MarkLogic®

インサイド MARKLOGIC サーバー

ジェイソン・ハンター、マイク・ウールドリッジ

MarkLogic 8対応 · 2016年4月

はじめに

本書ではデータモデル、インデックスシステム、更新モデル、および動作上の振る舞いを含め、MarkLogicサーバーの内部構造について説明します。MarkLogicの初心者でその機能を理解したい方や、MarkLogicは使い慣れているが、内部でどのような処理が行われているのかを理解したい方を対象としています。

本書はMarkLogicサーバーの入門書ではありません。使用方法については、製品マニュアルを参照してください。本書ではMarkLogicがどのような原理に基づいているのかを示します。目的はコードの書き方を示すことではなく、作成するコードがどのように処理されるのかについて理解を促し、より適切で堅牢なアプリケーションの作成を支援することです。

第1章では、MarkLogicサーバーの概要を示しています。第2章では、MarkLogicのコアインデックス、トランザクショナルストレージシステム、マルチホストクラスタリング、そして各種コーディングオプションについて説明しています。それほど専門的な情報を求めていない場合は、ここまで読めば十分かもしれません。第3章では高度なインデックスに加え、バイテンポラル、セマンティック、リバランス、階層型ストレージ、Hadoopとの統合、フェイルオーバー、レプリケーションなどについて説明しています。また、MarkLogicに関連するツール、ライブラリ、プラグイン（その多くがオープンソース）の体系についても説明しています。

第3版となる本書には、MarkLogic 7と8で新しく導入された機能に関する説明が追加されています。これにはJSONとJavaScriptのサポート、セマンティック、バイテンポラル、リバランス、フォレスト割り当てポリシー、階層型ストレージ、スーパーデータベース、増分バックアップ、QBFR（Query-Based Flexible Replication）、JavaとNode.jsのクライアントAPI、カスタムトークン化、関連度のスーパーブースト、モニタリング履歴、そして新しいタイムスタンプ分散オプションが含まれます。また、いくつかの従来からの機能（「マイルドNOT」やワイルドカードマッチング）や、検索の関連度についても取り上げています。

コード例

本書は、MarkLogicの機能について概念的に説明するものであり、プログラミングのマニュアルではありません。しかし、特定の概念がコードを通じて最もよく伝わると判断された場合は、XQueryやJavaScriptのコード例を示しています。コード例の完全版はGitHubからダウンロードできます。

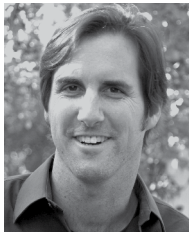
MarkLogicのビルトイン関数を参照するときは、XQuery版の関数を参照します（例：xdmp:document-load()）。ほとんどの場合、同等のJavaScript版があります。JavaScript版の関数を利用するには、「:」を「.」に置き換え、ハイフンをキャメルケース表記に変えます（例：xdmp.documentLoad()）[1]。

1　ヒント：関数の説明を簡単に確認するには、http://docs.marklogic.com/function-nameにアクセスします。名前空間も必要ありません。

APIのドキュメント一式、詳細なマニュアルやチュートリアルは、「MarkLogic Product Documentation」（http://docs.marklogic.com）にあります。このサイトには優れた検索機能が用意されていますが、この機能も当然、MarkLogicを使用して作成されています。

執筆者について

ジェイソン・ハンターは、MarkLogicアジア・パシフィックのCTOであり、会社設立当初からの社員の1人です。セールス、コンサルティング、パートナーシップ、そしてエンジニアリング（MarkMail.orgの開発を統括）の業務に横断的に従事した経験を持ちます。『Javaサーブレットプログラミング』（O'Reilly Media）の著者として、またJava向けに最適化されたXML操作のJDOMオープンソースプロジェクトの作成者としておそらく最も名が知られています。Apache Software Foundationのメンバーかつ元副社長、Apache TomcatとApache Antの初期のコントリビュータでもありました。講演も数多く行っています。



マイク・ウールドリッジは、MarkLogicのシニアソフトウェアエンジニアとして、フロントエンドアプリケーションの開発に従事しています。MarkLogicに付属する、Monitoring Historyなど一部のソフトウェアを開発したほか、MLPHPなどのオープンソースプロジェクトを生み出しています。また、『Teach Yourself Visually Photoshop CC』や『Teach Yourself Visually HTML5』など、グラフィックスソフトウェアやwebデザインに関する数十冊の著作をWileyから出版しています。MarkLogic開発者ブログを頻繁に執筆し、またMarkLogic Worldで講演も行っています。

目次

第1章

MARKLOGICサーバーとは

MarkLogicサーバーは、エンタープライズNoSQLデータベース[1]です。データベースの内部構造、検索タイプのインデックス、アプリケーションサーバーが1つのシステムに統合されています。マルチモデルのデータベースマネージメントシステム（DBMS）であり、ドキュメントモデルがセマンティックトリプルモデルと組み合わさっています。すべてのデータが完全にACID準拠のトランザクショナルレポジトリに格納されます。読み込まれるドキュメントの語と値、そしてドキュメント構造にはインデックスが付けられます。MarkLogic独自のユニバーサルインデックスがあるため、ドキュメント構造（「スキーマ」）に関する事前の知識も、特定のスキーマへの完全な準拠も必要ありません。そして、そのアプリケーションサーバー機能を通じて、プログラム開発と拡張が可能です。

MarkLogicサーバー（以後「MarkLogic」）は、ミッションクリティカルなアプリケーションに必要なエンタープライズ向け機能を維持しながらも、シェアードナッシングアーキテクチャを使用してコモディティハードウェア上でクラスタを構成し、大きな規模と優れたパフォーマンスをサポートすることで市場で差別化を図っています。

それでは、各機能について詳しく見ていきます。

> 「MarkLogicは、ドキュメント指向、マルチモデル、トランザクショナル、検索対応、構造認識型、スキーマ非依存、プログラム可能、ハイパフォーマンスのクラスタ型データベースサーバーです」

1　NoSQLは当初、「No SQL」、つまりSQLに頼らない非リレーショナルデータベースを表していました。現在は、MarkLogicを含む多くの非リレーショナルシステムが、特定目的のためにSQLインターフェイスを用意しているため、NoSQLは「Not Only SQL」（SQLだけではない）に意味を変えています。

ドキュメント指向

MarkLogicは、一般にXMLやJSONで記述されたドキュメントをコアデータモデルの1つとして使用します。非リレーショナルデータモデルを使用し、接続の主要な手段としてSQLに依存しないので、「NoSQLデータベース」と見なされています。金融契約、医療記録、法的書類、プレゼンテーション、ブログ、Twitterのつぶやき、プレスリリース、ユーザーマニュアル、書籍、記事、webページ、メタデータ、スパースデータ、メッセージトラフィック、センサーデータ、積荷目録、旅行プラン、契約書、メールは、自然にドキュメントとしてモデル化されます。これに対して、リレーショナルデータベースはテーブル指向のデータモデルを採用しているので、これらのデータを同じように自然に表すことができません。このため、データを多数のテーブルに分散するか（複雑さが増し、パフォーマンスが低下する）、インデックスのないBLOBまたはCLOBとしてデータを保持する必要があります。

MarkLogicには、XMLとJSONに加えてテキストドキュメント、バイナリドキュメント、セマンティック（RDF）トリプルを格納できます。テキストドキュメントは、それぞれが親のないXMLテキストノードのようにインデックスが付けられます。バイナリドキュメントはデフォルトではインデックスが付けられませんが、メタデータと抽出可能なコンテンツにはインデックスを付けるオプションがあります。RDFトリプルは、バックグラウンドでXML表現に変換されてからXMLドキュメントとして格納されます。

マルチモデル

データモデルによって、情報の格納方法が決まり、組織内に実在する人、物、そのやり取り、そしてそれぞれの関係がドキュメント化されます。MarkLogicのドキュメント指向モデルでは、XML形式とJSON形式を使用して、リレーショナルデータベースの行よりも豊かにレコードを表すことができます。また、MarkLogicでは、世界に関するファクトを表現するのに理想的なセマンティックデータを、主語、述語、目的語の構造、つまりRDFトリプルとして格納できます。トリプルは、データエンティティ間の関係を表します。例えば、特定の人物が、特定の国の特定の都市にある特定の格付けの特定の大学で教育を受けた、といった関係です。トリプルは、トリプルのみのドキュメントとして格納するか、ドキュメントの一部として埋め込むことができます。ドキュメントモデルとセマンティックモデルの両方を提供することで、MarkLogicはマルチモデルデータベースを実現しています。

トランザクショナル

MarkLogicでは、独自のトランザクショナルレポジトリにドキュメントが格納されます。このレポジトリは、リレーショナルデータベースやその他のサードパーティ技術を利用して作成されたものではなく、パフォーマンスを最大化することに重点を置いて作成されています。

このトランザクショナルレポジトリによって、複数のドキュメントの挿入や更新をアトミックな処理(原子性の単位)として実行することが可能で、その直後のクエリでその変更点をレイテンシなしで認識できます。MarkLogicは、ACIDのプロパ

ティを完全にサポートしています。ACIDとは、原子性（Atomicity：一連の変更がまとめて実行されるか、まったく実行されない）、一貫性（Consistency：2つのドキュメントが 同じ識別子を持たないといったようなシステムルールが適用される）、分離性（Isolation：完了していないトランザクションは認識されない）、永続性（Durability：一度コミットされたものは失われない）の4つです。

ACIDトランザクションは、リレーショナルデータベースでは当たり前と見なされていますが、ドキュメント指向のデータベースや検索タイプのクエリでは画期的です。

検索対応

MarkLogicは、そのテキスト検索機能がよく知られています。MarkLogicの設立チームは検索における経験が豊富です。創業者のChristopher Lindblad（クリストファー・リンブラッド）はUltraseek Serverのアーキテクトで、Paul Pedersen（ポール・ペダーセン）はGoogleでエンタープライズ検索担当の副社長を務めていました。MarkLogicは、単語検索とフレーズ検索、ブール型検索、近接と「マイルドNOT」、ワイルドカード、ステミング（語幹処理）、トークン化、複合語の分解、大文字小文字の区別のオプション、句読点による区別のオプション、発音符号による区別のオプション、ドキュメント品質の設定、数々の関連度アルゴリズム、個々のタームの重み付け、トピックのクラスタ化、ファセットナビゲーション、カスタムインデックスフィールド、地理空間情報検索など、多数の検索機能をサポートしています。

構造認識型

MarkLogicでは、テキストと構造の両方にインデックスが付けられるので、この両方に対して効率的にクエリを実行できます。例えば、脅威分析のために傍受されたメッセージトラフィックのクエリと分析を行う場合を考えます。

> 4月11日から13日の間にIP 74.125.19.103から送信されたメッセージのうち、「wedding cake」と「empire state building」の両方の語句を含み（大文字小文字と句読点を区別しない）、2つの語句が15語以内の距離にあり、「presents」（ステミングにより「present」も一致）などの別の重要な語句を含まないものをすべて検索するとします。件名が「Congratulations」となっているメッセージは除外します。また、一致する語句が、メール内の引用ブロック内で見つかったメッセージも除外します。そして、一致するメッセージについて、最も多かった送信者と受信者を返します。

MarkLogicでは、XMLまたはJSONのドキュメントを使用して各メッセージを表現し、また構造認識型インデックス付けを使用して、IPとは何か、日付とは何か、件名とは何か、どのテキストが引用され、どこが引用されていないのかを理解できるので、このようなクエリは実際には簡単に作成でき、ハイパフォーマンスで実行できます。他の例も考えてみます。

画像の確認
「herniated disc」という語句との関連度が高い10件の研究論文から大判サイズの画像をすべて抽出します。語句がタイトルに出現した場合は本文中よりも関連度が5倍高くなり、要旨に出現した場合は関連度が2倍高くなるように、関連度に重みを付けます。

メールから電話番号を検索
メールの大規模コーパスから、特定のユーザーから送信されたものを検索し、時系列の逆順に並べ替え、フッターに電話番号が含まれる最後のメールを特定します。この電話番号を返します。

2つの時間軸に沿ってトラッキングされている情報を抽出（バイテンポラルデータ）
インテリジェンス活動のデータベースより、対象となる人物は1月1日にどこにいたのか、そしてその人物の場所に関する情報は1月15日時点と比べてどのように違うのかを判断します。

スキーマ非依存

MarkLogicは、読み込み時に認識したXMLまたはJSONの構造にインデックスを付けます。検索エンジンに辞書内に存在する語を伝える必要がないのと同じように、MarkLogicにスキーマについて伝える必要はありません。MarkLogicでは、構造とテキストのクエリの難しさが基本的に同等と認識されています。インデックスレベルでは、XPath式の/a/b/cのマッチングは、語句「a b c」のマッチングと同じように実行できます。これがユニバーサルインデックスの要です。

スキーマに関する事前の知識なしでインデックス付けとクエリを実行できることは、次のような非構造化データや半構造化データにとってメリットとなります。

1. スキーマが存在するが明確に定義されていないか、定義はあるが従っていない
2. スキーマが存在し、ある時点で適用されているが、時間とともに変化を続け、常に最新の状態が保たれていない
3. スキーマを完全に特定できない。例えば、対象者について収集するインテリジェンスの情報は何もかも重要である可能性があるなど

ソースデータが高度に構造化され、ソーススキーマが存在する場合にも次のメリットがあります。

1. ほとんどまたは全く予告なく時間とともに変化するスキーマから、アプリケーションを分離できる
2. すべてのソースのすべてのデータに対応できるマスタースキーマを設計することなく、さまざまなソースのデータを容易に統合できる
3. 既存のデータのスキーマを再設計したり、データベースを再構築したりすることなく、新しいデータを既存のデータベースに追加できる

このような特徴によって、MarkLogicは、サイロ化したデータを実用のためにまとめる複雑または大規模なデータ統合プロジェクトに最適となります。

当然、MarkLogicはスキーマに完全に従ったデータにも適しています。MarkLogicを使用してスキーマを適用することもできます[2]。

プログラム可能

MarkLogicサーバーを末端で操作し、プログラムするには、W3C標準プログラミング言語であるXQuery、XSLT、JavaScript、SPARQLの4つのいずれかを使用できます。XQueryは、XMLのクエリ、取得、操作を実行できる、XML指向の関数型言語です。XSLTは、読み込み時と出力時にコンテンツを容易に変換できるスタイルシート言語です。JavaScriptは、JSONと相性のいい動的なオブジェクトベースの言語です。SPARQLは、セマンティックデータを取得できるSQLに似たクエリ言語です。

MarkLogicでは、複数の言語を組み合わせて使用できます。XSLTからXQueryへ、またその逆方向のインプロセス呼び出しが可能です。JavaScriptモジュールではXQueryのライブラリをインポートし、関数や変数をJavaScriptと同じように利用できます。XQueryとJavaScriptのビルトイン関数では、SPARQLのクエリを実行できます。また、MarkLogicでは、REST APIと、ODBCを介したSQLインターフェイスが公開されています。

MarkLogicはホスト1台につきシングルプロセスとして動作し、外部通信用に各種ソケットポートを開きます。アプリケーション用に新しいソケットポートを構成するときは、以下からプロトコルを選択できます。

HTTPとHTTPSの各webプロトコル

MarkLogicはHTTPとHTTPSにネイティブで対応しています。外部からのweb呼び出しは、XQuery、XSLT、またはJavaScriptのプログラムを、他のサーバーがPHPやJSP、ASP.NETのスクリプトを起動するのと同じように実行できます。これらのスクリプトは外部からのデータを受け付け、更新を実行し、出力を生成できます。これらのスクリプトを使用して、完全なwebアプリケーションやRESTful webサービスエンドポイントを作成できます。フロントエンドレイヤーは必要ありません。

XDBCワイヤプロトコル

XDBCは、他の言語コンテキストからMarkLogicへのプログラムによるアクセスを可能にします。これは、リレーショナルデータベースにおけるJDBCとODBCの役割に似ています。MarkLogicでは、XCCというJavaと.NETのクライアントライブラリが正式にサポートされています。これらは、他の言語に関しては、コミュニティによって開発されているオープンソースのライブラリが存在します。XDBCとXCCのクライアントライブラリによって、MarkLogicを既存のアプリケーションスタックに容易に統合できます。

2　構造化情報が実際には半構造化情報である詳細な理由については、http://www.kellblog.com/2010/05/11/the-information-continuum-and-the-three-types-of-subtly-semi-structured-information/を参照してください。

REST API

MarkLogicでは、一連のコアサービスがHTTPベースのREST APIとして公開されています。内部的には、RESTのサービスはXQueryで記述され、HTTPまたはHTTPSのポートに配置されますが、標準機能として提供されるので、REST APIのユーザーがXQueryを意識する必要はありません。ドキュメントの挿入、取得、削除、ページング、スニペット、強調表示を使用したクエリ実行、ファセットの計算、SPARQLを使用したセマンティッククエリ、そしてサーバー管理のサービスが用意されています。

MarkLogicでは、JavaクライアントAPIとNode.jsクライアントAPIもサポートされています。これらのライブラリはREST APIを囲むラッパーであり、JavaとNode.jsの開発者は使い慣れた言語でアプリケーションを作成できます。

SQL/ODBCアクセス

MarkLogicには、ビジネスインテリジェンスツールとの統合のために、読み取り専用のSQLインターフェイスが用意されています。各ドキュメントは1行または複数の行となり、内部の値が列となります。

WebDAVファイルプロトコル

WebDAVは、MarkLogicのレポジトリをWebDAVのクライアントにファイルシステムのように見せるプロトコルです。WebDAVのクライアントは多数あり、ほとんどのオペレーティングシステムにはクライアントが組み込まれています。WebDAVのマウントポイントを使用して、ネットワークファイルシステムと同じように、MarkLogic内外にファイルをドラッグアンドドロップできます。これは小規模なプロジェクトに便利です。大規模なプロジェクトでは通常、読み込みパイプラインを作成し、XDBCまたはRESTを介してデータを送信します。

ハイパフォーマンス

スピードと拡張性は、MarkLogicの信念です。これらは、後から追加できる機能ではなく、標準設計の一部である必要があります。実際、この信念は高度に最適化されたネイティブのC++コードから後述するアルゴリズムに至るまで貫かれています。MarkLogicのお客さまにとって、数ペタバイトのデータや数十億のドキュメントを対象とする高度なクエリを作成して1秒以内に応答を受け取ることは、無謀なことではありません。

クラスタ化

サーバー1台の性能を超えるスピードと拡張性を達成できるように、MarkLogicはLANに接続された複数のコモディティハードウェアを束ねてクラスタとして利用できます。コモディティサーバーには、（開発作業の大半に使われている）ラップトップやシンプルな仮想インスタンスから、12コアのCPU2基、512GBのRAM、大規模なローカルディスクアレイまたはSANへのアクセスを装備した(2016年時点での)ハイエンド機まで利用できます。このようなハイエンド機には一台あたりテラバイト単位のデータを格納できます。

「私たちがMarkLogicを新しい追跡システムの基盤として選んだのは、事前検証で、ピーク負荷時でも応答時間が非常に短かったからです。この技術が拡張性に優れ、オンラインショッピングの需要拡大に対応するためにシステムを拡大しても応答時間に過度な影響を及ぼさないことはすぐにわかりました」

DHLパーセル・ベネルクス、ITディレクター、Ad Hermans

クラスタ内のすべてのホストで同じMarkLogicプロセスが実行されますが、2種類のロールがあります。一部のホスト（データマネージャ、つまりDノード）はデータのサブセットを管理します。その他のホスト（エバリュエーター、つまりEノード）は外部からのユーザークエリを処理し、内部で連携してDノードのデータにアクセスします。ロードバランサーが複数のEノードにクエリを分散します。保持するデータが増えれば、Dノードを増やし、ユーザーアクセスによる負荷が増加すればEノードを増やします。一部のクラスタアーキテクチャ構成では、同じホストがDノードとEノードの両方として機能します。シングルホスト環境では、必ずそうなります。

クラスタ化によって高可用性が実現します。Eノードで障害が発生した場合、ホスト固有の状態は失われません。実行中のリクエストは失われますが、これは再試行が可能です。ロードバランサーは、残りのEノードにトラフィックを転送できます。Dノードで障害が発生した場合は、別のDノードがそのデータのサブセットをオンラインにする必要があります。これには、クラスタ化したファイルシステムを使用する（別のDノードから問題のDノードのストレージに直接アクセスし、ジャーナルを再生できるようにする）か、クラスタ内データレプリケーションを使用する（複数のDノードディスクに更新を複製する、つまりライブバックアップを行う）方法があります。

データベースサーバー

MarkLogicは根本的にはデータベースサーバーですが、通常のDBMSにはない機能が多数用意されています。あらゆるソースの構造化、非構造化、半構造化の情報を1か所に格納できる柔軟性を備えています。データベースタイプのクエリ、検索タイプのクエリ、そしてセマンティックのクエリ（あるいはそのすべての組み合わせ）を全データを対象に実行することも、非常に分析的なクエリを実行することもできます。また、水平拡張（スケールアウト）が可能です。ゼロから構築されているこのプラットフォームによって、サイロ化した混合データの統合を大幅に迅速化、簡素化でき、またリアルタイムの情報アプリケーションやデータサービスを作成、導入できます。

第2章

基本トピック

高速検索のためのテキストと構造のインデックス

MarkLogicとは何かを説明したところで、ここからはその仕組みを、独自のインデックス付けモデルから説明します。MarkLogicは、データベースドキュメント内の語、語句、構造の要素、メタデータにインデックスを付けて、ドキュメントを効率的に検索できるようにします。このインデックス情報を総称してユニバーサルインデックスと呼んでいます。MarkLogicは、他のタイプのインデックス、例えば、レンジインデックス、リバースインデックス、トリプルインデックスも使用します。これらについては後述します。

語のインデックス

まずは思考実験をしてみましょう。プリントアウトした10種類のドキュメントを渡されたとします。私が1つの語を尋ねたら、あなたはその語がどのドキュメントに含まれているかを探して答えなければなりません。この場合、どのような準備を行いますか。検索エンジンの方式を使うとしたら、すべてのドキュメントに出現する語の一覧を作成し、それぞれの語について、その語が含まれるドキュメントを記します。この方法を転置インデックスといいます。ドキュメントに対する語ではなく、語に対するドキュメント識別子であるため、「転置」と呼ばれています。転置インデックス内の各エントリをタームリストと呼びます。タームは、語などを表す専門用語です。どの語を尋ねられても、適切なタームリストを見つけることで、対応するドキュメントを返すことができます。MarkLogicではこのような方法で、シンプルな語のクエリを解決しています。

次に、2つの異なる語を含むドキュメントを尋ねられたとします。これは難しくありません。同じデータ構造を使用できます。最初の語を含むドキュメントIDと2番目の語を含むドキュメントIDを見つけ、2つの一覧の積集合を求めます（図1を参照）。結果は、両方の語を含むドキュメントの集合です。

否定のクエリも仕組みは同じです。1つの語を含み、別の語を含まないドキュメントを尋ねられたら、インデックスを使用して、最初の語を含むすべてのドキュメントIDと、2番目の語を含むすべてのドキュメントIDを探し、差集合を求めます。

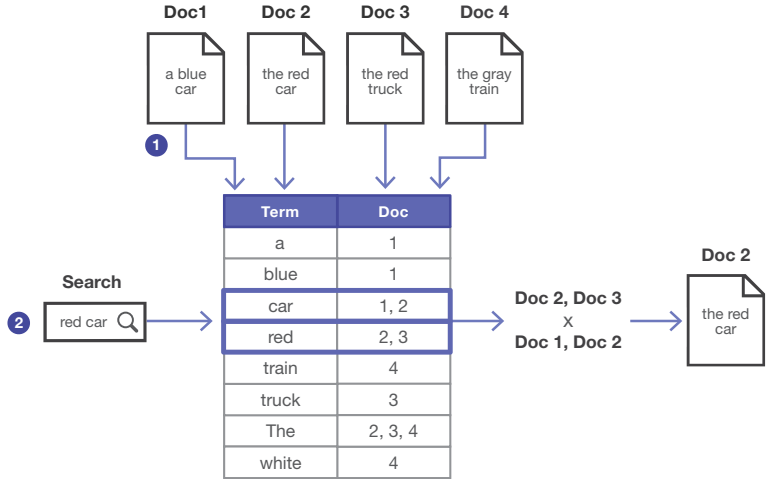


図1: MarkLogicでは、読み込みの際に、ドキュメント内のタームから転置インデックスを作成し(**1**)、各タームを、そのタームが出現するドキュメントに対応付ける。MarkLogicではこのインデックスを使用して検索クエリを解決する(**2**)

語句のインデックス

次に、2つの単語からなる語句を含むドキュメントを尋ねられた場合を考えてみましょう。このクエリは次の3通りの方法で解決できます。

1. 既存のものと同じデータ構造を使用します。両方の語を含むドキュメントを探し、候補ドキュメント内で2語が語句として一緒に出現するかどうかを確認します。

2. タームリストの各エントリに語のポジション情報を加えます。この方法では、ドキュメント内での各語のポジションがわかります。隣り合っている一致タームを探せば、インデックスを使用するだけで正解が見つかります。この場合、ドキュメント内を探す必要はありません。

3. 転置インデックス内に新しいエントリを作成し、ルックアップキーとして1語を使用するのではなく、2つの単語からなる語句を使用します。この場合、2つの単語からなる語句が「ターム」になります。後で2つの単語からなる語句を与えられたとき、その2つの単語からなる語句のタームリストを探せば、その他の処理を行わずに、その語句を含むドキュメントがすぐにわかります。

MarkLogicでは、この中のどの方法も使用できます。その選択は、データベースのインデックス設定に従って実行時に行われます。MarkLogic管理画面には、有効または無効にできるインデックスオプションが表示されます。また、RESTベースの管理APIからインデックスオプションを設定することもできます。これらの設定によって、使用するタームリストと、タームリスト内の各エントリについてポジションデータを保持するかどうかを制御します。

[fast phrase searches]オプションが有効になっている場合、MarkLogicは2つの単語からなるタームを転置インデックスに組み入れます。このインデックスが有効であれば、MarkLogicは前述の3番目の方法を使用できます。この場合、語句のクエリの効率が大きく向上しますが、インデックスのエントリ数が増えるので、ディスク上のインデックスがわずかに拡大し、ドキュメント読み込み時のパフォーマンスがわずかに下がります。

[word positions]のオプションが有効になっている場合、MarkLogicはポジション情報を使用して語句を解決します。これは、前述の2番目の方法です。このインデックス解決は[fast phrase searches]ほど効率的ではありません。なぜなら、ポジションを照合する作業が必要になるからです。ただし、[word positions]は、隣り合っていなくても、近くにある語を検索する近接クエリにも対応します。[word positions]では「マイルドNOT」も使用できます。これは、例えば「Mexico」を検索するときに「New Mexico」は除外できるというものです。

FAST PHRASE SEARCHES

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

WORD POSITIONS

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

図2: [fast phrase searches]が有効になっている場合、MarkLogicは語のペアにもインデックスを付ける。[word positions]の設定では、ドキュメント内でのタームのポジション情報にインデックスが付けられる

[fast phrase searches]と[word positions]のいずれのインデックスも有効になっていない場合、MarkLogicは、シンプルな1語のインデックスを最大限に活用して、結果のフィルタリングを行います。MarkLogicにおけるフィルタリングは、ドキュメントを開き、本当に一致するかどうかを確認する作業です。インデックスを無効にして、フィルタリングに頼った場合、インデックスを小さく保つことはできますが、候補ドキュメントのうち、実際には一致しないドキュメントの数、つまり候補として読み取る必要があるが、フィルタリング時に破棄する数に比例してクエリの処理速度が低下します。

重要な点として、いずれの場合も、クエリ結果に含まれるドキュメントは同じになります。問題はパフォーマンスのトレードオフだけです。関連度順は、インデックスの設定によってわずかに異なる場合があります。なぜなら、インデックスの量が多いと（[fast phrase searches]の場合）、MarkLogicにおける関連度の計算の精度が上がります。この計算は、ドキュメントをディスクから読み取る前にインデックスを使用して行う必要があるからです。関連度の計算については、後述します。

長い語句のインデックス

語句が2語ではなく、3語、4語だった場合はどうなるでしょうか。紙上で手動で行うとしたら、どうするでしょうか。フィルタリングだけに頼るか、位置の計算を使用するか、3語または4語の語句のタームリストエントリを作成することができます。

3語や4語の語句のタームリストを維持しようとすると、逆に効率が悪くなるので、これはMarkLogicでは選択肢になりません。[fast phrase searches]オプションでは2つの単語からなる語句だけがトラッキングされますが、長い語句を検索するときには2語でも役に立ちます。1語目と2語目、2語目と3語目、3語目と4語目を続けて含むドキュメントの検索に使用できるからです。この3つの制約を満たすドキュメントは、4語すべてを含むがそれぞれの位置が不明であるドキュメントよりも、有力な候補となります。

[word positions]が有効になっている場合は[fast phrase searches]も使用されるのでしょうか。答えは「はい」です。位置の計算には、対象となるタームの数に比例する時間とメモリが必要であるからです。MarkLogicでは、[fast phrase searches]を使用して、処理が必要なドキュメント数、そして結果的にターム数を絞り込みます。MarkLogicでは、モノリシックなインデックスは不要です。その代わりに、多数の小さなインデックスを必要に応じて有効または無効にすることで、クエリを解決します。通常は、クエリを確認し、どのインデックスが役立つかを判断し、それらのインデックスを合わせて使用して、結果を候補ドキュメントの集まりにまで絞り込みます。その後、必要に応じてドキュメントのフィルタリングを行い、ドキュメントが実際に一致することを確認し、誤判定を排除できます。有効になっているインデックスオプションが多いほど、候補の結果セットが厳密になります。誤判定がまったく発生しないように（あるいは許容できる程度にしか発生しないように）インデックス設定を調整できる場合もあります。この場合、フィルタリングは不要です。

構造のインデックス

これまでの説明は、そのほとんどが標準的な検索エンジンの動作です（ただし、従来型の検索エンジンは、ソースデータを持っていないため、フィルタリングを行うことができず、結果は常にインデックスからフィルタリングなしで返す点を除きます）。ここからは、MarkLogicがシンプルな検索の域を超えてドキュメント構造のインデックス付けを行う方法を見ていきます。

例えば、`<title>`要素が含まれるXMLドキュメントを探すように頼まれたとします。この場合、どうするでしょうか。MarkLogicであれば、語の場合と同じように、要素`<title>`のタームリストを作成します。各要素のタームリストがあるので、どの要素名を求められたとしても、迅速に回答できます（JSONドキュメントの場合も同様です。詳細については「JSONのインデックス」を参照）。

もちろん、特定の要素名だけを含むドキュメントを求めるクエリは多くありません。そこで、質問がもっと複雑だった場合を検討してみましょう。XPath `/book/metadata/title`と一致するXMLドキュメントを探し、それぞれについてタイトル

ノードを返すものとします。つまり、<book>ルート要素と、子<metadata>、さらにその下に子<title>があるドキュメントを探すということです。この場合はどうするでしょうか。前述のシンプルな要素のタームリストで、<book>、<metadata>、<title>の各要素を持つドキュメントを探すことはできますが、この場合、必要な階層関係が考慮されません。これでは、位置に関係なく語を探す語句クエリのようになります。

MarkLogicでは、独自の親子インデックスを使用して、要素の階層構造がトラッキングされます。これは[fast phrase searches]インデックスに似ていますが、タームキーとして近接する語を使用するのではなく、親子の名前を使用します。book/metadataの関係性(<book>が<metadata>の親)を持つドキュメントをトラッキングするタームリストと、metadata/title用のタームリストがあります。特定のルート要素があるドキュメントをトラッキングするタームリストもあります。この3つのリストの積集合を求めることで、候補ドキュメントの絞り込みが可能です。このように、XPath /a/b/cは語句「a b c」と似た方法で解決できます。

親子インデックスによって、パスが事前にわかっていない場合でも、XPathに対する検索が可能です。このインデックスは非常に有用であるため、MarkLogicで常に保持されます。このインデックスを無効にする設定オプションはありません。

ただし、親子インデックスを使用しても、実際には一致しないドキュメントが候補集合に含まれる可能性がわずかに残ります。XMLドキュメント内のどこかに<book>を親とする<metadata>があり、<metadata>を親とする<title>があることがわかっていても、それが同じ<metadata>であるとは限りません。このような場合にフィルタリングが役立ちます。MarkLogicは、プログラマに結果を返す前に、ドキュメント内でそれぞれの結果を確認します。また、ドキュメントを開いた際に、クエリと一致するノードを抽出するのです。

インデックス解決の目標は、候補集合を小さく、正確にすることで、フィルタリングを最小限に抑えることです。

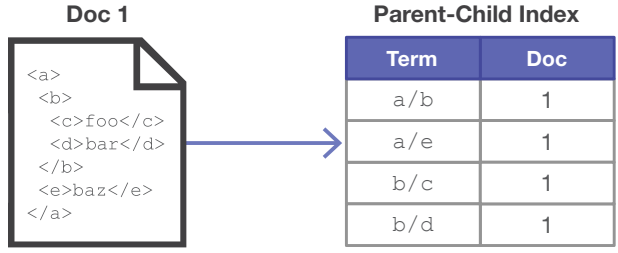


図3: MarkLogicは、XML要素の親子関係にインデックスを付けることで、ドキュメント構造に基づく検索を可能にする(JSONドキュメントのプロパティも同様)

値のインデックス

次に、要素の値を検索する場合について考えます。特定の年に出版された書籍について尋ねられたとします。これはXPathでは`/book[metadata/pubyear = 2016]`と表せます。これを効率的に解決するにはどうすればいいでしょうか。

紙ベースの方法を考えたとき、各XML要素の値（またはJSONプロパティの値）のタームリストが必要です。言い換えると、`<pubyear>`が`2016`であるタームリストに対応するドキュメントを追跡することができます。同じようにイ ンデックス付けの間に見つかった任意の値を持つ要素名のタームリストを追跡できます。したがって、特定の値を持つ要素を検索するクエリがあった場合は、インデッ クスから直接、対応するドキュメントを迅速に見つけることができます。その値のインデックスと、前述の親子構造のインデックスの積集合を求めれば、複数の小さなインデックスを組み合わせて大きなクエリを解決したことになります。この方法はスキーマやクエリが事前にわかっていない場合にも使用できます。このように、検索エンジンの核を使用してデータベースを構築します。

要素と値のインデックスは効率的に格納できるでしょうか。ハッシングを使えば可能です。完全な要素名と値を格納する代わりに、要素名と値を簡潔な整数にハッシングして、それをタームリストのルックアップキーとして使用できます。すると、要素名と値の長さに関わらず、インデックスでは小さなエントリになります。MarkLogicは、背後でハッシュを使用してすべてのタームリストのキー、要素と値、またはその他を格納することで、効率を高めています。要素と値のインデックスは効率的で有用であることが実証されているため、MarkLogic内では常に自動的に有効になっています。

上記の例では、`2016`は整数としてクエリが実行されます。MarkLogicでは、値が実際に整数として格納されているのでしょうか。デフォルトでは違います。ドキュメント内の表記と同じように整数値のテキスト表現として格納されており、上記のクエリは`2016`が引用符で囲まれている場合と同じように実行されます。この程度のあいまいさで通常は十分です。データ型のエンコーディングが重要である場合は、レンジインデックスを使用できます。レンジインデックスについては後述します。

JSONのインデックス

MarkLogicでは、JSONドキュメントのインデックスはどのように処理されるのでしょうか。その方法はXMLの処理と非常によく似ています。JSONプロパティは、XMLの要素と同じように考えることができます。インデックスにおいては、次のJSONは、

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

MarkLogicでは次のXMLと同等です。

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

MarkLogicは、XMLとJSONをいずれも階層構造のノードツリーという同じ方法で格納し、JSONの親子関係はXMLの場合と同じようにインデックス付けできます。このため、[fast element word searches]やその他の要素に関連するインデックスを有効にすると、JSONドキュメントの構造にもインデックスが付けられます。結果的に、XMLの場合と同様に、XPath `/person/last`を使用して、JSONから「Smith」を取得できます。

ただし、違いもあります。XMLの場合は、すべての要素と属性の値がテキストとして格納されるのに対して、JSONプロパティのリーフ値はテキスト、数値、ブール型、NULL値である場合があります。MarkLogicは、JSONの数値、ブール型、NULL値を別個の型固有のインデックスに格納します。このことは、検索において重要な意味を持ちます。例えば、JSONプロパティで値「12」を検索した場合、数値として等価である数値「`12.00`」が一致します。データがXMLの場合はこのような一致は発生しません。なぜなら、テキストとして処理した場合、値「`12`」と「`12.00`」は等しくないからです（XMLの検索で数値が等しいかどうかを確認するには、レンジインデックスを使用できます）。

別の違いとして、JSONのプロパティでは、次のように値が配列になっている場合があります。

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

インデックスにおいては、JSON配列の各要素がプロパティの値と見なされ、別個のテキストノードとして参照されます。その結果、配列要素に対する別個の検索、例えば`json-property-value-query("fruits", "apple")`と`json-property-value-query("fruits", "banana")`はいずれも一致します。また、XPath `/fruits/text()` は、`"apple"`および`"banana"`の双方のテキストノードに一致します。

構造を持つテキストのインデックス

データベース内に「Good Will Hunting」というタイトルのドキュメントがあるとします。特定の値を持つタイトルではなく、「Good Will」など、語または語句を含むタイトルを探す場合について考えます。前述の要素と値のインデックスは、値全体のみが照合されるので、役立ちません。要素に含まれる語や語句を確認するためのインデックスが必要です。このインデックスオプションを[fast element word searches]と呼びます。これを有効にすると、要素名とともに、要素内の個々の語をトラッキングするためのタームリストが追加されます。つまり、XMLドキュメントの場合、語「Good」が含まれる`<title>`があった場合にタームリストエントリが追加され、語「Will」が含まれる`<title>`には別のエントリ、語「Hunting」が含まれる`<title>`にはさらに別のエントリが追加されます。JSONドキュメントの場合は、`title`プロパティに関連付けられている語についてエントリが作成されます。このインデックスを使用して、さらに「Good」と「Will」の語がいずれもドキュメントのタイトルにあることを確認し、インデックス解決の精度を上げることができます。

インデックスでは、各語が同じ語句に含まれるかどうかは認識されていません。この詳細レベルでインデックスを解決したい場合は、MarkLogicの［fast element phrase searches］と［element word positions］の各インデックスオプションを使用できます。［fast element phrase searches］オプションを有効にした場合、すべての要素と、要素内の語のペアについてタームリストが保持されます。タイトルの要素またはプロパティのテキスト内に語句「Good Will」が含まれる場合にタームリストが作成されます。［element word positions］では、すべての要素と、そこに含まれる語のタームリストが保持され、すべての語の位置がトラッキングされます。これらのインデックスのいずれかまたは両方を使用して、各語がドキュメントのタイトルとして同じ語句に含まれることを確認できます。これらのインデックスが有用であるかどうかは、「Good Will」（またはその他のクエリ対象の語句）がタイトル以外の場所に出現する頻度によって異なります。

有効になっているインデックスオプションに関わらず、MarkLogicは、インデックスを自動的に最大限に活用してクエリを解決します。興味があれば、`xdmp:plan()`を使用して特定のXPath式または`cts:search()`式に関する制約を確認してみてください。

