XML データベース
NeoCore ベンチマークレポート

スキーマレスデータの検索・格納処理における、RDB の XML 機能に対する優位点

2016 年 9 月

株式会社サイバーテック

目次

1. はじめに

本文書はベンチマークプロジェクト(本プロジェクト)の実施内容を記述したものである。本プロジェクトでは XML データベース「NeoCore」と「製品 D（RDB）の XML 機能」について、CRUD 処理（挿入、検索、更新、削除）のパフォーマンスについて比較/検証を実施した結果に ついて記述したものである。

2. NeoCore について

2 – 1. NeoCore の用途

XML データベース「NeoCore」は、構造が一定ではない XML データの格納に最適な"やわらかい"データベース・エンジンです。国内出荷ライセンス数は 500 ライセンスを超え、国内シェアー位（富士キメラ総研調べ「ソフトウエアビジネス新市場 2015 年版」）の XML データベースです。

「NeoCore」は、XML をハンドリングすることに特化、RDB では実現不可能な水準のパフォーマン スを発揮する事ができる、柔軟性と拡張性に優れた XML データベース製品です。製造業の製品データやドキュメントに付随するメタデータなど、多様で変化しやすいデータをスキーマレスの XML としてそのままデータベースに格納する事ができるため、システム運用中のデータベースの変更コストを最小限に抑える事が可能です。

「NeoCore」の特長は、データ量やデータ構造に依らない「超高速の検索性能」（速い）、すべてのタグに対して自動的にインデックスをはる「フルオートインデックス機能」（カンタン）、XML 形式のデータを柔軟に吸収できる「スキーマレス機能」（やわらかい）です。これらの特長は、ビジネスサイドからの要求によって発生する仕様変更に対して、システム側で即座に対応する事ができます。

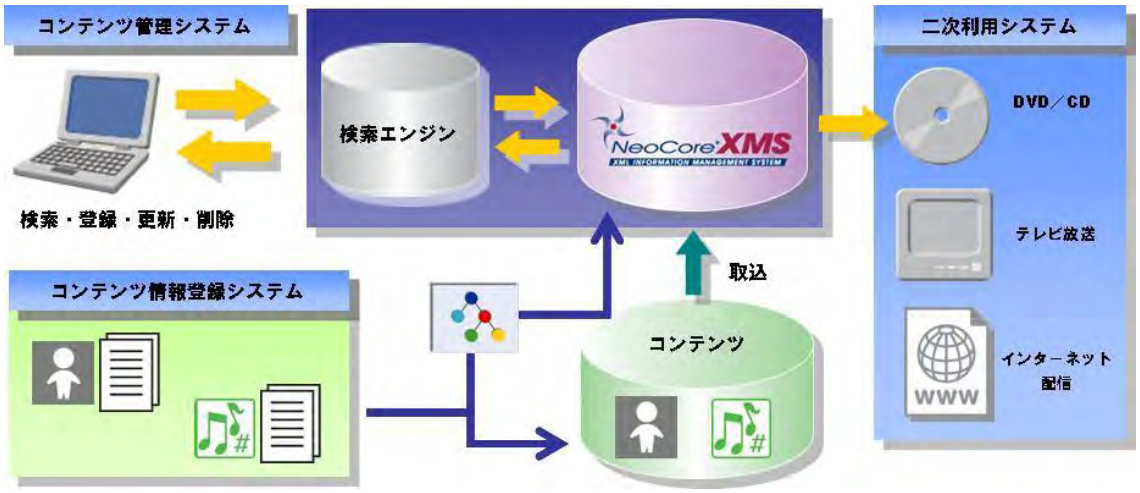
「NeoCore」は、企業情報システムの中で仕様変更が頻繁に発生する情報系システムの開発・運用フェーズにかかるシステムエンジニアコストと時間を大幅に削減する事を可能にしました。また、「NeoCore」は、アプリケーション開発者向けの情報やノウハウが多数公開されているだけでなく、初級者向けのトレーニングメニューを取り揃えているため、技術者の入門用またはプロトタイプ構築用に最適な XML データベースです。

■「NeoCore」の主な用途

＜ドキュメント管理に強い！＞

「NeoCore」は、XML ドキュメントを「そのまま」格納できるため、マニュアル・約款・規定集や教材などのドキュメント管理データベースに最適で、多くの導入実績があります。「内部統制による業務の明確化」や、「紙の電子化による印刷･配布･保管コストの削減」、「DTP などの制作現場の業務改善」など、ドキュメント管理に関する課題を解決します。

