XQuery required.
- Build applications using whatever language you like. Keep your favorite IDE, web framework, unit testing tools, and all the rest.

WebDAV: Remote Filesystem Access

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

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

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

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 or Tableau, which still expect 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 and it's possible to do text matching against that column using the SQL `MATCH` operator.

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.[18] 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. (Or for testing you can use the XQuery `xdmp:sql()` function or a 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 not "nullable" then the document doesn't match the required constraints and the document produces no rows.

QC for Remote Coding

Not actually a protocol into itself, but still widely used by programmers wanting raw access to MarkLogic, is the QC web-based code execution environment. QC stands for

[18] For XQuery programmers there's a little-known `cts:value-tuples()` function that also does the n-way co-occurrence.

"Query Console". It's a web application included with the server, just a set of XQuery scripts, that when accessed (password protected of course) enables you to run ad hoc XQuery or XSLT code from a text area in your web browser. It's a great administration and coding tool.

It includes syntax highlighting, multiple open queries groupable into workspaces, history tracking, state saved to the server, beautified error messages, the ability to switch between any databases on the server, and 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 under `http://yourserver:8000/qconsole`.

Advanced Topics

Advanced Text Handling

At the start of this paper 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 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 it's important to specify you desire a case-sensitive match. For example "Polish" and "polish" mean different things.[19] It's easy to specify this as part of your query, just pass a "case-sensitive" or "case-insensitive" option to each query term.[20] 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 lower-cased). If you conduct a query with a case-sensitive term MarkLogic relies 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

---

[19] Words like this that have different meanings when capitalized are called “capitonyms”. Another example: “March” and “march”. Or "Josh" and "josh". I notice that last one because my iPhone never auto-corrects my son's name to capitalize it. Any future children will have names that aren't capitonyms.

[20] 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 lower case, MarkLogic assumes case doesn't matter to you and treats it as case-insensitive. If the query term includes any upper case characters, MarkLogic assumes case does matter and treats it as case-sensitive.

they're more common you can gain efficiency by turning on the *fast case sensitive searches* index option. It tells MarkLogic to additionally maintain case-sensitive term list entries. With the index enabled, case-insensitive terms will use the case-insensitive term list entries while case-sensitive terms will use the case-sensitive term list entries, and all results will resolve quickly and accurately out of indexes.

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 match "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.

Stemmed Indexes

Stemming is another situation where MarkLogic provides optimized indexing options. Stemming is the process for 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". Running 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 searching for proper nouns or precise metadata tags.

MarkLogic gives you the choice on whether it should maintain stemmed or unstemmed indexes, or both. The options reside at the top of the database configuration, named *stemmed searches* and *word searches*. These indexes act somewhat like master index settings in that they impact all 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.

MarkLogic by default enables the *stemmed searches* option and leaves *word searches* disabled. Usually that's fine, but it can lead to surprises. Searching for "Good Will Hunting", if stemmed, will also match "Good Willing Hunt". If you don't like that, you can enable the *word searches* index and pass `"unstemmed"` as an option while constructing the search query constraint. The same query can include both stemmed (description) and unstemmed (title and author) constraints.

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 (encoded as XML) to modify the stemming and tokenization behavior. For details, see the *Search Developer's Guide*.[21]

---

[21] 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. This obscure fact has cost me real money because Amazon user searches for "Java servlets" doesn't stem to match my book title "Java Servlet Programming".

MarkLogic 6 provides basic language support for hundreds of languages, and provides advanced support — stemming, tokenization (breaking text into words), and collation (sorting) — for 13 languages: English, French, Italian, German, Spanish, Arabic, Persian/Farsi, Traditional Chinese, Simplified Chinese, Japanese, Korean, Russian, Dutch, and Portuguese. MarkLogic supports Unicode Level 5.2-0 and so can process, store, and run unstemmed searches against any language represented by Unicode. MarkLogic 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. 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 gets 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 just deals in stem roots.[22]

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's actually a few stemming options: There's "Basic" which indexes the shortest stem of each word; "Advanced" which indexes all stems of each word; and "Decompounding" which indexes all stems, and smaller component words of large compound words are also indexed. Each successive level of stemming improves the recall of word searches, but expands the index size.

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 are the target word. A higher term frequency increases relevance, but only logarithmically so the

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

effect gets 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 having a list of document ids, possibly locations (for things like proximity queries and phrase searches), and also term frequency data. All the numerical values are written using delta coding, to keep it as small on disk as possible. 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. You can watch as the server does the scoring math if you use the admin console to turn on the diagnostic flag `"relevance"`, which dumps the math to the server error log file.[23]

Most `cts:query` search constructs include a weight parameter for indicating the importance of that part of the query. A higher weight means it matters more. A zero weight means 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).