インデックスサイズ

このように多数のインデックスオプションがありますが、MarkLogicのインデックスはディスク上でどの程度の容量を占めるのでしょうか。デフォルトのインデックスが有効になった最初の状態では、ディスク上のサイズは、多くの場合、ソースのXMLよりも小さくなります。MarkLogicでは、読み込まれるXMLが圧縮されますが、多くの場合、インデックスは圧縮によって節約できる容量よりも小さくなります。有効にするインデックスが増えると、インデックスサイズはXMLのソースの2倍または3倍の大きさになる可能性があります。ワイルドカードをサポートするインデックスでは、インデックスサイズが3倍を超える場合もあります。

MarkLogicでは、ドキュメントの異なる部分ごとに異なるインデックス機能を有効にできる方法として、「フィールド」がサポートされています。例えば、論文のタイトルと要旨にはワイルドカードインデックスが必要で、本文には必要がない場合に有用です。

再インデックス付け

コンテンツの読み込み後にインデックス設定を変更したい場合はどうすればいいでしょうか。このような場合は、必要な変更を行うだけで、MarkLogicによって更新がバックグラウンドで管理されます。これは、すべてのデータをトランザクショナルレポジトリに格納する主な利点です。バックグラウンドで再インデックス付けが実行されるとき、システムによるクエリや更新の処理は停止しません。システムでは、変更の影響を受けるすべてのデータの再読み込みと再インデックス付けがバックグラウンドで実行されるだけです。

新しいインデックスオプションを追加した場合、そのインデックス機能は、再インデックス付けが完全に完了するまでリクエストをサポートできません。データが不整合をもって参照されるのを防ぐためです。インデックスオプションを無効にすると、すぐに使用が中止されます。

再インデックス付けは管理画面から監視できます。また、[Reindexer Throttle]を5（優先度高）から1（優先度低）の間で制御したり、再インデックス付けを一時的に停止したりすることもできます。ヒント：再インデックス付けを停止してインデックスを無効にした場合、そのインデックスはクエリで使用されませんが、インデックスに関連するデータは保持されます。この方法で、特定のインデックスを使用した場合と使用しなかった場合のパフォーマンスへの影響を簡単に確認できます。

関連度

全文クエリを実行するとき、特定の制約と一致するドキュメントを見つけるだけでは不十分です。結果は関連度順で返される必要があります。関連度は、シンプルな数学的な概念です。一致が多いほどドキュメントの関連度が高くなります。一致の数が同じであれば、短いドキュメントのほうが長いドキュメントよりも関連度が高くなります。検索に複数の語が含まれ、一般的な語と珍しい語がある場合、珍しい語の出現のほうが、一般的な語の出現よりも重要となります。関連度の計算は非常に複雑ですが、MarkLogicでは、自分でその計算を行う必要はなく、関連度順を選択すると自動的に行われます。また、クエリを準備し、実行するときには、関連度の計算を調整する多数の機能が用意されています。関連度については、「高度なテキスト処理」のセクションで詳しく説明しています。

クエリのライフサイクル

それでは、手順を追って実際のクエリのライフサイクルを確認することで、インデックスに関する理解を深めていきます。ここでは、以下の基準への関連度が最も高い上位10個のドキュメントを見つけるテキスト指向の検索を行います。

- ドキュメントは記事である
- 2010年に発行されている
- 説明に「pet grooming」の語句が含まれる
- 「cat」と「puppy dog」の各語句が10語以内の距離で出現する
- キーワードセクションに語「fish」が含まれない