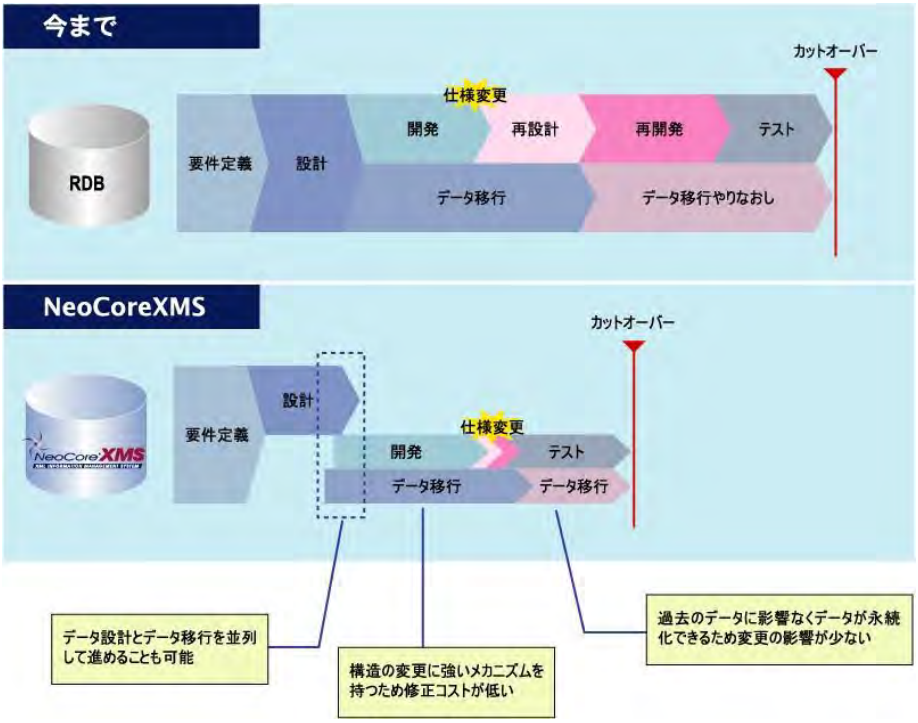
＜メタ情報の一元管理・検索に強い！＞ デジタルコンテンツの社内管理とユーザーへの提供サービス向上の実現での利用ケースがこれに該当します。デジタルコンテンツの音楽、写真に作曲者、著作権者、収録日などのメタ情報を付加すること で、社内用途ではコンテンツ管理として利用でき、デジタルコンテンツを再利用したいユーザーにとって は、付加されたメタ情報が検索キーワードとなり、効率良く目的のコンテンツにたどり着くことができるの です。Web カタログや製造業の製品情報管理、電子カルテなどその応用範囲は多岐にわたります。



■XML を活用した新しい開発スタイル

＜アプリケーションの設計変更コストを削減する「NeoCore」＞ ビジネス環境がスピード化・多様化する中で、情報系システムの開発・運用フェーズで「仕様変更が頻繁に発生する」事は、もはや避けられない状況です。ビジネスが変われば当然管理すべきデータも変わり、さらにデータを支えるアプリケーションにも影響が及びます。従来の RDB は、厳密なスキーマ情報を 前提とするデータベースであるため、「仕様変更＝スキーマ変更」となり、さらに適正なパフォーマンスを 確保するために「インデックス再設計」が必要でした。

「NeoCore」は、厳密なスキーマ設計を必要としないため、ビジネスサイドからの要求によって発生するスキーマ設計とインデックス再設計にかかるシステムエンジニアコストと時間を大幅に削減する事を可能にしました。



2 – 2. NeoCore の特長

「NeoCore」は、まったく新しいタイプの XML データベースマネジメントシステム（DBMS）です。スキーマ定義は一切不要、XML 形式のデータであれば、すべて柔軟に吸収。フルオートインデックス機能により、すべてのタグに対して自動的にインデックスを作成します。システム開発者の悩みどころであったインデックス設計を不要にし、システム開発の効率を飛躍的に向上させることができます。