Documents can also 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. Negative quality documents get suppressed. Quality is programmatically set and can be based on anything you like. MarkMail.org, for example, weights results toward more recent mails by giving them higher quality. It 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 score based on quality and terms. By having a wide gamut you keep granularity.

---

[23] Once upon a time the error log (`ErrorLog.txt`) only held errors. These days it's more a general purpose log file.

Stop Words

A stop word is a word considered so common and with such 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 with MarkLogic 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 titled *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 if 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 provides administrators with the ability to turn on and off various indexes. The list 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 `<redacted>` 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 to something declared by an administrator. Let's say 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)
))
```

In a field definition you can include these weights and they get 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 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 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 its child elements. Its singular value will be the person's full name as a string. You can also 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 additional term lists, one for each value.

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 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, and then use it repeatedly by intersecting it with the day constraint.

For the code below, you can assume we have a set of documents in the `fact` namespace that record data about visiting browsers: their name, version, various attributes, and the session id in which they appear. You can think of those documents as the XML-equivalent of a fact table. You can assume we also have a set of documents in the `dim` namespace (the XML equivalent of a dimension table) that record 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* to join between 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 the exact 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 Server 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 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 changes so often that it doesn't make sense to use the MarkLogic 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 register that `cts:query` for repeated optimized use. As the user interacts with the data, all 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 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 re-generated. Since in-memory stands are very small compared to on-disk stands, re-generating 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). It supports efficient intersection between regions (does this polygon intersect or contain that polygon), as well as calculating shortest distances between a point and a region.

The geospatial indexes fully support the polar regions, 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.

At a technical level, MarkLogic's geospatial indexes don't use quad trees or R-trees; instead they 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 a 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.[24]

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

[24] The worst-case performance on bounding boxes? A thin vertical slice.

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.

Polar regions and the anti-meridian complicate matters. In those cases the server generates multiple smaller regions that don't cross the special boundaries, and unions the results.

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

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

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

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

Reverse Query Use Cases

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

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

Without reverse query indexing, for each new document or set of documents you'd have to loop over all your queries to see which 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 paper, but suffice to say it's based on document training sets and finding similarity between document term vectors. Reverse queries provide a rule-based alternative to training-based classifiers. You define each classification group as a `cts:query`. That query must be satisfied for membership. With reverse query, each new or modified document can be placed quickly into the right classification group or groups.

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

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 8AM, who listens to rock, pop, and hip-hop, and wants $10 for gas. She requires a female passenger within five miles of her start and end points. You have a passenger, a woman who will pay up to $20. Starting at "3001 Summit View Dr, San Ramon, CA 94582" and traveling to "400 Concourse Drive, Belmont, CA 94002". She requires a non-smoking car, and won't listen to country music. A matchmaking query can match these two women to each other, as well as any other matches across potentially millions of people, in sub-second time.

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

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