デフォルトでは、フィルタリングが有効になっています。各結果について、記事のタイトル、日付、そして計算された関連度スコアを含むシンプルなHTMLの段落を返すこととします。XQueryコードで記述したプログラムは次のとおりです。

```
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では、タームリストを使用して検索が実行されます。どのタームリストが使用されるかは、データベースで有効にしたインデックスによって異なります。わかりやすいように、役立つ可能性があるすべてのインデックスが有効になっているとします。この場合、次のタームリストが検索に使用されます。

A. `<article>`のルート要素
B. 属性`year`が`2010`の要素`<article>`
C. 「`pet grooming`」を含む要素`<description>`
D. 語「`cat`」
E. 語句「`puppy dog`」
F. 「`fish`」を含む要素`<keyword>`

MarkLogicでは、タームリストに対して設定されている演算処理が実行されます。

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

これで、ドキュメントIDの集合が作成されます。この集合以外のドキュメントは一致する可能性がなく、この集合内のすべてのドキュメントが有力な一致候補です。

この検索には位置の制約もあります。MarkLogicでは、位置インデックスを使用してこの制約を候補ドキュメントに適用します。その結果、候補ドキュメント集合が、「cat」と「puppy dog」が10語以内の距離で出現するドキュメントだけに絞り込まれます。

続いて、関連度スコアに基づいてドキュメントIDが並べ替えられます。関連度スコアは、タームリストに保持されているタームの頻度データとタームの頻度統計を使用して算出されます。これで、スコア順の候補ドキュメントIDの集合が作成されます。

この時点で、並べ替えられたドキュメントIDが順番に処理されます。これはフィルタリングを使用した検索であるため、最高スコアのドキュメントが開かれ、内容を見て、一致することが確認されます。一致しないドキュメントはスキップされます。一致する場合は、return節まで進みます[1]。

各ドキュメントがreturn節まで到達したら、結果シーケンスのために、ドキュメントからいくつかのノードが抽出され、新しいHTMLノードがメモリ内に組み立てられます。ヒットが10個に達すると（[1 to 10]述語があるため）、検索式は完了し、処理が停止します。

すべてのインデックスが有効でなかった場合はどうなるでしょうか。MarkLogicでは、有効になっているインデックスが最大限に活用されます。例えば、位置インデックスがなかった場合は、フィルタリング段階で位置の制約が適用されます。[fast element phrase searches]を<description>語句の一致に使用できない場合は、[fast element word searches]と[element word positions]が使用されます。これらもなかった場合、[fast phrase searches]など、その他のインデックスが使用されます。インデックスの具体性が低いほど、検討する候補ドキュメント数が多くなり、フィルタリング段階で破棄する必要があるドキュメント数も多くなります。ただし、フィルタリング後の最終結果は常に同じです。

ドキュメントメタデータのインデックス

ここまでで、MarkLogicが、テキストと構造の両方のインデックスにタームリストをどのように使用するかを理解できたと思います。MarkLogicでのタームリストの活用方法はこれだけではありません。タームリストは、コレクション、ディレクトリ、セキュリティルールなど、その他多くのもののインデックス付けに有用であることがわかっています。「ユニバーサルインデックス」と呼ばれているのはこのためです。

コレクションのインデックス

MarkLogicにおけるコレクションは、ドキュメント自体を変更せずに、ドキュメントが特定のグループ（分類上の指定、出典、草案であるか、出版済みであるか、など）に属していることを示すタグを付ける方法です。各ドキュメントは、任意の数のコレ

1　以前に読み込まれたタームリストとドキュメントは効率のためにキャッシュに格納されます。キャッシュについては後述します。

クションに属しているとしてタグ付けできます。クエリ制約では、検索の範囲を、特定の1つまたは複数のコレクションに限定できます。

コレクションの制約は、内部では積集合を求めるタームリストの1つとして実装されます。コレクションごとに、そのコレクション内のドキュメントをトラッキングするタームリストがあります。クエリの範囲を1つのコレクションに限定すると、そのコレクションのタームリストと、クエリ内の他の制約との積集合が求められます。クエリの範囲を2つのコレクションに限定すると、2つのタームリストが1つに結合されてから、積集合が求められます。

ディレクトリのインデックス

MarkLogicには、データベース「ディレクトリ」の概念が含まれます。ディレクトリはコレクションに似ていますが、階層構造を持ち、重ならず、一意のドキュメント名、専門的にはURI（Uniform Resource Identifier）に基づきます。MarkLogic内のディレクトリは、多くの面でファイルシステムのディレクトリと似た特性を持ちます。それぞれに任意の数のドキュメントやサブディレクトリが含まれます。多くの場合、クエリはディレクトリに制約をかけ、特定のディレクトリまたはそのサブディレクトリにビューを限定します。

MarkLogicでは、ディレクトリはコレクションと似た方法でインデックスが付けられます。ディレクトリごとに、ディレクトリ内のドキュメントの一覧を示すタームリストがあります。同時に、そのディレクトリおよびその下位に保持されているドキュメントの一覧を示すタームリストも持っています。したがって、タームリストの積集合を求めるだけで、ディレクトリパスに基づいてビューを限定できます。

セキュリティのインデックス

MarkLogicのセキュリティモデルも、タームリストの積集合を求める手法を利用しています。MarkLogicでは、ロールベースのセキュリティモデルが採用されています。各ユーザーに任意の数のロールを割り当て、これらのロールにパーミッションや権限を関連付けます。パーミッションは、ユーザーが読み取り、挿入、および更新できるドキュメントを制御するもので、権限は、ユーザーが実行できる操作（サーバーの再起動など）を制御するものです。ほとんどのセキュリティチェックの実装はシンプルです。ユーザーが必要な認証情報を持っていることを確認してから、求められた操作を許可するだけです。それでは、タームリストを使用した手法はどのような役割を果たすのでしょうか。

MarkLogicでは、タームリストを使用して、ドキュメントの読み取りが管理されます。読み取りはクエリをサポートし、クエリは拡張時においても動作する必要があるからです。ドキュメントの読み取りパーミッションを1つずつ確認するのは現実的ではありません。各ロールごとに、そのロールが読み取りを許可されているドキュメントを管理するタームリストがあります。MarkLogicは、各クエリを実行する際に、呼び出し元ユーザ ーのロールに課せられた参照可能な範囲に関する暗黙的な制約と、

そのユーザーに明示的に化せられた制約とを組み合わせます。ユーザーアカウントに3つのロールがある場合、各ロールのタームリストが収集され、それらを結合してそのユーザーのドキュメント領域が作成されます。このリストと、ユーザーが実行する任意のクエリとの積集合が求められ、ユーザーの参照可能な範囲のドキュメントだけが結果に表示されます。暗黙的かつ非常に効率的に、すべてのユーザーの可視領域がセキュリティ設定に基づいて形作られ、そのセキュリティ設定にはタームリストが使用されます。

MarkLogicでは、「コンパートメントセキュリティ」も利用できます。このシステムでは、ユーザーがドキュメントを表示できるロールを1つ持っているだけでは不十分です。ユーザーには、ドキュメントの表示に必要なすべてのロールが必要です。複数の最高機密プロジェクトに関わるドキュメントがあるとします。すべての最高機密プロジェクトへの可視性がないと、このドキュメントは開くことができません。内部的にはセキュリティ機能により、ロールの可視性のタームリストの和集合（OR）ではなく積集合（AND）が求められます[2]。

プロパティのインデックス

MarkLogic内の各ドキュメントには、オプションでXMLベースのプロパティシートが用意されています。プロパティは、XML記述用の場所がないJSON、バイナリ、テキストの各ドキュメントに関するメタデータを保持できる便利な場所です。また、スキーマを変更できないXMLドキュメントにXMLメタデータを追加する場合にも便利です。設定によっては、各ドキュメントの最終変更日の自動トラッキングにプロパティを使用することもできます。

プロパティは通常のXMLドキュメントとして表され、対象ドキュメントとURI（ドキュメント名）が同じですが、特定の呼び出しを介してのみ使用できます。プロパティには通常のXMLドキュメントと同じようにインデックスが付けられており、通常のXMLドキュメントと同じようにクエリを実行できます。クエリが、メインドキュメントとプロパティシートの両方に対して制約を宣言している場合（例えば、クエリと一致し、かつ1時間以内に更新されたドキュメントを検索する場合）、MarkLogicはインデックスを使用して、プロパティの一致とメインドキュメントの一致を別個に探し、最終的な一致結果を決定するために（共有するURIに基づいて）ハッシュ結合を実行します。プロパティ値がメインドキュメント内に保持されている場合と比べると効率は下がりますが、それほど差はありません。

断片化

ここまで、コンテンツの各単位を単純化して説明するためにドキュメントという言葉を使用してきましたが、実際には、MarkLogicはフラグメントと呼ばれるものにインデックスを付け、取得および格納しています。デフォルトのフラグメントサイズはドキュメントであり、ほとんどの場合はこのままで問題ありません。しかし、XMLドキュ

2　MarkLogicのセキュリティは、ITセキュリティを対象としたNIAP（全米情報保証パートナーシップ）CCEVS（共通基準評価/検証スキーム）のプロビジョンに従って評価および検証されています。MarkLogicは、検証済みの唯一のNoSQLデータベースです。

メントでは、管理画面またはAPIを使用し、fragment root または fragment parent の設定を通じて、ドキュメントをサブドキュメントフラグメントに分割することもできます。これは、インデックス、取得、格納、関連度スコアの単位がドキュメントよりも小さい必要がある大きなドキュメントを処理する場合に便利です。QName（XML要素名を表す専門用語）をフラグメントルートとして指定すると、システムによってドキュメントがそのブレークポイントで自動的に分割されます。あるいは、QNameをfragment parentに、その子要素をfragment rootにできます。JSONドキュメントはサブドキュメントフラグメントに分割できません。

フラグメントとドキュメント

フラグメントの使い方を、書籍にたとえてみましょう。1冊の書籍をインデックス、取得、更新の単位にすることは可能ですが、それではおそらく大きすぎます。検索や表示を章ごとにできるようにするには、`<chapter>`をフラグメントルートにしたほうが適切です。この変更によって、各書籍ドキュメントは一連のフラグメントになります。書籍のメタデータを含む`<book>`ルート要素が1つのフラグメントになり、各`<chapter>`がそれぞれ異なる一連のフラグメントになります。書籍は、引き続き1つのURIを持つドキュメントとして存在し、1つの項目として格納および取得できますが、内部では複数の部分に分かれています。

フラグメントはすべて、自己完結した1つの単位として機能します。フラグメントがインデックスの単位となります。タームリストはドキュメントIDを実際に参照するわけではなく、フラグメントIDを参照します。フィルタリングと取得のプロセスでは、実際にはドキュメントではなく、フラグメントが読み込まれます。

章単位で書籍をフラグメントに分割するのと、ドキュメントを読み込むときに各章要素をそれぞれ別個のドキュメントに分割するのとでは、実際にはほとんど違いはありません。このため、断片化を避け、各ドキュメントをそれぞれ1つのフラグメントとして維持するのが一般的です。そのほうがモデルとして理解しやすくなっています。

MarkLogicの資料（本書を含む）に「フラグメント」と書かれている場合、断片化が有効ではないデータベースについては、「ドキュメント」に置き換えれば正しい記述になります。

断片化されたドキュメントと、個別のドキュメントに分割されたドキュメントとでは、1つだけ重要な違いがあります。同じドキュメントにある2つのフラグメントからデータを引き出すクエリは、2つのドキュメントからデータを引き出すクエリよりもわずかに効率よく実行されます。詳細については、クエリの`cts:document-fragment-query()`要素の説明を参照してください。しかし、このメリットを考慮しても、断片化は、本当に必要であると確信がない限り、有効にすべきものではありません。

推定とカウント

xdmp:estimate()とfn:count()の各関数の違いがわかれば、MarkLogicのインデックスと断片化のシステムをきちんと理解していることがわかります。そこで、これらについてこれから見ていきます。いずれも式を受け取り、その式と一致する項目の数を返します。

xdmp:estimate()の呼び出しでは、インデックスだけを使用して項目数が推定されます。このため、処理が短時間で完了します。この関数は、指定された式をインデックスを使用して解決し、タームリストの制約をすべて満たすとインデックスが見なしたフラグメント数を返します。

fn:count()は、実際のドキュメントフラグメント数に基づいて項目数を返します。この場合、インデックスと、フラグメントのフィルタリングを使用して、式と本当に一致するものと、ドキュメントあたりの一致回数が確認されます。フィルタリングには時間がかかるため（ほとんどディスクI/Oに起因）、常に高速であるとは限りませんが、常に正確です。

興味深いのは、xdmp:estimate()の呼び出しが、fn:count()の結果と比べて大きい値、小さい値、または同じ値を返す可能性がある点です。推定の結果は、フィルタリングで除外されるフラグメントがインデックスシステムから返される場合に大きくなります。例えば、大文字小文字を区別するインデックスを使用せずに大文字小文字を区別する検索を行った場合、大文字小文字が違う候補が結果として返される可能性があります。一方、同じフラグメント内に複数の一致があると、結果が小さくなる場合があります。例えば、xdmp:estimate(//para)の呼び出しは、<para>要素の合計数ではなく、<para>要素が1つ以上含まれるフラグメント数を返すように定義されています。これは、インデックスでは、各フラグメント内の<para>要素の数がトラッキングされないからです。インデックスだけではそのような情報はわかりません。fn:count()の呼び出し時には、各ドキュメントの内容を確認して正確な数が返されます。

大規模な環境においては、一般にxdmp:estimate()が唯一適しているツールであり、MarkLogicのエキスパートの共通の目標は、xdmp:estimate()がfn:count()と一致する返答を返すシステムを作ることです。そのためには、インデックスを適切にモデル化し、インデックスを最大限に活用するクエリと、優れたインデックスに適用できるデータモデルを作成する必要があります。これを実現できたら、高速かつ正確に件数を取得できます。また、クエリを実行するときに、ディスクから読み取られたドキュメントが、結局フィルタリングで除外されることはありません。

フィルタリングなし

フィルタリングは一定の範囲で手動の制御も可能です。cts:search()関数には多数のオプションフラグを指定できますが、その中の1つが「unfiltered」です。通常のcts:search()は、フィルタリングを適用し、結果を返す前にそれぞれの結果が

本当に一致しているかどうかを確認します。「unfiltered」は、フィルタリング手順を省略するようにMarkLogicに指示するフラグです。フィルタリング手順の省略は一般的であるため、新しいsearch:search()関数は、デフォルトでフィルタリングなしで実行します。

レンジインデックス

ここで、MarkLogicに用意されている他のインデックスオプションを見ていきます。最初はレンジインデックスです。レンジインデックスを使用すると、次の操作が可能です。

1. 高速レンジクエリを実行します。例えば、指定した2つの日付の間に収まるドキュメントを取得する検索を実行できます

2. データ型を認識した検索を実行します。例えば、小数値または日付値を、文字の並びとしての値ではなく、意味を表す値に基づいて比較できます（JSONドキュメントの場合は、数値型とブール型のデータが等しいかどうかをレンジインデックスなしで比較できます。詳細については、前述の「JSONのインデックス」を参照してください）

3. 結果セットのエントリから特定の値を迅速に抽出します。例えば、結果セット内のドキュメントから、メッセージ送信者のリストや、各送信者が出現する頻度を取得できます。これらは一般にファセットと呼ばれ、検索を操作しやすいように、検索結果とともに表示されます。また、抽出された値に対して高速集計を実行し、標準偏差や共分散などを計算できます

4. 最適化されたorder byを行います。例えば、大規模な製品の結果集合を価格で並べ替えることができます

5. 効率的なドキュメント間結合を実行します。例えば、人を表すドキュメントの集合と、これらの人が作った作品を表すドキュメントの集合がある場合、レンジインデックスを使用して、特定のタイプの人が作った特定のタイプの作品を探すクエリを効率的に実行できます

6. 複雑な日時のクエリをバイテンポラルドキュメントに対して実行します。バイテンポラルドキュメントには4つのレンジインデックスが含まれ、これらのインデックスによって、イベントが実際にはいつ発生したか、またイベントがMarkLogicにいつ格納されたかが記録されます。この4つのレンジインデックスに対してクエリを実行し、結果を統合することが、バイテンポラルクエリを解決する鍵となります。詳細については「バイテンポラル」を参照してください

7. 結果セット内のエントリから、同時発生の値を迅速に抽出します。例えば、ドキュメント内に同時に出現する頻度が高い2つのエンティティ値のレポートを迅速に入手できます。2つのエンティティ値が事前にわかっている必要はありません。これは高度なユースケースであるため、本書では説明を省きます

レンジインデックスの役割を本当に理解するために、パーティーに参加している場面を想像してみてください。会場で、誕生日が自分の誕生日から1週間以内の人を探すとします。特定の日付を順番に叫べば、誰に変な目で見られるか確認できるでしょう。これが、あるファクトを含むドキュメントを探す通常の転置インデックスの手法です。ファクトを特定し、どのドキュメントが一致するかを確認します。ただし、この手法は範囲に対しては不向きであり、ある種の「分析」からパーティーの全参加者の誕生日一覧を取得する場合にも適していません。それに、過去の日付を無闇に叫ぶことは避けたいはずです。歩き回って1人1人に誕生日を聞くこともできます。これは、格納されているすべてのドキュメントをディスクから読み取り、内部の値を確認するデータベースの処理と同等です。間違った方法ではありませんが、長い時間がかかります。もっと効率的なのは、パーティーの入口と出口で参加者の誕生日を記録する方法です。こうすれば、誕生日の完全なリストをいつでも確認でき、多くの用途に活用できるでしょう。これが、レンジインデックスの概念です。

レンジインデックスを構成するときは、次の3つの情報を指定します。

1. ドキュメント内でレンジインデックスを適用する部分（要素名、要素属性名、JSONプロパティ、簡素化されたXPath形式のパス、またはフィールド）
2. データ型（int、date、stringなど）
3. 文字列の場合はコレーション（特殊なURIで識別される文字列の比較とソートのアルゴリズム）

各レンジインデックスに対し、MarkLogicでは、2つの処理を容易にするデータ構造が作成されます。任意のドキュメントIDについてドキュメントのレンジインデックス値を取得する処理と、任意のレンジインデックス値について、その値を持つドキュメントIDを取得する処理です。

概念的には、レンジインデックスは、2つのデータ構造によって実装され、ディスクに書き込まれてから、効率的なアクセスのためにメモリにマッピングされると考えることができます。一方のデータ構造は、ドキュメントIDと値をドキュメントID順に並べ替えた構造の配列、そしてもう一方は、値とドキュメントIDを値順に並べ替えた構造の配列と考えることができます。実際にはこれほどシンプルではなく、メモリとディスクに無駄がありませんが（現実には値が1回だけ格納される）、モデルとしてはこのように理解できます。パーティーの例でいえば、誕生日と参加者の対応を誕生日順で示すリストと、参加者と誕生日の対応を参加者順で示すリストになります。

レンジクエリ

高速レンジクエリを実行するために、MarkLogicでは「値からドキュメントID」のルックアップ配列が使用されます。ルックアップ配列は値順に並んでいるため、この配列には、ユーザー 定義による2つのエンドポイント間の値を含む特定のサブ

シーケンスが存在します。範囲内の各値にはドキュメントIDが関連付けられています。このドキュメントIDを迅速に収集し、総合的なタームリストのように使用することで、ユーザーが指定した範囲内に一致する値があるドキュメントに検索結果を限定できます。

例えば、誕生日が1980年1月1日から1980年5月16日の範囲内の参加者を探すには、日付順のレンジインデックスで開始日と終了日のポイントを探します。配列内でこの2つのエンドポイント間にあるすべての日付は参加者の誰かの誕生日であり、各誕生日の横にはその参加者が記されているので、参加者名を取得するのは簡単です。複数の参加者の誕生日が同じであった場合、配列には、値が同じで対応する名前が異なる複数のエントリがあります。MarkLogicでは、参加者の名前ではなく、ドキュメントIDがトラッキングされます。

レンジクエリは、MarkLogicの他の任意のタイプのクエリと組み合わせることができます。例えば、前述の日付範囲に収まるだけでなく、特定のメタデータタグを持つものに結果を限定するとします。MarkLogicは、レンジインデックスを使用して範囲内のドキュメントIDの集合を取得し、タームリストを使用してメタデータタグを持つドキュメントIDの集合を取得し、IDの各集合の積集合を求めることで、両方の制約を満たす結果セットを判断します。MarkLogicでは、すべてのインデックスが相互に構成可能です。

プログラマは、`cts:element-range-query()`などの関数を通じてのみレンジインデックスを使用していると思っているかもしれません。実際には、通常のXPath式やXQuery式の高速化にもレンジインデックスは使用されています。XPath式/`book[metadata/price > 19.99]`は、一定の価格を上回る書籍を探しますが、`<price>`にdecimal（またはdouble、またはfloat）のレンジインデックスが存在する場合はそれを利用します。レンジインデックスが存在しない場合は、インデックスで価格を制約できないため（タームリストは役に立たない）、MarkLogicは`<price>`要素が含まれるすべての書籍を確認します。このため、パフォーマンスに大きな差が出る可能性があります。

図4: 概念的には、レンジインデックスは2つのデータ構造で実装される。一方は値順で並んだ値とドキュメントIDの配列、もう一方はドキュメントID順で並んだドキュメントIDと値の配列である。これらの構造を使用して、値範囲に対応するドキュメント**(1)**と、1つあるいは複数のドキュメントに対応する個々の値**(2)**を探すことができる

データ型を認識する等式クエリ

同じ「値からドキュメントID」のルックアップ配列を、データ型を認識する必要がある等式のクエリにも使用できます。誕生日だけでなく、それぞれの参加者が生まれた正確な時刻をトラッキングするとします。この場合の問題は、末尾のタイムゾーンの表記が原因で、同じタイムスタンプの値を順番に並べる（シリアライズ）方法が多数あることです。`2013-04-03T00:14:25Z`と`2013-04-02T17:14:25-07:00`の各タイムスタンプは、意味的には同じです。これは、数字の1.5と1.50の関係と同じです。すべての値が同じ方法でシリアライズされていれば、タームリストインデックスを使用して文字列表現を照合できますが、シリアライゼーションが異なる場合は、レンジインデックスを使用することが推奨されます。レンジインデックスはデータ型の値に基づいているからです（ただし、一致する範囲を指定するのではなく、単一の値を指定します）。

データ型を認識する等式のクエリを実行するには、`cts:element-range-query()`と「=」演算子を使用するか、XPathとXQueryを使用できます。前述のXPath `/book[metadata/pubyear = 2013]`を考えてみましょう。`2013`は整数値であるため、`<pubyear>`に整数としてキャストできる型のレンジインデックスがある場合は、それを使ってこのクエリを解決します。

JSONドキュメント内の数値とブール型値については、レンジインデックスなしで等式の比較を行うことができます。これらのデータ型は型固有のインデックスが付けられているからです（詳細については「JSONのインデックス」を参照してください）。ただし、不等式の比較にはレンジインデックスが必要です。

値の抽出

結果セット内のドキュメントから特定の値を迅速に抽出するために、MarkLogicは、ドキュメントIDと値を対応付けるデータ構造を使用します。最初に転置インデックス（および場合によってはレンジインデックス）を使用して、クエリと一致するドキュメントIDを抽出します。次に、「ドキュメントIDから値」のルックアップ配列を使用して、各ドキュメントIDに対応する値を確認します。ディスクへのアクセスがないため、迅速な処理が可能です。また、それぞれの値が出現する頻度を数えることもできます[3]。

抽出された値はバケット化も可能です。バケット化では、値ごとに数が返されるのではなく、値の範囲に収まる値の数が返されます。例えば、特定の価格を示すソースデータに対して、価格帯ごとに数を取得できます。バケットはクエリの一部として指定します。すると、MarkLogicでは、レンジインデックスが確認されるときに、各バケット内に出現する値の数がトラッキングされます。

パーティーの例で、シアトルに住む参加者全員の誕生日が知りたい場合は、最初に転置インデックスを使用して、シアトルに住む参加者の（ターム）リストを作ります。次に、名前と誕生日のルックアップ配列を使用して、該当する参加者の誕生日を探します。誕生日ごとにその人数を数えることができます。参加者の誕生日を月ごとにグループ化する場合は、バケット化を使用できます。

「ORDER BY」の最適化

XQueryにおける、最適化されたorder byの計算によって、MarkLogicでは、大規模な結果セットを、レンジインデックスがある要素を基準に迅速に並べ替えることができます。XQueryのシンタックスは、次のように、特定の型である必要があります。

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

最適化されたorder byの計算を行う場合も、「ドキュメントIDから値」のルックアップ配列が使用されます。どのような結果セットでも、MarkLogicは検索からドキュメントIDを取得し、それをレンジインデックスに供給します。その結果、結果を並べ替える値への高速アクセスが可能になります。並べ替え対象の値はレンジインデックスから取得されるため、数百万個の結果項目もほんの一瞬で並べ替えることができます。

3　鋭い読者の方は、ルックアップのためにレンジインデックスに渡されるドキュメントIDがフィルタリングされていないことにお気づきでしょう。これは純粋にインデックスに基づく処理です。レンジインデックスからの値の抽出については、cts:element-values()、cts:element-attribute-values()、cts:frequency()の説明を参照してください。

シアトル在住のパーティー参加者を年齢で並べ替え、最も高齢の、または最も若い10人を特定するとします。この場合、最初に結果をシアトルからの参加者に絞り込み、その誕生日を抽出し、誕生日で並べ替えてから、最後に該当者の集合を昇順または降順で返します。

レンジインデックス処理のパフォーマンスは、その大部分が結果セットのサイズ、つまりルックアップする件数に依存します。データ型もパフォーマンスに少し影響しますが、コア1個あたり、1秒におよそ1000万のルックアップが可能です。整数は浮動小数点数よりも高速で、浮動小数点数は文字列よりも高速です。文字列は、シンプルなUnicodeコードポイントコレーションを使用した場合のほうが、より高度なコレーションを使用した場合よりも高速です。

最適化された`order by`式の詳細と、その適用のルールについては、『Query Performance and Tuning Guide』を参照してください。

レンジインデックスを使用した結合

レンジインデックスは、ドキュメント間結合に便利であることがわかっています。その方法は次のとおりです。最初のクエリと一致するIDの集合を取得したら、その集合を制約として次のクエリに供給します。レンジインデックスを両端で使用できるので、この作業は比較的、効率よく実行できます（ドキュメント間結合はRDFトリプルを使用してさらに効率よく実行できます。詳細については、「セマンティック」のセクションを参照してください）。

この方法を理解するには、コード例が必要です。一連のTwitterのつぶやきがあり、各つぶやきに日付、投稿者ID、テキストなどがあるとします。また、投稿者に関する一連のデータ、例えば投稿者ID、登録日、本名などがあるとします。その中から、Twitterを1年以上使用し、特定の語句をつぶやいた投稿者を探し、その語句を含むつぶやきを返すとします。この場合、投稿者データとつぶやきデータの結合が必要です。

XQueryコードの例：

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

最初のコードブロックで、少なくとも1年間Twitterを利用している人の投稿者IDがすべて見つかります。ここでは`signup-date`レンジインデックスを使用して`cts:element-range-query()`制約を解決し、`author-id`レンジインデックスを`cts:element-values()`の取得に使用しています。これで、`$author-ids`の長いリストが取得できます。

2番目のブロックでは、その`$author-ids`の集合を検索制約として使用し、実際のテキスト制約と組み合わせています。ここで、レンジインデックスの機能がなかった場合、MarkLogicは投稿者IDごとに別個のタームリストを読み込んで、その投稿者に対応するドキュメントを見つける必要があり、投稿者ごとにディスクシークが必要になる可能性があります。レンジインデックスがあれば、インメモリルックアップだけを使用して投稿者IDをドキュメントIDに対応付けることができます。これを一般にショットガン、または（道徳的な言い方をすれば）分散クエリといいます。長いリストでは、個々のタームリストを検索するよりも大幅に効率的です。

パスやフィールドにレンジインデックスを使用してさらに最適化

従来のレンジインデックスでは、レンジインデックスを構築する要素または要素と属性の名前を指定できました。パスのレンジインデックスでは、さらに詳細な設定が可能です。同じ名前のすべての要素または要素と属性を含める代わりに、特定のパスにあるものに限定できます。この方法は、要素の名前が同じであっても、配置によって意味がわずかに異なる場合に特に有用です。例えば、DocBook標準には<title>要素がありますが、それは書籍名、章タイトル、セクションタイトル、ま

たはその他を表している可能性があります。この違いに対応するために、book/title、chapter/title、section/titleにレンジインデックスのパスを定義できます。別の例として、通貨によって価格が異なるため、レンジインデックスをそれぞれ分けたほうがいい場合があります。この場合、product/price[@currency = "USD"]やproduct/price[currency = "SGD"]などの述語を使用して定義できます。パスの定義は非常に柔軟です。相対パスまたは絶対パスを指定でき、ワイルドカードのステップ(*)を含めることができ、また述語(角括弧内)を含めることもできます[4]。

パスレンジインデックスの主な目的は、レンジインデックスに何を含めるかをより具体的に制御できるようにすることです。しかし、XPathのさらなる最適化も可能です。前に式/book[metadata/pubyear > 2010]で、<pubyear>に対するレンジインデックスを使用してクエリを解決できる方法を確認しました。一致するパスレンジインデックスもある場合は、より具体的なこちらのインデックスが代わりに使用されます。/book/metadata/pubyearに整数のパスレンジインデックスがある場合は、このレンジインデックスだけで完全なXPathを解決できます。タームリストは必要ありません。

ドキュメントのどの部分のインデックス付けを行うかを指定する別の方法として、フィールドがあります。フィールドを使用すると、インデックス可能な単一の単位として複数のXML要素やJSONプロパティを追加または除外できます。例えば、<first-name>と<last-name>の各要素をフィールドとして含め、<middle-name>を除外できます。このフィールドにレンジインデックスを追加すると、組み合わせた値に対してレンジ操作を行うことができます。また、複数のドキュメントで、<last-name>と<lastName>の各要素を使用して、別の方法で姓を定義しているような場合でも、これらの要素にレンジインデックス付きフィールドを作成すれば、マークアップが違っていても、ドキュメント全体を対象とした検索結果を姓の順に並べることができます。フィールドの詳細については、「詳細トピック」の章を参照してください。

レキシコン

データベースを構成するとき、「URIレキシコン」と「コレクションレキシコン」を構成するオプションがあります。これらは形を変えたレンジインデックスです。URIレキシコンはシステムに保持されているドキュメントURIをトラッキングすることで、ディスクにアクセスすることなく、クエリと一致するURIを迅速に抽出できるようにします。コレクションレキシコンは、コレクションURIをトラッキングし、同様の効果をもたらします。内部的には、値がURIである点を除き、いずれも他のレンジインデックスとほぼ同じです。レキシコン取得呼び出しは、cts:queryオブジェクト制約を利用できるので、クエリと一致するドキュメントのURIやコレクションを容易に、かつ効率的に見つけることができます。

4 パスインデックスは、直接的な要素インデックスと比べて計算とフィルタリングに必要な処理が多いため、コストが伴います。多数の重複するパスが定義されている場合、インデックスサイズとクエリ時間が大きくなります。このため、直接的な要素インデックスで十分である場合は、それを使用してください。

こうした場合に、なぜレンジインデックスが必要なのでしょうか。レンジインデックスは、インデックスの中の話ではないのでしょうか。デフォルトではそうではありません。タームリストのルックアップキーはハッシュであるため、ユニバーサルインデックスで、コレクション内のすべてのドキュメントを検索することは可能ですが（コレクション名をハッシュしてタームリストを検索し、ドキュメントIDを確認する）、すべてのコレクションを検索するのは効率的ではありません（ハッシュは一方向）。レキシコンでは、通常のレンジインデックスがドキュメント内から値を抽出するのと同じ方法でドキュメントとコレクションのURIを計算できます。

また、「語レキシコン」を設定して、データベース内の（または特定の要素に限定して）個々の語をトラッキングすることも可能です。メモリの効率のために、一般的な語に多数の連続エントリが必要とならないように、語レキシコンはドキュメントIDが関連付けられていない語のフラットリストとして保持されます。その場合も、`cts:query`オブジェクト制約を語レキシコンの取得呼び出しに渡して、ドキュメント内でクエリと一致する語だけを取得することはできます。この制約を適用するために、MarkLogicでは、簡単なタームリストの検索を通じて、レキシコンから各語を引き出し、タームリスト内のドキュメントIDが、クエリと一致するドキュメントIDに含まれているかどうかが確認されます。含まれている場合はその語が返され、含まれていない場合は次の語に進みます。語レキシコンは、ワイルドカードクエリに役立ちます。ワイルドカードクエリについては後述します。

データ管理

次のセクションでは、MarkLogicで、ディスク上のデータが管理される方法と、同時の読み取りと書き込みが処理される方法を見ていきます。

データベース、フォレスト、スタンド

MarkLogicでは、データが階層構造に編成され、データベースが最上位の論理単位になっています。システムでは複数のデータベースを利用でき、通常はそうしますが、各クエリまたは更新リクエストは一般に特定のデータベースに対して実行されます（例外として、スーパーデータベースでは、複数のデータベースを1つの単位としてクエリを実行できます。詳細については後述します）。MarkLogicをインストールすると、デフォルトでいくつかの補助データベースが作成されます。`App-Services`、`Documents`、`Extensions`、`Fab`、`Last-Login`、`Meters`、`Modules`、`Schemas`、`Security`、そして`Triggers`です。

データベースは1つあるいは複数のフォレスト[5]から構成されます。フォレストは、ドキュメントの集合であり、ディスク上で物理的なディレクトリとして実装されます。フォレストには、ドキュメントに加えて、それらのドキュメントのインデックスと、トランザクション処理をトラッキングするジャーナルが格納されます。1台のマシンで複数のフォレストを管理することもあれば、クラスタ内にある（エバリュエーターで

5　なぜ「フォレスト」と呼ばれるのでしょうか。それは、含まれるドキュメントがツリー構造で格納されているからです。

あるEノードとし て機能する）場合はまったく管理しないこともあります。フォレストはクエリを並列実行できるので、マルチコアサーバーに複数のフォレストを配置すると同時操作に役立ちます。フォレストの最適な数はさまざまな要因に依存しますが、そのすべてがパフォーマンスに関連するわけではありません。最新のサーバーハードウェアでは、サーバー1台につきプライマリフォレスト6つと、フェイルオーバー用のレプリカフォレスト6つ（詳細は後述）が出発点として適切です。各フォレストのサイズは、ハイパフォーマンス環境であるか、大容量環境であるかによって異なります。ハイパフォーマンス環境の場合は（例：ファセット付きユーザー向け検索サイト）、各フォレストに800万のドキュメントと100GBのディスク容量を含めることがあります。大容量環境の場合は（例：組織内の分析アプリケーション）、各フォレストに1億5000万のドキュメント と500GBのディスク容量を含めることもあります。クラスタ環境では、それぞれが各々のフォレスト群を管理する複数のサーバーをすべて単一のデータベースに統合できます[6]。フォレストの設定の詳細については、『Performance: Understanding System Resources』を参照してください。

各フォレストはスタンドから構成されます。スタンドには、フォレストデータのサブセットが含まれます。各フォレストにはインメモリスタンドが1つと、いくつかのオンディスクスタンドがあります。オンディスクスタンドは、フォレストディレクトリ下で物理サブディレクトリとして存在します。複数のスタンドがあると、MarkLogicでデータを効率的に読み込みできます。1つのスタンドには複数の圧縮バイナリファイルが含まれ、`TreeData`、`IndexData`、`Frequencies`、`Qualities`などの名前になっています。圧縮された実際のXMLデータは`TreeData`に、インデックスは`IndexData`に含まれます。

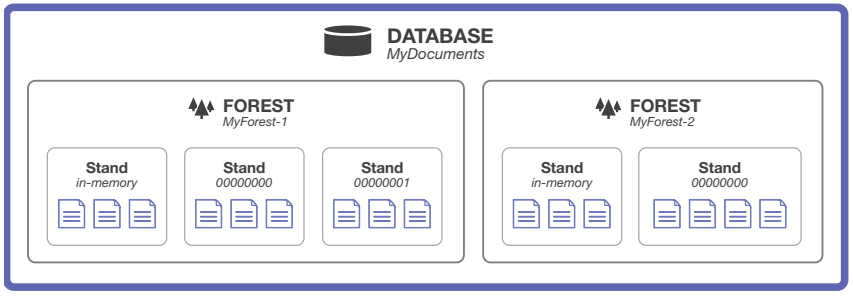


図5: MarkLogicにおけるデータベース、フォレスト、スタンドの階層関係

6　フォレストは、リレーショナルデータベースのパーティションのようなものでしょうか。どちらともいえません。データのサブセットが含まれるという点では、そうといえます。通常はデータの特定の側面に基づいてフォレストにデータを割り当てない点では、そうではないといえます。フォレストは、特定のクエリを事前に最適化するためのものではありません。多数のフォレストにまたがって、多数のコア上で、また場合によっては多数のディスク上で、リクエストの並列実行を可能にします。

データの読み込み

MarkLogicでデータがどのように読み込まれるのかを見ていきます。最初にフォレストが1つの空のデータベースを考えます。ドキュメントがないので、インメモリスタンドだけがあり、オンディスクスタンドはありません。ある時点で、XCC、XQuery、JavaScript、REST、またはWebDAVの呼び出しを通じて新しいドキュメントがMarkLogicに読み込まれます。どの言語やプロトコルを使用しても、結果は同じです。MarkLogicでは、このドキュメントがインメモリスタンドに格納され、システム障害が発生した場合に永続性とトランザクションの整合性を維持するために処理がフォレストのオンディスクジャーナルに書き込まれます。

その後、新しいドキュメントが読み込まれると、それらのドキュメントもインメモリスタンドに格納されます。この時点でのクエリリクエストでは、ディスク上の全データ（厳密にはまだ何もない）と、インメモリスタンドの内容すべて（少数のドキュメント）が認識されます。クエリリクエストでは、データの場所はわかりませんが、この時点までに読み込まれたデータ全体を確認できます。

十分なドキュメントが読み込まれたら、インメモリスタンドがいっぱいになり、チェッ クポイントが発行されます。つまり、オンディスクスタンドとして書き出されます。新しいスタンドには、フォレストディレクトリの下にそれぞれ専用のサブディレクトリが作成されます。名前は単調増加する16進数です。最初のスタンド名は`00000000`となります。このオンディスクスタンドには、それまでに読み込まれたドキュメントのデータとインデックスがすべて含まれます。これはインメモリスタンドからディスクへ、効率の最大化のために順次書き込み(シーケンシャルライト)されます。書き込みが完了すると、インメモリスタンドに割り当てられていたメモリが解放され、ジャーナル内のデータがリリースされます。

その後、読み込まれるドキュメントは、新しいインメモリスタンドに格納されます。このインメモリスタンドもいっぱいになると、新しいオンディスクスタンドに書き出されます。このスタンドの名前は`00000001`となり、最初のスタンドと同じ程度のサイズになります。負荷が高いときはインメモリスタンドが2つになり、最初のスタンドのディスクへの書き出し中に、追加ドキュメント用の新しいスタンドが作成されます。投入されたクエリまたは更新リクエストは、常にすべてのスタンドにまたがってすべてのデータを認識できます。

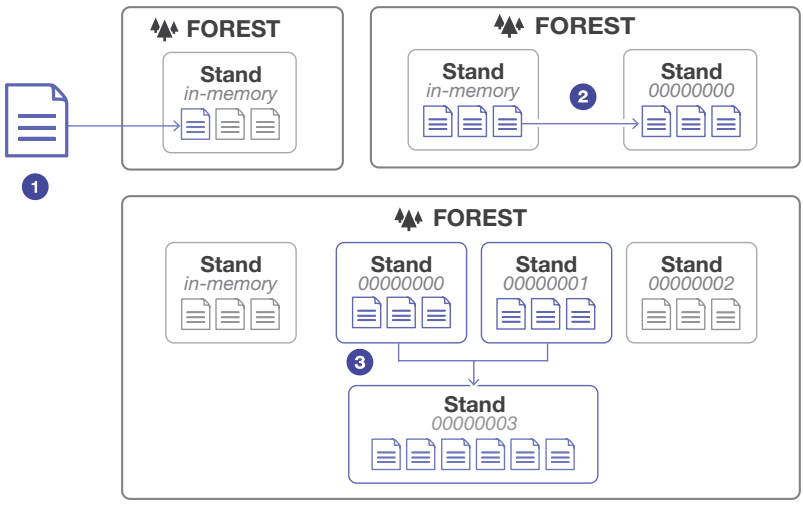


図6: 新たに読み込まれたドキュメントは、インメモリスタンドに格納される**(1)**。インメモリスタンドが一定のサイズに達すると、オンディスクスタンドにドキュメントが書き込まれる **(2)**。オンディスクスタンドは、MarkLogicによってマージ段階で結合されるまで蓄積する**(3)**

スタンドのマージ

インメモリスタンドがいっぱいになると、オンディスクスタンドに書き込まれるというメカニズムが続きます。オンディスクスタンドの合計数が増えると、効率の問題が生じる恐れがあります。MarkLogicで単一のタームリストを読み込むには、各スタンドからタームリストデータを読み込んで結果を統合する必要があります。この統合がパフォーマンスの問題にならない程度にスタンド数を維持するために、MarkLogicでは、マージがバックグラウンドで実行されます。マージ時には、ディスク上のいくつかのスタンドから新しいスタンドが作成され、インデックスやデータが統合および最適化され、以前に削除されたフラグメントが除去（詳細は後述）されます。マージが完了し、新しいオンディスクスタンドへの書き込みが完了し、古いオンディスクスタンドを使用して実行中のリクエストがすべて完了したら、古いオンディスクスタンドが削除されます。

MarkLogicでは、アルゴリズムを使用して、マージするタイミングが決定されます。この基準となるのは、各スタンドのサイズ、および、各スタンド内でまだアクティブであるデータと、すでに削除されたデータの量です。時間が経過すると、複数の小さなスタンドがマージされて大きなスタンドになります。「最大マージサイズ」は設定可能で、デフォルトでは32GBです。マージ時に、この制限を超えるスタンドが作成されることはないので、マージ処理に大量の空きディスク容量が必要になることはありません[7]。一般に、フォレストには、時間とともにマージ最大サイズに近い複

7　ここで説明している「マージ最大サイズ」はMarkLogic 7で新たに追加された設定です。旧バージョンでは、マージ時に、すべてのスタンドの合計サイズに等しい単一のスタンドが新たに作成される可能性があり、その場合はマージを行うために大量の空きディスク容量が必要でした。新しい「マージ最大サイズ」は、単一のスタンドのサイズを制限し、結果的に必要な空きディスク容量を制限します。大規模なフォレストに必要な追加容量はわずか50%です。オンディスクデータ1TBあたり空きディスク容量0.5TBです。

数のスタンドと、いくつかの小さなスタンドが蓄積されます。マージはCPUとディスクの消費量が多いので、システム管理を通じていつマージを行うかを制御できるようになっています。

各フォレストには、それぞれ専用のインメモリスタンドと、複数のオンディスクスタンドがあります。コンテンツの読み込みとインデックス付けは多くの場合、並列処理が可能であるため、フォレスト間やクラスタ内のマシン間で読み込み処理を分散させることが、読み込み作業の拡張に役立ちます。

ドキュメントの圧縮

`TreeData`ファイルには、XMLドキュメントのデータが非常に効率よく格納されます。ドキュメントのツリー構造は、コンパクトなバイナリエンコーディングを使用して保存されます。テキストノードは、辞書ベースの圧縮スキーマを使用して保存されます。このスキーマでは、テキストが（語、スペース、句読点に）トークン化され、各ドキュメントで独自のトークン辞書が作成されて、数値のトークンIDがトークン値に対応付けられます。文字列は文字のシーケンスとしてではなく、数値のトークンIDのシーケンスとして格納されます。元の文字列は、辞書をルックアップテーブルとして使用して再構成できます。辞書内のトークンは頻度の順で格納され、最も頻繁に出現するトークンのトークンIDが最小になります。ツリー構造の表現とテキスト文字列におけるすべての数値が可変長のエンコーディングを使用してエンコードされ、小さい数値のほうが大きい数値よりも必要なビット数が少ないため、これは重要です。トークンの数値表現は単項のニブルカウントで、その後に数ニブルのデータが続きます（1ニブルは1バイトの半分）。最も頻度の高いトークン（通常はスペース）がトークンID 0となり、文字列内で表すのに1ビットしか使用しません。トークン1〜16には6ビットが必要です（2ビットのカウント（10）と1個の4ビットニブル）。トークン17〜272には11ビットが必要です（3ビットのカウント（110）と2個の4ビットニブル）。以降も同様です。コンパクトなトークンIDは、それぞれ任意の長さのトークンを表します。その結果、XMLの非常にコンパクトなシリアライゼーションになり、通常のファイルのXMLよりも大幅に小さくなります。

データの更新

ドキュメントを削除したり変更したりすると、どうなるでしょうか。ドキュメントを削除すると、そのドキュメントはMarkLogicによって削除済みとマークされますが、すぐにディスクから削除されるわけではありません。削除されたドキュメントは、削除のマークに従ってクエリ結果から除外されるようになります。そして、該当ドキュメントが格納されているスタンドを次にマージするとき、新しいスタンドへの書き込み段階でそのドキュメントは無視され、結果的にディスクから削除されます。

ドキュメントを変更した場合、MarkLogicでは、ドキュメントの旧バージョンが現在のスタンドで削除済みとマークされ、ドキュメントの新しいバージョンがインメモリスタンドに作成されます。ドキュメントがその場で変更されることはありません。ドキュメントに対する1回の変更がどれだけ多くのタームリストに影響するかを考えると、その場での更新は、完全に非効率的なことです。したがって、MarkLogicでは、変更されたドキュメントは新しいドキュメントとして処理され、旧バージョンは削除されたドキュメントとして処理されます。

ここでは、説明を少し単純化しています。前述のように、クエリ、取得、更新の基本単位はフラグメントであり、ドキュメントではありません。このため、断片化ルールが有効になっているときに、フラグメントがあるドキュメントを変更すると、MarkLogicによって変更が必要なフラグメントが判断され、そのフラグメントが削除済みとマークされ、必要に応じて新しいフラグメントが作成されます。また、MarkLogicには、削除されたフラグメントを一定期間残し、データベースのロールバックや、データのポイントインタイムリカバリを高速化できる設定があります。

データベース分野ではこのアプローチをMVCC（Multi-Version Concurrency Control）と呼びます。この後説明するように、ロックなしのクエリを実行できるなど、いくつかのメリットがあります。

MVCC（MULTI-VERSION CONCURRENCY CONTROL）

MVCCシステムでは、変更点がタイムスタンプ数値でトラッキングされます。この数値はクラスタ内でトランザクションが発生するたびに増分されます。フラグメントごとに作成時刻（作成時のタイムスタンプ、コミットされていないフラグメントは無限から開始）と削除時刻（削除済みとマークされたときのタイムスタンプ、削除されていないフラグメントは無限から開始）があります。ディスク上では、これらのタイムスタンプは`Timestamps`ファイルにあります。スタンドディレクトリで読み取り専用ではない唯一のファイルです。

データを変更しないリクエスト（「クエリ」、これに対して変更を行う場合があるのは「更新」）の場合、URIロックを回避することでシステムのパフォーマンスが向上します。クエリは、特定のタイムスタンプでの実行が想定され、その実行中は、他の（更新）リクエストが進行してデータを変更しても、タイムスタンプ時のデータベースを一貫して認識します。

MarkLogicではこのとき、通常のタームリスト制約に2つの制約が追加されます。1つめは、返されるフラグメントが「リクエストのタイムスタンプ以前に作成」されたものであること、2つめは、「リクエストのタイムスタンプ後に削除」されたものであることです。この2つの単純な条件から、特定のタイムスタンプに存在する新しい暗黙的タームリストといえるものを容易に作成できます。このタイムスタンプベースのタームリストがすべてのクエリに暗黙的に追加されます。ハイパフォーマンスで、ロックを取得する代わりになります。

ポイントインタイムクエリ

クエリは通常、その開始時刻に基づいて、タイムスタンプマーカーを自動的に取得します。しかし、以前の特定のタイムスタンプにおけるポイントインタイムクエリを実行することも可能です。この機能では、過去の任意の時点におけるデータベースの状態に対して、現時点でのクエリと同じ程度の効率でクエリを実行できます。この機能の一般的な用途の1つとして、新しいデータが読み込まれ、テストされている間にパブリックワールドを特定のタイムスタンプでロックできます。承認されたら、パブリック

ワールドのタイムスタンプが再び現在に戻ります。承認されなかった場合は、すべての変更を過去のタイムスタンプの状態に戻すことができます（「データベースのロールバック」）。

ポイントインタイムクエリの実行時には、マージを考慮に入れる必要があります。マージ時には通常、削除されたドキュメントが除去されます。削除されたドキュメントも含む過去の状態に対してクエリを実行する場合は、管理者としてmerge timestampの設定を調整し、その前ならドキュメントを取り戻すことができ、それを過ぎると取り戻せない日付を指定する必要があります。このタイムスタンプがポイントインタイムクエリを実行できる最も古い時点になります。

ロック

更新リクエストは読み取り専用ではないため、対象ドキュメントをロックし、変更時のシステムの整合性を維持する必要があります。このロック動作は暗黙的に行われ、ユーザーが制御することはできません[8]。読み取りロックは、他の操作による書き込みをブロックします。書き込みロックは、読み取りと書き込みの両方の操作をブロックします。更新時には、ドキュメントの読み取り前に読み取りロック、ドキュメントの変更(追 加、削除、修正)前に書き込みロックが取得されます。ロックの取得は先着順に処理され、書き込みロックを待っているリクエストが、同じリソースに対して新規にリクエストされた読み取りロックをブロックします（そうしないと、いつまでも書き込みロックを取得できないためです）。ロックは、リクエストの終了時に自動的にリリースされます。

どのようなロックベースのシステムでも、デッドロックを考慮する必要があります。デッドロックとは、複数の更新が互いに他方のロックを待ち、処理が停止した状態です。MarkLogicでは、デッドロックはバックグラウンドスレッドで自動的に検出されます。デッドロックが発生すると、最も進んだ更新（書き込みロック数と過去の再試行回数に基づく）が優先され、その他の更新はやり直しになります。

8　ほとんどの場合、これはユーザーの制御下にはありません。唯一の例外が`xdmp:lock-for-update($uri)`呼び出しです。これは、ドキュメントURIに対する書き込みロックをリクエストしますが、書き込みを実際に実行する必要はなく、URIが存在する必要もありません。ドキュメントに書き込まれることになる負荷の高い計算を行う前にドキュメントの書き込みロックを明示的に取得することで、デッドロックが検出され、ステートメントをやり直す必要が生じた場合も計算を繰り返す必要がありません。また、この呼び出しを使用して、各ステートメントが同じURIの書き込みロックを取得して開始するようにすれば、ステートメントを実行順序にシリアライズすることができます。この方法で、述語ロックを取得し、ファントムリードを回避できます（「分離レベルとMarkLogic」を参照）。

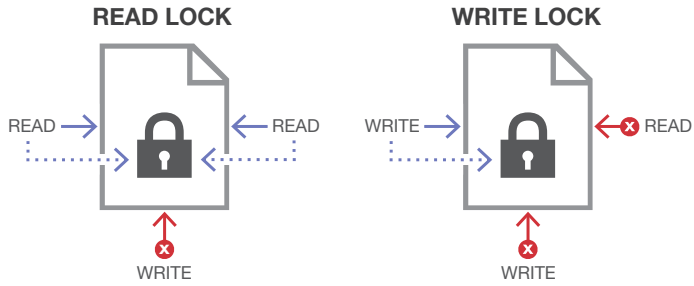


図7: 読み取りロックは、他の読み取りは許可するが、書き込みはブロックする。書き込みロックは、読み取りと書き込みの両方をブロックする。同時リクエストが可能なシステムでは、一部のリクエストが順番を待たなければならない場合がある

XQueryでは、MarkLogicは静的分析を使用してクエリと更新を区別します。リクエストの実行前に、コードに更新関数への呼び出しが含まれるかどうかが確認されます。含まれる場合は更新で、含まれない場合はクエリです。実行時に実際には更新関数が呼び出されない場合も、更新として実行されます（高度なヒント：リクエストを強制的に更新と見なす`xdmp:update`プロローグステートメントがあります。これは、静的アナライザで認識できない評価済みまたは起動済みのコードからの更新をリクエストで実行する場合に便利です）。JavaScriptでは、システムで静的分析が使用されないので、更新を実行するコードでは`declareUpdate()`関数でその意図を宣言する必要があります。

更新

更新の実行時にはロックが取得されますが、実際のコミット処理は更新が正常に完了してから行われます。更新がエラーで終了した場合、その更新に含まれていた保留中の変更はすべて破棄されます。デフォルトでは、各ステートメントがそれぞれ独立した自動コミットトランザクションです。マルチステートメントトランザクションについては後述します。

更新リクエスト中に、実行中のコードは行っている変更を認識できません。厳密には、それがまだ行われていないからです。更新関数を呼び出してもすぐにデータは変更されず、更新が正常に終了した場合に行うべき処理のキューに「作業指示」が追加されるだけです。

理論上、コードがその変更を認識できないのは、XQueryが関数型言語であり、関数型言語では、異なるコードブロックは相互依存していない場合に並列実行できるという興味深い最適化が可能であるからです。コードブロックを並列実行できる場合、更新（基本的に副作用）が任意の時点で完了していると想定できません。

ステートメント内の更新のバッチは競合しない必要があります。「競合しない」の定義は、簡単にいえば、どのような順序で実行しても同じ結果が得られる、というものです。例えば、ノードに子を追加し、そのノードを削除することはできません。実行

が逆だった場合、意味がないからです。しかし、同じ更新操作中に同じドキュメントに多数の変更を加え、それと同時に他の多くのドキュメントに対しても多数の変更を加えることは、同じアトミックコミットの一環として可能です。

> 更新の分離
>
> リクエストが数百万個のドキュメントにアクセスする可能性がある場合（大規模なデータセットのソートを行い、最近の項目を見つける場合など）、ロックなしで実行されるクエリリクエストのほうが、読み取りロックと書き込みロックを取得する必要がある更新リクエストよりもパフォーマンスが高くなります。場合によっては、更新処理を専用のトランザクショナルコンテキストに分離することで、クエリ処理を高速化できます。
>
> この手法は、更新が外部クエリに依存していない場合にのみ機能しますが、実際には依存している場合が一般的です。例えば、コンテキスト検索を実行し、ユーザーの検索文字列をトラッキング目的でデータベースに記録するとします。データベースの更新が、検索自体と同じトランザクショナルコンテキスト内にある必要はなく、同じコンテキスト内にあった場合は処理速度が低下します。この場合、1つのコンテキストで検索を実行し（読み取り専用でロックなし）、更新を別のコンテキストで実行したほうが適切です。
>
> 別のリクエスト内にあるリクエストを呼び出し、その2つのトランザクショナルコンテキストを管理する方法については、`xdmp:eval()`、`xdmp:invoke()`、`xdmp:spawn()`の各関数の説明を参照してください。

ドキュメントは行に類似

MarkLogic用にデータをモデル化するときは、ドキュメントをテーブルよりも行と考えます。言い換えると、数千個の項目がある場合は、数千個の子要素がある単一のドキュメントではなく、数千個の別個のドキュメントとしてモデル化します。これには2つの理由があります。

まず、ロックがドキュメントレベルで管理される点です。項目ごとにドキュメントが分かれていると、ロックの競合を回避できます。次に、すべてのインデックス、取得、および更新の操作がフラグメントレベルで実行される点です。このため、項目を検索、取得、または更新するときは、各項目がそれぞれ別個のフラグメントにあるほうが適切です。これを実現する最も簡単な方法は、それぞれ別個のドキュメントに分けることです。

もちろん、MarkLogicのドキュメントは単純なリレーショナルの行よりも複雑に記述できます。XMLやJSONのデータ形式は表現性に優れているからです。1つのドキュメントに、1つのエンティティ（マニフェスト、契約書、メール）が完全に記述されていることも稀ではありません。

ドキュメントのライフサイクル

ここで、以上のことを現実問題として考えるため、ドキュメントのライフサイクルを最初の読み込みから、削除して最終的にディスクから除去されるまでをトラッキングします。

XMLドキュメントの読み込みリクエストから、ドキュメントのライフサイクルが始まるとします。リクエストは、xdmp:document-load()関数呼び出しの一環として、ターゲットURIの書き込みロックを取得します。同じURIに対して別のリクエストがすでに書き込み中である場合は、読み込みはブロックされます。逆の場合も同様です。ある時点で、更新リクエストが最後まで正常に（エラーが発生して暗黙的なロールバックを発生させずに）完了すると、実際の挿入処理が開始し、更新処理のキューが処理されます。

MarkLogicでは、最初にドキュメントコンテンツの解析とインデックス付けが行われ、シリアライズされたXMLから、XMLデータモデル（厳密にはXQueryデータモデル[9]）の圧縮されたバイナリフラグメント表現にドキュメントが変換されます。フラグメントはインメモリスタンドに追加されます。この時点で、フラグメントは初期段階（nascent）のフラグメントと見なされます。この用語は管理画面のステータスページに表示されることがあります。初期段階とは、スタンドに存在するが、完全にコミットされていないことを指します。厳密には、初期段階のフラグメントの作成と削除のタイムスタンプはいずれも無限に設定されるので、システムによって管理はされていますが、まだクエリの対象とはなりません。大規模なトランザクショナル挿入を行う場合は、ドキュメントの処理中に初期段階のフラグメントが多数、蓄積されます。これらは、コミットされるまで、初期段階が維持されます。

ある時点で、ステートメントが正常に完了し、リクエストをコミットする準備が整います。MarkLogicは、次のタイムスタンプ値を取得し、トランザクションをコミットする意図をジャーナルに記録してから、新しいフラグメントの作成タイムスタンプをトランザクションのタイムスタンプに設定することで、フラグメントを使用可能にします。この時点で、この時点で、これは耐久性のあるトランザクションとなり、サーバーの障害が発生した場合は再生可能です。また、このタイムスタンプ以降に実行される新しいクエリに加えて、この時点以降の更新（進行中のものも含む）の対象となります。リクエストが終了すると、書き込みロックがリリースされます。

ドキュメントは、インメモリスタンド内で一定期間、完全にクエリを実行可能で永続性のある状態で維持され、インメモリスタンドがいっぱいになると、ディスクに書き込まれます。これで、ドキュメントがオンディスクスタンドに格納されます。

しばらくすると、マージアルゴリズムに基づいて、オンディスクスタンドが別のオンディスクスタンドとマージされ、新しいオンディスクスタンドが作成されます。フラグメントは引き継がれ、ツリーデータとインデックスはより大きなスタンドに統合されます。これは数回発生する場合があります。

その後、新しいリクエストが、xdmp:node-replace()呼び出しなどを使用してドキュメントに変更を加えます。変更を行うリクエストはドキュメントに最初にアクセスするときにURIの読み取りロックを取得してから、xdmp:node-replace()呼

9　データモデルは、XQueryとXPathデータモデルで詳細に定義され、多くの面でシリアライズされた形式とは異なる場合があります。例えば、XMLでシリアライズされているときは、属性値が一重引用符または二重引用符で囲まれている場合があります。データモデルでは、この違いは記録されません。

び出しを実行するときに、読み取りロックを書き込みロックに昇格します。別の実行中の更新によって、そのURIに別の書き込みロックが既存していた場合は、その書き込みロックがリリースされるまで、読み取りロックがブロックされます。別の読み取りロックが既存していた場合は、書き込みロックへのロック昇格がブロックされます。

更新リクエストが正常に完了した場合、処理は前述と同様に行われます。ドキュメントの解析とインデックス付けが行われ、初期段階のフラグメントとしてインメモリスタンドに書き込まれ、タイムスタンプが取得され、処理がジャーナルに記録され、開始タイムスタンプが設定されてフラグメントがライブ状態になります。更新であるため、古いフラグメントを削除済みとマークする必要があります。このとき、元のフラグメントの終了タイムスタンプが、トランザクションのタイムスタンプに設定されます。この組み合わせで、古いフラグメントが新しいフラグメントに置き換えられます。リクエストが完了すると、ロックがリリースされます。ドキュメントが削除され、新しいモデルに置き換えられます。

もちろん、まだディスクには存在します。更新によってタイムスタンプが増分される前に処理中であったクエリや、古いタイムスタンプのポイントインタイムクエリでは、まだドキュメントを認識できます。最終的に、フラグメントが保持されているオンディスクスタンドが再びマージされたときがこのドキュメントの最期になります。ドキュメントを残すように「merge timestamp」が設定されていない限り、新しいオンディスクスタンドに書き込まれることはありません。「マージのタイムスタンプ」が設定されていた場合、ドキュメントは後続のポイントインタイムクエリで使用できるように維持され、最終的に、終了タイムスタンプより後にマージが発生したときにマージから除外されます。

マルチステートメントトランザクション

前述のように、各ステートメントはデフォルトでそれぞれ独立した自動コミットトランザクションとして機能します。場合によっては、トランザクションをステートメントの境界にまたがって拡張することが有用です。マルチステートメントトランザクションを使用すると、例えばドキュメントを読み込んで処理し、処理が失敗した場合にトランザクションをロールバックし、問題のドキュメントをデータベースから除外できます。また、Javaプログラムを使用して複数の連続するデータベース更新処理を1つの単位としてまとめて実行できます。マルチステートメントトランザクションは、更新用だけではありません。マルチステートメントの読み取り専用クエリトランザクションは、複数の読み取りを同じタイムスタンプにまとめる簡単な方法です。

Javaからマルチステートメントトランザクションを実行するには、`session.setTransactionMode(Session.TransactionMode.UPDATE)`または`session.setTransactionMode(Session.TransactionMode.QUERY)`を呼び出します。前者では、読み取り書き込み更新モード（ロックを使用）、後者では読み取り専用クエリモード（固定タイムスタンプを使用）

になります。トランザクションを終了するには`session.commit()`または`xdmp:rollback()`を呼び出します。XQueryでトランザクションモードを制御するには、特殊な`xdmp:set-transaction-mode`プロローグを使用するか、オプションを`xdmp:eval()`、`xdmp:invoke()`、または`xdmp:spawn()`に渡し、`xdmp:commit()`または`xdmp:rollback()`で終わらせます。

マルチステートメントトランザクションの重要な側面として、各ステートメントでは前のステートメントの結果が認識されます。これらの変更は、データベースに完全にコミットされておらず、また他のトランザクションコンテキストでは認識できません。MarkLogicでは、このために、各更新トランザクションコンテキストに対応する、「追加」および「削除」されたフラグメントのIDが保持されます。これらは、データベースの通常のビューでオーバーライドとして機能します。マルチステートメントトランザクション内のステートメントでドキュメントを追加すると、そのドキュメントは初期段階のフラグメントとしてデータベースに格納されます（開始タイムスタンプと終了タイムスタンプが無限）。通常のトランザクションではこのようなドキュメントは見られませんが、同一トランザクション内の後続ステートメントでは新しいフラグメントIDがトランザクション固有の「追加」リストにも加えられます。これが、タイムスタンプビューを無効にします。同じトランザクション内の後続のステートメントがデータベースを見たとき、「追加」リストのフラグメントを、初期段階であっても認識すべきであることがわかります。削除の場合も同じです。マルチステートメントトランザクション内のステートメントがドキュメントを削除するとき、終了タイムスタンプがすぐに更新されるのではなく、フラグメントが「削除」リストに追加されます。これらのフラグメントは、タイムスタンプデータに関わらず、トランザクションコンテキストから隠されます。ドキュメント修正の場合はどうでしょうか。MVCCでは、削除と追加の組み合わせと同じです。ロジックは実際には非常にシンプルです。「削除」リストにあるフラグメントは認識できません。「追加」リストにある場合は認識できます。それ以外は、フラグメントの通常の開始タイムスタンプと終了タイムスタンプのデータに従います[10]。

マルチステートメントトランザクションでは、デッドロックの処理が興味深いものになっています。シングルステートメントの自動コミットトランザクションでは、検出したものがトランザクション全体であるため、通常はサーバーがステートメントを自動的に再試行できます。しかし、Javaクライアントからのマルチステートメントトランザクションのステートメントは、MarkLogicに情報がない任意のアプリケーション固有のコードが組み込まれる可能性があります。このような場合、MarkLogicでは、自動的に再試行せずに、`RetryableXQueryException`がスローされます。すると、呼び出し元のアプリケーションがその時点までのトランザクション内のリクエストを再試行する責任を負います。

マルチステートメントトランザクションに関与しているサーバーがトランザクション

10 マルチステートメントトランザクションは、MarkLogic 6で採用されましたが、これは新しいコードではありません。MarkLogicでは、同じシステムを使用したポストコミットトリガーが何年もの間、提供されていました。

中に再起動する必要が生じた場合は、自動的にロールバックされます。

XAトランザクション

XAトランザクションは、複数のシステムが関わる特殊なマルチステートメントトランザクションです。XAは「拡張アーキテクチャ（eXtended Architecture）」の略語であり、複数のバックエンドシステムが関わる「グローバルトランザクション」の実行を対象としたThe Open Groupの標準規格です。MarkLogicはXAトランザクションをサポートしているので、複数のMarkLogicデータベース間、またはMarkLogicデータベースとサードパーティ製データベースの間でトランザクションの境界をまたぐことができます。

MarkLogicでは、Java JTA（Java Transaction API）仕様を通じてXAが実装されています。技術的には、MarkLogicが提供するオープンソースの`XAResource`がJTA Transaction Manager（TM）にプラグインし、XAプロセスのMarkLogic固有部分を処理します。Transaction Manager（MarkLogicではなくJava EEベンダーが提供）は、一般的な2段階のコミットプロセスを実行します。TMは、すべての対象データベースにコミットの準備を指示します。各データベースは、準備ができているかどうかを返します。すべてのデータベースの準備が整ったら、TMはコミットを指示します。1つあるいは複数のデータベースでコミットの準備ができていなかった場合、TMがすべてのデータベースのロールバックを指示します。グローバルコミットが実行されたどうかは、TMの視点で判断されます。重要なときにTMで持続的な障害が発生し、コミットの状態があいまいになった場合に備えて特殊な処理機能があります。

フォレスト内のストレージタイプ

次の短いセクションでは、MarkLogicがさまざまなストレージタイプをどのように使用してパフォーマンスを最大化しながら、同じフォレスト内のストレージコストを最小化しているかを見ていきます。「階層型ストレージ」のセクションで、異なるストレージ階層間でフォレスト全体を移動する方法について説明しています。

SSD上の高速データディレクトリ

SSD（ソリッドステートドライブ）は、回転式ハードディスクと比べて非常に高性能で、価格も大幅に高くなっています。価格が高いため、すべてのフォレストデータをSSDに格納できるのは、小規模なデータセットに対する導入の場合のみです。コストをかけずにSSD水準のパフォーマンスを実現できるように、MarkLogicには、各フォレストに設定可能な「高速データディレクトリ」が用意されています。これを、高速ファイルシステム（SSDを使用したものなど）上に構築したディレクトリに設定します。この設定は必須ではありません。この設定が存在しなくても特別なことは起こらず、すべてのデータが通常の「データディレクトリ」に配置されるだけです。ただし、存在した場合は、フォレストでチェックポイントまたはマージが行われるたびに、MarkLogicが新しいスタンドを高速データディレクトリに書き込もうとします。容量がなく、書き込めなかった場合は、通常のデータディレクトリが使用されます。

もちろん、データディレクトリへのマージが行われると、高速データディレクトリに将来的にスタンドを格納できる空き容量ができます。フォレストのジャーナルも高速データディレクトリに配置されます。このため、すべてのチェックポイントと、より小規模でより頻繁なマージがSSDで行われ、ディスクI/O帯域幅の使用率が向上します。頻繁に更新されるドキュメントは小規模なスタンドに格納される傾向があるため、SSDに格納される可能性が高くなります。

テストにより、システムのストレージ容量の5～10%をソリッドステートにすると、高い費用対効果が得られることがわかっています。ソリッドステートストレージをシステムに導入する場合は、SLCフラッシュ（パフォーマンス、信頼性、寿命が優れている）であり、またPCIベースであるか、専用のコントローラがある必要があります。

バイナリ用の大規模なデータディレクトリ

MarkLogic内にバイナリドキュメントを格納すると、いくつかのメリットがあります。それは、ドキュメントがクエリ可能なデータベースの一部になるということです。バイナリをトランザクションの対象にし、またトランザクショナルデータベースのバックアップに含めることができます。更新に失敗した場合は、迅速なポイントインタイムリカバリで修正できます。フェイルオーバーや高可用性を実現するために必要なデータレプリケーション処理に含めることができ、またMarkLogicのセキュリティシステムでセキュリティを確保できます。

ただし、前述のマージの説明を思い出すと、大規模なバイナリはマージ処理に含めないことが推奨されます。大規模なバイナリドキュメントをスタンド間でコピーするのは非効率的であり、また不必要です。

このため、MarkLogicでは大規模なバイナリ用に特別な処理方法が用意されています。「large size threshold」（デフォルトでは1MB、32KB～512MBで設定可能）を超えるバイナリドキュメントは特別な方法で処理され、ディスク上でスタンド内ではなく別個のディレクトリに配置されます。このディレクトリはデフォルト名が`Large`で、スタンドのディレクトリと並んでいます。この`Large`ディレクトリの下で、バイナリドキュメントは通常のファイルとして配置され、基本的にランダムに名前が付けられます。データベース内には別個のファイルを参照するハンドルが維持されます。プログラマの観点からは、バイナリドキュメントは他のドキュメントと同様にデータベースの一部ですが、MarkLogicでは、最適化として、マージの対象とならないように`Large`ディレクトリに別個に維持されます。データベースドキュメントを削除する必要が生じた場合、データベース内のハンドルとディスク上のファイルの両方がMarkLogicによって削除されます。

大規模なバイナリの格納に高速なディスクは必要ないので、MarkLogicではフォレストごとに「large data directory」を設定できるようになっています。これは、「高速データディレクトリ」の反対です。容量は多いがギガバイトあたりのコストが低く、おそらくはシーク時間も遅く、1秒あたりのI/O処理数が少ないファイルシステムを指定できます。通常は、リモートのNASまたはHDFSにホストします。「大規模データディレ

クトリ」が構成されている場合、MarkLogicでは、大規模なバイナリが、スタンドと並んでいるLargeディレクトリではなく、ここに保存されます。

MarkLogicでは、「外部バイナリドキュメント」もサポートされています。これは多くの点で大規模なバイナリドキュメントと似ていますが、ディスク上の別個のファイルをプログラマが管理するという違いがあります。ファイルへのハンドルはMarkLogicで管理されますが、物理的なファイルへのパスと、その作成と削除はプログラマの責任になります。この選択肢は、アマゾンウェブサービスに展開し、大規模なバイナリをS3に配置する場合に適しています。これらを外部バイナリドキュメントとしてホストすることで、クライアントがMarkLogicのノードを経由せずにAWSサーバーから直接ドキュメントを引き出すことができます。

クラスタとキャッシュ

データのサイズが拡大し、リクエストの負荷が増加すると、ソフトウェアベンダーから次の2つのうちいずれかのようなことを伝えられることがあります。サーバーがモノリシックな構成であるため、より大規模なボックスの購入を求められるか（買い替え時は常に前のものよりも大幅に高価になる）、システムを、複数のコモディティサーバーをまとめて単一運用できるクラスタ構成にしたと言われることです（低コストを維持できる）。MarkLogicはどちらでしょうか。MarkLogicはクラスタ化の仕様になっています。

クラスタ化には主に4つのメリットがあります。

1. 手頃な価格で購入したコモディティサーバーを使用できる
2. 必要に応じて（データまたはユーザーの増加に応じて）サーバーを徐々に追加（または削除）できる
3. サーバーをそれぞれ異なるロール向けに最適化し、データの異なる部分を管理することで、キャッシュのローカル性を最大化できる
4. フェイルオーバー機能を持たせ、サーバーの障害に対処できる

クラスタ化されたMarkLogicサーバーには、2つのロールのうちいずれかを割り当てます。エバリュエーター（Eノード）またはデータマネージャ（Dノード）です。Eノードは、ソケットで待機し、リクエストを解析し、応答を生成します。Dノードは、対応するインデックスとともにデータを保持し、Eノードがリクエストに対応したり、更新を処理したりするために必要なデータを提供することでEノードを支えます。

ユーザー負荷が増加したら、Eノードを追加できます。データサイズが拡大したら、Dノードを追加できます。中規模のクラスタならば、Eノード2台とDノード10台を構成し、Dノードにそれぞれデータ全体の約10%を割り振ることができます。

通常は、ロードバランサーが入ってきたリクエストをEノード間で分散します。Eノードはリクエストを処理し、リクエスト内でデータの取得または格納に関わる部分式をDノードに任せます。例えば、リクエストで、クエリと一致する最近の10項目が求

められているとします。Eノードが、データベースのフォレストを持つすべてのDノードにこのクエリ制約を送信すると、各Dノードがそのデータ部分における最近の結果を返します。Eノードはこれらの部分的な応答を単一の統合された応答、つまりクラスタ全体における最近の項目にまとめます。すべてのリクエストで、クラスタ内の往復が頻繁に発生します。この往復は、インデックスの処理、ドキュメントのロック、フラグメントの取得、またはフラグメントの格納が必要な部分式がリクエストに含まれる場合に必ず発生します。

Dノードは、効率のために、Eノードで本当に必要ではない限り、フラグメント全体を送信しません。例えば、関連度に基づく検索を行うときは、各Dノードのフォレストがイテレータを返します。イテレータ内には、インデックスから抽出され、順番に並べられた一連のフラグメントIDとスコアがあります。EノードはDノードから返される各イテレータからエントリを引き出し、最高スコアに基づいて処理するフラグメントを判断します。Eノードが特定の結果を処理する必要がある場合は、そのフラグメントを適切なDノードからフェッチします。

クラスタ管理

各サーバーにインストールするMarkLogicサーバーソフトウェアは、ロールに関わらず常に同じです。サーバーが、入ってくるリクエスト（HTTP、XDBC、WebDAV、ODBCなど）をソケットで待つように構成されている場合、それはEノードです。データを管理する（1つまたは複数のフォレストが関連付けられている）場合、それはDノードです。MarkLogicの管理ページで、構成が同じであるサーバーから名前付きグループを作成できるので、「Eグループ」と「Dグループ」を簡単に設定できます。ラップトップなど、サーバー1台の簡単な環境では、この単一のMarkLogicインスタンスがEノードとDノードの両方の役割を担います。

クラスタの設定は簡単です。新しいMarkLogicインスタンスの管理画面に初めてアクセスするとき、インスタンスを既存のクラスタに追加するかどうかを尋ねられます。追加する場合、クラスタ内の他の任意のサーバー名と追加するグループを指定すると、そのグループの設定に従って新しいシステムが自動的に構成されます。クラスタ内のノードは、ポート7999でXDQPという専用のプロトコルを使用して通信します。

キャッシュ

MarkLogicでは、データベース作成時に、使用ハードウェア向けに最適化されたデフォルトのキャッシュサイズが割り当てられます。このとき、サーバーがEノードとDノードの両方として機能すると想定されます。各グループのキャッシュサイズを最適化すると、クラスタ環境のパフォーマンスを向上できます。Eノードグループでは、データ管理に関連するキャッシュを犠牲にして、リクエストの評価に関連するキャッシュを増やします。データマネージャでは逆のことを行います。主なキャッシュ、その機能、クラスタ環境で変更すべき点を以下に示します。

リストキャッシュ
このキャッシュは、ディスクから読み取られた後のタームリストを保持します。インデックスの解決はDノードでのみ行われるので、Dノードグループではこのサイズを拡大し、Eノードでは最小値に設定するのが適切です。

圧縮ツリーキャッシュ
このキャッシュは、ディスクから読み取られた後のドキュメントフラグメントを保持します。ドキュメントフラグメントは、容量を削減し、I/O効率を改善するために、圧縮して格納されています。ディスクからのフラグメントの読み取りは完全にDノードのタスクであるので、Dノードではこのサイズを拡大し、Eノードでは最小値に設定するのが適切です。

展開ツリーキャッシュ
Dノードがネットワークを介してフラグメントをEノードに送信するときは、格納時と同じ圧縮形式で送信します。Eノードはこのフラグメントを使用可能なデータ構造に展開します。このキャッシュには、この展開ツリーインスタンスが格納されます。

展開ツリーキャッシュは、Eノードではサイズを拡大し、Dノードでは大幅に縮小するのが適切です。それでは、ゼロまで縮小しないのはなぜでしょうか。Dノードにも、バックグラウンドでの再インデックス付けをサポートするために展開ツリーキャッシュが必要だからです。また、Dノードグループに管理ポート8001が含まれる場合は（クラスタを離れたボックスを直接管理する必要が生じた場合のため）、この管理作業に十分な展開ツリーキャッシュが必要です。経験則として、Dノード上の展開ツリーキャッシュは128MBに設定します。

Eノードは、フラグメントが必要になると、最初にローカルの展開ツリーキャッシュを確認します。そこになかった場合、Dノードに送信を依頼します。Dノードは最初に圧縮ツリーキャッシュを確認します。そこになかった場合は、Dノードがフラグメントをディスクから読み取り、ネットワーク経由でEノードに送信します。各Dノードで、そのデータのサブセットのリストキャッシュと圧縮ツリーキャッシュが維持されるため、キャッシュのローカル性のメリットがあります。また、MVCCがあるため、更新時に展開ツリーキャッシュを無効化する必要がありません[11]。

バイナリドキュメントのキャッシュ
大規模な外部バイナリドキュメントは、ツリーキャッシュに関して特別な方法で処理されます。大規模なバイナリ（設定可能なサイズのしきい値を超え、MarkLogicによってディスク上で特別な方法で管理されているもの）や外部バイナリ（外部に保持され、MarkLogic内から参照されているもの）は、圧縮ツリーキャッシュに格納されますが、それはチャンク単位です。チャンク化によって、大量のバイナリでも、キャッシュの問題が発生しません。これらのバイナリが展開ツリーキャッシュに格

11 「XDMP-EXPNTREECACHEFULL: Expanded tree cache full」というエラーが発生しても、展開ツリーキャッシュのサイズを拡大する必要があるとは限りません。このエラーメッセージは、クエリによって読み取られているデータ量が多すぎてメモリバッファがいっぱいになっていることを示します。適切な解決方法は、より効率的に処理ができるようにクエリを書き直すことです。

納されることはありません。データは圧縮ツリーキャッシュ内でネイティブのバイナリ形式になっているため、意味がありません。

外部バイナリは、Eノードによって外部ソースから取得されます。このため、外部バイナリがあるシステムでは、Eノードに十分に大規模な圧縮ツリーキャッシュがあることを確認する必要があります。

キャッシュのパーティション

各キャッシュサイズを設定すると同時に、キャッシュパーティション数も設定できます。デフォルトでは、各キャッシュのパーティションはメモリサイズに応じて1つまたは2つ（場合によっては4つ）に設定されています。この数を増やすと、キャッシュの並列性を向上できますが、効率は下がります。その仕組みは次のとおりです。キャッシュに変更を加えるスレッドは、更新をスレッドセーフな状態に維持するために、キャッシュの書き込みロックを取得する必要があります。短期的なロックですが、書き込みアクセスをシリアライズできる効果があります。パーティションが1つの場合、その単一の書き込みロックを通じてすべてのスレッドをシリアライズする必要があります。パーティションが2つの場合、2つの異なるキャッシュと2つの異なるロックがあるので、2倍の数のスレッドがキャッシュの更新を同時に行うことができます。

スレッドでは、エントリを格納または取得するキャッシュパーティションをどのように判断するのでしょうか。これは、キャッシュのルックアップキー（検索する項目の名前）に基づいて決まります。キャッシュにアクセスするスレッドは、最初にルックアップキーを確認し、そのキーがあるキャッシュを確認し、そのキャッシュにアクセスします。キーに適切なパーティション以外の読み取りロックや書き込みロックは必要ありません。

パーティション数が多すぎると、キャッシュの効率が下がります。各キャッシュパーティションが、それぞれのエントリのエージアウトを管理する必要があり、そのパーティションで最も古いエントリだけを削除できます。別のパーティションにもっと古いエントリがあったとしても何もできません。

キャッシュのチューニングの詳細については、『Query Performance and Tuning Guide』を参照してください。

グローバルキャッシュ無効化が不要

一般的な検索エンジンでは、管理者が更新の頻度とキャッシュのパフォーマンスの間でトレードオフを行う必要に迫られます。エンジンではグローバルキャッシュが保持され、ドキュメントが変更されるとキャッシュが無効になるからです。MarkLogicでは、キャッシュをスタンド対応にすることでこの問題を防止しています。前述のように、スタンドは、フォレストの読み取り専用の構成要素です。新しいドキュメントが読み込まれても既存のスタンドのコンテンツは変わらないので、更新が発生したときに、パフォーマンスに影響するグローバルキャッシュの無効化は必要ありません。

クラスタ内のロックとタイムスタンプ

クラスタ全体でロックと、トランザクションタイムスタンプを管理することは、ボトルネックをもたらし、パフォーマンスを損なうタスクのように思えるかもしれません。幸い、MarkLogicにはそれは当てはまりません。

MarkLogicでは、ロックは分散的に管理されます。各Dノードが、そのフォレストの下のドキュメントのロックを管理する責任を負います。これは、通常のデータアクセス処理の一環として行われます。例えば、更新リクエストを実行しているEノードがドキュメントセットを読み込む必要がある場合、そのドキュメントがあるDノードはデータを返す前に必要な読み取りロックを取得します。Dノードは、ロック処理についてすぐにEノードに通知しません。その必要がないからです。事実、Dノードは、クラスタ内の他のどのホストにも確認せずに、データのサブセットのロックを取得できます。このため、特定のドキュメントのすべてのフラグメントが同じフォレストに配置されます。

クラスタ内のホスト間で送信される通常のハートビート通信で、各ホストは、(ロックされているため)待っているトランザクションを報告し、バックグラウンドのデッドロック検出スレッドをサポートします。

トランザクションタイムスタンプも分散的に管理されます。更新処理の最後に、変更元の1つまたは複数のDノードが最新のタイムスタンプを確認し、増分し、そのタイムスタンプを新しいデータに使用します。タイムスタンプを取得するためにクラスタ全体の調整は必要ありません。他のホストは新しいタイムスタンプを、各ホストから送信されるハートビート通信の一部として認識します。各ホストは認識している中で最新のタイムスタンプをブロードキャストし、クラスタ全体での最大値をトラッキングします。タイムスタンプは以前は単調増加する整数でしたが、現在は、通常の時刻と厳密に一致する値になっています(`xdmp:timestamp-to-wallclock()`と`xdmp:wallclock-to-timestamp()`を参照)。このため、特定の時刻までのデータベースのロールバックが簡単になります。ただしそのためには、MarkLogicクラスタ内のすべてのマシンのクロックが正確であることが重要です。

2つの更新がほぼ同じ頃に発生し、Dノードの各セットが次のハートビートをまだ見ていないため、同じ新しいタイムアウト値が選択されたとしたらどうでしょうか。これは発生する可能性がありますが、実際には問題ありません。いずれの更新も、同じタイムスタンプで発生したとマークされます。これは、更新がそれぞれ完全に独立している場合にのみ発生します(そうでなければ、更新に関わるDノードがより大きなタイムスタンプを認識します)。この場合、一方をシリアライズして、他方よりも前に発生したように見せかける必要はありません。一連の独立した更新の間でシリアライゼーションが必要であることが明らかな特別な場合は、更新時に同じURI書き込みロックを取得し、異なる数値のタイムスタンプにトランザクションを自然にシリアライズできます。

1つのリクエストが書き込みを実行し、その直後、1秒未満の間にハートビート通信で別のリクエストが読み取り専用のクエリを、最初の書き込みの対象ではなかった別のホストに対して実行した場合はどうでしょうか。この場合、2台目のホストは増加したタイムスタンプのことを知らず、2つめのリクエストは古いデータを読み取る可能性があります（ただし、このシナリオは2つめのリクエストが純粋に読み取り専用の場合にのみ発生するので、2つめのリクエストが古いデータの読み取り時に誤った更新を行うことはありません）。このようなことが問題である場合は、タイムスタンプ分散アプリケーションサーバー構成オプションを調整できます。デフォルトでは［fast］で、更新はできるだけ迅速に返され、トランザクションの対象ではないホストに特別なタイムスタンプ通知はブロードキャストされません。［strict］に設定されている場合、すべての更新で、タイムスタンプ通知メッセージが、グループ内の他のすべてのホストにすぐにブロードキャストされ、更新は、タイムスタンプが分散されるまで返されません。

クラスタ内でのクエリのライフサイクル

クエリのライフサイクルについては前述しました。ここでもう一度、EノードとDノードのやり取りとキャッシュの位置付けに着目して、ライフサイクルを見ていきます。

前述と同じ`cts:search()`を実行するとします（猫と子犬を検索、魚は除外）。タームリストを収集するために、Eノードは検索式の表現をデータベース内の各フォレストに並行してプッシュします。Dノードの各フォレストは、そのスコア順の結果へのイテレータを返します。タームリストの選択と積集合の作成はすべてDノード上で行われます。必要な場合はロックもDノード側で発生しますが、この例は読み取り専用のクエリであるため、ロックなしで実行されます。

Eノードは、最高スコアを持つイテレータから選択し、引き出すことで、データベース全体で最も関連度の高い結果を特定します。ネットワーク経由でフラグメントIDとスコアデータを受け取ります。Eノードがフラグメントに対応する実際のフラグメントを取得するときは、最初に展開ツリーキャッシュを確認します。見つからなかった場合、フラグメントIDを受け取ったDノードにフラグメントをリクエストします。Dノードは圧縮ツリーキャッシュ内でフラグメントを探し、そのキャッシュにもなかった場合、ディスクから読み取ります。

この時点で、Eノードがフラグメントをフィルタリングできます。Eノードはツリー構造を見て、本当に検索の制約を満たしているかを確認します。フィルタリングを通過した場合にのみ、return節に進み、生成された結果になります。クエリがフィルタリングなしで実行されている場合、この確認は行われません。

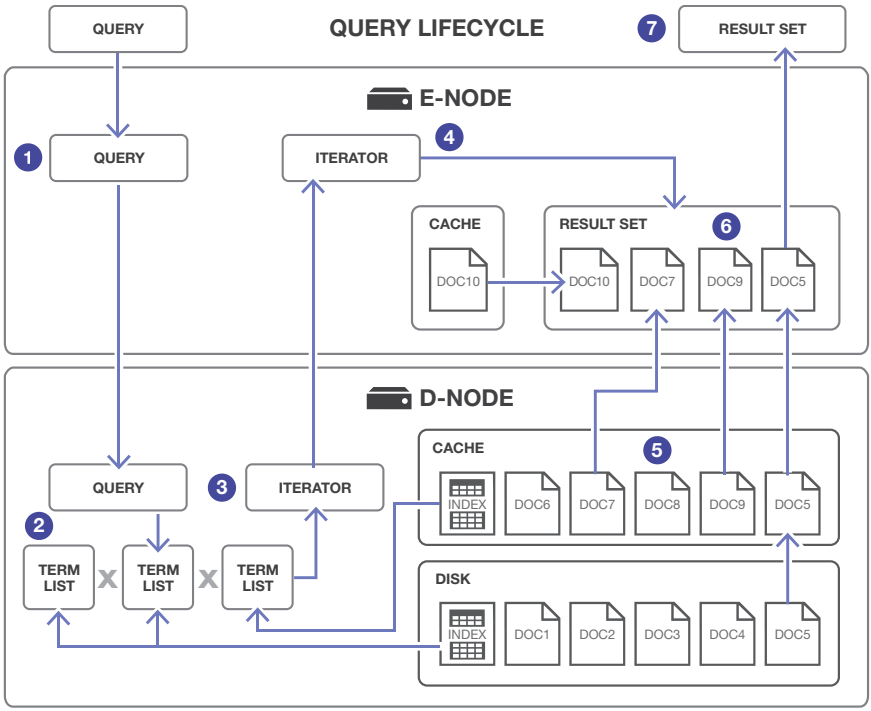


図8: クエリがEノードに送信され、クエリの表現がDノード内のフォレストにプッシュされる**(1)**。タームリストの選択と積集合の作成がDノードで行われる**(2)**。タームリストがキャッシュまたはディスクから取得される。各Dノードはスコア順の結果を指すイテレータを返す**(3)**。Eノードがイテレータを使用して、最も関連度の高い結果をDノードから取得する**(4)**。Dノードがキャッシュまたはディスクから結果フラグメントを取得する**(5)**。ディスクから取得されたフラグメントは将来的なアクセスのためにキャッシュに格納される。ドキュメントのフラグメントが返されたら、Eノードはフィルタリングを実行して、フラグメントが本当にクエリと一致していることを確認できる**(6)**。フィルタリングを正常に通過した結果を含む結果セットが返される**(7)**

クラスタ内での更新のライフサイクル

更新対象のドキュメントのライフサイクルについても前述しました。同じシナリオを振り返り、クラスタ環境の場合をより詳細に見ていきます。MarkLogicがACID準拠であること、つまりMarkLogicのトランザクションが次の特性を持つことがわかります。

- **原子性:** トランザクション内のデータ修正がすべて実行されるか、まったく実行されないかのいずれか
- **一貫性:** トランザクションの完了時に、すべてのデータが一貫した状態で残る
- **分離性:** トランザクションは、同時に実行している他のトランザクションの影響から保護される
- **永続性:** トランザクションのコミット後は、予想外の中断があっても、その影響は残る。永続性を持つデータベースは、システム障害から回復するメカニズムを持つ

挿入呼び出しの時点で、リクエストは書き込みロックを取得する必要があります。ロックを取得するために、すべてのフォレストにリクエストを送信し、ドキュメントが既存するかどうかを確認します。既存する場合は、そのフォレストが書き込みロックを取得し、ドキュメントの新しいバージョンがフォレストに戻されます。URIが新しい場合は、Eノードが、ドキュメントを配置するフォレストを選択し、そのフォレストが書き込みロックを取得します。Eノードは、デフォルトで、URIと抽象「バケット」の対応に基づいて決定論的な方法でフォレストを選択します（詳細については、「リバランス」のセクションを参照）。これにより、他の同時リクエストが同じDノードの書き込みロックを確認しにいくことが保証されます（フォレストは異なる方法で選択することもできます。詳細については、「リバランス」および「フォレスト配置におけるロック」の各セクションを参照してください）。

MarkLogicでは、書き込み時にロックを取得することで、更新が確実に分離されます。同じドキュメントを更新したいトランザクションは順番を待つ必要があります。また、ロックは一貫性の確保にも役立ちます。ロックがなかった場合、同じドキュメントを同時に更新する複数のトランザクションが、値の一貫性を損なう可能性があります。

更新リクエストのコードが正常に完了したら、Eノードがコミット処理を開始します。ドキュメントを解析してインデックスを付け、圧縮したドキュメントを、事前に選択したDノードのフォレストに送信します。Dノードはインメモリスタンドを更新し、タイムスタンプを取得し、変更をディスク上のジャーナルに記録し、ドキュメントのタイムスタンプを変更してドキュメントをライブ状態にしてから、書き込みロックをリリースします。

MarkLogicは、更新されたドキュメントをライブ状態にする前にディスク上のジャーナルに更新を書き込むことで、永続性を確保します。システム障害が原因でメモリ内の変更が取り消されることになっても、ジャーナルの内容を再生することでデータを再構築できます。

挿入または更新に複数のドキュメントが関わる場合はどうなるのでしょうか。また、トランザクションが複数のフォレストにまたがった場合はどうでしょうか。MarkLogicでは、更新が原子的に行われるか、まったく行われないかのいずれかになるように、標準的な2フェーズコミットをDノード間で使用します。詳細については、図9を参照してください。

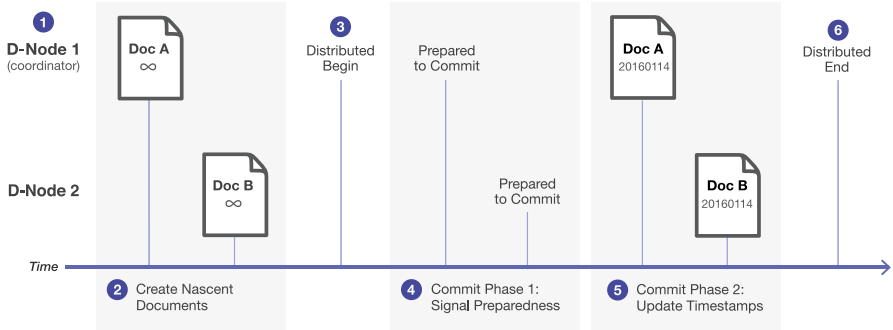


図9: 複数のDノードが関わるコミット時には、1つのDノードがコーディネーターとして機能する**(1)**。最初に、システムが初期段階のドキュメントを作成（タイムスタンプは無限に設定）し**(2)**、コーディネーターがdistributed beginエントリをジャーナルに記録する**(3)**。コミットの最初の段階で、更新が行われ、コミットの準備ができたことを対象ノードが通知する**(4)**。すべてのホストの確認が済んだら、第2段階に進み、タイムスタンプが更新され、ドキュメントがライブ状態に移行する**(5)**。準備ができていないと通知するノードがあった場合は、MarkLogicがトランザクションをロールバックする。最後に、コーディネーターがdistributed endエントリをジャーナルに記録し、コミットが完了する**(6)**

フォレスト配置におけるロック

MarkLogicではデフォルトで、選択されている割り当てポリシーに従って、新しいドキュメントの配置場所が選択されます。挿入呼び出しでは、ドキュメントを特定のフォレストに配置することをリクエストすることもできます。これには2通りの方法があります。1つめの方法では、ドキュメント挿入呼び出しのパラメータとして、フォレスト配置キー（フォレストID）を指定し、ドキュメントに特定のフォレストを指定できます。この場合、ロックは、配置キーがなかった場合と同じように、決定論的ハッシュで選択されたフォレストによって管理されます。このため、2つの異なるフォレストに対する2つの同時フォレスト配置で同じロックが尊重されます。2つめの方法として、ドキュメント挿入呼び出しを「フォレスト内eval」（`eval()`関数への非公開のパラメータを使用）内で実行できます。この場合、実行されるコードはすべて選択されたフォレストのコンテキスト内でのみ実行されます。この高度な方法では、選択されたフォレストだけがロックを管理します。実行速度が少し早いですが、他のフォレストで重複するURIは確認されません。それはアプリケーションの責任になります。Hadoopコネクタでは、この高度な方法がときどき使用されます。

分離レベルとMARKLOGIC

ACIDプロパティにおける分離は、開発者のコードが、対象データベースに対して実行されている唯一のコードであるかのように振る舞うことです。データベースでこれを実現する1つの方法は、すべてのトランザクションをシリアライズし、各トランザクションが、前のトランザクションの完了を待つようにすることです。ただし、このようなデータベースは通常はパフォーマンスが許容できないほど低くなります。このため、MarkLogicは、他のハイパフォーマンスのデータベースと同じように、トランザクションを並列実行します。1つのトランザクションが別のトランザクションの処理を認識できる範囲を、データベースの「分離レベル」といいます。1つのトランザクションが別のトランザクションを認識する方法は3つあります。

ダーティリード：1つのトランザクションに、別のトランザクションが作成または変更したがまだコミットされていないドキュメントの読み取りを許可すること。他方のトランザクションが失敗してロールバックを実行した場合、最初のトランザクションは、存在しないドキュメントを読み取ったことになります

非リピータブルリード：トランザクション中にドキュメントを複数回読み取ると、他のトランザクションによる同時更新が原因で、その都度、値が異なること

ファントムリード：トランザクション中に、2つの同じ検索クエリを実行すると、返されるドキュメントセットが異なること。別の同時実行のトランザクションが、クエリと一致するデータを挿入、更新、または削除したことに起因します

ANSI SQLでは、データベースマネージメントシステムが上記の問題にどの程度、取り組んでいるかを表す4つの分離レベルが定義されています。

- **読み取り非コミット**：3つの現象すべてが起こり得ます。基本的にまったく分離がない状態です
- **読み取りコミット：**ダーティリードは起こりませんが、非リピータブルリードとファントムリードは起こり得ます
- **リピータブルリード：**ダーティリードと非リピータブルリードは起こりませんが、ファントムリードは起こり得ます
- **シリアライズ可能**：3つのどの現象も起こりません

MarkLogicの対応はどうなっているでしょうか。読み取り専用トランザクションについては、MarkLogicは、MVCCによって「シリアライズ可能」な分離を行います。MarkLogicにおける読み取り専用トランザクションは、トランザクションが認識可能なドキュメントを決定するタイムスタンプに関連付けられています。同時実行のトランザクションが、読み取り実行時にドキュメントを更新した場合、更新されたドキュメントにはそれより後のタイムスタンプが割り当てられます。このため、ドキュメントの更新されたバージョンは、読み取り時に認識できません。

更新トランザクションについては、MarkLogicは「リピータブルリード」で、コードパターンで「シリアライズ可能」になります。書き込みロックによって、トランザクションが非コミットデータを読み取ることはできず、ダーティリードが防止されます。読み取りロックによって、他のトランザクションが読み取り中のドキュメントを変更することはできず、非リピータブルリードが防止されます。MarkLogicでは、検索結果内のドキュメントに対する読み取りロックを取得することで、一部のファントムリードが防止されます。これにより、あるトランザクションが以前に検索したドキュメントを別のトランザクションが削除したり、更新したりできないようになります。ただし、あるトランザクションが以前に検索した結果セットに、別のトランザクションがドキュメントを挿入することは防止されません。これが1つの抜け穴です。

すべてのファントムリードからシステムを保護するにはどうすればいいでしょうか。このようなことは通常は必要ありません。しかし、2つのトランザクションを完全にシリ

アライズ可能にする必要がある場合は、それぞれがセンチネルドキュメントへのロックをプログラムで取得することで、シリアライズ可能な実行を実現できます。合成URIの読み取りロックは`fn:doc()`で、書き込みロックは`xdmp:lock-for-update()`で取得できます。

コーディングと、MARKLOGICへの接続

MarkLogicのデータモデル、インデックスシステム、更新モデル、動作について説明したので、次にMarkLogicでプログラミングと操作に使用できるさまざまなオプションを見ていきます。まず、MarkLogicでシングルティアのアプリケーションを開発できるサーバー側の言語オプション（XQuery、XSLT、JavaScript）を示します。その後、MarkLogicのREST APIとのやり取りに基づくマルチティアのオプションを見ていきます。これには、JavaとNode.js クライアントAPIが含まれます。これらを使用して、MarkLogicで標準的な3層アプリケーションを構築できます。

XQUERYとXSLT

MarkLogicには、XQuery 1.0とXSLT 2.0のサポートが含まれます。これらはW3C標準、XML指向の言語であり、XMLの処理、クエリ、変換のために設計されています。

サーバーは、実際にはXQueryの次の3つのダイアレクトに対応します。

1.0-ml

最も一般的な言語です。XQuery 1.0の完全な実装に、MarkLogic固有の拡張を加えることで、検索、更新、try/catchエラー処理、およびXQuery 1.0言語に含まれないその他の機能がサポートされています。

1.0

[strict]と表されることもあります。拡張のないXQuery 1.0の実装であり、XQuery 1.0の他のプロセッサとの互換性のために用意されています。1.0でもMarkLogic固有の関数を使用できますが、関数の名前空間を自分で宣言する必要があり、またそのような呼び出しは移植可能ではありません。try/catchエラー処理のような言語拡張は使用できません。

0.9-ml

下位互換性のために用意されています。XQuery 1.0の前にMarkLogicが実装していた2003年5月のXQueryプレリリース仕様に基づいています。

各XQueryファイルの先頭で、ファイルが記述されているダイアレクトを宣言できます。宣言がなかった場合、アプリケーションサーバーの構成によって、デフォルトのダイアレクトが判断されます。同じプログラム内に複数のダイアレクトを混在させることも可能です。これは非常に便利です。新しいプログラムで古いライブラリを活用し、古いプログラムで、新規作成されたライブラリを活用できます。

XQueryに加えて、XSLTも使用できます。両方を一緒に使用することもできます。XQueryをXSLTから呼び出し、XSLTをXQueryから呼び出せます。つまり、特定のタスクに最適な言語を常に使用し、サポートするライブラリを最大限に再利用できます。

詳細については、『Application Developer's Guide』およびXQuery and XSLT API Documentationを参照してください。

JAVASCRIPT

MarkLogicには、JavaScriptのサーバーサポートも含まれます。JavaScriptは、Webブラウザのプログラミング言語として始まり、広く普及している言語です。JavaScriptは近年、Node.jsなどのサーバー側プラットフォームも対象とするようになりました。MarkLogicのネイティブなJavaScriptサポートは、この使い慣れた言語をさらに1歩進めてデータベースに採用しています。

JavaScriptをサポートするため、MarkLogicにはGoogle V8 JavaScriptエンジンが統合されています。これは、JavaScript言語のハイパフォーマンス、オープンソースのC++実装です。MarkLogicで記述するJavaScriptプログラムは、次のものを利用できます。

- データ管理、サーバー管理を実行するためのビルトインのMarkLogic JavaScript関数
- MarkLogicに格納されたJavaScriptとXQueryのカスタムライブラリ
- MarkLogicのXQueryライブラリ

XQueryやJavaScriptのライブラリは、`require()`関数を使用してプログラムにインポートします。これは、モジュールインポートに関するCommonJS標準に従っています。

重要な注意点として、データベースからデータを読み取るほとんどのJavaScript関数は、データの完全なインスタンシエーションではなく、ValueIteratorのインスタンスを返します。これにより、評価環境が、必要に応じてレイジーかつ非同期でデータを読み込むことができます。ValueIteratorインターフェイスはECMAScript 6 イテレータインターフェイスを実装します。他のイテレータと同様に、ValueIteratorは、`for...of`ループを使用してループできます。その結果、アプリケーションで効率的に検索結果をストリーミングできます。

MarkLogicにおけるJavaScriptプログラミングの詳細については、『JavaScript Reference Guide』を参照してください。

モジュールと展開

XQuery、XSLT、およびJavaScriptのコードファイルは、ファイルシステム上またはデータベース内に配置できます。ファイルシステム上にコードを置くことは、シンプルであるというメリットがあります。コードを（.xqyまたは.sjsのスクリプト、または.xsltテンプレートとして）ファイルシステムディレクトリに配置するだけで完了です。これに対して、コードをデータベース内に格納する方法は、展開に便利です。クラスタ環境では、すべてのEノードが同じコードベースを使用していることを簡単に確認できます。各ファイルはデータベース内で1回のみ存在し、Eノード間で複製したり、ネットワークファイルシステムにホストしたりする必要がありません。また、複数ファイルへの大規模な変更を原子的な更新として展開できます。ファイルシステムに展開した場合、一部のリクエストが、コード更新を完了していない状態で認識する可能性があります。また、データベースでは、MarkLogicのセキュリティルールを使用して、更新を許可するユーザーを指定し、またシェルアカウントなしで安全なリモートアクセスを（WebDAV経由で）公開できます。

プログラマがXQuery、XSLT、またはJavaScriptのコードを明示的にコンパイルする必要はありません。ただし、MarkLogicでは、同じコードの実行の繰り返しを最適化するために、「モジュールキャッシュ」が維持されます。

出力オプション

MarkLogicでは、多数の異なる形式で出力を生成できます。

- XML：1ノードまたは一連のノードを出力できます
- JSON（Ajaxアプリケーションで一般的なJavaScript Object Notation形式）：XMLとJSONは容易に変換できます。MarkLogicには変換機能が組み込まれています
- HTML：HTMLはXML指向のXHTMLまたは標準のHTMLとして出力できます
- RSSおよびAtom：XMLの形式です
- PDF：PDFの生成を目的としたXSL-FOというXML形式があります
- Microsoft Office：Officeファイルには、Microsoft Office 2007より、XMLがネイティブ形式として採用されています。XMLファイルの読み取りと書き込みを直接できますが、複雑な形式をわかりやすくするためには、MarkLogicのOffice Toolkitsの使用を推奨します
- Adobe InDesignおよびQuarkXPress：Microsoft Officeと同様に、これらの出版形式はネイティブのXML形式を使用します

1層のWEB展開

MarkLogicには、SSLのサポートが組み込まれたネイティブのwebサーバーが含まれます。外部からのwebリクエストは、他のサーバーがPHP、JSP、またはASP.NETのスクリプトを呼び出すのと同じ方法で、XQuery、XSLT、またはJavaScriptのコードを呼び出すことができます。

MarkLogicには、スクリプトが一般的なwebタスクを処理するために使用する一連のビルトイン関数があります。タスクには、パラメータのフェッチ、ファイルアップロードの処理、リクエストヘッダーの読み取り、応答ヘッダーの書き込み、応答本体の書き込みに加えて、ユーザーセッションのトラッキング、URLの書き換え、およびエラーページのスクリプトなどの詳細が含まれます。

MarkLogicのビルトインHTTP/HTTPSサーバーには、完全なwebサイト（またはデータ指向のRESTエンドポイントの集合）を 1 層アプリケーションとして作成するオプションがあります。それは得策でしょうか。次のメリットはあります。

- アーキテクチャがシンプルになる：可動部が少なくなります
- インピーダンスミスマッチがない：バックエンドに構造化マークアップが保持されていて、フロントエンド用に構造化マークアップを生成する必要がある場合、構造化マークアップを処理するための言語が間にあると便利です。テーブルをオブジェクトにモデル化してから、すぐにオブジェクトをマークアップとしてモデル化する必要がないと、webプログラミングを大幅に簡素化できます
- コードの実行がデータに近くなる：web応答を生成するスクリプトは、通常のようにシステム間通信のオーバーヘッドを生じずに、バックエンドデータに対して小さなリクエストを頻繁に実行できます
- 整形式のエスケープしたHTML出力を容易に作成できます：XQueryとXSLTはXMLがネイティブ言語であるため、web出力を単なる文字列ではなく構造化ツリーとして認識する点が大きなメリットです。すべてが必然的に整形式になり、正しくエスケープされます。これに対して、PHPでは、開始タグと終了タグの配置を自分で管理し、ユーザー指定の文字列をHTML本体のテキストとして出力するときはエスケープ関数htmlspecialchars($str)を呼び出す必要があります。これを忘れると、サイト間スクリプトという脆弱性の可能性が生じます。XQueryまたはXSLTでは、何も忘れるものがありません

詳細については、『Application Programming in XQuery and XSLT』と『Server-Side JavaScript in MarkLogic』を参照してください。もちろん、アーキテクチャが 1 層である必要はありません。MarkLogicを多層アーキテクチャに展開することもできます。

マルチティア開発用のREST API

XQuery、XSLT、またはJavaScriptをサーバーでプログラミングしたくない場合はどうすればいいでしょうか。このような場合のために、MarkLogicには標準のREST webサービスインターフェイスが用意されています。REST APIは、MarkLogicに接続するアプリケーションに必要な主要機能を公開しています。これには、ドキュメントの挿入、取得、削除、クエリの実行とページング、スニペット、強調表示、ファセットの計算とサーバー管理が含まれます。

REST APIのエンドポイントは、HTTPやXDBCの場合と同じように構成します。その後、任意のクライアント言語（またはプログラム）からリモートweb呼び出しを行うことでエンドポイントと通信します。MarkLogicは、JavaとNode.jsのライブラリを提供およびサポートします。ライブラリはREST APIを囲むラッパーであり、基盤となるネットワーク呼び出しとデータマーシャルを開発者から隠します。他の言語は、RESTと直接通信するか、複数のコミュニティ開発のオープンソースライブラリの中から1つを使用できます。

REST APIのメリットとして、お気に入りのIDE、webフレームワーク、ユニットテストツール、およびその他を維持できる点が挙げられます。REST APIが提供する機能以外の機能が必要な場合のために、拡張性フレームワークが含まれています。この場合、開発者がカスタムのXQueryまたはJavaScriptを記述する必要があります。

詳細については、『REST Application Developer's Guide』を参照してください。

JAVAクライアントAPI

JavaクライアントAPIは、MarkLogic向けのデータ管理、クエリ、集約、アラートの機能を標準で装備しています。パフォーマンスと拡張性のベストプラクティスが、エンタープライズJava開発者が使い慣れ、使いこなせる包括的なAPIにカプセル化されています。APIには、Javaの充実したI/Oエコシステムを活用し、MarkLogicのネイティブなJSONやXMLと、またJavaのPOJO（Plain Old Java Object）と効率的に連携するためのフックがあります。

サーバーサイドでコードを実行してパフォーマンスを向上する必要があるアプリケーションのために、JavaクライアントAPIは、RMI的な拡張メカニズムならびにJavaからサーバーサイドライブラリを管理するAPIを提供します。このフックにより、データベース内のデータの近くで実行できるJavaScriptやXQueryコードをパッケージ化できます。

Java APIは、後述のJava XCCと共存します。Java XCCは、リモートまたはアドホックな XQueryまたはJavaScriptの呼び出しを実行するための低レベルのインターフェイスを提供します。

JavaクライアントAPIの使用方法については、MarkLogic Samplestackプロジェクトに示しています。Samplestackは3層リファレンスアーキテクチャであり、Javaをミドルティアとして使用しています（プロジェクトにはNode.jsクライアントAPIに基づくミドルティアも含まれる）。

JavaクライアントAPIは、MarkLogicがGitHubで公開開発中です。詳細については、『Java Application Developer's Guide』を参照してください。

NODE.JSクライアントAPI

Node.jsは、ミドルティアアプリケーションをJavaScriptで開発するためのオープンソースのランタイム環境です。Node.jsでは、拡張可能なwebアプリケーションの作成に役立つイベント駆動型のノンブロッキングI/Oモデルが使用されます。

Javaクライアントと同様に、Node.jsクライアントAPIは、MarkLogic REST APIを囲むラッパーです。NodeとMarkLogicの両方でベストプラクティスに従います。例えば、完全に非同期のI/Oが適用され、MarkLogicの充実したインデックス機能を活用する複雑なクエリを構築しやすくなっています。Node.jsクライアントは、外部Node.jsライブラリをインポートする標準の方法であるnpmを使用してプロジェクトに統合できます。

以下に、MarkLogicデータベースからドキュメントを読み込み、その内容をコンソールに書き込む、典型的なNode.jsクライアント処理を示します。

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

Node.jsの非同期プログラミングモデルは、データベースからの応答を待つ間のプログラムブロッキング実行ではなく、Node.jsプログラムが、応答が返されたときに実行するコールバック関数（前述のresult()の最初の引数として渡される）を定義することを意味します。これによってプログラム実行を続行でき、多くのデータベース処理を効率的に並列実行できます。

Node.jsクライアントを使用する利点：

- Node.jsに詳しく、MarkLogicを多層のJavaScript Webアプリケーションに統合したり、Node.jsコマンドラインを介してMarkLogicを操作したりしたいと考える開発者にとって有用である
- HTTP REST呼び出しに関連する複雑性がカプセル化されているため、開発者はMarkLogicの機能に集中できる。これには、ダイジェスト認証と接続プーリングのサポートが含まれる
- プロミス、オブジェクトストリーム、チャンクストリームの組み込みサポートが含まれる
- JSONドキュメントとJavaScriptオブジェクトの間でシリアライゼーションとデシリアライゼーションが自動的に実行される

Node.jsクライアントAPIも、GitHubでオープンに開発されています。詳細については、『Node.js Application Developer's Guide』を参照してください。

SEARCH APIとJSEARCH API

Search APIは、検索アプリケーションの開発を容易にすることを目的としたコードレベルのXQueryライブラリです。REST APIによって使用され、検索、検索構文解析、拡張とカスタマイズが可能な検索文法、ファセット化、スニペット化、検索語補完などの検索アプリケーション機能が1つのAPIにまとめられています。`cts:query`構造に詳しい開発者も、ユーザーが入力した文字列を`cts:query`階層に簡単に変換できるので便利です。詳細については、Search APIのドキュメント（Search Developer's Guide / 2.0 search API：概要と使い方）を参照してください。

JSearch APIは、JavaScriptの開発者が複雑な検索をMarkLogicで簡単に実行できるようにするためのJavaScriptライブラリです。JSearchには、Search APIおよびcts:関数で利用できるものと同じ機能が多数含まれていますが、結果をJavaScriptオブジェクトとしても返します。開発者はメソッドの呼び出しを連鎖させることができ、スニペットの生成とファセット化のための便利な手法を手に入れることができます。詳細については、Creating JavaScript Search Applicationsを参照してください。

トリプルのクエリのためのSPARQL

MarkLogicでセマンティックデータにアクセスするには、SPARQLを使用します。これは、RDFトリプルに対してクエリを実行するための標準言語です（セマンティック機能の詳細については、「セマンティック」のセクションを参照）。SPARQLによりRDFトリプルのパターンを記述し、そのパターンに一致するトリプルを返すことができます。そのシンタックスはSQLのものと似ています。例えば、以下のSPARQLクエリは、パリで生まれた人々のリストを返します。

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

MarkLogicでは、SPARQLを使用してさまざまな方法でクエリを実行できます。

- Query Consoleは、XQueryとJavaScriptを評価できるのと同様、SPARQLを直接評価できます
- SPARQLクエリを引数として受け入れる、サーバー側のXQueryおよびJavaScriptセマンティック関数を使用できます
- SPARQLクエリをセマンティックエンドポイントに送信し、REST APIを介してトリプルデータにアクセスできます

MarkLogicは、SPARQL Update演算もサポートしています。SPARQL Updateとは、RDFトリプルを挿入および削除することによってセマンティックデータ（セマンティックグラフ）のコレクションを管理できるようにする、独立した標準です。

SPARQLおよびMarkLogicの詳細については、『Semantics Developer's Guide』を参照してください。

JAVAアクセス用のXDBC/XCC

XDBCワイヤプロトコルは、XCCというオープンソースのJavaクライアントライブラリを使用して、JavaからMarkLogicにプログラムによってアクセスできるようにします。XCCクライアントライブラリは、接続およびセッションの管理、コード呼び出しの実行、コンテンツの読み込み、結果ストリームの引き出し、MarkLogicとのやり取り全般を行うためのJavaオブジェクトを提供します。JDBCやODBCに詳しい開発者なら、違和感なく使用できます。

XDBCワイヤプロトコルはデフォルトでは暗号化されていませんが、SSLをXDBCの上に重ねて、信頼されていないネットワーク全体でプロトコルのセキュリティを確保できます。

MarkLogic 7より、クライアントで特別な`xcc.httpcompliant`設定オプションを有効にすることで、HTTPを介してXDBCを実行する機能が追加されています。これにより、XDBCトラフィックが標準的なWebロードバランサーを通って流れるようになります。このロードバランサーは、Eノードの状態を監視できるだけでなく、セッションアフィニティを実行できます（同じクライアントから繰り返されるリクエストが同じEノードに渡される—トランザクション内のすべてのリクエストが同じEノードを使用しなければならない、マルチステートメントトランザクションの重要な要件）。HTTPを介してXDBCを実行することの唯一の欠点は、HTTPではサーバーがクライアントにコールバックしてXML外部エンティティを解決できないことです。

詳細については、『XCC Developer's Guide』を参照してください。

WEBDAV：リモートファイルシステムアクセス

WebDAVは、MarkLogicとのインターフェイスのためのもう1つのオプションです。WebDAVは、ファイルの読み込みと書き込みのために幅広く採用されているワイヤプロトコルです。MicrosoftのSMB（Sambaで実装）に少し似ていますが、オープン標準です。MarkLogicでWebDAVポートを開き、WebDAVクライアントで接続すると、MarkLogicデータベースをファイルシステムのように表示したり操作したりして、ファイルのプルとプッシュを行うことができます。

WebDAVは、ドラッグアンドドロップによるドキュメントの読み込みや、MarkLogicからのコンテンツの一括コピーに最適です。主要なオペレーティングシステムすべてにWebDAVクライアントが組み込まれていますが、多くの場合、サードパーティ

製クライアントの方が堅牢です。WebDAVは、WebDAVクライアントが想定している数千のドキュメントの管理は適切にできますが、それよりも大きなデータセットでは問題が発生します。

開発者によっては、WebDAVを使用して、データベースから展開されたXQuery、XSLT、またはJavaScriptファイルを管理することもあります。多くのコードエディタがWebDAVに対応しており、そのコードが使用されたデータベースをマウントすることで、ローカルエディタを使用してリモートMarkLogicシステムでホスティングされるコードを簡単に作成できます。

WebDAVには、XQuery、XSLT、またはJavaScriptコードを実行するメカニズムは含まれていません。単にファイル転送用です。詳細については、WebDAV Serversを参照してください。

ビジネスインテリジェンスのためのSQL/ODBCアクセス

MarkLogicにSQL/ODBCインターフェイスがあると聞いて、驚かれるかもしれません。SQLの「S」は「Structured（構造化）」を表します。ドキュメント指向のデータベースで、一体それがどう機能するのでしょうか。そして、なぜ必要なのでしょうか。

SQLシステムの目的は、データベースがリレーショナルモデルに従うことを求める、IBM Cognos、Tableau、MicroStrategyなどのビジネスインテリジェンス（BI）ツールの実行に適したデータベースの読み取り専用ビューを提供することです。MarkLogic内でアクセスされる「テーブル」は完全に合成です。すべてがドキュメントのまま残ります。テーブルは、レンジインデックス上に配置される「ビュー」として構築されます。例えば、データベースに金融取引を保存し、日付、取引先、価値要素に関するレンジインデックスを設定した場合、これらの値が列になります。ドキュメントのデータ全体を表す列（名称はビューの名称と一致）は特別です。この特別な列を使用して、ドキュメントのデータ全体に対する制約を、SQL MATCH演算子を介して指定することができます。これはちょっとした機能に思えますが、大きな意味があります。補助ドキュメントの任意の部分（SQLビューに公開されていない部分でも）に基づいて、取得される行を制限できるからです。

すべてがメモリから実行されます。このシステムは、行ではなく列がまとめて保存されるという点で、列指向データベースに似ているところがあります。また、クエリの一部として、n-wayコオカレンスを使用してタプルが作成されます。SQLバージョンはSQL92で、SQLiteで実装され、SET、SHOW、DESCRIBEの各ステートメントが追加されています。

MarkLogicは、ODBCを介してSQLを公開します。ODBCは、リレーショナルデータベースにアクセスするための標準のCベースAPIです（テスト目的では、XQuery `xdmp:sql()` 関数またはMLSQLコマンドラインツールを使用できます）。ODBCドライバは、オープンソースのPostgreSQL ODBCドライバをベースとしています。

ドキュメントのバックエンドに加えてリレーショナルビューのデータモデリングを行う際は、注意が必要です。ドキュメントが1つのレンジインデックスに対して複数の値を持っている場合（通常のリレーショナルモデリングでは不可能）、複数の行にわたる外積として表す必要があります。レンジインデックスの値が欠けているか無効である場合、その処理方法に関する設定オプションがあります。ビューが列を「nullable」と定義した場合、指定された列の値は「null」として表示されます。「nullable」でない場合、ドキュメントは必要な制約を満たさず、行を一切作成しません。

詳細については、『SQL Data Modeling Guide』を参照してください。

リモートコーディング向けQUERY CONSOLE

それ自体はプロトコルではありませんが、MarkLogicへの直接のアクセスを希望するプログラマによって幅広く使用されているのが、webベースコード実行環境のQuery Consoleです。サーバーに付属するwebアプリケーションであり、基本的にはXQueryおよびJavaScriptのスクリプトのセットに過ぎません。アクセスすると（もちろんパスワード保護されています）、webブラウザのテキスト領域からアドホックコードを実行できます。JavaScript、SPARQL、SQL、およびXQueryで記述されたスクリプトを実行できます。優れた管理およびコーディングツールです。

Query Consoleには、シンタックス強調表示、ワークスペースにグループ化可能な複数のオープンクエリ、履歴追跡、サーバーへの状態の保存、整えられたエラーメッセージ、サーバー上のあらゆるデータベース間を切り替える機能などが含まれており、XML、HTML、プレーンテキストの出力オプションがあります。また、コード内のスロースポットを特定するために役立つプロファイラ（MarkLogicのプロファイラAPIのwebフロントエンド）と、データベース内のファイルを表示するエクスプローラも含まれています。こちらを確認してください。