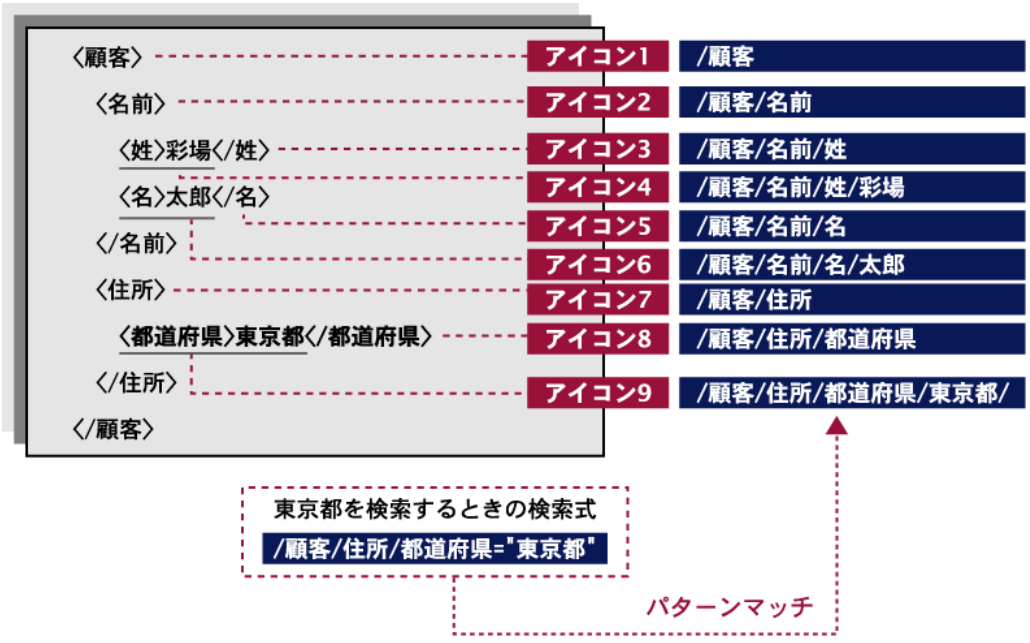
XML の超高速検索を実現する、DPP（Digital Pattern Processing）を搭載。元データを見ることなく、フラットな構造のアイコンをパターンマッチさせる事でデータ量やデータ構造の複雑さに依存しない安定した検索パフォーマンスを実現します。XML 形式のデータをスキーマレスで格納、XML の柔軟性を活かしたデータ管理が可能です。

■XML 超高速検索機能

リレーショナルデータベース（RDB）のテーブル構造に XML データをマッピングした場合や、XML データ型の領域に格納した場合に、著しく検索性能が低下してしまう場合があります。NeoCore は、独自の特許技術である DPP（Digital Pattern Processing）により、XML のデータ量や XML の階層構造の深さに依存しない安定した検索が可能です。

NeoCore は、XML データを格納する際、Parsing を通過した XML データを、Flattener と呼ばれるモジュールにより、パスとデータに分解します。DPP は、この分解した 1 つ 1 つのパスとデータを、64bit の固定長のデータに変換し、全てのタグに対してユニークな「アイコン」として生成します。

NeoCore では、データベースに格納された XML データを検索する際に、元データを見ることなく、フラットな構造の「アイコン」をパターンマッチさせる方式を採用。この「アイコン」が XML データを検索する際のインデックスとして機能し、実データはこの「アイコン」が指し示す場所に格納されます。