```
    <preferences>{
      cts:and-query((
        cts:element-value-query(xs:QName("gender"), "female"),
        cts:element-geospatial-query(xs:QName("from"),
          cts:circle(5, $from)),
        cts:element-geospatial-query(xs:QName("to"), cts:circle(5, $to))
      ))
    }</preferences>
  </driver>)
```

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

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

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

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

It searches across passengers requiring that your preferences match them, and also that their preferences match you. The combination of rules is defined by the `cts: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 on both parties must match each other. Within MarkLogic even a complex query such as this (notice the use of negative queries, range queries, and geospatial queries, in addition to regular term queries) runs efficiently and at scale.

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

To support the first phase, MarkLogic maintains a custom index in which it gathers all 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 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 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 the required constraints are true, short-circuiting whenever possible. If all the constraints are satisfied, MarkLogic determines that the nominated query document id is in fact a reverse query match to the source document.

At this point, depending on the user's query, MarkLogic can return the results for processing as a final answer, or feed the document ids into a larger query context. In the matchmaker 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 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 just as special leaf nodes on the DAG, able to resolve directly and with no need for the cutpoints.

Lastly, what about geospatial queries? MarkLogic uses the same cutpoint approach as for range queries, but in two dimensions. To support the first phase, MarkLogic generates a set of geographic bounding boxes, each with its own set of query document ids to

nominate should the source document contain a point within that box. For the second phase, like with range queries, the geographic constraint acts as a special leaf node on the DAG, complete with precise geospatial comparisons.

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

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 using the time-based scheduled backup feature, or they can be initiated programmatically via an XQuery script run by an administrator. You specify a database to backup and a target location. Backing up a database backs up its configuration files, all the forests in the database, as well as the corresponding security and schemas databases. It's particularly important to backup the security database because MarkLogic tracks role identifiers as `xs:long` values and the backup forest data can't be read without the corresponding roles existing in the security database.

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

Typical Backup

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

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

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 duration of the snapshot. Any changes to those files go to different disk blocks. It 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. Its default is "all", meaning that the forest is fully read-write. 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 generating an error immediately. 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, into which are placed a record of all the database transactions committed after the backup. Note that this is a separate journal from the main rotating journal kept with the database, and should be on a separate storage medium.

Having the journal in the backup provides two major advantages. First, it reduces how much data you can lose in the event of a catastrophe happening to 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, to restore things very much like they were 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 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 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 restore after a primary database failure has the potential to be "jagged" -- with some forests having journal frames referencing transactions that other forest journals (which were a bit behind in their writes) know nothing about but should. By default a restore will recover all the data from the backup, even if jagged. That means you get all data, but some transactions might be only partially present. 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 that put things back into the last "safe" timestamp, which will revert any jagged transactions.

Failover and Replication

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

A cluster uses a voting algorithm to determine if a host is down. The voting algorithm gets its data from each host's view of the cluster. If there is a quorum of hosts (more than 50%[25]) whose view of the cluster is such that they believe a particular host is down (because it's been unresponsive for some configurable timeout period, default 30 seconds), then the other hosts in the cluster treat that host as if it is down and the cluster goes on without it, disconnecting it 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 its recently committed transactions. To do this you have your choice of two approaches: Shared-Disk Failover and Local-Disk Failover. (For administrative details see the *Scalability, Availability, and Failover Guide*.)

Shared-Disk Failover

Shared-Disk Failover requires a clustered filesystem: Veritas VxFS, Red Hat GFS, or Red Hat GFS2. Every D-node stores its forest data on a 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. 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.

---

[25] Notice that the need for a majority vote requires a minimum of three hosts in a cluster where failover is required. If it weren't for the majority vote, 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 you have all writes to one forest 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 re-synchronized 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 re-synchronized 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 it's not yet caught up and so 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 for example that one forest has been told to merge while the other hasn't.) Commits across replicated forests are synchronous and transactional, so a commit to the primary is a commit to the replica.[26]

A D-node usually hosts several forests, and it's good practice to "stripe" its forest replicas across the other D-nodes. That ensures that if the primary fails, the work of managing its forests gets shared across numerous other machines.

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 disk, 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 due to 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 also may be competing with other applications for bandwidth.

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

---

[26] 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.

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 you first "couple" two clusters together. Each cluster offers one or more "boostrap hosts" to initiate communication. Communication runs over the same XDQP protocol that is used for intra-cluster communication, but 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 should stall new transaction commits. That's needed because between clusters there's usually a significant latency and often a less reliable network, and this lag accommodates for that. If the replica cluster goes fully down, the primary keeps going. (The idea isn't to 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 any number of replicas across any number of 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 shard data across geographies? Imagine 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.