```
http://yourserver:8000/qconsole
```

詳細については、『Query Console User Guide』を参照してください。

第3章

詳細トピック

高度なテキスト処理

本書の前半では、MarkLogicのユニバーサルインデックスを紹介し、MarkLogicがタームリストを使用して語と語句、そして構造をどのようにインデックス化するかについて説明しました。そのセクションでは、テキストのインデックス化に関するMarkLogicの機能について簡単に説明しただけでした。このセクションでは、その部分についてもう少し深く掘り下げます。

これらのインデックスは、すでに学習したものと同じように機能します。新しい各インデックスオプションは、新しいタイプのタームリストを追跡して、インデックス解決をより効率的に、`xdmp:estimate()`呼び出しをより正確にするようMarkLogicに指示します。

テキストの区別のオプション

テキストをクエリするとき、大文字と小文字を区別するかどうかの指定が必要な場合があります。例えば、「Polish（ポーランド人）」と「polish（磨いて艶を出す）」は意味が異なります[1]。クエリの一部としてこれを指定するのは簡単です。「case-sensitive」または「case-insensitive」オプションを各クエリタームに渡すだけです[2]。問題は、MarkLogicがこれらのクエリをどのように解決するかです。

デフォルトでは、MarkLogicは大文字小文字の区別が必要ないタームリストエントリのみを維持します（すべて小文字と考えてください）。大文字小文字の区別が必要なタームでクエリを実行すると、MarkLogicはインデックスを利用して大文字小文字の区別が必要ない一致を探し、フィルタリングして大文字小文字の区別が必要な真の一致を特定します。大文字小文字の区別が必要な検索がめったにない場合は問題ありませんが、よく行われる場合は、fast case sensitive searchesインデックスオプションを有効にして効率を高めることができます。このオプションは、大文

1　このように、先頭を大文字にすると意味が異なる語を「キャピトニム」と呼びます。ほかにも、「March」と「march」、「Josh」と「josh」などの例があります。

2　「case-sensitive」または「case-insensitive」を指定しない場合、興味深いことに、MarkLogicはクエリタームの大文字小文字を確認します。すべて小文字の場合は、大文字小文字の区別は重要でないとみなし、「case-insensitive」として扱います。クエリタームに1つでも大文字が含まれている場合は、大文字小文字の区別が必要であるとみなし、「case-sensitive」として扱います。

字小文字の区別が必要なタームリストエントリを区別の必要がないものと一緒に維持するようMarkLogicに指示します。このインデックスを有効にすると、大文字小文字の区別が必要でないタームは大文字小文字の区別が必要でないタームリストエントリを使用し、大文字小文字の区別が必要なタームは大文字小文字の区別が必要なタームリストエントリを使用し、すべての結果がインデックスから迅速かつ正確に解決されるようになります。

fast diacritic sensitive searchesオプションは、発音符号の有無に関して同様に機能します（発音符号とは、ウムラウト記号やアクセント記号のように、文字に追加される付随的記号です）。デフォルトでは、「resume」を検索すると、「résumé」もヒットします。通常はそれで問題ありませんが、必ずしもそうとは限りません。発音符号を区別するインデックスをオンにすることで、発音符号による区別が必要なタームリストエントリと発音符号による区別が必要でないタームリストエントリの両方を維持するようにMarkLogicに指示して、インデックスから発音符号による区別が必要な一致を解決できます[3]。

ステムインデックス

ステミング（語幹処理）は、MarkLogicが最適化されたインデックスオプションを提供するまた別のケースです。活用語（あるいは派生語）を原形に戻すプロセスです。「run」をクエリすると、「runs」、「running」、「ran」などもヒットします。これらすべての語に同じステムルートの「run」が含まれているためです。ステム検索を実行すると、返される結果セットの範囲が広がるため、再現率が高まります。多くの場合はそれが望ましいのですが、そうではないケースもあります。例えば、固有名詞やメタデータタグそのものを検索している場合などです。

MarkLogicには、ステムインデックスを維持する、非ステムインデックスを維持する、両方を維持するというオプションがあります。これらのインデックスは、ほかのテキストインデックスすべてに影響するという意味で、マスターインデックス設定のように機能します。例えば、[fast phrase searches]のオプションは、マスターインデックス設定を参照して、語句を語のペア、ステムのペア、または両方のいずれとしてインデックス化すべきかを判断します。

バージョン8より前のMarkLogicでは、デフォルトでstemmed searchesオプションが有効になっており、word searchesが無効のままでした。通常はそれで問題ありませんが、想定外の事態が発生することもあります。例えば、ステミングが不要なメタデータフィールドでタームを検索する場合などです。そのような場合は、word searchesインデックスを有効にして、検索クエリ制約を構築する際に「`unstemmed`」をオプションとして渡します。バージョン9では設定が逆になり、デフォルトでstemmed searchesが無効に、word searchesが有効になっています。

3　MarkLogicは、発音符号による区別があるかどうかに関して、Unicodeの定義に従います。これは、特定の文字をインデックス化する際に重要になることがあります。例えば、「ł (U+0142, "LATIN SMALL LETTER L WITH STROKE")」には、Unicodeの分解マッピングがありません。結果的に、その文字については発音符号による区別があるとはみなされません。

MarkLogicは、言語固有のステミングライブラリを使用して、各語の語幹（複数の場合もあります）を特定します。「chat」のように、英語とフランス語で意味が違い、語幹も異なる語もあるため、言語固有のものでなければなりません。また、言語固有のカスタム辞書を作成して、ステミングの動作を変更することも可能です。詳細については、『Search Developer's Guide』を参照してください[4]。

MarkLogicは、数百もの言語に対して基本言語サポートを提供し、14言語（英語、フランス語、イタリア語、ドイツ語、スペイン語、アラビア語、ペルシア語/ファルシ語、中国語（簡体字および繁体字）、日本語、韓国語、ロシア語、オランダ語、ポルトガル語、ノルウェー語）については、ステミング、トークン化（テキストを語に分解）、コレーション（並べ替え）といった高度なサポートを提供しています。Unicode Level 5.2-0をサポートしているため、MarkLogicは、Unicodeで表されるあらゆる言語に対して非ステミング検索を処理、保存、実行できます。`xml:lang`属性、charset推論、文字シーケンスヒューリスティック、データベースのデフォルト設定をこの順で使用して、テキスト言語を特定します。

ステムインデックスでは、ステミング結果をユニバーサルインデックスに埋め込むという、パフォーマンスを最大限に高めるためのシンプルな手法を利用しています。MarkLogicは、ステムインデックス内の語のすべての語幹をステムルートとして扱います[5]。テキスト内に表示される、ルートではなかった語はすべて、インデックス付けされる前にルートに簡略化されます。つまり、「run」も「ran」も、ステムルート「run」に基づいて単一のタームリストで管理されることになります。クエリ時点では、「ran」をステム検索しても「run」というルートに簡略化され、そのルートが適切なタームリストを特定するためのルックアップキーとして使用されます。タームリストは、語幹の任意のバージョンでドキュメントのリストを事前計算しています。インデックスはステムルートのみを扱うため、これが語をその他のステム形式に一致させるための効率的な手法となります[6]。

word searchesとstemmed searchesの両方が有効になっている場合、各語にそれぞれエンコードの異なるエントリが追加されます。一方は実際に表示されるもので、もう一方はステムルート用です。検索クエリオプションは、使用するタームリストを制御します。

4　ドメイン固有の技術用語は、デフォルトのステミングライブラリに含まれていないことがほとんどです。例えば、「servlets」という語の語幹は「servlet」であるべきですが、デフォルトではそうなっていません。

5　MarkLogicは、語幹処理されたバージョンを同じ品詞として維持しない派生ステミングではなく、これを維持する屈折ステミングを実行します。例えば、「functioning（動詞）」は語幹「function（動詞）」として格納されますが、「functional（形容詞）」はそのまま変わらず格納されます。

6　`cts:stem`("running", "en")を使用して、語のステムルートを確認してください。2番目の引数は言語コードです。

ステミングオプション

ステミングオプションは少数しかありません。[basic]は各語の最短の語幹をインデックス付けし、[advanced]は各語のすべての語幹をインデックス付けします。また、[decompounding]は、大きな複合語の中の小さな複合語とともに、すべての語幹をインデックス付けします。連続的な各ステミングレベルによって、語検索の再現率が向上しますが、インデックスのサイズが大きくなります。

[advanced]オプションは英語では便利なことがあります。例えば、「running」という語は、動詞としても(語幹「run」)名詞としても(語幹「running」)使用できます。また、多義性(1つの語に複数の意味があること)が一般的であるその他の言語では、さらに便利です。例えば、フランス語では、「bois」は「woods(木)」を意味する名詞(語幹「bois」)であると同時に、「drinks(飲む)」を意味する動詞(語幹「boire」)でもあります。

[decompounding]オプションは特定の言語(ドイツ語、オランダ語、ノルウェー語、日本語、スウェーデン語など)にしか適用されませんが、これらの言語では必須です。既存の名詞のシーケンスから新しい名詞を作成し、これらを全て利用して検索を実行します。もしこのオプションを有効にしなければ、乏しい結果しか得られないでしょう。

トークン化

MarkLogicは、テキストを文字のシーケンスとしてだけでなく、トークンのシーケンスとしてもみなします。各トークンは「word」、「space」、「punctuation」などの型分類を取得します。トークン化のロジックを確認するには、`cts:tokenize()`を使用します。

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

これによって、次が生成されます。

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

`cts`:名前空間内の検索機能は(人間の場合と同様に)、文字ではなくトークンに対して機能します。これにより、「230」を検索しても「2300」がヒットしなくなり、「foo」を検索しても「food」がヒットしなくなります。スペースと句読点のトークンは語を分割するために使用されますが、テキストのインデックス付けでは無視されます。このため、メールアドレス「foo@bar.com」を検索すると、「foo bar com」を語句検索した場合と同様に、インデックスから解決されます。この場合、クエリで空白/句読点を区別するかどうかが指定されていれば、スペースと句読点のトークンがフィルタリング時に役立ちます。同様に、検索で句読点を区別しないように指定されていても「don't」は「dont」と一致しません。なぜなら、最初は3つのトークンで、2番目は1つのトークンだからです[7]。

7 トークン化は、ステミングと同様に値のクエリにも適用されます。ステミングを有効にすると、次が「true」を返します。`cts:contains(<text>I do run</text>, cts:element-value-query(xs:QName("text"),"me done ran")`

トークン化は言語を認識し、言語に依存します。英語のネイティブスピーカーは、語のトークンをスペースと句読点で分割しますが、ほかの言語の場合はさらに複雑です。例えば、日本語では、語間にスペースがないことがほとんどです。しかし、MarkLogicは日本語のテキストを`cts:word`トークンの適切なシーケンスに変換できます。これは、日本語に対応したトークナイザーのおかげです。また、日本語に対応したステマーのおかげで、`cts:word`トークンをステミングすることもできます。MarkLogicと他製品の大きな差別化要因は、このようにテキストを深く理解できる点です。

カスタムトークン化

MarkLogic 7以降、特定の文字のトークン化を優先させて、異なる動作を行わせることができます。あらゆる文字を「word」、「space」、「punctuation」、「remove」、「symbol」の各文字として強制トークン化できます。

「remove」とマークされた文字は、トークン化時に無視され、その文字がテキスト内に一度も現れていないかのように読み取られます。電話番号の場合、ハイフン、スペース、丸括弧、プラス記号などを「remove」できます。このようにして、あらゆる電話番号は、ドキュメント内にどのように記載されていようと、シンプルで一致しやすい単一の`cts:word`にトークン化されます。もちろん、スペースのように一般的な文字をすべてのテキストから削除すると大変なことになるため、カスタムトークン化を設定するときは、電話番号フィールドのようなドキュメントの一部にのみ適用する必要があります。フィールドについては、「フィールド」セクションを参照してください。

ところで「symbol」の役割とは何でしょうか。記号であることによって、その文字が特別なものになります。句読点はテキストインデックスに表示されないため、記号は単なる句読点ではありません。記号はテキストインデックスに表示され、1つの文字からなる語のように機能します。インデックスで使用できますが、その近くの語から分離したままです。

記号の価値を理解するために、Twitterデータの処理方法を例に挙げます。Twitterのハッシュタグによるトピック分類では、#（ハッシュ記号）を先頭に付けます。カスタムトークン化ルールがない場合、このハッシュ記号は句読点として扱われ、インデックス付けでは無視されます。それで構いませんが、「#nasa」を検索した場合、MarkLogicはインデックスを使用して「nasa」のオカレンスを検索してから、結果をフィルタリングしてハッシュ記号のある「#nasa」を特定します。これは効率的ではありません。「#」を語としての文字としたらどうなるでしょうか。その場合「#nasa」が望みどおりに完璧に解決されますが、残念なことに、単純に「nasa」を検索してもハッシュタグがヒットしなくなります。語としての文字が、修飾対象の語に内在するものとなり、このケースではあまり適切ではありません。

ここで本当に必要なのは「#」を記号とすることです。その場合、テキスト「#nasa」が`(cts:symbol("#"),cts:word("nasa"))`にトークン化されます[8]。「nasa」を検索すると一致し、「#nasa」を検索するとインデックスから完璧に解決されます。

詳細と例については、『Search Developer's Guide』の「28.0 カスタムトークン化」セクションを参照してください。

ワイルドカード

MarkLogicでは、ワイルドカード文字を使用してトークンのレベルより細かなテキスト制約を指定できます。アスタリスク(*)は任意の数の文字を表し、疑問符(?)は1文字を表します。例えば、`cts:word-query("he*")`は「he」で始まるすべての語に一致し、`cts:word-query("he?")`は3文字の語にしか一致しません。ワイルドカード文字は、どこにでも制限なしで置くことができます。例えば、検索パターン*ark??g*cというものがあっても、まったく問題ありません。

複数のインデックスにより、ワイルドカードクエリの速度が改善します。最も単純なのは、trailing wildcard searchesインデックスです。これを有効にすると、語の先頭の3文字以上の全シーケンスのタームリストが作成されます。例えば「MarkLogic」がソーステキストに表示される場合、このインデックスは「mar*」、「mark*」、「markl*」などのタームリストエントリを作成します(大文字小文字を区別しないか、大文字小文字を区別するエントリが追加されるかは、別の大文字小文字の区別に関するインデックス設定によって決まります)。このトレイリングワイルドカードインデックスは、最後が1つのアスタリスクで終わる検索パターンという一般的なケースで、完璧な答えを迅速に提供します。関連インデックス設定は、trailing wildcard word positionsとfast element trailing wildcard searchesです。これらは、一致したテキストの位置と要素の場所を追跡します。

より汎用性が高いのは、three character searchesインデックス(およびその仲間であるthree character word positionsとfast element character searches)です。これらのインデックスはソーステキストに表示されるすべての3文字のシーケンスのタームリストエントリを作成するため、3文字以上の文字からなるシーケンスを含む検索パターンがスピードアップします。前述したワイルドカードクエリ、*ark??g*cには3文字のシーケンス(「ark」)があり、これを使用して結果を取り出すことができます。また、このインデックスは、あらゆる語および値の最初と最後にある3文字のシーケンスもトラッキングします。このため、`cts:word-query("book*ark")`は、単語の先頭に「boo」があり、最後に「ark」があり、どこかに「ook」があるドキュメントのみを取り出します。また、存在すれば、book*トレイリングワールドカードインデックスを使用することもできます。ポジションが有効になっている場合は、それによって一致するシーケンスが同じ語内にあることを保証することができます。

8 `cts:tokenize()`関数は、言語を2番目の引数として、フィールド名を3番目として受け入れます。

これが最善の方法でしょうか。いいえ、違います。語レキシコンこそ、ワイルドカードクエリを解決するための優れた手法です。語レキシコンは、データベース内に表示されるすべての語をトラッキングすることを覚えておいてください。ワイルドカードパターンを、語レキシコンの既知の語のリストと比較することで[9]、MarkLogicはパターンに一致するすべての語をデータベースに列挙するかクエリを構築できます。レキシコンを完全に拡張することで、インデックスから正確な結果を得られます（正確なファセットが必要な場合は便利です）が、タームリストが長くなり、解決に必要なタームリストルックアップの回数が多くなります。

完全に拡張するのではなく、レキシコンを使用してパターンに一致するデータベース内のすべての語の3文字のプレフィックスとポストフィックスのリストを推測し、これらのパターンからorクエリを構築することもできます。一致する語の大部分はプレフィックスとポストフィックスの文字シーケンスを共有する傾向があるため、完全な拡張よりもわずかに正確性は劣るものの、こちらの方が効率的です。この手法では、three character searchesインデックスを有効にする必要があります。

デフォルトで、MarkLogicはインデックス設定に基づくヒューリスティックとデータ統計を使用して、ワイルドカードによる各ワードクエリの解決方法を判断します。その方法とは、完全なレキシコン拡張、プレフィックスとポストフィックスの拡張、拡張なし（文字シーケンスのみを使用する）のいずれかです。ワードクエリの構築時にワイルドカードを解決する方法に関して、オプションでサーバーにヒントを与えることができます[10]。

パターンを含むドキュメントのマッチングではなく、前述の拡張のように、パターンに一致するデータベース内の語や値そのものを抽出したい場合は、`cts:word-match()`や`cts:value-match()`などの関数を使用して自分で対処できます。これらは、レキシコンの値に対して機能します。

詳細については、『Search Developer's Guide』の「20.0 ワイルドカード検索の概要と使い方」セクションを参照してください。

関連度に基づくスコアリング

結果の関連度スコアについては前述しましたが、テキストベースの関連度がどのように機能するかについては説明しなかったので、ここで説明しましょう。関連度アルゴリズムの数式は、以下の公式です。

log(term frequency) * (inverse document frequency)

9　ワイルドカードはコードポイント（つまり、各文字）に反して機能するため、語レキシコンをコードポイントコレーションで設定するのが最適です。

10　背後でどのようにワイルドカード検索が実行されたかを調べるには、`xdmp:plan()`でクエリ式をチェックします。ヒューリスティックはデータサイズに依存するため、なぜワイルドワードクエリが特定の方法で処理されたか、また自分の用途でそれが問題ないかどうかを確認できると便利です。

term frequency因数は、ドキュメント内の語の何パーセントがターゲットの語と一致するかを示します。term frequencyが高いほど関連度が高くなりますが、対数的なので、frequencyが高くなるほど効果は下がります。これにinverse document frequencyを乗算すると（document frequencyで除算した場合と同じ）、その語が通常はデータベース全体でどの程度登場するかが正規化されるため、出現頻度の低い語の影響が大きくなります。

MarkLogicではこのアルゴリズムを「`score-logtfidf`」と呼びます。これは、`cts:search()`式のデフォルトのオプションです。その他のオプションには、「`score-logtf`」（inverse document frequencyの計算をしない）、「`score-simple`」（頻度に関係なく、一致するタームの数を単純にカウントする）、`score-random`」（ランダムな順序を生成し、サンプリングを行う場合に便利）、「`score-zero`」（スコア計算を行わず、順番が重要でない場合は結果を少し速く返す）があります。

MarkLogicはterm frequency情報をインデックス内に維持し、検索中にそのデータを使用して関連度スコアを迅速に計算します。各タームリストには、ドキュメントID、場合によっては場所（近接クエリや語句検索などの場合）、さらにterm frequencyデータのリストがあると想定できます。すべての数値は、ディスク上にできる限り小さく保存できるように、デルタコーディングを使用して記述されています。MarkLogicは、タームリストの積集合を求める一方で、term frequencyデータと取得されたdocument frequencyデータに基づいて、どの結果が最高のスコアになるかを特定するためのちょっとした計算も行います。

すべてのリーフノード`cts:query`オブジェクトには、クエリの該当部分の重要性を示す重みパラメーターが含まれています。重みが大きいほど、その重要性も高まります。重みがゼロの場合は、満たす必要があるものの、スコアリングに関してはまったく重要ではありません。重みを使用することで、あるタームをほかのタームよりも重くしたり、ある配置（タイトル内など）をほかの配置（本文内など）よりも重くしたりできます。

タームの重みは、`xs:double`として指定されます。値の範囲は-64.0～+64.0、デフォルトで+1.0です。その効果は多大であるため、小さな変更でも大きな影響力を持ちます。+/-16.0を超える重みは「超重み付け」とみなされます。なぜなら、その影響力が強すぎて、検索結果を支配してしまうからです。これは、クエリの該当部分に一致する結果を強制的に上位に持ってきたい場合に便利です。タームの重みが負である場合、そのタームを含むドキュメントのスコアを下げることで、そのタームを含まないドキュメントのヒット率が高まります。スパムに似た結果や関連性の低い結果に関連付けられた語や語句については、タームの重みを負にするとよいでしょう。

cts:queryオブジェクト

MarkLogicは、cts:queryオブジェクトを使用して検索の制約を表します。これらのオブジェクトは、コンストラクタ関数を介して構築されます。それらの名前は、cts:word-query、cts:element-attribute-value-query、cts:element-range-queryなどです。一部の特殊なクエリタイプは、ほかのクエリを階層に配置することを目的としたアグリゲーターです。それらの名前は、cts:and-query、cts:or-query、cts:and-not-query、cts:boost-queryなどです。リーフノード（つまり、集約的でない）cts:queryタイプは、内部のインデックスと1対1でマッチすることに気付くでしょう。これにより、MarkLogicは簡単にそのインデックスを利用してcts:query階層を実行できます。

もちろん、ユーザーはcts:queryオブジェクトの観点から考えません。ユーザーは、GUIをクリックしたり、平文の検索文字列を入力したがります。GUI設定と検索文字列があり、実際の実行ではcts:query階層にそれらを変換できることが、アプリケーションの要件です。Search APIのようなツールにより、これが簡単で設定可能になります。

シンプルな1語のユーザークエリですら、一致が許容される場所をリストするために、内部的にcts:or-queryに拡張されることがあります。また、これを暗黙的に外部のcts:and-queryに配置し、GUI設定に基づいて表示可能な範囲を制御することができます。数千ものリーフノードに拡張する階層は多くのお客様によくある状況で、効率的に実行されます。

cts:queryを構造化しプログラム可能にすることで（SQLの場合のように、扱いにくい文字列の連結から構築されるのに反して）、MarkLogicは大きな柔軟性とパワーを獲得します。例えば、シソーラス関数thsr:expand()は、提供されたcts:query階層を進み、拡張が必要なリーフノードクエリすべてを特定し、同義語クエリを持つcts:or-queryにそれぞれを置き換えることで動作します。

MarkLogic内のドキュメントには、Google PageRankに似た内在的なクオリティがあります。各ドキュメントのクオリティが、そのドキュメント用に計算されたスコアに追加されます。クオリティが高いドキュメントの方が検索結果の上位になり、クオリティが低いドキュメントは抑制されます。クオリティはプログラム的に設定され、任意の項目をベースとすることができます。例えばMarkMail.orgでは、コードのチェックインメッセージに非常に低いクオリティを与えているため、ほかの通常メッセージが一致しない場合のみ結果に表示されます。

検索には、クオリティにどの程度の重要性を与えるかを示すクオリティの重み付けパラメータが含まれます。これは、浮動小数点数にクオリティを乗算して計算します。値「0」は、そのクエリのクオリティ値を完全に無視することを意味します。ドキュメントに幅広いクオリティスコアを設定し（最大1,000など）、少しずつクオリティの重み付けを行い、クオリティとタームに基づいて最適なスコアの組み合わせになるまで調整するとよいでしょう。幅広い範囲を設定することで、精度を維持できます。

管理画面を使用して診断フラグ「relevance」をオンにすると、サーバーが行うスコアリングの計算を監視することができます。このフラグは計算をサーバーのエラーログファイルにダンプします。プログラムによるアクセスの場合、「relevance-trace」をcts:search()にオプションとして渡し、cts:relevance-info()を使用してXML形式のトレースログを抽出します。トレースはパフォーマンスに影響を与えるため、実稼働にはお勧めできません。

ストップワード

ストップワードとは、あまりにも一般的でそれ自体ではあまり意味がないため、検索のインデックス付けを目的とする場合は無視できる語のことです。MarkLogicではストップワードは不要であり、使用しません。つまり、ストップワードのみからなる「to be or not to be」という語句や、ストップワードを1語含む「Vitamin A」のような語句も検索できます。

ただし、MarkLogicは、特定のタームについて、位置リストをどの程度の大きさにできるかを制限します。位置リストは、タームが表示されるドキュメント内の場所をトラッキングし、近接クエリと長い語句の解決をサポートします。最大サイズは、positions list max sizeと呼ばれる構成可能なデータベース設定であり、通常は数百メガバイトです。タームが何度も表示され、その位置リストがこの制限を超えると、そのタームの位置データが削除され、クエリの解決では使用されなくなります（なぜなら、非常に非効率だからです）。タームがこの制限を超えているかどうかを検出するには、`StopKeySet`という名前のファイルを確認します。このファイルは、オンディスクスタンドの1つにゼロ以外の長さを持っています。

フィールド

MarkLogicを使用する管理者は、さまざまなインデックスの有効と無効を切り替えることができます。インデックスには、stemmed searches、word 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、one-character searchesなど、さまざまなものがあります。各インデックスは、特定のタイプのクエリのパフォーマンスを改善しますが、ディスク使用量が増えて読み込み時間が長くなるという犠牲を伴います。

状況によっては、ドキュメントの一部で特定のインデックスを有効にし、それ以外の場所では有効にしないという方法が理にかなっていることがあります。例えば、タイトル、作者、要約にはワイルドカードインデックスが適切ですが（つまり、オーバーヘッドを正当化できる）、より長い本文全体についてはそうではない場合があります。

フィールドを使用すると、ドキュメントの異なるサブセットに異なるインデックス設定を定義できます。各フィールドは、一意の名前、フィールドに含める要素および属性のリスト、除外するもののリストをもちます。例えば、フィールドにタイトル、作者、要約は含め、要約内のすべての脚注は除くということができます。または、フィールドにすべてのドキュメントコンテンツを含めるけれど、`<encoded-blob>`要素だけはフィールドのインデックスから削除することも可能です。リクエスト時に、`cts:field-word-query()`のようなフィールド認識関数を使用して、フィールドに対するクエリ制約を表現することができます。

より効率的なインデックス化に加えて、ドキュメントのどの部分を検索するかの定義を、XQueryまたはJavaScriptでコーディングされたものから、管理者が宣言したものへと移すオプションも、フィールドによって提供されます。例えば、AtomおよびRSSフィードを、異なるスキーマでクエリしているとしましょう。スキーマの複数のタイプに渡ってクエリを実行するように、このコードを記述できます。

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

非常にシンプルですが、明日新しいスキーマが登場すれば、コードの変更が必要になります。代わりに、フィールドでフィード項目とみなす要素のセットを定義し、そのフィールドに対してクエリを実行することができます。フィールドの定義は変わりますが、コードはそのままで利用できます。

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

フィールド定義の一環として、関連する各要素または属性に重みを追加し、それぞれを検索の一致の関連性を高めたり、低くしたりすることが可能です。これにより、また別のパフォーマンス上のメリットを得ることができます。通常、作者よりもタイトル、要約よりも作者に重み付けをしたい場合、クエリ時に重みを付けます。例えば、次のようになります。