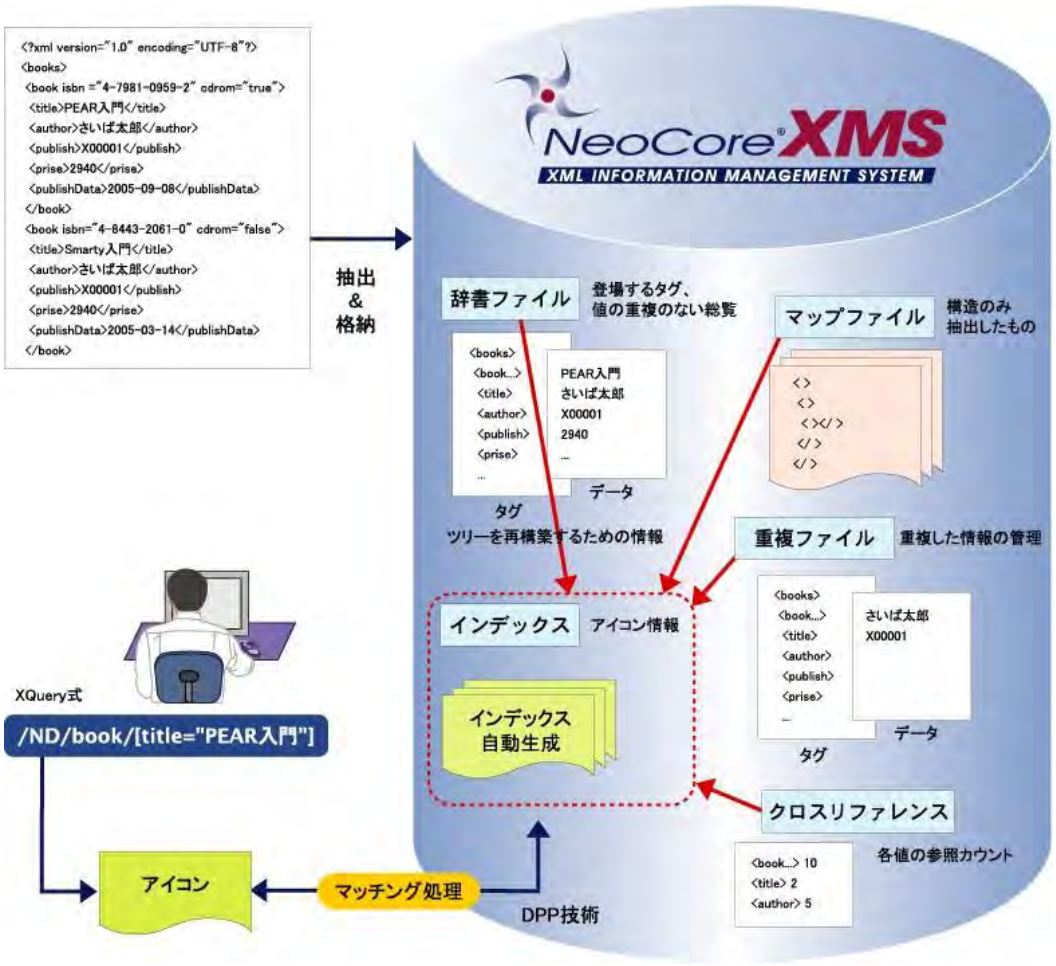


■フルオートインデックス機能

NeoCore は、XML データをデータベースに格納する際、DPP（Digital Pattern Processing）により、すべてのタグに対して自動的にインデックスを生成します。

フルオートインデックス機能は、リレーショナルデータベース（RDB）で必要とされるインデックス設計や生成のための作業を不要とします。つまり、開発者はどの項目にインデックスを設定すれば良いかを考える必要はなく、インデックスを上手に使用するような検索方法を考えるだけで済むため、システム開発時の時間とコストを飛躍的に向上する事が可能です。NeoCore は、格納時や更新時に DPP により 分解された XML の構造を、独自の方式でバイナリデータとして種類別に分類して格納します。NeoCore の内部は、下記の複数ファイルで構成・管理されます。

- タグ部及びデータ部の実体を格納する「辞書ファイル」
- タグ部及びデータ部のインデックス情報を格納する「インデックスファイル」
- タグ部及びデータ部の重複インデックス情報を格納する「重複ファイル」
- 辞書ファイルとインデックスファイルを相互参照する「クロスリファレンス」
- XML データの物理的な構造情報を格納する「マップファイル」
- データベースファイルのロケーション情報や、格納されている文書数などの管理者情報を格納する「Admin ファイル」



■スキーマ定義不要

NeoCore は、XML 形式のデータを格納する際、スキーマを定義する必要がありません。格納、更新等の操作の際に入力する XML については、妥当性検証（Validation チェック）を行わず、整形式であるかどうかのチェック（Parsing）のみを行います。従って、DTD や XMLSchema とは関係なく、あらゆる XML 形式のデータを格納し、またエレメントの追加や削除を自由に行うことができます。

リレーショナルデータベース（RDB）は、格納するデータ全てのデータについて、厳密なスキーマ定義を必要とするため、仕様変更のたびにスキーマを再定義する必要がありました。これに対して、NeoCore を用いたデータベース設計では、ノードや属性を Tree 型に配置した構造を設計し、パフォーマンスを考慮した Tree 構造の変更を行います。リレーショナルデータベース（RDB）の場合はインプリメントの前に構造設計が完結している必要がありますが、NeoCore の場合は、基礎が決まっていれば、ある程度の変更はアプリケーションで吸収できてしまいます。

これにより、ビジネスサイドからの要求によって発生するスキーマ設計とインデックス再設計にかかるシステムエンジニアコストと時間を大幅に削減する事が可能となり、システム開発の初期の段階でデータ構造を厳密に定義できない場合や、システム運用フェーズでのデータ項目の追加変更の際には、大きなメリットとなります。