Contemporaneous vs Non-Blocking

Each application server has a configuration option "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 for that 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, an application server running against a replica database should be configured "nonblocking" to ensure it doesn't have to wait excessively as transactions stream in from the primary, or possibly wait forever if contact with the primary gets severed in the middle of a transaction. This is *not* the default, so it's an important item for an administrator to configure! For an application server running against a primary database it's usually best to stay "contemporaneous".[27]

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), single-master, 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 an HTTP-friendly protocol. Documents can be pushed (to the replica) or pulled (by the replica), depending on your configuration choice.

[27] Fun fact: When clusters couple their timestamps will synchronize.

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 not to, and which documents should have only pieces replicated. The filter can even wholly transform the content as part of the replication, using something like an XSLT stylesheet to automatically adjust from one schema to another.

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

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

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 as 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 a coded "map" task that takes in key-value pairs and outputs a series of intermediate key-value pairs, and 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. The simple model makes it easy to parallelize the work, running the map and reduce tasks in parallel across machines, yet the model has proven 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.[28] 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 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

[28] MarkLogic 6 supports only specific versions of Hadoop, and only certain operating systems. See the documentation for the specifics.

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.

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 same 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 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.

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, or 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 the MarkLogic nodes, and that speeds up how quickly data can flow into and out of MarkLogic.

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 the above aggregate functions 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 above? 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 plugin, and install the plugin in MarkLogic Server. You gain the same parallel execution as 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. Turn off the "Last Modified" and "Directory Last Modified" options on your database. By default MarkLogic tracks 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.

2. Change the "Directory Creation" option from "automatic" to "manual" on your database. When set to "automatic" MarkLogic creates directory entries automatically, so if you create a document with the path */x/y/z.xml* MarkLogic creates the directory entries for */x/* and */x/y/* if they don't already exist. When 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 aren't necessary otherwise. Turning 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.

3. Change the "Locking" option from "strict" to either "fast" or "off" on your database. The default behavior, "strict", enforces mutual exclusion on existing documents and on new documents. It makes sure the same URI doesn't appear multiple times, a very important fact. This guarantee involves cross-cluster lock coordination, a potential bottleneck during bulk loads. If you're absolutely certain all the documents you're loading have distinct URIs, as often happens with a bulk load, you can relax this restriction. When set to "fast", locking enforces mutual exclusion on existing documents but not on new documents. When set to "off", locking does not enforce mutual exclusion on existing documents or on new documents.

4. Change the "Journaling" option from "fast" to "off". MarkLogic actually 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. Configure Linux Huge Pages if on Linux. Linux Huge Pages are 2 Megabyte blocks of memory, as opposed to normal pages which are just 4 Kilobytes. Besides being bigger, they also are locked in memory and cannot be paged out. Employing Huge

Pages on a MarkLogic system gives MarkLogic more efficient access to memory and also ensures its in-memory data structures remain pegged in memory. MarkLogic recommends setting Linux Huge Pages to 3/8 the size of your physical memory. You'll see advice to this effect in the ErrorLog.txt should MarkLogic detect anything less.

6. Ensure you've configured a proper amount of swap space, which can be up to twice the amount of physical memory. Because of how MarkLogic forks processes and uses memory mapped files, operating systems generally want there to be a good amount of swap space available even if in normal operations it's not going to be used. See the *Installation Guide* for details on proper configurations for each operating system.

7. Ensure you've configured the "Deadline" scheduler if on Linux. The Deadline scheduler improves performance and ensures operations won't starve for resources and possibly never complete. The Deadline scheduler is a required configuration option.

8. Ensure you've configured a sufficient number of forests. Forests are in most ways independent from each other, so with multi-core systems you can often benefit by having more forests (up to as many cores as you have) to get more parallel execution. Also during queries, 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.

Outside the Core

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

Application Services

Application Services includes a set of services that make it easier to load, manage, query, and display content with MarkLogic, as well as manage a MarkLogic installation. The services are written in XQuery and XSLT (and sometimes JavaScript) but are officially supported and provided in the MarkLogic Server distribution. There are several services:

Application Builder, a browser-based application for building search-centric web applications. It guides you as you choose a user interface, define a search syntax, show and sort the search results, manage facets, embed charts and graphs, and adjust rendering rules. It's a way for non-programmers to build a search-centric site or for programmers to get a head start.

Information Studio, a browser-based application for managing document loads and transformations. It makes it easy to load in bulk from a directory or by dragging and

dropping onto a browser applet, and includes a set of APIs and a framework for plugging in your own specific document collectors, transformers, and load policies. It also includes XQuery functions to expose these features to programmers; see the functions in the `info:` and `infodev:` namespaces.

Library Services API, an interface for document management services, letting you do check-in/check-out and versioning of documents.