```
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)
))
```

これらの重みをフィールド定義に含めると、インデックスに埋め込まれます。

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

これは、実行時には計算があまり行われないことを意味します。欠点は、フィールドの重み付けを調整するには、新しい値をインデックスに埋め込むために、管理者が変更を行わなければならないこと、そして、バックグラウンドでのコンテンツの再インデックス化が必要であることです。ほとんどの場合、適切な重み付けに満足できるまで、アドホックの重み付けを試し、その重み付けをフィールド定義に埋め込む方法が最善です。

フィールドに関する追加情報

フィールドは、複雑なXMLからインデックス化された特異値を作成するためにも使用できます。以下のように構造化されたXMLがあるとします。

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

「fullname」というフィールドを、子要素すべてを含むように定義された<name>に対して作成できます。その特異値は、文字列としての人間のフルネームとなります。「simplename」という別のフィールドを<name>に対して作成できますが、<mname>は除きます。その特異値は、文字列としての単なる姓および名となります。cts:field-value-query()を使用すると、これらの計算値のいずれかに対して最適化されたクエリを実行できます。これは、クエリの一部として値を計算する場合よりずっと高速です。ユニバーサルインデックス内で、各フィールドは各値について1つずつ、タームリストを追加するだけです。

前述したように、フィールドに対してレンジインデックスを作成することもできます。その後、ドキュメント内に存在すらしない値ごとに並べ替えたり、制約したり、それらの値を抽出したりできます。

登録済みクエリ

登録済みクエリは、パフォーマンスを最適化するのためのまた別の手法です。これにより、cts:queryを繰り返し使用するものとして登録し、その結果をMarkLogicに保存して後で使用することができます。

例えば、自分のサイトのindex.htmlページをヒットする1日あたりのFirefoxブラウザの数に関するレポートを生成するとしましょう。日付を含まないクエリの一部を登録しておけば、日にちの制約との積集合を求めることで繰り返し使用できます。

以下のコードでは、アクセスするブラウザに関するデータを記録するfact名前空間に一連のドキュメントがあるとします。そのデータとは、名称、バージョン、各種属性、表示されるセッションIDです。これらのドキュメントは、ファクトテーブルのXML版であるとみなすことができます。また、dim名前空間（ディメンションテーブルのXML版）にも一連のドキュメントがあるとします。この名前空間は、特定のページヒットに関するデータを記録します。そのデータとは、ヒットしたページ、表示された日付、記録されたパフォーマンス特性といったリクエストのその他の側面です。

このクエリを効率的に実行するために、まずshotgunを使用するか、セッションIDに基づいて2つのドキュメントセットを結合する必要があります。その後、これは手のかかる操作であるため、クエリを登録します。

```
(: Firefoxユーザーのすべてのindex.htmlページビューを含むクエリを登録 :)

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")

(: ここで各日について、クエリに一致するヒットをカウント。 :)

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))
)
```

このクエリは迅速に実行できます。なぜなら、MarkLogicは日々、登録済みクエリからキャッシュされた結果に対し、日付のタームリストの積集合を求めるだけでよいからです。登録はクエリ間でも持続するため、恐らく日付以外の要素によって制限を受ける同様のクエリにおいて、2回目に実行してもメリットがあります。

cts:register()呼び出しは、登録のためのxs:long識別子を返します。cts:registered-query()呼び出しは、そのxs:longをライブのcts:queryオブジェクトに変換します。前述のコードは、登録済みのサーバー上のクエリとまったく同じクエリを登録すれば、同じxs:long識別子を返すファクトを利用しています。これによって、クエリ間でxs:long値を覚えておかないで済むようになります。

登録済みクエリはメモリキャッシュでトラッキングされ、キャッシュが大きくなり過ぎると、一部の登録済みクエリがキャッシュから古い順に削除されることがあります。また、MarkLogicが停止または再起動した場合、登録されていたクエリはすべて失われ、再登録が必要となります。ここの例で示すように、使用する直前に登録(または再登録)すれば、この問題を回避できます。

登録済みクエリには多数のユースケースがあります。よくある例は、ユーザーの参照可能なデータに制約を課す必要があり、制約は (LDAPシステムなどによって)外部的に定義されているか、制約があまりにも変わるのでMarkLogicに組み込まれているセキュリティモデルを使用して規則を適用しても意 味がないような場合です。ユーザーがログインすると、このアプリケーションはユーザーの参照可能な範囲をcts:queryとして宣言し、そのcts:queryを最適化された状態で繰り返し使用するために登録します。 ユーザーがデータを操作するときに、ユーザーのすべてのアクションと登録済みクエリとの積集合から、可視性が制限されたビューが表示されます。ビルトインのセキュリティの場合と異なり、可視性の規則をその場で自由に変更できます。また、やはりビルトインのセキュリティの場合と異なり、最初の実行で初期コストが発生します。

登録済みクエリを実行する初回は、登録されていない場合と同じだけの時間が解決までにかかります。結果(cts:queryによって返されるフラグメントIDのセット)が内部にキャッシュされるため、後からの実行の方がメリットがあります。

登録済みクエリは合成タームリストのように動作します。複雑なcts:queryは、場合によっては数百、数千ものタームリストとの間で積集合と和集合を求め、無数のレンジインデックスに対する処理を実行する必要があります。クエリを登録することで、MarkLogicはそのすべての演算およびレンジインデックス処理の結果を取り込み、シンプルなキャッシュ済み合成タームリストを作成します(実際には、結果はリストキャッシュに配置されます)。常に"unfiltered"を登録呼び出しに渡し、この合成タームリストがフィルタリングなしで機能することを承認する必要があります。

驚くべきことに、ドキュメントが追加または削除されるとキャッシュが更新され、常にトランザクション的に一貫性のあるビューを取得できます。なぜこのように動作するのでしょうか。

最下層のレベルでは、登録済みの各クエリに関して、すべてのフォレスト内のすべてのスタンドで、合成タームリストが維持されています。これが、更新時にMarkLogicがキャッシュを最新に保つ機能に隠された秘密です。

オンディスクスタンド内では、登録済みクエリが最初に使用されたときに合成タームリストが生成されます。ドキュメントが更新または削除された場合、オンディスクスタンド内のどのフラグメントにも起こり得る唯一の現象は、「削除済み」としてマークされることです（MVCCのメリットのひとつ）。これは、合成タームリストを変更する必要がないことを意味します。削除後でも、同じリスト値を報告できます。削除されたフラグメントを取り除くには、タイムスタンプベースのタームリストを使用します。

インメモリスタンドの場合、削除はオンディスクスタンドと同様に処理されますが、フラグメントは更新を介して挿入することができます。サーバーは、フラグメントが挿入された場合は常にインメモリスタンドに固有の合成タームリストを無効にすることで、これに対処します。その後、登録済みクエリがフラグメントの挿入後に使用されると、合成タームリストが再生成されます。インメモリスタンドはオンディスクスタンドと比べると非常に小さいため、合成リストの再生成は短時間で済みます。

インメモリスタンドがディスクに書き込まれた後、またはバックグラウンドでのスタンドのマージが完了した後、合成タームリストがない新しいオンディスクスタンドとなります。その合成タームリストは、次のクエリ実行の一環として（時間をかけて）再生成されます。使用されていない登録済みクエリがあっても、オーバーヘッドは発生しません。

登録済みクエリの使い方の詳細については、『Search Developer's Guide』を参照してください。

位置インデックス

MarkLogicの位置インデックスを使用すると、ドキュメントで言及されている地理的な位置に基づいてクエリ制約を追加できます。地理的なポイントはXMLドキュメントで（GML、KML、GeoRSS/Simple、またはMetacartaのマークアップ標準を使用した便利な関数を使用して）明示的に参照できます。あるいは、サードパーティのツールを使用したテキスト分析ヒューリスティックに応じて、場所の名前が自動的に特定、ジオコーディングされるエンティティ識別により、自動的に追加できます。

MarkLogicの位置インデックスを使用すると、ポイントごとに（つまり、緯度/経度で正確に）、ポイントと半径に対して（円）、緯度/経度ボックスに対して（メルカトルの「長方形」）、または任意の多角形に対して（大量の頂点がある場合に効率的、都市の境界線や道路からある程度離れた場所にある地域などの特性を作図する場合に便利）、一致させることができます。また、複雑な多角形に対して（イタリアの国境からバチカンを引いたような、多角形の内側にギャップのあるもの）一致させることもできます。

位置インデックスは、極地および日付変更線に近い経度180度線に完全に対応しており、地球の球形ではなく楕円形の形状を考慮に入れています。地理空間情報インデックスも、ほかのイン デックスと組み合わせて利用可能であるため、検索語と最も関連性が高く、指定の期間内に記述され、抽象的な多角形の内外の場所をソース

とするドキュメントを検索できます。また、地理空間情報バケット化に基づいて頻度のカウントを生成することもできます。この目的は、例えば、地理的な境界ボックス内に表示されるクエリに一致するドキュメントの数を効率的にカウントすることなどです。その後、このカウントを使用してヒートマップを作成することができます。

技術的なレベルでは、MarkLogicの位置インデックスは、ポイントをデータ値とするレンジインデックスのように機能します。位置レンジインデックスの全エントリは、単一のスカラ値だけでなく、緯度と経度のペアも保持しています。緯度/経度の値と関連ドキュメントIDを保持し、緯度メジャーと経度マイナーの順に並べ替えられ、メモリマップファイルに格納されている構造の長い配列を想像してみてください（緯度メジャーと経度マイナーとは、同じ緯度のポイントの場合、まず緯度で並べ替えられ、次に経度で並べ替えられることを意味します）。

ポイントクエリは、ソート済みのインデックス内で一致するポイントを特定し、対応するドキュメントID（複数の場合もあり）を抽出することで、簡単に解決できます。ボックスクエリ（2つの緯度の値と2つの経度の値の間の一致を探す）は、まず緯度境界内の位置インデックスのサブセクションを特定し、次に経度境界内にもあるそのレンジ内のセクションを特定することで解決できます[11]。

円と多角形の制約について、MarkLogicは、レンジインデックス内の特定のポイントが円または多角形の制約の内側と外側のどちらにあるかを判断するために、高速の比較機能を備えています。位置インデックスは、文字列ベースのレンジインデックスが文字列コレーション比較機能を使用する場所で、この比較機能を使用します。この比較機能はコアごとに1秒あたり100万～1,000万ポイントを比較できるため、レンジインデックス全体を高速でスキャンできます。コツは、特定のポイントから北または南の方向を見て、境界の形状を持つアーク交差をカウントすることです。交差が偶数の場合ポイントは外側にあり、奇数の場合は内側にあります。

円と多角形のアクセラレータとして、MarkLogicは初回の境界線ボックスセット（円または多角形を完全に含むボックスまたはボックスシリーズ）を使用して、詳細な比較機能で実行する必要があるポイントの数を制限します。このため、円の制約ではすべてのポイントを比較する必要がなく、円の周囲の境界ボックス内にあるものだけを比較すればよいことになります。

地球上の特定の場所を検索するのは少々厄介です。極は、すべての経線が集まる特異点を表します。このため、MarkLogicでは特別な三角法を使用して、この部分の検索の解決を支援します。180度経線（経度の値が負から正に切り替わる場所）をまたぐ位置形状に関しては、サーバーはこの特別な境界線をまたがない複数の小さな領域を生成し、結果を統合します。

この内容の詳細については、Geospatial Search Applications（Search Developer's Guide / 13.0 位置検索アプリケーション）を参照してください。

11 境界ボックスでパフォーマンスが最も悪いのは、細長い縦型のスライスです。

高度なコツ

API呼び出しで座標系「raw」を使用すると、MarkLogicは世界を平坦なものとして（同時に無限のものとして）扱い、複雑な大円の楕円形計算をせずにポイントを簡略的に比較します。これは、患者の身長と体重など、地理以外の何かを表す2つの値のレンジインデックスが必要な場合に便利です。身長は経度（x軸）として、体重は緯度（y軸）として表されます。データをチャートとしてレイアウトすると、大人数の患者から得た結果は散布図のようになります。MarkLogicの地理空間情報呼び出しを使用すると、チャート内の意味のある（異なる統計値に応じて、どの患者を正常、体重過多、体重過少と判断すべきか、など）部分の周囲に任意の多角形を描画し、これらの領域をクエリ制約として使用することができます。このコツは、データを散布図として捉え、ポイント、ボックス、円、または多角形で結果を制限するのが妥当な、2つの値を持つデータシリーズすべてに有効です。

リバースインデックス

ここまで取り上げてきたインデックス化戦略は、すべてフォワードクエリと呼ばれるものを実行します。これは、クエリから始めて、一致するドキュメントのセットを特定するものです。リバースクエリには、この反対の機能があります。つまり、まずドキュメントから始めて、一致するクエリ（実行すればこのドキュメントに一致する保存済みクエリのセット）すべてを特定します。

プログラム的には、まず、シリアライズされたクエリ表現をMarkLogic内に保存します。シンプルなドキュメントとしても、大きなドキュメントの中の要素としても保存できます。便利な機能として、`cts:query`オブジェクトはすべて、XMLコンテキストに配置されると、XMLとして自動的にシリアライズします。このXQueryが...