2 – 3. 機能詳細

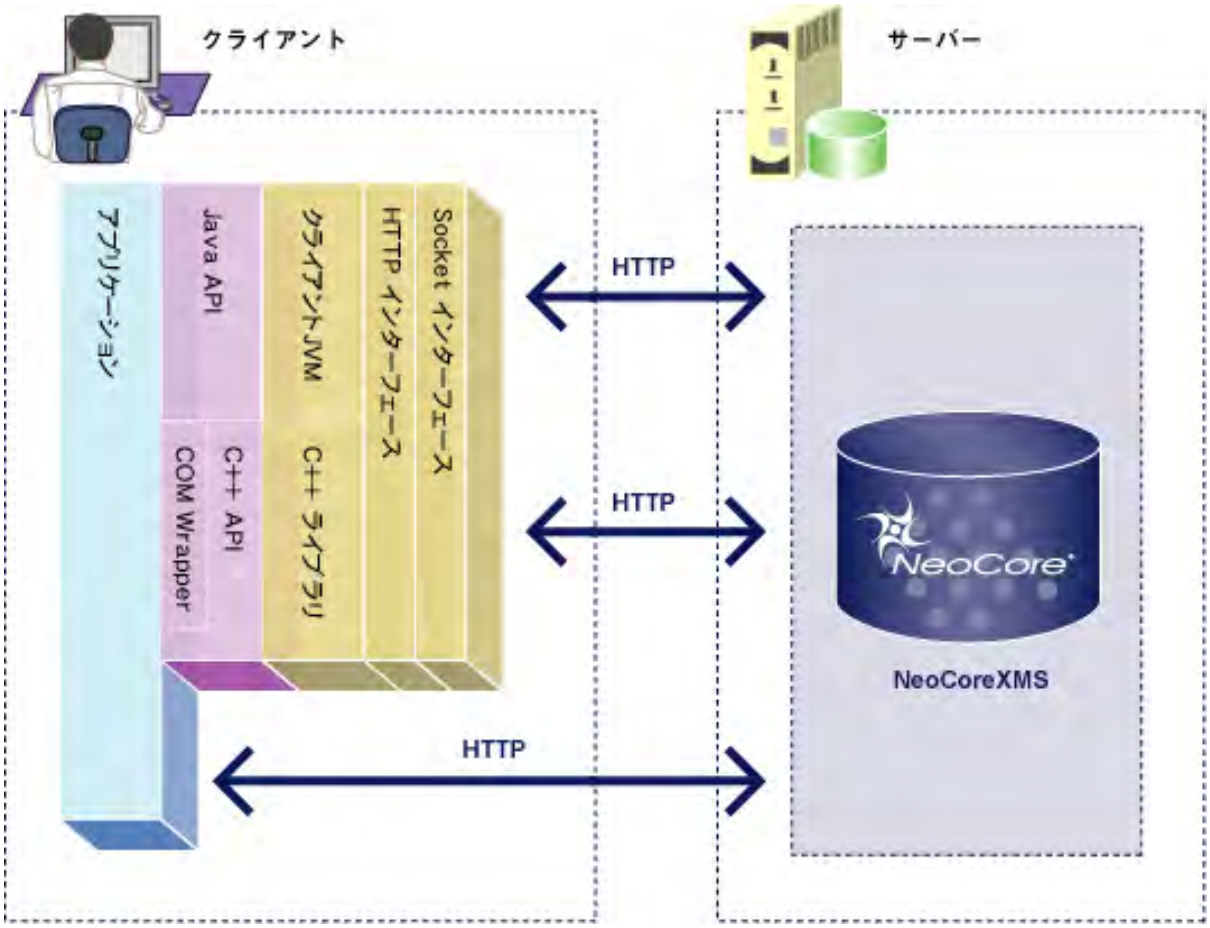
■HTTP によるアプリケーション通信（プロセス・アーキテクチャ）

NeoCore のサーバープロセスは、各コンポーネントが実行を制御します。各コンポーネント内部で はマルチスレッドで処理され、処理効率を向上させています。クライアントアプリケーションは JavaAPI、C++API、または C++API をラップした COM ラッパーから、HTTP クライアントのソケットを実装した各ライブラリを介してサーバープロセスへ接続します。Perl や PHP などのスクリプト言語からは HTTP を介して接続します。

アプリケーション開発者は、NeoCore を利用する場合、使用する言語を Java または C++から選択できます。これらの言語からは、NeoCore より提供される API ライブラリを使用する形で開発が可能です。さらに、HTTP I/F を用いることで、C#、PHP、Perl、Ruby、JavaScript や Ajax、Flash などの RIA

をはじめとする、ソケット通信が可能なあらゆる言語を使ったアプリケーション開発が可能です。

API ライブラリおよび HTTP I/F いずれの手段を使っても NeoCore の持つ機能全体を使用できるため、アプリケーション開発者は最も慣れている言語、または開発タスクに最も適した言語を選択できます。また、NeoCore は、HTTP 通信のオーバーヘッド自体が非常に小さいため、パフォーマンスへの影響が出にくいのも特長です。



■XPath／XQuery によるデータアクセス

NeoCore へのデータ格納および格納されたデータに対するアクセスを行う場合には、SQL ではなく XPath／XQuery を使います。これを使うことで、複数存在する要素のうちアクセスしたい任意の要素を指定して検索結果を取得することができます。

また、引数として XPath 式を与えることで、XML データの一部を更新することや、削除・追加をすることができます。XPath はパス形式の記述言語なので、複数の文書にまたがって存在する同名の要素に対し、一回のコマンドで横断的にアクセスすることもできます。XPath/XQuery は共に W3C の正式勧告を受けた世界標準規格なので、長く安心してお使い頂く事ができます。

■トランザクション・データロック機能

＜(1)トランザクション機能＞

NeoCore は、リレーショナルデータベース（RDB）同等の ACID レベルのトランザクションをサポート しています。全てのトランザクションは、トランザクションログに記録され、電源障害などの予測できない障害が発生した場合におけるデータベースの復元・復旧を可能とします。明示的なトランザクションの開始、コミットおよびロールバックも可能で、分離レベルの設定も可能です。

(2)データロック機能

一般的なリレーショナルデータベース（RDB）と同様に、NeoCore においてもデータロックを使用したデータへの同時アクセス制御が可能です。情報の更新時には、更新がコミットされるまで、その情報はロックされ、発行またはコミットされるまで、ロックされている情報は誰も変更できません。これにより、システムのデータ整合性が保証されます。