Search API, a code-level library designed to simplify creating search applications. Used by the Application Builder and 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 on `cts:query` constructs can appreciate its help in transforming user-entered strings into `cts:query` hierarchies.

Configuration Manager, a read-only view of system settings suitable for sharing with non-administrator users. 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.

Visualization Widgets, a set of drop-in HTML5 components that simplify adding interactive data visualizations (line chart, bar chart, pie chart, heat map, point map) to a MarkLogic web-based application. They're used by Application Builder, or in your own code.

Monitoring, a mechanism to track all kinds of performance statistics on what's going on within a MarkLogic Server cluster. Includes a built-in web-based monitoring dashboard, as well as plugins for Nagios. All the data is also available via a REST API.

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). When 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, 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 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.

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, 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 the steps 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. Based on technology from Perceptive Software (formerly known as ISYS) it 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. Information Studio includes an optional "filter" transformation that leverages this feature.

Unofficial Tools, Libraries, and Connectors

The MarkLogic developer site also hosts or references a number of unofficial but still highly useful projects. All are open source under the Apache License v2.0.

Roxy

Roxy (RObust XQuerY Framework) is a lightweight framework for developing XQuery applications based on the MVC model familiar to users of Ruby on Rails and CakePHP. It includes an MVC framework, a unit testing framework, and a command-line deployment system.

xray

xray is a framework for writing XQuery unit tests. There's also xquery-unit, xqut, and PerformanceMeters (which despite the name can be used for more than just testing performance).

xmlsh

xmlsh is a command line shell for XML, with extensions for MarkLogic to store/retrieve documents and run queries. It has a syntax similar to `sh`, and can run in batch mode or interactive.

Ant Tasks

Apache Ant is a widely used build tool originally used for building Java programs more efficiently than a Makefile, but now used to drive any sort of automated process. It includes a large set of built-in tasks that you combine to create a build recipe. The Ant Tasks give an Ant build script the ability to interact with MarkLogic to load documents,

evaluate queries, and invoke modules. It's an easy way to script interaction with MarkLogic.

XQDebug

XQDebug is a customer-contributed open source browser-based debugger for MarkLogic applications. It presents a convenient user interface on top of MarkLogic's low-level debugger API. You can connect to any request, view the current expression and variables, view the call stack, set breakpoints, create watch expressions, and step through your code. Note: The Oxygen XML Editor (commercial) also supports XQuery debugging.

XQDT

XQDT is a set of plugins for the Eclipse IDE, providing support for syntax highlighting and content-assisted editing of XQuery modules, and a framework for executing and debugging modules.

xqDoc

xqDoc is like Javadoc but for XQuery. It suggests a convention for decorating XQuery functions with API documentation, and provides tools that convert the inline documentation into a user-friendly web presentation.

Converter for MongoDB

The converter is a Java-based tool to import data from MongoDB into MarkLogic. It reads JSON data from a mongodump and pushes it into MarkLogic via XCC.

MLJAM and MLSAM

MLJAM enables the evaluation of Java code from the MarkLogic environment. (JAM stands for Java Access Module.) MLJAM gives XQuery programs access to the vast libraries and extensive capabilities of Java, without any glue coding. The Java code does not run within the MarkLogic Server process. It runs in a separate Java servlet engine process, probably on the same machine as MarkLogic but not necessarily. The MLJAM distribution contains two halves: an XQuery library module and a custom Java servlet. The XQuery makes HTTP GET and POST calls to the servlet to drive the interaction between the two languages.

MLSAM (a SQL Access Module, formerly known as MLSQL) allows easy access to relational database systems from within the MarkLogic environment. MLSAM lets you execute arbitrary SQL commands against any relational database and captures the results as XML for processing within the MarkLogic environment. It enables pure XQuery applications to leverage a relational database without having to resort to Java or C# glue code to manage the interaction. MLSAM, like MLJAM, uses HTTP calls to send the SQL query to a servlet, and the servlet uses JDBC to pass the query to the relational database.

REST Endpoint Library

The REST Endpoint Library is a set of XQuery modules that makes it easy to deploy custom RESTful web services on MarkLogic. It consists of two parts: (1) an XML vocabulary for describing web service endpoints and (2) a library module that uses the vocabulary to perform URL rewriting and parameter parsing for incoming requests. It handles the boilerplate work so you can focus on the endpoints.

AtomPub Server

The Atom Publishing Protocol is a simple HTTP-based protocol for creating and updating web resources, such as the entries in an Atom feed. (Atom is a more modern version of RSS.) An Atom server is responsible for maintaining the collection of entries, responding to requests to create, retrieve, and update the entries, and manage some ancillary XML representations describing the set of services available to clients. The AtomPub Server project is an open source pure-XQuery implementation of an AtomPub Server. It uses the REST Endpoint Library.

Corb

Corb is an open source Java-driven bulk reprocessing system. Essentially, it lists all the documents in a collection (or all the documents in the database) and uses a pool of worker threads to apply an XQuery script to each document. The Hadoop Connector and ability to bulk process via Hadoop has reduced the importance of Corb.

XQuery Commons

The XQuery Commons is an 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. Some of the best libraries fall into this category. For example, there's a `json` library for custom serializing JSON, and a `properties` library for reading externally declared deployment properties.

But Wait, There's More

There's far more contributed projects than can be mentioned in this paper. You'll find a full list at http://developer.marklogic.com/code. 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 at http://docs.marklogic.com. Good luck!

About the Author

Jason Hunter is Chief Architect with MarkLogic, and one of the company's first employees. He works across sales, consulting, partnerships, and engineering (he led development on MarkMail.org). He's 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 also an Apache Software Foundation Member and former Vice-President, and as Apache's representative to the Java Community Process Executive Committee he established a landmark agreement for open source Java. He's an original contributor to Apache Tomcat, a member of the expert groups responsible for Servlet, JSP, JAXP, and XQJ API development, and was recently appointed a Java Champion. He's also a frequent speaker.