```
<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>
```

このXMLを生成します。

```
<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>
```

このようなドキュメントの長いリストがあると考えてください。それぞれが、何らかの`cts:query`制約を定義する異なる内部XMLを持っています。特定のドキュメント$docについては、次のXQuery呼び出しで一致するクエリのセットを特定できます。

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

これですべての<query>要素が返されます。その中には、実行された場合、ドキュメント$docに一致するシリアライズされたクエリが含まれています。もちろん、ルート要素の名前は何でも構いません。

MarkLogicは、[fast reverse searches]インデックスを有効にすることで、リバースクエリを効率的に広範囲で実行します。保存されたクエリが多数あり、1秒あたり多数のドキュメントが読み込まれる場合でも、外部からのドキュメントごとにリバースクエリを実行できます。しかも、オーバーヘッドはほとんどありません。これがどのように機能するか後で検証しますが、まずはリバースクエリが役立つ状況をいくつか見ていきましょう。

リバースクエリのユースケース

リバースクエリの一般的なユースケースの1つは、アラートです。特定の条件に一致する新しいドキュメントが現れた場合に、常に関係者に通知することができます。例えば、政治専門誌であるCongressional Quarterlyでは、MarkLogicのリバースクエリを使用してアラートに対応しています。米国連邦議会で誰かが特定の語や語句を口にした場合は常に、その場で通知を受けるように依頼することができます。議事録が追加されるとすぐに、アラートを発することができます。

アラートの対象は、語または語句の単純なクエリのみとは限りません。一致の条件は、任意の`cts:query`コンストラクトにすることができます。これには、ブール型、構造認識型クエリ、近接クエリ、レンジクエリ、さらには位置クエリが揃っています。会社のXBRL書類に興味ある内容が含まれているなら直ちに通知を受けたいという場合でも、アラートなら対応できます。特定の都市で地震が発生した場合はどうでしょうか。その都市をリバースインデックスの位置クエリとして定義し、地震が発生した際は一致する緯度/経度データを確認してください。

リバースクエリのインデックス化を行わない場合、新しいドキュメントまたはドキュメントセットごとに、すべてのクエリを繰り返してどれが一致するかを確認しなければなりません。クエリの数が増えるのに応じて、このあまりにも単純な手法が次第に非効率になっていきます。

リバースクエリは、ルールベースの分類にも使用することができます。MarkLogicにはSVM（サポートベクタマシン）分類子が含まれています。SVM分類子の詳細については本書では取り上げませんが、ドキュメントトレーニングセットに基づき、ドキュメントタームベクタ間の類似性を特定するものと理解すれば十分でしょう。リバースクエリは、トレーニングベースの分類子に代わってルールベースの分類子を提供します。各分類グループを`cts:query`として定義します。このクエリは、グループに属する条件に相当する必要があります。リバースクエリを使用すると、新しいドキュメントや修正されたドキュメントをそれぞれ適切な分類グループ（複数の場合もあります）に迅速に配置できます。

リバースクエリの最も興味深く驚くべきユースケースは恐らく、マッチメーキング（仲介）でしょう。車の相乗り（運転者/同乗者）、雇用（仕事/履歴書）、投薬（患者/薬）、検索のセキュリティ（ドキュメント/ユーザー）、恋愛（男性/女性、または混合）、さらには戦闘（標的/射手）などを一致させることができます。マッチメーキングでは、各エンティティをドキュメントとして表します。そのドキュメント内で、そのエンティティ自体に関するファクトと、それに一致すべきほかのドキュメントに関するエンティティの優先順位を定義し、`cts:query`としてシリアライズします。リバースクエリとフォワードクエリを組み合わせて使用すると、双方向の効率的なマッチングを行い、お互いの条件に一致するエンティティのペアを特定することができます。

リバースクエリによる車の相乗りのマッチング

このアイデアを具体的に理解するために、車の相乗りの例を取り上げましょう。ある女性の運転者がいるとします。彼女は煙草を吸わず、朝8時にサンラモンを出発して、サンカルロスまで運転します。音楽はロック、ポップス、ヒップホップを聴き、燃料代として10ドルを求めています。女性の同乗者を希望しており、出発地と到着地のそれぞれ周囲5マイル以内でピックアップしたいと考えています。一方、ある女性の同乗者がいます。最高で20ドルまで支払うと言っています。出発地は「3001 Summit View Drive, San Ramon, CA 94582」で、「400 Concourse Drive, Belmont,

CA 94002」まで行きます。禁煙車を希望しており、カントリーミュージックは聴きません。マッチメーキングクエリにより、このような女性2人をマッチングさせるだけでなく、1秒もかからずに数百万もの人々の中からほかの一致も見つけることができます。

このXQueryコードは、運転者の定義、つまり彼女の属性と優先事項を挿入します。

```
let $from := cts:point(37.751658,-121.898387) (: サンラモン :)
let $to := cts:point(37.507363, -122.247119) (: サンカルロス :)
return xdmp:document-insert( "/driver.xml",
   <driver>
      <from>{$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>
)
```

この挿入は同乗者、つまり彼女の属性と優先事項を定義します。

```
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>
)
```

あなたが運転者である場合、以下のクエリを実行して同乗者を探すことができます。

```
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)
```

このクエリは複数の同乗者候補を対象に、運転者（あなた）の優先事項と同乗者の優先事項が一致し、同乗者の優先事項が運転者の優先事項と一致する条件で検索を実行します。ルールの組み合わせは、cts:and-queryによって定義されます。前半は、優先事項要素に保持されているシリアライズされたクエリから、ライブの cts:queryオブジェクトを構築します。後半は、リバースクエリ制約を構築します。$meをソースドキュメントとして渡し、$meに一致するシリアライズされたクエリを含むほかのドキュメントに検索を制限します。

あなたが同乗者である場合、以下のクエリで運転者を見つけることができます。

```
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)
```

この場合も、両者の優先事項が互いに一致しなければなりません。MarkLogicでは、このような複雑なクエリ（通常のタームクエリに加えて、否定のクエリ、レンジクエリ、位置クエリが使用されています）でも効率的に広範囲で実行できます。

リバースインデックス

リバースクエリを効率的に解決するために、MarkLogicではカスタムインデックスと2段階評価を使用しています。第1段階はソースドキュメントから始まり（新たに読み込まれるドキュメントを知らせるアラートを伴う）、ソースドキュメント内で見つかった少なくとも1つのクエリターム制約との一致を含むシリアライズされたクエリドキュメントセットを特定します。続けて第2段階では、シリアライズされたそれらのクエリドキュメントそれぞれを調べ、存在するその他の制約すべてに基づいて、どれが実際にソースドキュメントと完全に一致するかを判断します。基本的には、まずいずれかの一致を含むドキュメントを迅速に特定してから、すべての一致を含むドキュメントを抽出します。

第1段階に対応するために、MarkLogicでは個別のリーフノードクエリターム（つまり、cts:and-queryのような複合クエリタームではなく、語や値のようなシンプルなクエリターム）すべてを集めるカスタムインデックスを維持しています。対象範囲は、シリアライズされたすべてのcts:queryドキュメント全体です。各リーフノードについて、MarkLogicはタームがドキュメント内に存在するときにリバースクエリの一致候補として指定するためのドキュメントIDのセットと、（否定のクエリで）タームが明示的には存在しない場合に指定するための別のIDセットを維持しています。それは、クエリタームのタームリストです。

その後、ソースドキュメントを示されたときに、MarkLogicはそのドキュメント内に存在するタームのセットを収集します。ドキュメント内のタームを構築済みのリバースインデックスと比較し、ソースドキュメント内の少なくとも1つのタームと一致するクエリを持つ、シリアライズされたクエリドキュメントIDすべてを特定します。シンプルな1語のクエリの場合、これが最終的な答えとなります。それ以上複雑なクエリの場合、MarkLogicは、ソースドキュメントがその複雑なクエリの制約全体に対して本当に一致するかを確認するために、第2段階を実行する必要があります。

第2段階のために、MarkLogicはカスタムの有向非巡回グラフ（DAG）を維持しています。これは、ブランチが重なり合っている可能性があり、無数のルートを持つツリーです。クエリドキュメントIDごとにルートが1つあります。MarkLogicは指定されたクエリドキュメントIDのセットを取得し、ドキュメントIDのルートノードからこのDAGを介してこれらを実行して、必要な制約すべてが真であるかどうかを下方に向けて確認し、可能な場合は常にショートサーキット評価を行います。すべての制約が満たされると、MarkLogicは、指定されたクエリドキュメントIDが実際にソースドキュメントに対するリバースクエリに一致すると判断します。

この時点で、ユーザーのクエリに応じて、MarkLogicは処理の結果を最終的な回答として返すか、ドキュメントIDをより大きなクエリコンテキストに供給することができます。マッチメーキングの例では、`cts:reverse-query()`制約が`cts:and-query()`の半分のみを表し、双方向の一致を特定するためには、その結果とフォワードクエリの結果との積集合を求めなければなりませんでした。

シリアライズされたクエリが、`cts:near-query`または位置を使用する必要がある語句のいずれかを使用して正確に解決しなければならない位置制約を含んでいるしたらどうでしょうか。MarkLogicでは、DAGを進める際に位置を考慮します。

複雑ではありますが、良好に、しかも迅速に機能します。

リバースインデックスにおけるレンジクエリ

リバースクエリで使用されているレンジクエリについてはどうでしょうか。これらのクエリには特別な対応が必要です。なぜなら、レンジクエリにはシンプルなリーフノードタームがないからです。処理の第1段階では、「存在する」ものも「存在しない」ものも見つかりません。これに対してMarkLogicは、シリアライズされたレンジクエリで使用されているカットポイントに基づいて、ルックアップ用のサブレンジを定義します。シリアライズされた3つのクエリがあり、それぞれに異なるレンジ制約があるとしましょう。

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>
```

上記のレンジには、5、7、10、15、20、+無限というカットポイントがあります。隣接するカットポイント間の値のレンジにはすべて、同じマッチングクエリドキュメントIDがある可能性があります。これによって、処理の第1段階でこれらのレンジをリーフノードのように使用できます。

	存在する
5〜7	4
7〜10	4 5
10〜15	5
15〜20	5 6
20〜+無限	6

8という価格を持つソースドキュメントがある場合、MarkLogicはクエリドキュメントIDの4と5を指定します。なぜなら、8は7〜10のサブレンジに入っているからです。2という価格の場合、MarkLogicはどのドキュメントも指定しません（少なくとも、価格の制約に基づいては）。第2段階で、レンジクエリはDAGの特殊なリーフノードのように機能し、カットポイント不要で直接的に解決できます。

最後に、位置クエリについてはどうでしょうか。MarkLogicはレンジクエリの場合と同じカットポイント手法を使用しますが、2次元である点が異なります。第1段階に対応するために、MarkLogicは地理的な境界ボックスのセットを生成します。それぞれに、ソースドキュメントでそのボックス内にポイントが含まれている場合に指

定するクエリドキュメントIDの独自のセットがあります。第2段階では、レンジクエリの場合のように、地理的な制約が、精密な地理空間情報比較機能を備えたDAG上の特殊なリーフノードのように機能します。

概して、第1段階では、シリアライズされた多数のクエリの中から、ソースドキュメントに対する一致が少なくとも1つあるものだけに迅速に制限します。第2段階では、指定された各ドキュメントが実際に一致するかどうかを確認するためのチェックを行います。このために、ショートサーキット評価および式を最大限に再利用して迅速な判断を可能にする、特殊なデータ構造を使用します。シリアライズされた対象クエリの数にかかわらず、リバースクエリのパフォーマンスは一定しており、特定される実際の一致数に対して線形である傾向があります。

バイテンポラル

MarkLogicのバイテンポラル機能を使用すると、2つの時間軸に沿って同時にデータベースドキュメントをトラッキングできます。ファクトが真であった時間（実際の時間）とデータベースがその真実を認識した時間（システム時間）をトラッキングできます。金融、保険、インテリジェンスなどの業界では、2つの時間軸でデータをトラッキングすることが不可欠です。これにより組織は、特定の時点における世の中についての情報に基づいて判断を下した理由を正当化できます。

MarkLogicでは、バイテンポラルデータをどのように管理しているのでしょうか。テンポラルドキュメントにはそれぞれ、ドキュメントデータの実際の時間とシステム時間を定義する4つのタイムスタンプの値（valid start、valid end、system start、system end）があります。新しい期間のテンポラルドキュメントを挿入すると、MarkLogicはシステムのタイムスタンプの値をトラッキングし、バージョン履歴全体にデータが確実に入力されるようにすることができます。デフォルトでは、ドキュメントのバイテンポラル履歴を構成するバージョンがなくなることはありません（ただし、タイムスタンプが変わることはあります）。

実際の時間とシステム時間のレンジを指定することで、バイテンポラルデータに対してクエリを実行することができます。バイテンポラルクエリを解決するために、MarkLogicは各タイムスタンプに1つずつ、合計4つのレンジインデックスを使用して、一致するドキュメントを迅速に特定します。

バイテンポラルの例

テンポラルドキュメントを使用して、ある人物の居場所をある一定期間にわたってトラッキングするというシンプルな例を考えましょう。この場合の実際の時間は、その人物がある場所に本当にいた時間を示し、システム時間は、私たちがその情報を知った時間をトラッキングします（デフォルトでは、これはドキュメントがデータベースに保存された時間に基づいています）。その人物がいる場所が変わるたびに、その情報を表す新しいテンポラルドキュメントを読み込みます。MarkLogicは、バックグラウンドで既存のテンポラルドキュメントに対してハウスキーピング処理

を実行し、「何を知ったか」および「いつ知ったか」の履歴を残せるように、タイムスタンプを最新の状態に保ちます。

例えば、ある人物が1月1日にアラメダ市に移動したとします。私たちはこのことを知り、1月5日にこのデータをデータベースに追加します。ドキュメントはテンポラルドキュメントなので、4つのタイムスタンプ（valid start、valid end、system start、system end）が含まれています。この時点では、この人物がそれ以外にどこに移動したかが分からないため、開始時間だけを設定します（終了時間はデフォルト設定の無限のままにしておきます）。valid start（実際の開始時間）はこの人物がいつからアラメダ市にいたかを示し、system start（システム上の開始時間）は私たちがいつこれを知ったかを記録します。

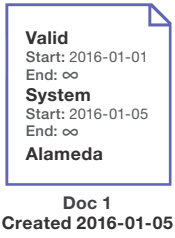


図10: ある人物の特定の時点における場所を表すドキュメントは、バイテンポラルデータベースに追加される

今度は、この人物が1月10日にバークレーに移動し、私たちが1月12日にこのことを知ったとしましょう。この情報を新しいテンポラルドキュメントとして挿入します（Doc 3）。この新しいドキュメントに合わせて、MarkLogicは、私たちが現在知っているように、この人物がアラメダ市で過ごした時間を反映したドキュメントを作成します（Doc 2）。また、MarkLogicは最初のドキュメントを更新してsystem end（システム上の終了時間）を追加します。これで、最初のドキュメントが、1月12日までのこの人物について私たちが知っていることの履歴になります。

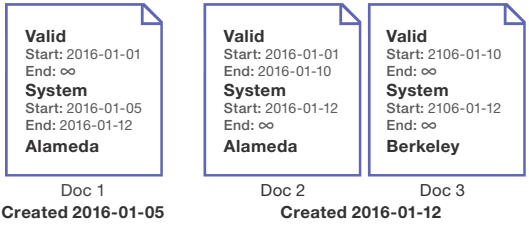


図11: 人物の新しいドキュメント（Doc 3）に、新しい地理空間情報が反映される。MarkLogicはドキュメント（Doc 2）を追加し、元のバージョン（Doc 1）を更新して履歴データを入力する

1月15日の時点で、私たちは間違いを発見します。この人物は、実は1月8日にバークレーに移動していたのです。しかし、問題はありません。新しい実際の時間を記した修正済みドキュメントを挿入します（Doc 5）。MarkLogicはバックグラウンドでドキュメント（Doc 4）を作成し、既存のドキュメント（Doc 2、Doc 3）のsystem endを編集して、正確な履歴を維持します。

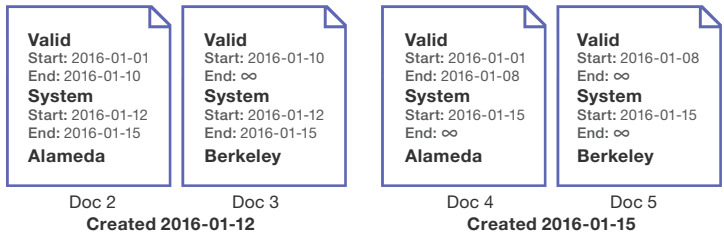


図12: この人物の地理空間情報に再び変更が加わる（Doc 5）。今回も、MarkLogicがドキュメント（Doc 4）を追加し、時間を更新して履歴データを入力する

1月18日、私たちがトラッキングしているこの人物は、実際にはその当人ではないことが判明しました。前の情報は無効であるため、バイテンポラル削除を実行します。MarkLogicは、バックグラウンドで最新のドキュメントセットの終了値を更新します（データベースから実際にドキュメントを削除することはありません）。結局、ファクトは間違っていましたが、私たちが追っていた一連の情報は完全に記録されました。

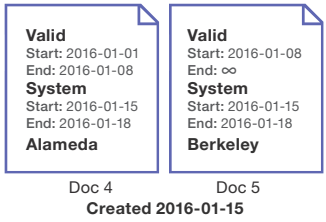


図13: バイテンポラル削除を実行しても、ドキュメントが実際に削除されることはない。MarkLogicはドキュメントバージョンのタイムスタンプを更新し、ある一定期間に知り得た内容を正確に記録する

バイテンポラルデータは、システム時間を横軸、実際の時間を縦軸にとる2次元のチャートとして表すことができます。以下のチャートの色付きボックスは、この例の5つのドキュメントバージョンを表しています。

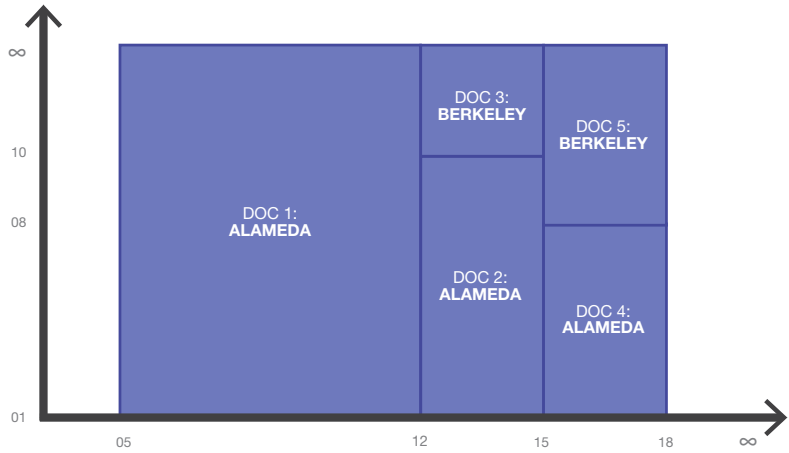


図14: ある一定期間にバイテンポラルデータベースに記録されたデータを表す2次元チャート

この例に示されているように、MarkLogicはドキュメントバージョンのタイムスタンプに一貫性を持たせ、新しい情報が追加されると常にバージョン履歴を入力します。興味深い点は、1つの期間のデータを挿入すると、最終的に複数のドキュメントが更新されることが多いことです。例えば、挿入したデータが、前に挿入したテンポラルドキュメントに完全に含まれている実際の時間のレンジに関連している場合、そのドキュメントは新しい挿入の前後のレンジに対応できるように2つに分割され、新しいドキュメントが中央のレンジに対応します。

バイテンポラルクエリ

バイテンポラルデータに対してクエリを実行する場合、4つのタイムスタンプそれぞれでレンジインデックスを利用します。例えば「対象人物が1月13日～14日にいた場所は知っているが、1月8日～9日にいた場所はどこか」というクエリがあるとしましょう。このクエリは、タイムレンジを定義する`cts:period`制約を使用して表現します。

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

このコードでは、ショートカットを使用して、Allenの区間代数の関係に基づく時間間隔を定義しています[12]。これらは、2つの時間間隔の間で考え得る関係すべてを定義します。例えば、preceding、overlapping、containing、meeting、finishingなどです。MarkLogicは、内部的にこのクエリを以下の不等式に分割することができます。

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

レンジインデックスを考慮すると、以下のようになります。

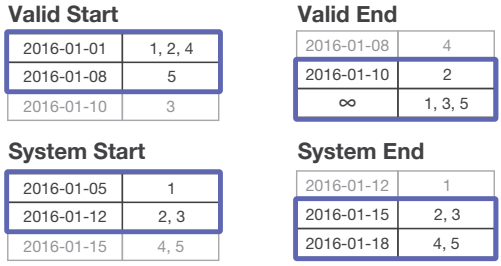


図15: バイテンポラルクエリを解決するために、MarkLogicはドキュメントのレンジインデックスに対して不等式の演算を実行できる

4つの結果セットの積集合を実行することで、1つのドキュメント（2番目）を得ることができます。これによって、この人物がこの期間にアラメダ市にいたことが分かります。

さらに、MarkLogicはテンポラルドキュメントを3つのコレクションに分類します。バイテンポラルデータはデータベース内の通常のデータと共存できるため、MarkLogicはすべてのテンポラルドキュメントを1つのテンポラルコレクショ

12 MarkLogicでは、時間の関係の定義にSQL 2011の各種演算子を使用することもできます。これらの演算子はAllenの演算子に似ていますが、制約はそれほど厳しくありません。

ン（名称は「temporalCollection」）に追加します[13]。テンポラルコレクションは、テンポラル関数を使用してしか追加や削除ができないという意味で特殊です（テンポラルコレクションを作成するときは、管理者がテンポラル以外の関数を使用してテンポラルドキュメントを操作できるようにするかを指定するオプションを渡します。これを指定しない場合、管理者を含むどのユーザーも、テンポラル以外の手法でテンポラルドキュメントを変更したり削除したりできません）。さらに、一定期間に作成されたテンポラルドキュメントの全バージョンはURIコレクションに割り当てられます。これは、全ドキュメントに異なる物理URIがあるとしても、これらのドキュメントを論理的に結合するものです。位置追跡の例では、元のドキュメントにURI「person.json」があった場合、そのドキュメントの各バージョンは「person.json」コレクションに割り当てられます。URI「person2.json」を持つ2番目のドキュメントを挿入した場合、そのドキュメントは「person2.json」コレクションに割り当てられます。最後に、有効な最新のドキュメントバージョン用のコレクションがあります。つまり、システム上の終了時間が「無限」に設定されているドキュメント（名称は「latest」）です。ドキュメントの新しいバージョンが挿入されると、このlatestコレクションに配置され、更新対象のドキュメントは削除されます。バイテンポラルコレクションで削除されたドキュメントには最新バージョンはありません。

詳細については、『Temporal Developer's Guide』を参照してください。

セマンティック

MarkLogicのセマンティックを使用すると、また別の方法でデータを表現することができます。それは、トリプルです。トリプルはデータ要素間の関係を記述するもので、主語、述語、目的語から構成されます（人間の言語と似ています）。

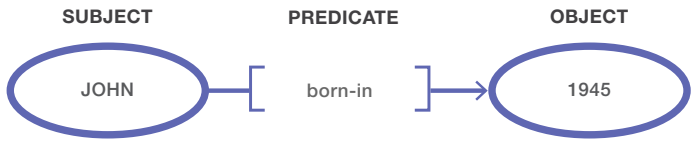


図16: セマンティックトリプルは、世界に関するファクトを主語-述語-目的語の関係で表す

セマンティックデータの重要な側面は、関係を記述できることだけでなく（これによって「ジョンはいつ生まれたのか？」という質問に回答できる）、それらの関係を相互に関連付けることができるということです。ジョンについて説明するトリプルを追加し、さらにメアリーについて説明するトリプルも追加して、そうしたファクトの一部から積集合を求めることができます。その結果、相互につながり合った情報網が生まれます。これは、以下のような図で表すことができます。

13 ドキュメントがテンポラルコレクションに入っていなくても、テンポラルクエリと比較演算子は使用できます。必要なのは、ドキュメントに開始と終了のタイムスタンプがあることだけです。レンジインデックスがこれらのタイムスタンプに適用され、軸がこれらのタイムスタンプを結び付けます。こうしたドキュメントの場合、（テンポラルドキュメントに対して行うように）データ変更時にMarkLogicがタイムスタンプを自動的に更新したり、履歴データを入力する追加のドキュメントを作成したりすることはありません。

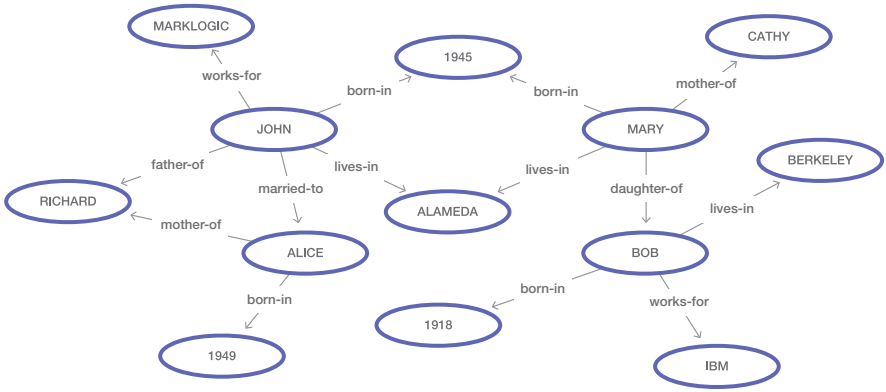


図17: 多数のセマンティックトリプル間の関係が情報網を形成する

MarkLogicのアプリケーションは、大量のトリプル（セマンティックグラフ）に対してクエリを実行し、興味深い情報を提供してくれます。グラフ内のトリプルをトラバースすることで、ジョンはメアリーと同じ年に生まれただけでなく、同じブロックに住み、同じ学校に通っていたという情報をアプリケーションから得られます。セマンティックデータを利用すれば、ジョンはメアリーを知っていたかもしれないという推測が可能になります。

トリプルの格納

RDF（Resource Description Framework）とは、主語-述語-目的語のシンタックスを使用してファクトを表現する方法を示すW3Cの標準です。これこそ、MarkLogicのトリプルが「RDFトリプル」と呼ばれている理由です。RDFでは、主語と述語が、国際化資源識別子（IRI）によって表現されます[14]。目的語はIRIにすることも、文字列、数字、日付などのリテラル値として入力することもできます。

MarkLogicが生のRDFトリプルを読み込むときは、そのトリプルをXML表現に変換してから、XMLドキュメントとして読み込みます。このため、MarkLogicは内部的に、通常のXMLドキュメントを表現するときと同様にセマンティックデータを表現します。結果的に、すべての機能と利便性をほかのMarkLogicコンテンツに関連付けた状態でトリプルを管理できます。トリプルはロールベースのセキュリティによって保護でき、コレクションとディレクトリに整理できます。また、ACID互換のトランザクション内で更新できます。さらに、`sem:triple`タグ（XMLの場合）またはtripleプロパティ（JSONの場合）で囲まれている場合、従来のXMLまたはJSONコンテンツに追加することもできます。これにより、主語-述語-目的語のファクトで既存のドキュメントを改良してから、通常の方法とセマンティックの方法の両方を使用してコンテンツを検索（または結合）することが可能になります。

14 IRIはURIに似ていますが、より幅広い国際文字をサポートしています。詳細についてはWikipediaを参照してください。

セマンティックをサポートするインデックス

セマンティック検索をサポートする特殊なインデックスは、トリプルインデックス、トリプル値インデックス、トリプル型インデックスの3種類です。

トリプルインデックス

トリプルインデックスでは、各トリプルの主語、述語、目的語に対してカラムがあります。また、見つかったドキュメントにトリプルをマッピングするためのカラム（タームインデックスの場合と同様）と、地理空間情報のためのカラムがあります（トリプル位置が有効になっている場合）。各トリプルは、主語-目的語-述語、述語-主語-目的語、目的語-述語-主語の順で並べ替えられた3つの異なる順列として、トリプルインデックスに格納されます（トリプルインデックスの追加のカラムで順列を指定）。

これら3種類の順列から選択できるため、MarkLogicでは可能な限り迅速にクエリを解決することができます。指定された主語を持つトリプルをクエリで特定する必要がある場合、主語で並べ替えられたトリプルのシーケンスを利用できます。このとき、一致するすべてのトリプルは隣接するブロックにあります。これは、特定の述語や目的語を持つトリプルを一致させる場合も同様です。指定された主語と述語を持つトリプルをクエリで特定する必要がある場合、まず述語で並べ替えられ、次に主語で並べ替えられたトリプルのシーケンスを利用できます。このとき、一致するトリプルはすべてまとめて隣接するブロックに置かれます。3つの順列があるため、トリプルの任意の2つの部分を隣接するブロックに置くことができます。また、クエリで特定の主語/述語/目的語の値を持つトリプルを必要としている場合、そのシーケンスも利用できます。

トリプルインデックスは多くの点で特殊なレンジインデックスのように機能しますが、レンジインデックスが常に完全にメモリにマッピングされるのに対して、トリプルインデックスは物理メモリよりもかなり大きくなるように、また、必要に応じて大きなトリプルブロックをメモリ内外にページングするように設計されています。

トリプル値インデックス

トリプルインデックスでは、各トリプル内で発生する値を実際には格納しません。代わりに、整数のIDでそれらの値を表し、ルックアップの高速化と使用スペースの削減を実現しています。IDは、トリプル値インデックス内の実際の値にマッピングされます。ある検索結果のトリプルインデックスからトリプルが取得された場合、MarkLogicはそのトリプル値インデックスを参照して値を再構築します。つまり、セマンティッククエリの結果を返すために、MarkLogicはトリプルインデックスにのみアクセスし、元のドキュメントにはアクセスしません。

トリプル型インデックス

トリプルの目的語の値は、型付きリテラル（文字列、日付、数値など）にすることができます。MarkLogicはこの型情報と、値の順序付けには不要なその他の情報（日付タイムゾーン情報など）を取り込み、トリプル型インデックスに格納します。トリプル型インデックスは通常、比較的小さく、メモリにマッピングされています。

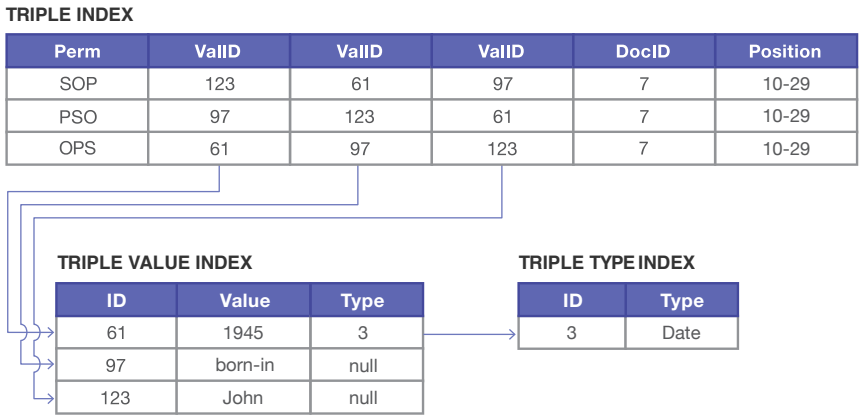


図18: MarkLogicは、トリプルインデックス、トリプル値インデックス、トリプル型インデックスという3つのインデックスを使用してセマンティッククエリに対応している。効率的なルックアップを行うために、各トリプルは主語-目的語-述語、述語-主語-目的語、目的語-述語-主語の3種類の順列に格納される

トリプルに対するクエリの実行

セマンティックデータを取得するためのクエリ言語、SPARQLを使用して、MarkLogicデータベース内のトリプルを検索できます。SQLを使用する場合とよく似ていますが、対象がトリプルである点が異なります。最も一般的なタイプであるSPARQL SELECTクエリを構築するには、IRI、リテラル値、変数を使用して1つあるいは複数のトリプルパターンを記述します。例えば、大学のマスコットに関するセマンティックデータがあるとします。

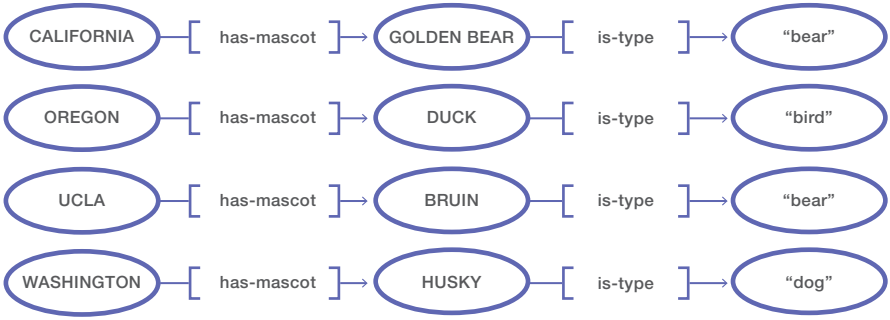


図19: 学校とそのマスコットを表すトリプルのセット

前述の各行がトリプルのペアを表しています。最初のトリプルが学校のマスコットを記述し、2番目がマスコットのタイプを記述しています。以下のSPARQLクエリは、マスコットがいて、それがクマである学校を返します。

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

MarkLogicは、WHERE節で2つのトリプルパターンを処理し、トトリプルインデックスで一致するパターンを特定します。最初のパターンはすべてに一致し、4つのトリプルを返します。2番目のパターンについては、`"bear"` がオブジェクトである2つのトリプルが返されます。MarkLogicは最終結果を得るために、共有変数である`?mascot`に基づいて2つの一致セットを結合します。これによって2つの結合結果、カリフォルニア大学とUCLAのIRIを得ることができます。これはシンプルな例ですが、セマンティッククエリを解決するための2つの重要なステップ、つまり、各パターンに対応するトリプルを取得してから、これらのトリプルセットを結合して最終結果を得るというステップを表しています。