■ブラウザ及びコマンドラインによる各種ユーティリティ

NeoCore では、インスタンスの起動・停止や、データベース管理のために、各種ユーティリティ プログラムを提供しています。

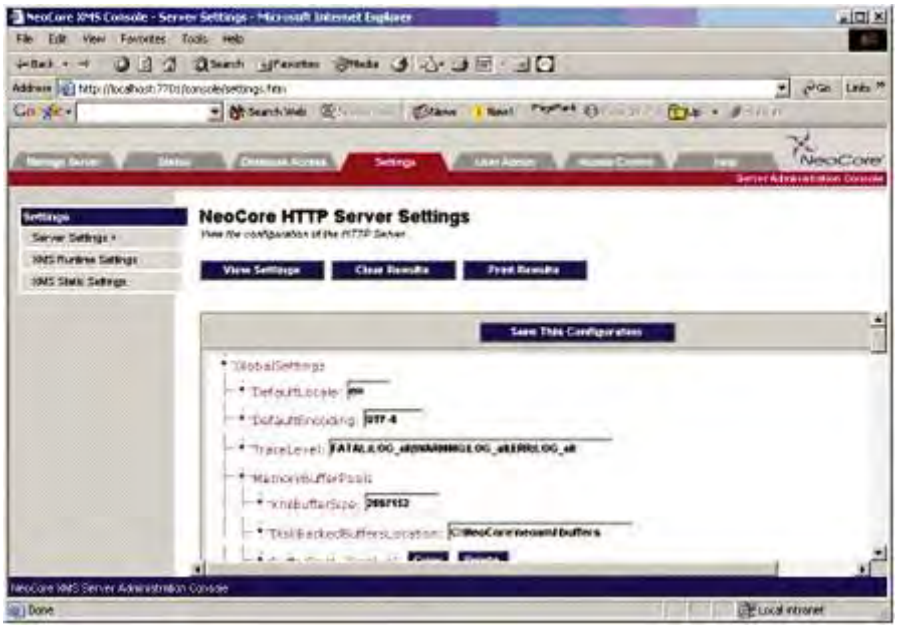
＜(a)NeoXMLUtils＞

NeoXMLUtils は、データベースの管理を行うユーティリティプログラムで、以下の機能を備えています。

- データベース生成・削除関連 データベース作成・空のデータベース作成・データベースクリア・データベース削除が可能です。
- データベースの保存・復元関連 データベースのバックアップ・リストア・インポート・エクスポート等が可能です。
- インデックス再構築・ファイル移動関連 インデックスの再構築や、データベースファイルの移動が可能です。

＜(b)管理コンソール＞ 管理コンソールは、データベース管理者のための、ブラウザベースの管理ツールです。

- サーバ管理
  NeoCore の起動、停止、バージョンの確認が可能です。
- データベースの状態表示・サーバログとアクセスログの表示
  NeoCore の内部で管理されている各ファイル(マップファイル等)の物理サイズや使用率等を確認することが可能です。
- XML 文書の操作とデータベースアクセス(Copy/Query/Insert/Modify/Delete)
  データベースへ XML データの登録、XPath／XQuery を使ったデータの参照などが可能です。
- サーバコンポーネントの設定表示・設定変更・チューニング メモリバッファサイズ、デフォルトエンコード値、JAVA 仮想マシンのヒープサイズ、HTTP アクセス のポート番号、セッションタイムアウト時間、メモリの自動拡張設定等、設定の確認と変更をきめ 細かく行うことが可能です。



■キーファイルによるライセンス管理

NeoCore は、ライセンスキーファイル(license.xml)でライセンス情報の管理を行います。購入前に無償で評価頂くための「 30 日評価版ライセンス」からデモやプロトタイプ用途に利用頂くための「Developer Edition」、さらには、それぞれのご利用シーンに応じた形でのご選択が可能な「Workgroup Edition」「Limited Edition」「Standard Edition」へのステップアップは、このライセンスキーファイルの入れ替えのみで可能です。