SPARQLクエリは、その他のタイプのMarkLogic検索クエリとも組み合わせることができます。トリプルはMarkLogicデータベースのその他すべてのコンテンツと同様にドキュメントとして格納されるため、こうしたクエリは簡単に組み合わせることができます。SPARQLクエリがトリプルのセットを返すと、これらのトリプルはドキュメントIDのセットに関連付けられます。MarkLogicでは、これらのドキュメントIDと、その他のタイプのクエリから返されたものとの積集合を求めることができます。これによって、SPARQLの検索結果を、特定のコレクション、ディレクトリ、セキュリティロールなどと関連のあるトリプルに制限できます。

SPARQLの詳細については、『Semantics Developer's Guide (6.0 セマンティッククエリ)』を参照してください。

適切なクエリプランの特定

MarkLogicは、関与する多数のトリプルのセットをいかに取得して結合するかに応じて、複雑なSPARQLクエリを無数の方法のいずれかで解決できます。MarkLogicでは、トリプルのセットをマージする際、さまざまな結合アルゴリズムから選択できます（ハッシュ結合、スキャッタ結合、ブルーム結合、マージ結合など）。結合は、さまざまな順序で実行できます。トリプルインデックスからトリプルを取得するときに、さまざまな順列を選択できます。MarkLogicがSPARQLクエリを解決するために演算を構築する手法のことをクエリプランと呼びます。クエリプランが効率的であるほど、結果が返ってくるのも速くなります。

MarkLogicでは、固定ルールセットを使用して最善のクエリプランを解明しようと試みることはありません。代わりに、シミュレーションアニーリングというアルゴリズムを使用します。これは、コンピュータベースの自然選択を利用して限られた時間内に最善のプランを特定するというものです。MarkLogicは、SPARQLクエリの記述方法をベースにしたデフォルトのプランから開始し、その後コスト分析によってそ

のプランの効率性を評価します。この分析では、トリプルのカーディナリティ、結合によって使用されるメモリ、そして何よりも、中間結果が順序付けされて返されるかどうかが考慮されます（MarkLogicは、トリプルを読み込んでインデックス化するときに、カーディナリティ情報などのコスト分析に使用する統計情報の一部を収集します）。

その後、プランを改善するために、MarkLogicは以下を行います。

1. プランの処理を変更して、引き続き同じ結果を返す新しいバージョンを作成する。例えば、異なる結合方法を使用したり、結合の順序を変更したりできます
2. コスト分析を使用して新しいプランの効率性を評価する
3. 新しいプランの効率性と現在のプランの効率性を比較し、どちらかを選択する

続けて、時間切れになったら停止し、選択したプランを使用してSPARQLクエリを実行します。それ以外の場合は、ステップを繰り返します。

選択プロセスにおいて、MarkLogicは、新しいプランの方が効率的であれば常に、現在のプランよりも新しいプランを選択します。ただし、効率性が勝っていなくても、新しいプランを選択することがあります。これは、クエリプランが予期しない方法で互いに関連しており、最適なプランを特定するには、まず効率性の低い中間プランを選択する必要がある場合があるためです。時間の経過に応じて、効率性が低い新しいクエリプランをMarkLogicが選択する確率は下がっていきます。プロセスの終了までには、MarkLogicは非常に効率の高い（ただし必ずしも最適ではない）プランを特定するようになります。

このクエリプラン選択方法にはメリットがあります。それは、SPARQLクエリで設定を渡すことで、MarkLogicが最適化に費やす合計時間を指定できることです。複雑なクエリの場合は効率的な方法の発見に長い時間を費やし、シンプルなクエリの場合は短い時間を費やすようにすることができます。また、長期間繰り返し実行されているクエリにも長い時間を費やすことができます。これは、MarkLogicがこうしたクエリを最初に最適化してから、最適化されたクエリプランをキャッシュに格納するためです。

ドキュメント間結合のトリプル

Twitterのつぶやきのデータに関して、ドキュメント間結合のためのレンジインデックスの使用について前述しました。同じ目的でRDFトリプルを使用することができます。URIでドキュメント間に関係性を定義してから、クエリ時にSPARQLを使用してこれらのドキュメントを結合します。リレーショナルシステムで異なるテーブルの行を結合する場合と似ていますが、MarkLogicのマルチモデル（ドキュメントおよびセマンティック）機能を利用している点が異なります。

Twitterのつぶやきのデータの場合、RDFトリプルを使用してつぶやきを投稿者に関連づけ、さらに投稿者どうしを互いに関連付けて誰が誰をフォローしているかを定義できます。JSONとしてのトリプルは以下のようになります。

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

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

例えば、「healthcare」（医療）について最近言及している投稿者を見つけ、その投稿者のフォロワー全員を取得したいとします。JavaScriptのコードは以下のようになります。

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

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

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

最初のブロックで該当するタームを含むつぶやきを取得し、作成時間の降順で並べ替えます。結果から最新のつぶやきを選択し、続けてそのつぶやきのIDを、SPARQLクエリに渡されるオブジェクトに配置します。2番目のブロックでは、つぶやきのIDをつぶやきの投稿者に結合し、続けて投稿者をフォロワーに結合します。最後に残ったフォロワーのURIセットを使用して、関連する投稿者のドキュメントにアクセスできます。

バックアップの管理

MarkLogicではオンラインのバックアップとリストアに対応しているため、システムをオフラインにしたりクエリやアップデートを中断したりせずに、データを保護およびリストアできます。バックアップは、管理webコンソール（ボタンを押すアクションまたは定期ジョブとして）、サーバー側のXQueryまたはJavaScriptスクリプト、あるいはREST APIを介して開始できます。ユーザーは、バックアップ対象のデータベースとターゲットロケーションを指定します。データベースをバックアップすると、設定ファイルのコピー、データベース内のフォレストすべて、対応するセキュリティおよびスキーマデータベースが保存されます。セキュリティデータベースのバックアップは特に重要です。なぜなら、MarkLogicはロール識別子を文字列の名称ではなく数値IDでトラッキングし、対応するロールの数値IDがセキュリティデータベースに存在しない場合、バックアップフォレストのデータを読み取ることができないためです。

また、データベース全体ではなく個々のフォレストを選択してバックアップすることもできます。1つのフォレストのデータのみが変わる場合は、便利なオプションです。

典型的なバックアップ

多くの場合、バックアップの実行中、すべてのクエリとアップデートは通常どおりに継続します。MarkLogicでは、スタンドデータをソースディレクトリからバックアップターゲットディレクトリに、ファイルごとにコピーするだけです。スタンドは、小さなTimestampsファイルを除いて読み取り専用です。このため、リクエストを一切中断することなく、この一括コピーを継続できます。バックアップの終わりにだけ、MarkLogicは入ってきたリクエストを短時間中断する必要があります。これは、バックアップの完全に一貫したビューを書き出し、メモリからディスクにすべてを流し込むためです。

データベースバックアップでは、常に、ターゲットバックアップディレクトリにそのデータ用の新しいサブディレクトリが作成されます。フォレストバックアップの場合は、毎回、同じバックアップディレクトリに書き込まれます。ターゲットバックアップディレクトリにすでに前のバックアップのデータがある場合（古いスタンドがまだ新しいスタンドにマージされていない場合など）、MarkLogicは、ターゲットに既存する同一ファイルを一切コピーしません。これにより、パフォーマンスが向上します。

フラッシュバックアップ

MarkLogicは、フラッシュバックアップにも対応しています。一部のファイルシステムでは、ある特定の時点に存在するファイルのスナップショットをとることができます。ファイルシステムは基本的に、使用中のディスクブロックをトラッキングし、スナップショットが有効な間、読み取り専用にします。これらのファイルに加えられた変更はすべて、異なるディスクブロックに保存されます。これには非常に迅速であるという利点があり、変更されないディスクブロックについては一切複製を必要としません。これは、MVCCアップデート時のMarkLogicの動作とよく似ています。

MarkLogicのデータに対してフラッシュバックアップを実行するには、バックアップ対象のフォレストを完全に一貫性のあるオンディスクステート（すべてがメモリからフラッシュされ、進行中のディスク書き込みがない）にするようにMarkLogicに指定しなければなりません。各フォレストに［Updates Allowed］設定があります。この設定の詳細は以下のとおりです。

`all`
デフォルトの設定。フォレストが完全に読み取りと書き込みに対応していることを示します。

`delete-only`
フォレストは新しいフラグメントを受け入れることができませんが、既存のフラグメントを削除できることを示します。通常のドキュメント挿入では、このフォレストでの新しいデータの読み込みは回避します。

`read-only`
フォレストが一切の変更を受け入れることができないことを示します。このフォレスト内にあるフラグメントを修正または削除しようとすると、エラーが発生します。

`flash-backup`
read-only設定と似ていますが、フォレスト内のフラグメントを修正または削除するようにリクエストすると、即座にエラーが発生するのではなく、［Retry Timeout］の制限（デフォルトでは120秒）が過ぎるまで再試行される点が異なります。その意図するところは、フォレストを数秒間フラッシュバックアップのステートにして、スナップショットをとってから、通常の［all］ステートに戻すということです。

データベースのバックアップとリストアの詳細については、『Administrator's Guide』を参照してください。

ジャーナルアーカイブとポイントインタイムリカバリ

データベースバックアップを実行すると、ジャーナルアーカイブを実行するかどうかの確認を促されます。実行するように選択すると、MarkLogicはバックアップの隣に大きさに制限のないジャーナルを書き込み、バックアップ後に実行されたすべてのデータベーストランザクションを記録します。これは、データベースで保持されるメインのローテーションされるジャーナルとは別のジャーナルである点に注意してください。

バックアップにジャーナルがあることには、主に2つの利点があります。まず、プライマリデータベースのファイルシステムで大規模障害が発生した場合に、失われるデータの量を削減することができます。通常のバックアップでは、バックアップイベントの発生時に存在していたデータしか復元できませんが、ジャーナルアーカイブを指定してバックアップすると、バックアップ後に発生したすべてをジャーナルから再生できるため、データベースを大規模障害のほぼ直前の状態に復元することができます。

2つ目の利点は、バックアップと現在の間であれば、任意の時点に復旧できることです。これは、ポイントインタイムリカバリと呼ばれます。多くのエンタープライズシステムでは、ディスク障害だけでなく、人的エラーに対しても復元力が必要とされます。人間が誤って、あるいは悪意を持って、またはコードのバグの結果として、データを不適切に修正してしまったとしたらどうなるでしょうか。ジャーナルアーカイブにより、MarkLogicは、バックアップのチェックポイントから目的の時点までジャーナルを再生するという方法で、データベースを過去の任意の時点に復元することができます。

これは、一度バックアップすれば、それ以降はバックアップが不要であることを意味するのでしょうか。いいえ、違います。なぜなら、ジャーナルを適用するには時間がかかるためです。ロックと通常のディスク同期が不要なため、ジャーナルの再生は最初の実行よりもスピードアップしますが、それでも時間はかかります。長い再生の間に、マージが発生する場合もあります。削除されたデータはなくならないので、バックアップも無制限に大きくなります。アクセスの多いシステムやデータが大量に保存されているシステムでは、バックアップを定期的に実行して、新しい開始時間のチェックポイントを作成する必要があります。ジャーナルアーカイブは、一度に1つのデータベースバックアップに対してのみ実行できる点に注意してください。新しいバックアップにアーカイブすると、（新しいアーカイブが終了した時点で）古いアーカイブへの書き込みは中断されます。

ジャーナルアーカイブにより、データベースへの書き込みの速度が落ちる可能性が生じます。ジャーナルアーカイブがデータベースからどの程度遅れることができるかを指定する設定可能な「ラグタイム」があります。デフォルトは15秒で、この時間が過ぎると、ジャーナルアーカイブが追いつくまでデータベースへの書き込みが遅くなります。このタイムラグの目的は、ジャーナルアーカイブが遅いファイルシステムにあっても大量の書き込みをデータベースで実行できるようにすることです。

興味深いことに、ジャーナルアーカイブを指定してバックアップを実行すると、メインのフォレストデータの書き込みが進行中でも、ジャーナルはほぼ即座にトランザクションの記録を開始します。これは、バックアップには数時間かかる可能性があり、バックアップが行われるタイムスタンプの後にジャーナルアーカイブがすべての更新を開始する必要があるためです。

ジャーナルアーカイブは常にデータベースバックアップの一部として設定しますが、ジャーナル処理そのものはフォレストごとに行われます。これは、プライマリデータベースで障害が発生した後に全体をリストアすると、整合性がとれなくなる（ジャーナルフレームを持つ一部のフォレストが、書き込みが若干遅いほかのフォレストジャーナルが何らかの情報を認識しているトランザクションを超えて、トランザクションを参照する）可能性があることを意味します。デフォルトでは、リストアによって、整合性がとれていなくてもバックアップのすべてのデータが復元されます。つまり、すべてのデータを取得できますが、一部のトランザクションは部

分的にしか存在しない可能性があります。ユースケースによってはこれで適切ですが、そうでない場合もあります。ジャーナルアーカイブは、1秒あたり1回以上書き込みを実行することで、その最新の状態をトラッキングします。『Administrator's Guide』に記載されているXQueryスクリプトを使用すると、状態を最新の「安全な」タイムスタンプに戻し、整合性のとれていないトランザクションすべてを元に戻すことができます。

増分バックアップ

増分バックアップを有効にすることで、バックアップ戦略を強化することもできます。MarkLogicは最後のフルバックアップ（または前回の増分バックアップ）以降に追加された新しいデータを定期的に保存します。フルバックアップを毎週、増分バックアップを毎日実行するよう設定し、さらにジャーナルアーカイブを実行するようにすれば、データ消失の可能性を最小限に抑えられます。MarkLogicでは、Timestampsファイルをスキャンして増分バックアップで何を保存するかを判断します。増分バックアップはデータのほんの一部を保存するだけなので、フルバックアップと比べてかなり短時間で実行できます。また、増分バックアップでは、ジャーナルアーカイブからフレームを再生する場合に比べて、データの復旧も迅速です。ユーザーは、フルバックアップ、増分バックアップ、ジャーナルアーカイブをさまざまに組み合わせ、必要な復旧精度とストレージ容量の要件のバランスを取ることができます。

インシデントが発生してデータをリストアする必要がある場合、MarkLogicは最新の増分バックアップを探します。ジャーナルアーカイブが有効になっている場合は、ジャーナルフレームの再生により、最後の増分バックアップ以降に更新されたデータが復旧されます。ある特定の時点まで復旧する場合は、その時点以降の最も近い増分バックアップが特定され（可能な場合）、そこからロールバックされます。これは、前のバックアップから先に進む場合、一般的にロールバックの方がジャーナルフレームの再生よりも高速であるためです。

増分バックアップは、ジャーナルアーカイブの機能と似たポイントインタイムリカバリにも対応しています。この機能を有効にするには、管理画面のデータベースのマージポリシーの下にある［retain until backup］設定をオンにします（または、`admin:database-set-retain-until-backup()`を呼び出します）。これにより、削除済みのドキュメントが、バックアップに保存されるまでスタンド内に保持されるようになります。増分バックアップでは、削除済みのドキュメントを残りのデータと一緒に含めることで、前回のバックアップ以降の任意の時点の状態にデータベースを再構築することができます。増分バックアップを使用するポイントインタイムリカバリでは、これらの期間のジャーナルアーカイブを破棄できるため、ストレージのフットプリントを削減できます。

フェイルオーバーとレプリケーション

フェイルオーバーにより、MarkLogicクラスタは、ホストの障害が長期化した場合でも、中断することなく稼働し続けることができます。

クラスタ内のホストが1秒ごとにハートビート情報を交換する理由の1つは、障害がいつ発生したのかを判断することです。設定されたタイムアウト（デフォルトは30秒）が経過してもホストが別のホストからハートビートを受信せず、ほかのホストがそのホストからハートビートを受信したことを示すハートビート情報も受け取らなかった場合、通信可能な状態にないそのホストはクラスタから切断されます。

切断されたホストがEノードであり、フォレストをローカルで一切マウントしていない場合、クラスタ内のその他のホストはすべて通常どおりに稼働し続けることができます。その切断されたホストに対して開始されたリクエストのみがエラーとなり、ロードバランサーはほかのEノードにリクエストを送信することで、障害を検出して伝達できます。再分配された負荷に対処するためにクラスタに必要なのは、ウォームスタンバイまたはホットスタンバイのEノードが十分にあることだけです。

切断されたホストがDノードである場合は、そのDノードでホスティングされているデータを、最近コミットされたトランザクションすべてと一緒にオンラインに戻す手段が必要です。このためのアプローチとして、共有ディスクフェイルオーバーとローカルディスクフェイルオーバーの2つがあります。フェイルオーバーが発生するのは、引き継ぐホストが、クラスタ内のホストの50%以上のクォーラムに属している場合のみです[15]（管理面での詳細については、『Scalability, Availability, and Failover Guide』を参照してください）。

共有ディスクフェイルオーバー

共有ディスクフェイルオーバーでは、1つのクラスタ内の複数のホストが同じフォレストデータにアクセスする必要があります。これは、クラスタファイルシステムであるVeritas VxFS、Red Hat GFS、またはRed Hat GFS2で可能です。すべてのDノードは、同じパスにあるクラスタ内のほかのサーバーでアクセス可能な、ストレージエリアネットワーク（SAN）にそのフォレストデータを格納します。1台のDノードサーバーで障害が発生すると、そのサーバーはクラスタから削除され、SANにアクセス可能なクラスタ内のほかのサーバーが各フォレストを「リモートでマウント」します。これは、サーバーがネットワーク経由でフォレストに接続するネットワークファイルシステム（NFS）でも可能です。MarkLogicクラスタ内の各フォレストは、一度に1台のホストによってしかマウントされない点に注意してください。

共有ディスクフェイルオーバーでは、フェイルオーバーのDノードはディスク上で障害が発生したサーバーと同じデータ（障害が発生した時点までのジャーナルを含

15 フェイルオーバーホストがマジョリティに属すためには、フェイルオーバーに対応するホストがクラスタ内に3台以上必要です。マジョリティが必要ない場合は、クラスタがスプリットブレインシンドロームに直面する可能性があります。これは、ハーフ間をつなぐネットワークで、各ハーフクラスタが自身を生存ハーフであるとみなす状態のことです。

む）を読み取ることができます。この場合、ファイルシステムのサーバー間の一貫性は、クラスタ化されたファイルシステムによって保証されます。各フォレスト設定の一環として、プライマリホストとそのフェイルオーバーホストを設定できます。すべてのフェイルオーバーホストで、自身のフォレストとリモートでマウントされている可能性のあるフォレストすべてに対処するために、十分な予備の稼働能力が必要です。

障害が発生したDノードがオンラインに戻っても、ほかのホストによってリモートでマウントされたフォレストは自動的に再マウントされません。これによって、解決までに時間のかかる問題がプライマリホストで繰り返し発生する場合も、フォレストがホスト間を「ピンポン」のように行き来することはなくなります。プライマリによるフォレストの再マウントが適切である場合、管理者は各フォレストを「再起動」する必要があります。

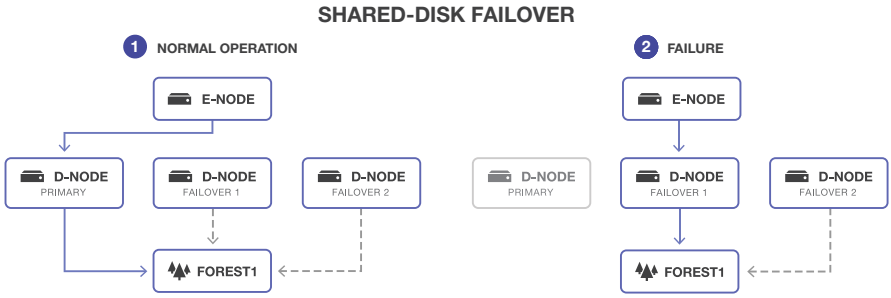


図20: 共有ディスクフェイルオーバーでは、フォレストデータはSANまたはNFSに一元的に格納される**(1)**。1つのDノードで障害が発生した場合、ほかのDノードがフォレストをリモートでマウントして引き継ぐことができる**(2)**

ローカルディスクフェイルオーバー

ローカルディスクフェイルオーバーでは、クラスタ内フォレストレプリケーションを使用します。フォレストレプリケーションにより、1つのフォレストに対するすべての更新が、別のフォレストまたはフォレストセットに自動的に複製されます。各フォレストは、冗長性を目的として、別のディスクセット（通常は安価なローカルディスク）上に物理的に保存されます。

フォレストのプライマリコピーを管理しているサーバーがオフラインになった場合、レプリカフォレストを管理する別のサーバー（データの完全なコピーを含む）が新しいプライマリホストとして処理を進めることができます。障害が発生したホストがオンラインに戻ると、レプリカフォレストで更新されたすべてのデータがプライマリに対して再同期され、再び同期のとれた状態になります。共有ディスクフェイルオーバーと同様、障害が発生したDノードは、データが再同期され、レプリカフォレストが管理者によって再起動されるまで、自動的にプライマリに戻ることはありません。

MarkLogicは、最初の「ゼロデイ」同期のために高速一括同期化を実行することで、フォレストレプリケーションを開始します（管理画面では、そのステータスが［async replicating］となっています）。また、フォレストが長期間オフラインになっている場合もこれを行います。この段階では、フォレストはまだ取り込まれていないため、フェイルオーバーの状況に移行できません。フォレストが同期されたら、MarkLogicはプライマリフォレストからレプリカフォレストにジャーナルフレームを持続的に送信します（管理ステータスは［sync replicating］）。この間、プライマリに問題が発生すると、レプリカフォレストが移行することができます。再生でも各フォレストで同等の結果が得られますが、フォレストの全バイトが同一になることはありません（あるフォレストがマージすることに決めても、ほかは違うこともあります）。複製されたフォレスト間のコミットは同期されておりトランザクショナルであるため、プライマリへのコミットはレプリカへのコミットとなります[16]。

Dノードは通常、6個のプライマリフォレストをホスティングします。すべてのDノードにさまざまなホストのレプリカフォレストが6個ある環境では、少なくとも2つのほかのDノード間でフォレストレプリカを「ストライピング」するとよいでしょう。これによって、プライマリで障害が発生した場合に、フォレスト管理作業が可能な限り多数のマシンに分散されるようになり、ホストに不具合がある間のパフォーマンスへの影響を最小限に抑えることができます。

どのタイプのクラスタ内フェイルオーバーにも長所と短所があります。共有ディスクフェイルオーバーは、ディスクに関して高い効率性を発揮します。ローカルディスクフェイルオーバーは設定が簡単で、安価なローカルディスクを使用でき、クラスタ化されたファイルシステムやフェンシングソフトウェアを必要としません。ローカルディスクフェイルオーバーでは、両方のフォレストが個別にインデックス化とマージを行うため、読み込み負荷が2倍になります。ローカルディスクは、コントローラとスピンドルの追加によりホストが追加されると、自動的に拡張します。その一方、共有ディスクは、SANまたはNASの帯域幅をごく小さく分割するだけです。共有ディスクは、ほかのアプリケーションと帯域幅を奪い合う場合もあります。レプリカフォレストがフェイルオーバー時に稼働できるよう準備ができているため、復旧はローカルディスクの方が高速です。共有ディスクの動作はホストの再起動に似ており、利用可能な状態になる前にジャーナルを再生しなければなりません。

フェイルオーバーを設定する際は、Security、Modules、Triggers、Meters、Schemasといった補助データベース向けにも設定することを忘れないでください。

16　［async replicating］の間、MarkLogicはプライマリフォレストへの書き込みを、ネットワーク経由でレプリカに送られるデータ量の最大50%に制限します。これによって、最終的にレプリケーションが追いつきます。

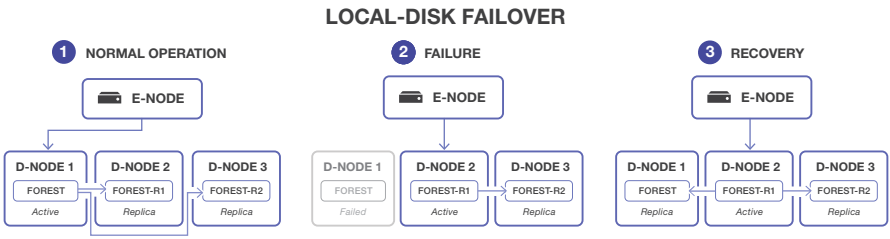


図21: ローカルディスクフェイルオーバーでは、フォレストデータがほかのDノードに自動的に複製される**(1)**。1つのDノードで障害が発生した場合、レプリカがプライマリホストとして作業を引き継ぐ**(2)**。障害が発生したDノードがオンラインに戻ると、レプリカから再同期できる**(3)**

データベースのレプリケーション

データセンターの停電など、クラスタ全体に障害が発生した場合はどうすればよいでしょうか。あるいは、効率性を高めるために、複数のクラスタを複数の地域に分散させたい場合はどうすればよいでしょうか。このようなクラスタ間レプリケーションを可能にするために、MarkLogicにはデータベースのレプリケーション機能があります。

データベースのレプリケーションは、ローカルディスクフェイルオーバーと同様、多数の役割を持ちます（別のクラスタでホスティングされるレプリカフォレストを除く）。管理画面で、またはAPIを介して、まず2つのクラスタを連結します。各クラスタには、通信を開始するための「ブートストラップ」ホストが1つあるいは複数あります。通信は、クラスタ内通信で使用されるものと同じXDQPプロトコルで行われますが、ポートは7999ではなく7998です。データベースのレプリケーションはデータベースレベルで設定されますが、その名前に反し、実際にはフォレストごとに実行されます。通常は、プライマリデータベースとレプリカデータベースのフォレストは名前で対応付けますが、必要に応じて手作業でオーバーライドすることができます。物理レプリケーション処理では、「ローカルディスクフェイルオーバー」のセクションで取り上げたものと同じテクニックを使用します。つまり、まず一括同期化を行い、次にジャーナルフレームを持続的に送信するというものです。

しかし、注目すべき相違点がいくつかあります。第一に、ローカルディスクフェイルオーバーでは、プライマリフォレストとレプリカフォレストが同期をとって更新されていきます。データベースのレプリケーションでは、設定可能なラグタイム（デフォルトで15秒間）があります。これは、レプリカがどの程度後ろに離されると、プライマリが新しいトランザクションコミットを遅らせるかを示すものです。これが必要になる理由は、クラスタ間には通常大きなレイテンシがあり、多くの場合、ネットワークの信頼性が低いことです。レプリカクラスタが完全にダウンしても、プライマリは稼働し続けます（障害点を2倍にしないため）。

第二に、ローカルディスクフェイルオーバーでは、フェイルオーバーの場合を除き、レプリカが使用されることはありません。データベースのレプリケーションでは、レプリカデータベースをオンラインにして、読み取り専用のクエリに対応することができます。このコツで、プライマリクラスタの負荷を減らすことができます。

最後に、各クラスタは独立して管理を行っているため、データベースのレプリケーションは設定の境界を超えて機能します。データベースのインデックス設定をプライマリとレプリカの間で連携させるかどうかは、グローバル管理者次第です。ユーザーアクセスを適切に機能させるには、セキュリティデータベースのコンテンツを連携させる必要もあります。もちろん、これは簡単です。セキュリティデータベースでデータベースのレプリケーションを実行するだけです。

プライマリフォレストは、複数のクラスタ間で複数のレプリカに複製できます。クラスタが相互に複製することすらできますが、各データベースが一度に持つことができるプライマリクラスタは1つだけです。残りは常に読み取り専用になります。

複数の地域でデータを共有したい場合はどうすればよいでしょうか。ロケーションAとBの間で双方向に複製したいと考えているとしましょう。ロケーションAの近くのアカウントはクラスタAのプライマリである必要があり、ロケーションBの近くのアカウントはクラスタBのプライマリである必要があります。また、どちらかのクラスタで障害が発生した場合はすべてのデータを複製しなければなりません。これは可能ですが、データベースが2つ必要です。

データベースをあるクラスタから別のクラスタにフェイルオーバーするには、外部での調整が必要です。なぜなら、各クラスタ内からのビューは世界を正確に読み取るには不十分であり、多くの場合、同時にフェイルオーバーする必要があるほかのコンポーネントがアーキテクチャ全体に存在するからです。外部のモニタリングシステムまたは人間が障害を検出し、災害復旧（ディザスタリカバリ）を開始するかどうかの判断を下す必要があります。このために、DNSやロードバランサーなどの外部ツールを使用してトラフィックを適宜ルーティングします。また、レプリカが新しいプライマリになるように、データベースのレプリケーション設定を変更するようMarkLogicを管理します[17]。

17 追加情報：クラスタを連結すると、そのタイムスタンプが同期されます。

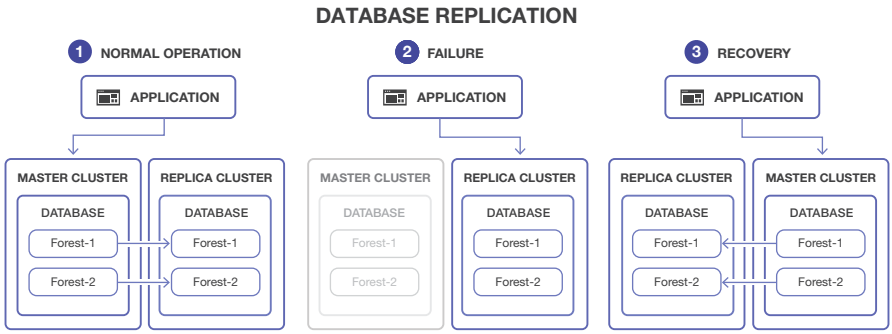


図22: データベースのレプリケーションによって複数のクラスタが連結され、それらの間でフォレストデータがコピーされる**(1)**。マスタークラスタで障害が発生した場合、アプリケーションがレプリカの使用に切り替える**(2)**。元のマスターがオンラインに戻ると、レプリカとして作業を引き受けることができる**(3)**

「同時」と「ノンブロッキング」

各アプリケーションサーバーには「Multi-Version Concurrency Control」と呼ばれる設定オプションがあります。これは、ロックなしの読み取り専用クエリに関して、最新のタイムスタンプを選択する方法を制御するものです。[contemporaneous]（デフォルト）に設定すると、MarkLogicはトランザクションが1つでもコミットされたと認識されている最新のタイムスタンプを選択します。ただしその場合、同じタイムスタンプに、まだ完全にコミットされていないその他のトランザクションがある可能性があります。クエリはタイムリーな結果を得ることができますが、トランザクションが完全にコミットするまでの待機をブロックする可能性があります。[nonblocking]に設定すると、MarkLogicはすべてのトランザクションがコミットされたと認識されている最新のタイムスタンプを選択します。たとえ、別のトランザクションがコミットされた、わずかに遅いタイムスタンプがある場合でも同様です。クエリはトランザクションの待機をブロックしませんが、タイムリーな結果は得られない可能性があります。

「ホット」レプリカのあるデータベースのレプリケーションでは、レプリカデータベースに対して実行されているアプリケーションサーバーを[nonblocking]として設定するとよいでしょう。これによって、プライマリフォレストからトランザクションが流れ込んでくるときに過度に待機する必要がなくなります。そうしなければ、プライマリが不整合な状態でデータを受け取っていないかどうか永遠に待ち続けることになり得ます。これはデフォルトではないので、管理者はこれをninし来ておくことが重要です。通常の場合，プライマリデータベースに対して実行しているアプリケーションサーバーの設定は[contemporaneous]のままにしておくのがベストです。

フレキシブルレプリケーション

データベースのレプリケーションでは、ジャーナルフレームを送信することでプライマリデータのトランザクション面で一貫性のある正確なコピーを別のデータセンターに保持しようとしますが、フレキシブルレプリケーションでは、ドキュメントを送信することでシステム間のカスタマイズ可能な情報共有を可能にします。

技術的には、フレキシブルレプリケーションは、非同期、非トランザクショナル、トリガーベース、ドキュメントレベルという特性を備えたクラスタ間のレプリケーションシステムであり、そのベースにあるのはContent Processing Framework（CPF）で

す。フレキシブルレプリケーションシステムを有効にすると、ドキュメントが変更されるたびトリガーが発せられ、そのトリガーコードによって、ドキュメントの変更内容がドキュメントのプロパティシートに記録されます。プロパティシートに「変更あり」とマークされたドキュメントは、HTTPプロトコルを使用してバックグラウンドプロセスでレプリカクラスタに転送されます。ユーザーの設定内容に応じて、ドキュメントを（レプリカに対して）プッシュまたは（レプリカによって）プルすることができます。

フレキシブルレプリケーションでは、オプションのプラグインフィルタモジュールをサポートしています。これこそ、このレプリケーションの「フレキシブル」なところです。フィルタは、ドキュメントの複製時に、コンテンツ、URI、プロパティ、コレクション、パーミッション、その他ドキュメントに関するあらゆるものを修正できます。例えば、プライマリの単一のドキュメントをレプリカでは複数のドキュメントに分割できます。あるいは、ドキュメントを単純にフィルタリングして、複製すべきドキュメントはどれで、一部のみを複製すべきドキュメントはどれかを判断できます。フィルタを使用すれば、レプリケーションの過程でコンテンツを完全に変換することも可能です。例えば、XSLTスタイルシートなどを使用して、あるスキーマから別のスキーマに自動的に調整することができます。

フレキシブルレプリケーションのオーバーヘッドは、ジャーナルベースのデータベースのレプリケーションの場合よりも大きくなります。スピードを保つには、タスクサーバーのスレッドを増やす（より多くのCPF作業を同時にこなせる）、ロードバランサーで負荷をターゲットに分散する（より多くのEノードが参加できる）、クラスタ間のネットワークをより太くする（配信スピードが高まる）などの対策があります。

フレキシブルレプリケーションの詳細については、『Flexible Replication Guide』およびflexrep関数に関するドキュメントを参照してください。

クエリベースのフレキシブルレプリケーション

どのドキュメントをどのノードに複製し、パーミッションに基づいて誰が何を表示できるかを細かく制御したい場合は、フレキシブルレプリケーションをMarkLogicのアラート機能と結び付けることができます。この機能は「クエリベースのフレキシブルレプリケーション（QBFR）」と呼ばれます。QBFRが有用なのは、現場でラップトップコンピュータを使うときのように、大きな中央サーバーと小さなシステムが混在する混合ネットワークです。すべてのシステムがMarkLogicサーバーを実行しますが、ラップトップでは、メインサーバーに格納されているドキュメントのサブセットのみを必要とします。QBFRのユースケースには、軍隊の配備（現場の部隊がその場所に固有の情報のみを必要とする）や科学分野への応用（離れた場所にいる研究者が自分の研究分野に関連する情報のみを必要とする）などがあります。

前述したように、アラート機能を使用すると、ユーザーが関心を持っているデータセットのドキュメントを指定するクエリを定義できます。このクエリをQBFRで使用すると、マスタークラスタからレプリカクラスタ（ラップトップの例のように、シンプルな1ノードのクラスタの場合もあります）に複製されるドキュメントを定義できま

す。例えば、軍隊の例のクエリでは地理空間座標を定義し、科学分野の例のクエリでは技術的なキーワードを定義できます。現場にある各レプリカクラスタにはクエリが関連付けられています。中央のマスタークラスタでドキュメントが挿入または更新されると、リバースクエリが実行され、保存されたクエリのうちどれがドキュメントと一致するかが判断されます。一致するクエリは、ドキュメントの複製先クラスタを判断します。保存された各クエリはユーザーにも関連付けられており、レプリケーションをロールベースで制御できます。

一般的に、プルベースのレプリケーションがQBFRで好まれる操作方法です。これにより、接続が制限されたり途絶えたりすることがあるフィールドクラスタが、ネットワークアクセスのあるときにドキュメントを取得できるようになります。レプリケーションは双方向にすることもできます。フィールドクラスタは接続時にマスターとして動作し、収集したすべての新しいデータをホームベースに送り返します。例えば、軍隊の現場部隊は、偵察データを送り返し、データの配布は追加のクエリで制御することができます。

リバランス

MarkLogicのリバランスでは、データベースを構成する各フォレストのドキュメント数が似たようなものになるように、データベース内のコンテンツを再分配します。フォレスト全体にドキュメントを均一に分散させることで、ホスト間の並行性を活用できます。1つのフォレストにドキュメントの大部分を置くことを許可されたデータベースは、ほとんどの処理をそのそのフォレストのホストで行いますが、ドキュメントを再配分することで、ホスト全体に処理を均一に分散させることができます。こうして、リバランスによりハードウェアをより効率的に使用し、クラスタのパフォーマンスを改善できます。

リバランスは、フォレストの追加時や廃止時など、データベースの再設定によってトリガされます[18]。フォレストを分離して削除する予定がある場合は、まずフォレストを廃止します。これによって、そのフォレストのすべてのコンテンツが、残りのフォレスト間でまず再配分されます。フォレストを廃止せずに分離して削除すると、その削除されたフォレストのコンテンツはすべて失われます。

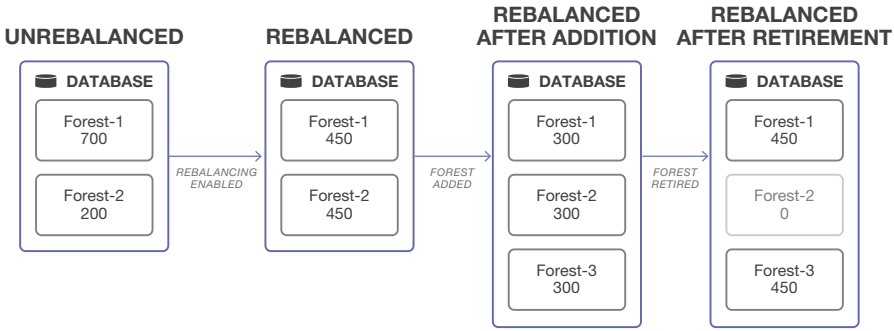


図23: リバランスにより、ドキュメントがデータベースのフォレスト全体に均一に配分され続ける

18 いかに積極的にリバランスを行うかを制御するには、スロットル値を設定します。この値は、システムリソースの利用に関するリバランサーの優先事項を定めるものです。

割り当てポリシー

リバランスは、割り当てポリシーに基づいて機能します。このポリシーは、どのドキュメントがどのフォレストに入るのかを決めるルールセットです。データベースの割り当てポリシーは、リバランスだけでなく、最初の読み込み段階におけるドキュメントのフォレストへの配分方法についても適用されます。

割り当てポリシーは、パフォーマンスを改善するために、複数のホスト間でデータを水平方向に配分する方法を定義します。これは、MarkLogicがデータベース内のドキュメントを複数のフォレスト間に割り当てる際の動作です。これらのフォレストは、異なるホスト上に存在します。しかし、MarkLogicを単一のホスト上で実行していても、ドキュメントを複数のフォレストに分散することでパフォーマンスが向上します。なぜなら、ホストのマルチコアで並列処理を活用できるためです。

割り当てポリシーには、バケット、レガシー、統計、レンジの4種類があります。

バケットポリシー（デフォルト）では、アルゴリズムを使用してドキュメントのURIを16,000個の「バケット」のひとつにマッピングします。各バケットには、フォレストが関連付けられています（バケットをフォレストにマッピングするテーブルは、迅速な割り当てを実現するためにメモリ内に格納されます）。このように、クラスタ内のフォレストの数よりもずっと多い大量のバケットを採用することで、再配分時に転送されるコンテンツの量を最小限に抑えることができます[19]。