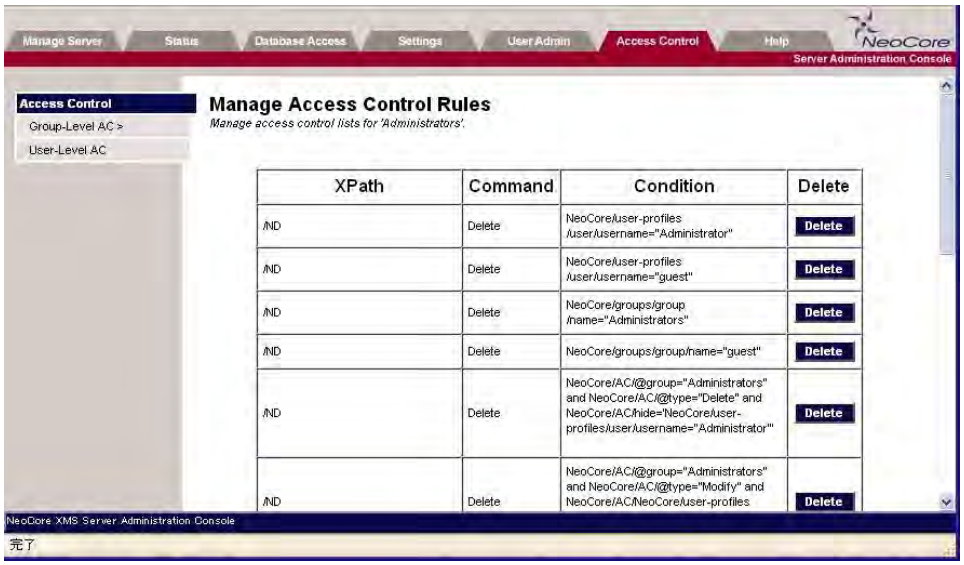
さらにこの仕組みを活用することで、容易にパッケージソフトへの組み込みが可能となります。 組み込むパッケージソフトの利用形態や運用形態に応じて、利用期間の制限、データベース容量の上 限値の設定、リモートサーバからのアクセス制御などをライセンスキーファイルのみで設定・管理を行うことが可能です。

■セキュリティ機能(アクセスコントロール)

NeoCore は、データベースアクセスを制御するためのセキュリティ機能を標準で提供しています。 こ

のアクセスコントロールを使用することで、XML データへのアクセスをノードレベルで自由自在に制御する事が可能となります。

アクセスコントロールの設定には、ブラウザベースの管理コンソールを使い、制御ルールを作成します。すべてのノードを対象とする単純なルールから、該当文書内の情報に基づく条件に合致するノードだけを対象とするといった複雑なルールにも対応する事が可能です。



■XML メタデータの格納

NeoCore は、XML 文書データの他に、その格納された XML 文書データのメタ情報を、MetaData セクション配下に格納します。これらメタデータは、規定集や約款など、文書の履歴管理や版管理が必要とされるアプリケーションから利用することが可能です。

- 変更時刻 ＜ModifyTime＞
  XML 文書が更新されたときのシステム時刻（秒単位）。XML 文書を初めて格納したときには、TimeStamp タグと同じ時刻を設定します。
- 格納時刻 ＜TimeStamp＞
  XML 文書が格納されたときのシステム時刻（秒単位）。XML 文書をコピーした場合、コピー先には新しい時刻を設定します。
- ソースファイル ＜SourceFile＞ データの元となるファイルの名前。格納時に
  設定したファイル名を設定します。
- 文書 ID ＜DocID＞
  格納順に XML 文書に割り当てられるユニークな値です。
- 文書コピーの番号 ＜CopyNumber＞
  XML 文書のコピー時に割り当てられる管理番号です。
- スキーマファイル ＜SchemaFile＞ 文書と一緒にスキーマを格納した場合に、その文書スキーマが入っているファイルの URI を設 定します。
- プレフィックスファイル ＜PrefixFile＞ プレフィックスファイル
  の名前（prefix.xml など）を設定します。

3. テスト方法

事前に用意したテストプログラム（C#）の実行により、NeoCore、および XML ネイティブ対応が一番進んでいるエンタープライズ向け商用 RDB として有名な、I 社が提供する RDB「製品 D」の2製品において、各オペレー ションにおける処理速度を計測。

テストは以下を考慮して実施した。

- ベンチマークが終了する度に VMware をベンチマーク実行前の状態に巻き戻し、ディスクアクセスが収まるのを待ってから次のベンチマークを実行する。但し、検索のみキャッシュの効果を排するためループ毎に巻き戻す。
- 一部の例外を除いて 5 回ループ実行し、最大値を除いた 4 回分の平均値を求める。

3 － 1. テストデータ

テストデータとして複数階層をもつ TREE 形式のサンプルデータを作成し使用。また、データ値には乱数値を設定して作成。（データ数：1 万件）

【データサンプル】

```
<tree id="1" tree-info="3,2,2,2,2,1" min="1" max="999" random-seed="12345">
  <branch idx="1">
    <branch idx="2">
      <branch idx="3">
        <branch idx="4">
          <branch idx="5">
            <item num="1">067</item>
          </branch>
          <branch idx="6">
            <item num="2">071</item>
          </branch>
        </branch>
        <branch idx="7">
          <branch idx="8">
            <item num="3">774</item>
          </branch>
          <branch idx="9">
            <item num="4">511</item>
          </branch>
        </branch>
      </branch>
       :   :   :
      <branch idx="87">
        <branch idx="88">
          <branch idx="89">
            <item num="45">635</item>
          </branch>
          <branch idx="90">
            <item num="46">295</item>
          </branch>
        </branch>
        <branch idx="91">
          <branch idx="92">
            <item num="47">370</item>
          </branch>
          <branch idx="93">
            <item num="48">852</item>
          </branch>
        </branch>
      </branch>
    </branch>
  </branch>
</tree>
```