レガシーポリシーでは、旧版のMarkLogicリリースが読み込みの際にドキュメントの割り当てに使用していたものと同じアルゴリズムを使用します。ドキュメントは、そのURIのハッシュに基づいて個別に配分されます。レガシーポリシーはバケットポリシーよりも多くのコンテンツを移動するため、効率性に劣ります（新しいフォレストを1つ追加すると、ほぼすべてのドキュメントの割り当てが大きく変わります）が、後方互換性のために用意されています。バケットポリシーとレガシーポリシーは、フォレストセットが同じならばドキュメントは最終的に常に同じ場所に置かれるため、確定的です。

統計ポリシーは、データベースの全フォレストの中で、ドキュメントの数が現在最も少ないフォレストにドキュメントを割り当てます。バケットポリシーに比べると、移動が必要なコンテンツはずっと少なくなりますが、確定的ではありません。つまり、ドキュメントが最終的にどこに置かれるかが、特定の時点のフォレスト内でのコンテンツ配分に応じて決まります。このため、統計ポリシーは、ロックを［strict］にした場合のみ使用できます[20]。

19 M個のバケットがあるとします（Mはかなり大きい数）。また、新しいフォレストが、すでにN個のフォレストを持つデータベースに追加されるとします。再びバランスのとれた状態にするために、バケットポリシーではデータのうち「N x (M/N – M/(N+1)) x 1/M = 1/(N+1)」を移動する必要があります。これはほぼ理想的な状態です。ただし、Mの値が大きいほど、（バケットからフォレストへの）マッピングの管理にかかるコストが高くなります。

20 ロックを［strict］にすると、MarkLogic Content Pump（MLCP）で高速読み込みオプションを使用できなくなります。

レンジポリシーは、階層型ストレージと関連付けて使用され、レンジインデックスに基づいてドキュメントを割り当てます。つまり、データベース内のドキュメントを、作成時間や更新時間などに基づいて配分できるということです。これにより、最近使用したドキュメントや頻繁にアクセスされるドキュメントを高速なソリッドステートメディアに、古いドキュメントを低速のディスクベースメディアに格納できます。詳細については、「階層型ストレージ」セクションを参照してください。レンジポリシーでは、通常、各レンジ内に複数のフォレストがあります。ドキュメントは、統計ポリシーで使用したアルゴリズムに基づいて、レンジ内のフォレストに割り当てられます。

データの移動

リバランスのバックグラウンドでは、フォレスト間でプログラム的にドキュメントを移動する方法に似たステップが実行されています。リバランスイベント中、ドキュメントは既存のフォレストから削除され、新しいフォレストに挿入されます。これら両方のステップは、データベースの一貫性を維持するために、単一のトランザクションで実行されます。

効率を高めるために、ドキュメントは一度に1つずつではなく、バッチでまとめて転送されます。各バッチのサイズは、割り当てポリシー（およびポリシーの運用に必要なリソース）によって異なります。さらに、統計ポリシーの場合、リバランスをトリガーするために、フォレスト間で相違のしきい値に達する必要があります。これにより、フォレストの数に大きな違いがない場合、不要なリバランスが実行されるのを回避できます[21]。

HADOOP

MarkLogicは、Apache Hadoop、特に、HadoopスタックのMapReduce部分を利用してデータの一括処理を強化しています。Hadoop MapReduceエンジンは、多数のノード間でJavaベースの計算量の多いプログラムを実行するためによく使われる手法です。MapReduceという名前は、処理全体を2つのタスクに分割するところから来ています。最初の「map」タスクでは、キー/バリューペアを取り込んで、一連の中間的なキー/バリューペアを出力します。2番目の「reduce」タスクでは、「map」段階の各キーとそのキーに対して作成された値すべてを取り込んで、最終的な一連の値を出力します。マップの実行の並列化を容易にし、マシン間で並行してタスクを削減できるこのシンプルなモデルは、一方で、多数の複雑なワークロードを処理するのに十分な堅牢性を備えていることが分かっています。MarkLogicでは、一括処理にMapReduceを使用します。つまり、大規模なデータ読み込み、変換、エクスポートが対象です（ライブのクエリや更新の実行にはHadoopを使用しません）。

21 フォレストの平均フラグメント数がAVG、フォレストの数がNであるとします。しきい値は「max(AVG/(20*N),10000)」として計算されます。つまり、フォレストAのフラグメント数が「AVG + しきい値」より多く、かつフォレストBのフラグメント数が「AVG – しきい値」より少ない場合のみ、AとBの間でデータが移動されます。

技術的なレベルで、MarkLogicはHadoop向けの双方向コネクタを提供し、サポートしています[22]。このコネクタはオープンソースで、Javaで記述されており、メインのMarkLogicサーバーパッケージとは別に提供されています。このコネクタには、MarkLogicクラスタとHadoopクラスタの間でアクティビティを調整するためのロジックが含まれています。これにより、マシン間、ノード間ですべての接続が直接行われるようになり、どのマシンもボトルネックになることはありません。

MarkLogic Hadoopのジョブは通常、次の3つのパターンのいずれかに従います。1. ファイルシステムやHDFS（Hadoop Distributed File System）などの外部ソースのデータを読み取り、MarkLogicにプッシュします。これは、従来からのETL（extract-transform-load）ジョブです。データは、必要に応じてHadoop内で処理の一環として変換（標準化、非正規化、非複製化、変形）できます。2. MarkLogicのデータを読み取り、ファイルシステムやHDFSなどの外部の保存先に出力します。これは、従来からのデータベースエクスポートです。3. MarkLogicのデータを読み込み、結果をMarkLogicに書き戻します。この方法では、一括変換を効率的に実行できます。例えば、MarkMail.orgプロジェクトで、メールのヘッダーに含まれるIPアドレスに基づき、メッセージ部分に緯度/経度で場所を示す「ジオタグ」を付けたいとします。IPから場所を判断するためのロジックはJavaで記述されており、小さなHadoopクラスタで実行されているHadoopジョブのすべてのメッセージに対して並列実行されます。

MarkLogicからデータを読み取るMapReduceジョブには、読み取りの効率を高めるためのいくつかのコツが取り入れられています。このジョブは、各フォレストから、フォレスト内にどのようなドキュメントが存在するかのマニフェストを受け取り（オプションで、アドホッククエリの制約に一致するドキュメントまたはサブドキュメントのセクションに制限可能）、ドキュメントを「スプリット」に分割して、これらのスプリットをHadoopノード間で並列実行可能なマップジョブに割り当てます。マップジョブを実行するHadoopノードと、フォレストをホスティングするマシンの間では、常に通信が行われています（メモ：Hadoopノードが、フォレストをホスティングするMarkLogicノードと通信するには、MarkLogicノードにオープンなXDBCポートがなければなりません。つまり、Eノード/Dノードの組み合わせである必要があります）。

コネクタコードも、あまり知られていない「unordered」機能を内部的に使用して、ディスクに格納された順序でドキュメントを引き出します。`fn:unordered($sequence)`関数は、シーケンス内のアイテムの順番は重要ではないというヒントをオプティマイザーに与えます。MarkLogicはそのライブラリを使用して、（内部のフラグメントIDを増やすことによって）ディスクに格納されたときと同じ順番で結果を並べます。その結果、ディスクのパフォーマンスが向上し、連続的な読み取りが最適化されます。

22 MarkLogicは特定のバージョンのHadoopとオペレーティングシステムのみをサポートしています。詳細については、ドキュメントを参照してください。

MarkLogicにデータを書き込むMapReduceジョブにも、パフォーマンスを高めるためのコツが取り入れられています。「reduce」のステップでドキュメントをデータベースに挿入し、コネクタがそれを安全とみなした場合、この挿入処理では、そのフォレストが存在するホストに対して直接通信を行いさえすればよいように、フォレスト内配置を使用します。

管理者の観点から言えば、Hadoopプロセスを各MarkLogicノードに配置するか別個にするかは、ワークロードによって決まります。コロケーションによりMarkLogicとMapReduceのタスクの間のネットワークトラフィックは減りますが、ホストに対する計算およびメモリの負担はさらに重くなります。

MarkLogic Content Pump（MLCP）については後述しますが、これはHadoopベースのコマンドラインプログラムで、一括インポート、エクスポート、データコピーといった作業の管理を目的としています。MarkLogicでは、データベースコンテンツの格納にHadoop Distributed File System（HDFS）も利用できます。これは、階層型ストレージの展開で役に立ちます。これについては、次のセクションで説明します。

Hadoopのない世界

Hadoopがない場合は、どうなるでしょうか。同じ作業を実行することはできますが、通常は簡略化するために、ユーザーは1台のマシンから読み込み、1台のマシンから一括変換を実行し、1台のマシンにエクスポートをプルダウンします。これによって、パフォーマンスが1台分のクライアントのパフォーマンスに制限されます。Hadoopを使用すると、任意の大型クラスタマシンをクライアントとして機能させ、すべてのMarkLogicノードに対して同時に通信できます。ここれによって、MarkLogicに出入りするデータフローのスピードを高められます。

階層型ストレージ

すべてのストレージメディアが同等に作られているわけではありません。高速ではあるけれど高価なストレージは、アクセス頻度の高い高価値ドキュメントに最適です。クラウド内のストレージなど、低速ではあるけれど安価なストレージは、アクセスはめったにされないが、履歴や監査の目的で維持する必要がある古いデータに適しています。

MarkLogicの階層型ストレージを使用すると、データベースのドキュメントを最適なメディアに自動的に格納できます。ドキュメントは、レンジインデックスのプロパティに基づいて、さまざまな場所に保存できます。例えば、ドキュメントの作成日や最終更新日などのプロパティです。これは、アルゴリズムを使用してフォレスト全体に均一に分散する、一般的なドキュメント割り当て方法とは対照的です。アクセスのニーズに応じてデータベースのドキュメントを格納する階層型ストレージは、より低いコストで高いパフォーマンスをユーザーに提供します。

階層型ストレージをセットアップするには、データベースのフォレストをパーティションに分類します。各パーティションには、レンジインデックスの開始値および終了値とストレージの場所が関連付けられています。例えば、最終更新日をインデックスとして使用している階層型ストレージデータベースがあるとします（MarkLogicの［maintain last modified］データベース設定で、ドキュメントのこの値を自動的にトラッキングすることができます）。最近修正されたデータ用に「Newest」パーティションを定義し、レンジを2014年～2016年に、場所をSSD上のディレクトリに設定します。2010年～2013年のデータを処理する「Intermediate」パーティションは、ローカルディスク上の通常のディレクトリに設定します。2000年～2009年のデータを処理する「Archival」パーティションは、クラウドストレージに設定します。

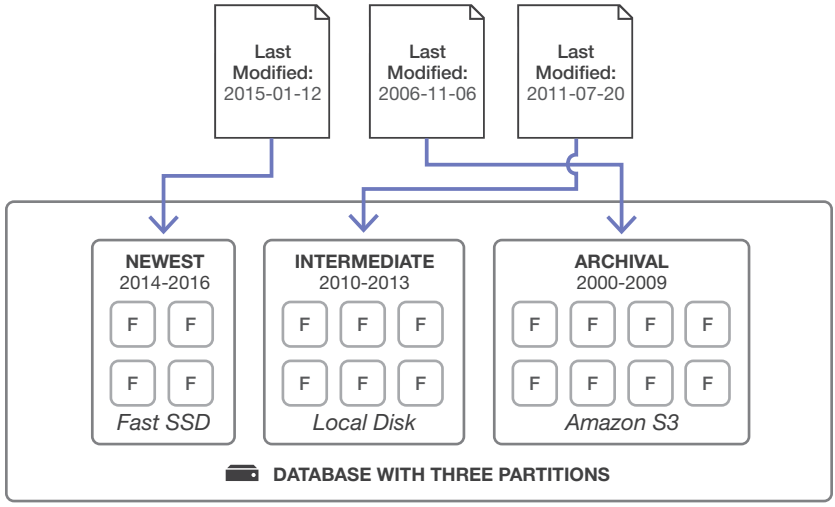


図24: 階層型ストレージを使用すると、データベースのフォレストをパーティションに分類してから、レンジインデックスの値に基づいてドキュメントをパーティションに挿入できる

ドキュメントの挿入を実行すると、そのレンジインデックス値に基づいてドキュメントがパーティションにマッピングされます。今回の例では、最終修正日が2015年1月12日のドキュメントは「newest」パーティションのフォレストに割り当てられ、SSDに保存されます。パーティションに複数のフォレストが含まれている場合、そのパーティションでドキュメントの数が最も少ないフォレストにドキュメントが割り当てられます。

階層型ストレージには、長期間にわたってデータを保持するためのさまざまな機能があります。

- 古くなったドキュメントを低コストの階層に移動する場合、異なるストレージロケーションに**パーティションを移行**できます。組み込み関数とRESTエンドポイントにより、ローカルのロケーションと共有ストレージのロケーションの間でも、これを簡単に実行できます

- データベースが大きくなるのに応じて、パーティションに**フォレストを追加**できます。MarkLogicはフォレスト内のデータをリバランスし、効率的に機能し続けるようにします（詳細については、「リバランス」のセクションを参照してください）

- 同様に、ストレージのニーズが減った場合は、パーティションの**フォレストを廃止**することができます。廃止されたフォレスト内のドキュメントは、パーティション内の別のフォレストに再配分されます

- パーティションの**レンジを再定義**することができます。これにより、パーティション内で、およびパーティション間で、ドキュメントがリバランスされます

- パーティションは**オンラインとオフライン**にすることができます。オフラインのフォレストは、クエリ、更新、その他の多くの処理から除外されます。パーティションをオフラインにしてもデータをアーカイブし、RAM、CPU、ネットワークリソースを節約できますが、再度クエリを実行する必要がある場合、すぐにオンラインに戻すことができます

古くなったドキュメントを安価なストレージに移動する場合など、ドキュメントを別の階層に移動するときは、リバランスよりも移行の方が効率的です。MarkLogicの移行処理では、パーティション内のフォレストすべてをまとめてコピーします。リバランスが関与する更新（例えば、パーティションレンジの再定義）は、ずっと小さなバッチでドキュメントを移動するため、計算コストが高くつきます。

また、レンジインデックスによるドキュメントのパーティション化は、データベースに高速データディレクトリを設定することとは異なります。これは、高速で高価なSSDなどのストレージを利用するための別の手段です。高速データディレクトリの定義は、データベースのセットアップ時のオプションです。これを定義すると、MarkLogicは重要な処理に高速ストレージを活用し、全体的なシステムパフォーマンスを改善することができます。詳細については、「フォレスト内のストレージタイプ」を参照してください。

スーパーデータベース

MarkLogicの階層型ストレージ戦略のもうひとつの側面は、スーパーデータベースです。スーパーデータベースを使用すると、異なるストレージメディアに格納された別のデータベースにドキュメントを分類でき、かつ、それらのドキュメントに対して単一のユニットとしてクエリを実行できます。クエリを適切に機能させるには、スーパーデータベースとサブデータベースの設定が同じでなければなりません。複数のデータソースに対してまとめてクエリを実行し、その結果を集約する処理は、横串検索と呼ばれます。

例えば、高速で高価なストレージ上にアクティブなドキュメントのデータベースAがあり、低速で安価なストレージ上にアーカイブドキュメントのデータベースBがあるとします。ここでスーパーデータベースを作成し、データベースAおよびデータベースBと

関連付けます（スーパーデータベース自体にドキュメントのフォレストを含めることもできますが、関連付けられたサブデータベースとは別にこれらのフォレストに対してクエリを実行できないため、推奨できません）。このように環境を整えることで、データベースAの高価値ドキュメントのみに対してクエリを実行することも、データベースBのアーカイブドキュメントのみに対してクエリを実行することも、スーパーデータベースに対してクエリを実行することでコーパス全体に対してクエリを実行することもできます。

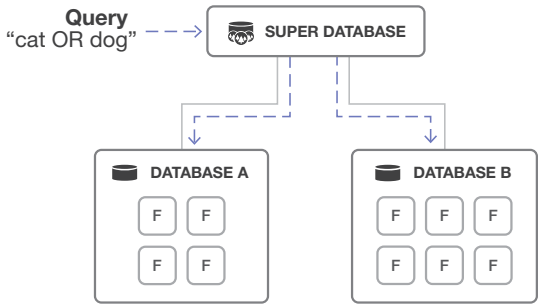


図25: スーパーデータベースを使用すると、複数のデータベースに対して単一のユニットとしてクエリを実行できる

スーパーデータベースには制限があります。サブデータベースにあるドキュメントに対して、スーパーデータベースを介して更新を実行することはできません。そのドキュメントがあるサブデータベースに対しては、直接更新を実行しなければなりません。また、スーパーデータベースの下に、複数のレベルのサブデータベースを指定することはできません。これは、循環参照（2つのデータベースが互いを参照する状態）を回避するためです。

HDFSとAMAZON S3

階層型ストレージの重要なポイントは、あまり使われていないデータを低コストのストレージメディアに保存する機能です。低コストのオプションとして、Hadoop Distributed File System（HDFS）とAmazon S3（Simple Storage Service）の2つがあります。

HDFSはHadoopのストレージ部分です。HDFSはオープンソースであり、コモディティハードウェア上で実行できるため、SAN（ストレージエリアネットワーク）やNAS（ネットワークアタッチトストレージ）などの従来の共有ストレージソリューションよりも安価にデータを格納できます。HDFSを使用するようにMarkLogicを設定すると、「hdfs:」プレフィックスを使用するデータベースのストレージディレクトリとしてHDFSを指定できるようになります。

Amazon S3は、アマゾンウェブサービスが提供するクラウドベースのストレージサービスです。S3のユーザーは、RESTなどの各種インターフェイスを介して、サービスに格納されているドキュメントを操作できます。S3のストレージは大規模であるため、

価格が安く設定されています。2013年、アマゾンは2兆を超えるオブジェクトがS3に格納されていることを報告しました。S3をMarkLogicクラスタにセットアップするには、S3認証情報を送信します。この情報は、セキュリティデータベースに保存されます。その後、「s3:」プレフィックスを使用するデータベースのストレージディレクトリとしてS3の場所を指定できます。

MarkLogicのストレージにS3を使用すると安価ですが、パフォーマンス面でのトレードオフがあります。S3に格納されたデータは「結果整合性」を保ちます。つまり、ドキュメントを更新した後、そのドキュメントを読み取り、変更内容を確認できるようになるまでに、遅延期間が生じます。このため、S3はドキュメントの更新やジャーナルには適しておらず、共有ディスクフェイルオーバーにも使用できません。S3の推奨分野は、ジャーナル処理を必要としない読み取り専用のフォレストです。階層型ストレージの場合は、アクセスがほとんどなく、二度と更新されないドキュメントを含むアーカイブパーティションに適しています。S3は、バックアップストレージにも適したオプションです。

集計関数とC++のUDF

集計関数は、長いデータシーケンスから特異値を計算します。一般的な集計関数は、count、sum、mean、max、minです。あまり一般的でないものは、median（中央値）、percentile、rank（順番に並んだ数値の配列）、mode（最も頻繁に発生するもの）、variance（どの程度の範囲まで値セットが分散するかを定量化）、standard deviation（分散の平方根）、covariance（2つの確率変数が互いにどの程度変化するか）、correlation（2つのデータセットが互いにどの程度近く増減するかを定量化）、linear model calculation（従属変数と独立変数の間の関係を計算）です。

MarkLogicには、これらの集計関数すべてとその他の関数が用意されています。あらゆるシーケンスに対してこれらの関数を実行できますが、通常はレンジインデックス内の値のシーケンスが対象です。関数が1つのレンジインデックスに対して機能することもあれば（countやstandard deviationなど）、複数の場合もあります（covarianceやlinear model calculationなど）。複数のレンジインデックスに対して実行すると、Dノード間、フォレスト間、さらにはスタンド間にわたって（基盤となるアルゴリズムで許容される範囲まで）並列処理されます。

自分がよく使用する集計関数がドキュメントに記載されていない場合はどうすればよいでしょうか。MarkLogicは、ユーザー定義関数（UDF）と呼ばれるカスタム集計関数を定義するためのC++インターフェイスを備えています。UDFをダイナミックリンクライブラリに構築し、ネイティブプラグインとしてパッケージして、そのプラグインをMarkLogicサーバーにインストールすることができます。UDFは、ネイティブの集計関数と同様の並列実行が可能です。実際、前述した高度な集計関数は、エンドユーザーに公開されているものと同じUDFフレームワークを使用して実装されたものです。 MarkLogicでは、この実行モデルを「インデータベースMapReduce」と呼びます。Eノードで最終結果が得られるまで、処理がフォレスト間にマッピングさ

れ、分割処理されるからです。UDFを記述する際は、メインのロジックをmap()およびreduce()関数に含めます。

低レベルのシステム制御

システムを拡張するとき、いくつかの管理設定を調整できます。これらの設定はMarkLogicの動作を制御するもので、賢明に使用すれば、読み込みとクエリのパフォーマンスを向上させることができます。

1. **[Last Modified]オプションと[Directory Last Modified]オプション：**これらのオプションは、変更があるたびに各ドキュメントと各ディレクトリのプロパティシートを更新することで、それぞれの最終修正時間をトラッキングします。あると便利な情報ですが、ドキュメントを更新するたびに余計なフラグメントの書き込みが必要になります。これらのオプションをオフにすると、このちょっとした処理を省略できます。MarkLogic 7以降、これらのオプションはデフォルトで無効になっていますが、旧バージョンや旧バージョンからアップグレードしたシステムでは、有効になっていることが多くあります

2. **[Directory Creation]オプション：**[automatic]に設定すると、ディレクトリエントリが自動的に作成されます。つまり、パス/x/y/z.xmlを指定してドキュメントを作成すると、/x/および/x/y/のディレクトリエントリが作成されます（存在しない場合）。[manual]に設定すると、これらのディレクトリを必要に応じて自分で作成しなければなりません。ファイルシステムと似た構造や動作がデータベースに必要とされるWebDAV経由でデータベースにアクセスするときは、ディレクトリは重要です。また、ディレクトリベースのクエリを実行している場合も重要です。しかし、それ以外の状況では必要ではありません。ディレクトリの作成を[manual]に設定すると、指定されたディレクトリがすでにパスにあるかどうかを確認し、存在しない場合はそれらを作成するという読み込みのオーバーヘッドを排除できます。このオプションは、MarkLogic 7以降、デフォルトで無効になっています

3. **[Locking]オプション：**MarkLogicのロック機能は、データベースに読み込まれるドキュメントに固有のURIを割り当てます。このとき必要な設定は、データベースの割り当てポリシーによって異なります（「リバランス」セクションを参照）。デフォルトのバケットおよびレガシーポリシーには、既存のドキュメントのURIのみの一意性をチェックする[fast]ロックが必要です。統計およびレンジポリシーには[strict]ロックが必要です。これは、新しいドキュメントと既存のドキュメント両方のURIをチェックするものです。一括読み込みでよく見られるように、読み込むドキュメントに一意のURIがあることを確信できる場合は、この制限を緩和し、ロックを[off]にすることができます

4. **[Journaling]オプション：**MarkLogicでは3つのジャーナル処理オプションをサポートしています。[strict]オプションでは、トランザクションがジャーナル化され、ジャーナルがディスクに完全にフラッシュされるまで、そのトランザクションがコミットされたとみなしません。これによって、MarkLogicサーバープロセスの障害、ホストオペレーティングシステムのカーネルの障害、ホストハードウェアの障害から保護できます。このオプションは、共有ディスクフェイルオーバー用に設定されたフォレストで常に使用されます。[fast]オプションは、トランザクションがジャーナル化されればコミットされたとみなしますが、オペレーティングシステムがジャーナルをディスクに完全に書き込んでいない可能性があります。[fast]オプションは、MarkLogicサーバープロセスの障害からは保護できますが、ホストオペレーティングシステムのカーネルの障害やホストハードウェアの障害には対応していません。[off]オプションでは一切ジャーナル処理を行わず、MarkLogicサーバープロセスの障害、ホストオペレーティングシステムのカーネルの障害、ホストハードウェアの障害からの保護も行いません。一括読み込み時など、データを復旧できる特殊なケースでは、ジャーナル処理を一時的に[off]に設定し、ディスクアクティビティとプロセスのオーバーヘッドを削減するとよいでしょう

5. **Linux HugePagesとTransparent HugePages：**Linux HugePagesは、2メガバイトのメモリブロックです（通常のページはわずか4キロバイト）。サイズが大きいだけでなく、メモリにロックされているため、ページアウトできません。HugePagesを使用することで、MarkLogicはより効率的にメモリにアクセスし、インメモリデータ構造をメモリ内に確実に固定することもできます。MarkLogicでは、Linux HugePagesを物理メモリの3/8のサイズに設定することを推奨しています。また、Red Hat 6ではデフォルトで有効になっているものの、まだバグが見られるTransparent HugePagesの使用は推奨していません。MarkLogicで問題が検出されると、この趣旨の助言がErrorLog.txtに記載されます

6. **スワップスペース：**MarkLogicプロセスが利用可能なすべてのメモリを使用している場合、OSはスワップスペースを一時メモリストアとして使用します。この状況になると、処理速度が非常に遅くなります。スワップスペースを使用すると、OSがより多くのメモリを割り当てることができない場合にクラッシュするのではなく、時間の経過とともにパフォーマンスが徐々に低下します。ハードウェアのRAMが32ギガバイト未満の場合、スワップ用に1xのRAMを使用します。ハードウェアのRAMが32ギガバイト以上の場合、32ギガバイトのRAMを使用します

7. **Red Hat Linuxのdeadline I/Oスケジューラー：**deadline I/Oスケジューラーはパフォーマンスを改善し、処理でリソースが不足したり完了しないことがないようにします。Red Hat Linuxプラットフォームでは、deadline I/Oスケジューラーが必須です

8. **フォレスト：**フォレストは多くの点で互いから独立しているため、マルチコアのシステムでは、フォレストの数を増やして並列実行を増やすことでメリットを得られることが多くあります。また、別のコアで実行されている別のスレッドから、フォレストに対して並列クエリを実行することもできます。フォレスト内のスタンドに対しても、別個のスレッドから並列クエリを実行することは可能ですが、マージ後にスタンドが1つだけになると並列処理は不可能になります。このため、展開時に十分な数のフォレストを設定しておくことが重要です

MarkLogicのパフォーマンスを最適化するベストプラクティスの詳細については、ホワイトペーパー『Performance: Understanding System Resources』を参照してください。

コア以外の技術

以上、MarkLogicの内部構造、つまりシステムのコア部分について説明してきました。MarkLogic関連のエコシステムには、実に数多くの優れた重要な技術が存在します。公式にサポートされているものもあれば、オープンソースのものもあります。この最後のセクションでは、その一部を取り上げます。

CONFIGURATION MANAGER

Configuration Managerは、管理者以外のユーザーと共有するのに最適な、システム設定の読み取り専用ビューを提供します。また、マシン間でアプリケーションサーバーおよびデータベースの設定を移動するためにこれらの設定を簡単にエクスポートしてインポートできる、パッケージング機能もあります。

モニタリング

モニタリングツールスイートにより、MarkLogicサーバークラスタ内の状況を示すあらゆる種類のパフォーマンス統計情報をトラッキングできます。これらには、サーバーの処理に関するリアルタイムの情報と履歴情報を表示するwebベースのインターフェイスと、人気の高いオープンソースのモニタリングツール、Nagios向けのプラグインが含まれます。パフォーマンスに関するデータはMetersデータベースに格納され、REST API経由でも利用できます。これにより、独自のカスタムモニタリングアプリケーションを構築できます。

MARKLOGIC CONTENT PUMP（MLCP）

MarkLogic Content Pump（MLCP）は、インポート、エクスポート、データコピーの各タスクに対応したオープンソースのコマンドラインツールです。MarkLogicでサポートされ、Javaドリブンで、Hadoopがベースとなっています。ローカル環境でも（1台のマシン上）、分散環境でも（Hadoopクラスタ上）実行できます。データのサイズが大きくなっていくと、インポート/エクスポート/コピーの各ジョブを並列実行できるこれらのツールが役立ちます。また、Hadoopは大規模なJava並列処理を実行するための標準的な手法となっています。

MLCPは、通常のXML/テキスト/バイナリのドキュメント、区切り文字テキストファイル、集約XMLファイル（分割すべき要素を繰り返すもの）、Hadoopシーケンスファイル、ZIP/GZipアーカイブ内に格納されたドキュメント、および「データベースアーカイブ」と呼ばれるMLCP固有の形式を入力として読み取ることができます。データベースアーカイブには、ドキュメントデータに加えて、各ドキュメントのMarkLogic固有のメタデータ（コレクション、パーミッション、プロパティ、クオリティ）が圧縮形式で含まれています。MLCPは、通常のドキュメント、一連の圧縮ZIPファイル、新しいデータベースアーカイブに出力できます。また、あるデータベースから別のデータベースに直接コピーすることもできます。

パフォーマンス上の理由から、MLCPはインポートとコピーの際、デフォルトでまとめて100更新、トランザクションごとに10バッチを実行します。これが、データを一括して出し入れするための最も効率的な方法であることがほとんどです。その理由は、どのトランザクションにもわずかなオーバーヘッドがあることです。このため、速度を最大限に高めるためには、各ドキュメントのトランザクションを個別に実行するのではなく、グループ化するのが適切です。しかし、ロック管理にも独自のオーバーヘッドがあるため、1つのトランザクションにあまりにも多くのロックがあるのも望ましくありません。1トランザクションで1,000ドキュメントの処理が妥当でしょう。各バッチはメモリに保持されるため（デフォルトではストリームも可能）、1つのトランザクションを複数の小さなバッチに分割することでメモリのオーバーヘッドを最小限に抑えることができます。MLCPではフォレストの直接配置も実行できますが（「高速読み込み」と呼ばれます）、これは必要な場合のみ行ってください。フォレストの直接配置では、完全な重複URIチェックを行わないためです。MLCPは、通常のHadoopの場合と同様、常にターゲットフォレストのあるノードと直接通信します。

さらに、MLCPの直接アクセス機能により、ディスク上のフォレストからドキュメントを直接読み込むことで、MarkLogicをバイパスしてデータベースからこれらのドキュメントを抽出できます。これは主に、階層型ストレージを介してアーカイブされるデータにアクセスすることを目的としています。長い期間が経過して古くなり、MarkLogicを介したリアルタイムクエリには必要がないデータがあるとしましょう。保存元のフォレストをオフラインにしてこのデータをアーカイブできますが、直接アクセス機能を使用すれば引き続きコンテンツにアクセスできます。

ほとんどの目的で、MLCPはオープンソース（ただしサポート対象外）のRecordLoaderおよびXQSyncプロジェクトの代わりになります。

CONTENT PROCESSING FRAMEWORK

Content Processing Framework（CPF）も、MarkLogicディストリビューションの公式サポート対象サービスです。これは、ドキュメントのライフサイクルを管理するための自動システムです。あるファイル形式の種類から別の種類に、またはあるスキーマから別のスキーマにドキュメントを変換するか、ドキュメントを分割します。

CPFは内部的にプロパティシートエントリを使用して、ドキュメントの状態をトラッキングします。また、トリガーとバックグラウンド処理により、ドキュメントの状態を移行させます。カスタマイズ可能な範囲が広く、独自の処理ステップセット（パイプライン）をプラグインして、ドキュメントの処理を制御することができます。

MarkLogicには「Default Conversion Option」パイプラインが付属しています。これは、Microsoft Office、Adobe PDF、HTMLのドキュメントをXHTMLおよび簡略化されたDocBookドキュメントに変換するものです。変換プロセスには多数のステップがあり、そのすべてが、プロセスのほかのステップの結果に基づいて自動的に実行されるように設計されています。

OFFICE TOOLKIT

MarkLogicでは、Word、Excel、PowerPoint向けのOffice Toolkitを用意しています。これらのツールキットにより、MarkLogicプログラムで、ネイティブのMicrosoft Officeファイル形式で簡単にドキュメントを読み取ったり、ドキュメントに書き込んだり、ドキュメントを操作したりできます。

これらのツールキットには、プラグイン機能もあります。独自のカスタムサイドバーやコントロールリボンをアプリケーションに追加できるため、例えば、MarkLogic内で検索して選択したコンテンツをWordドキュメントにドラッグするなどの操作が容易になります。

SHAREPOINT向けコネクタ

Microsoft SharePointは、ドキュメント管理に幅広く利用されているシステムです。MarkLogicは、SharePointに組み込むことでシステムに保存されているドキュメントへのさらに高度なアクセスが実現するSharePoint向けコネクタを提供（およびサポート）しています。このコネクタにより、MarkLogic内のSharePointドキュメントをミラーリングして検索、アセンブリ、再利用に対応したり、MarkLogicをSharePointワークフロー内のノードとして動作させたりできます。

ドキュメントのフィルタ

あまり目立たない関数ですが、`xdmp:document-filter()`を支えているのが、MarkLogicに組み込まれ、バイナリドキュメントからメタデータとテキストを抽出する、数百ものドキュメント形式に対応した堅牢なシステムです。オフィス文書、メール、データベースダンプ、動画、画像、その他のマルチメディア形式や、アーカイブファイルですらフィルタリングすることができます。フィルタプロセスではこれらのドキュメントをリッチXML形式に変換するのではなく、標準的なメタデータとファイル内にあるすべてのテキストを抽出します。これは、検索、分類、その他のテキスト処理のニーズに役立ちます。より詳細な抽出を行うには（画像内の顔認識や動画の文字起こしなど）、サードパーティのツールを利用してください。

ライブラリサービスAPI

ライブラリサービスは、ドキュメント管理のためのインターフェイスを提供します。ユーザーは、そこでチェックイン/チェックアウトおよびバージョン管理を行うことができます。ライブラリサービスの機能をロールベースのセキュリティおよびSearch APIと組み合わせると、MarkLogic上でコンテンツ管理システムを構築できます。

コミュニティがサポートするツール、ライブラリ、プラグイン

MarkLogic Developer Site（http://developer.marklogic.com）でも、非常に有用な多数のプロジェクトを主催または紹介しています。その多くは、GitHub（https://github.com/marklogic）で協同で構築されたものです。GitHubでは、関心があれば誰でも開発に協力できます。

Converter for MongoDB

MongoDBからMarkLogicにデータをインポートするためのJavaベースのツールです。MongoDBのmongodumpツールからJSONデータを読み取り、XDBCサーバーを使用してデータをMarkLogicに読み込みます。

Corb2

MarkLogicに格納されたドキュメントの一括コンテンツ再処理を行うJavaベースのツールです。データベースドキュメントのリストに基づいて、それらに対して処理を実行します。これらの処理には、全ドキュメントに及ぶレポートの生成、個々のドキュメントの操作、それらの組み合わせなどが含まれます。

ML Gradle

Gradleビルド自動化システムのプラグインです。ML GradleはMarkLogicアプリケーションを導入でき、MarkLogic Content Pumpなどのその他の機能と連携できます。ML Gradleは主に、MarkLogic REST API経由で機能します。

ml-lodlive

LodLiveは、分散エンドポイント間のセマンティックデータを視覚的に表すためのRDFブラウザです。ml-lodliveを使用すると、MarkLogicに格納されているデータを、MarkLogic REST APIを介してLodLiveブラウザ上で確認しやすくなります。

MarkLogic Workflow

CPFパイプラインをモデル化し、そのモデルからパイプラインを生成する手法など、CPFパイプラインを定義するための便利な手法を提供することを目的としたプロジェクトです。

MarkMapper

Rubyでオブジェクトをモデル化し、MarkLogicデータベースに永続させるためのツールです。MarkMapperは、RubyとMongoDBで同様に機能するMongoMapper ORMをベースにしています。

MLPHP
PHPが稼働するwebサーバー上のMarkLogicで、ドキュメントの格納、ドキュメントメタデータの管理、検索クエリの実行をするためのPHP APIです。MarkLogic REST APIを介してMarkLogicと通信します。

MLSAM
MarkLogic環境からリレーショナルデータベースシステムに簡単にアクセスできるようにするXQueryライブラリです。MLSAMを使用すると、あらゆるリレーショナルデータベースに対して任意のSQLコマンドを実行し、結果をXMLとして取り込んで、MarkLogicで処理できるようになります。 MLSAMは、HTTP呼び出しを使用してSQLクエリをサーブレットに送信します。そのサーブレットは、JDBCを使用してクエリをリレーショナルデータベースに渡します。

oXygen
MarkLogic統合によるXML編集環境です。複数のサーバー接続、XQueryの実行とデバッグ、WebDAVを介したリソースの管理と編集などがサポートされています。

Roxy
MarkLogicアプリケーションを設定して複数の環境に導入するための、非常に人気の高いユーティリティです。Roxyを使用すると、アプリケーションサーバー、データベース、フォレスト、グループ、タスクなどをローカルの設定ファイルで定義できます。続けてRoxyは、これらの設定をコマンドラインからリモートで作成、更新、削除します。このプロジェクトはまた、ユニットテストフレームワークとXQuery MVCアプリケーション構造も提供します。

Sublime Text Plug-in
人気の高いSublime Textソースコードエディタのアドオンです。MarkLogic組み込み関数のシンタックス強調表示およびコード補完機能、XQueryのエラーチェック機能、MarkLogicオンラインドキュメントとの統合などの機能があります。

xmlsh
XMLのコマンドラインシェルです。xmlshは、XMLプロセス自動化のためにカスタマイズされた馴染みのあるスクリプト作成環境を提供します。 MarkLogicに接続するための拡張モジュールが含まれています。

XQDT
Eclipse IDE向けのオープンソースのプラグインセットです。XQDTは、XQueryモジュールのシンタックス強調表示およびコンテンツ支援編集機能と、サポート対象XQueryインタープリターでのモジュールの実行およびデバッグのためのフレームワークをサポートしています。

XQuery Commons
自己完結型の小さなコンポーネントを網羅する、MarkLogicの包括的なプロジェクトです。これらの「断片」だけではアプリケーション全体を構成することはできませんが、これらは多数のさまざまなコンテキストで再利用できます。利用可能なXQueryコンポーネントには、HTTP、Date、Searchなどのユーティリティがあります。

XML Memory Operations
メモリ内のXMLに対して処理を実行するためのXQueryライブラリです。XPath軸、ノード比較、集合演算子を使用することで、このライブラリはXMLに変更を加えることができます。その一方で、ノードを再構築できるのは、変更対象ノードの直接パス内のみです。

xray
MarkLogicサーバー上でXQueryユニットテストを記述するためのテキストとして出力できるため、多数のCIサーバーと簡単に統合できます。

その他の情報の参照先

本書で紹介したもの以外にも、貢献度の高いプロジェクトは多数あります。その全リストは、MarkLogic Developer Community Toolsで参照できます。もちろん、いずれかのプロジェクトに参加して貢献することも可能です。あるいは、独自のプロジェクトを開始するのもよいでしょう。

ご質問がある場合は、MarkLogic Product Documentation（http://docs.marklogic.com）を検索するか、MarkLogicの開発全般に関するメーリングリストで質問するか、Stack Overflowに投稿してください。皆様のご成功をお祈りしています。

マークロジック株式会社 MARKLOGIC K.K.
150-0043 東京都渋谷区道玄坂 1-12-1渋谷マークシティウェスト 22 階
03-4360-5354 | jp.marklogic.com | MarkLogic-JP@marklogic.com

マークロジック株式会社 MARKLOGIC K.K
150-0043 東京都渋谷区道玄坂1-12-1渋谷マークシティウェスト22階 | 03-4360-5354
jp.marklogic.com | MarkLogic-JP@marklogic.com