3 – 2. テスト環境

Windows 7 マシン上のに VMware Workstation をインストールし、仮想マシン上のゲスト OS 側にデータベースサーバをインストールする。

ネットワーク(VMnet:十分高速)を介したクライアント・サーバー構成とする。

クライアント (VMware ホスト OS／ベンチマーク実行側)

OS:Windows 7 Ultimate Service Pack 1 (64 ビット)

メモリ: 32GB

CPU: AMD FX-8300 8 コア 3.30GHz

ベンチマークプログラム:

C# 6 (Visual Studio 2015)

.NET Framework 4.5

サーバ (VMware ゲスト OS／データベースサーバ側)

OS:Windows 8.1 Professional (64 ビット)

メモリ: 4GB を割り当て ※ 登録データに対して十分な量とする

CPU: 4 コアを割り当て

データベースサーバ:

NeoCore 3.1.3.125

製品 D の最新版(2016 年 9 月時点)

ゲスト OS 側は余計なディスク／ネットワークアクセスを行うサービスやタスクを停止。

・Windows Search サービス

・Windows Update サービス

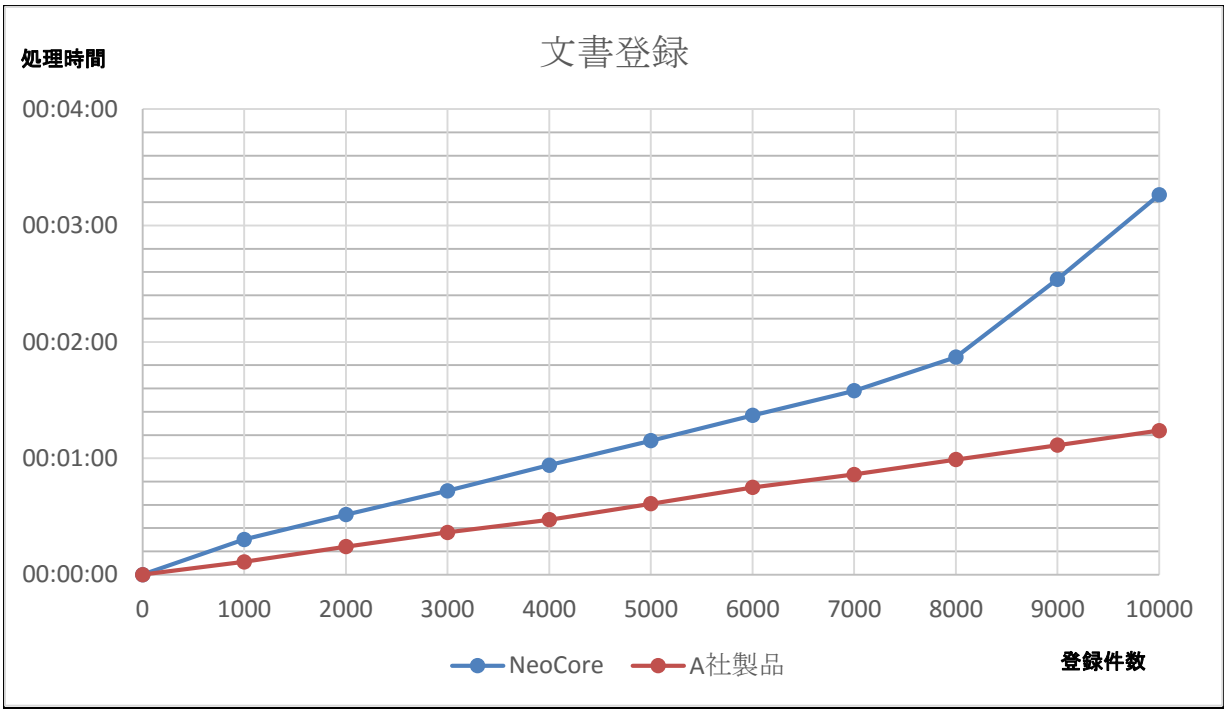
・Windows Defender

・カスタマーエクスペリエンス向上プログラム関連タスク

4. 各処理におけるテスト結果、検証

各処理におけるテスト結果は、以下の通り。

– 文書登録時の処理時間比較



【実行手順の概要:】

・入力ファイル 1 万件ループ

・XML 文書ファイルを 1 つ読み込む

・読み込んだ文書をデータベースに登録する。

NeoCore ： StoreXML を使用

製品 D ： INSERT INTO XML_DATA (ID, DOCUMENT) VALUES (XML 文書の@id 値, XML 文書)

・1000 件登録毎に経過時間を出力する。

【結果】

8000 件までは製品 D に対して、NeoCore は約 2 倍の登録時間がかかっている。

●8000 件登録時

製品 D:約 1 分

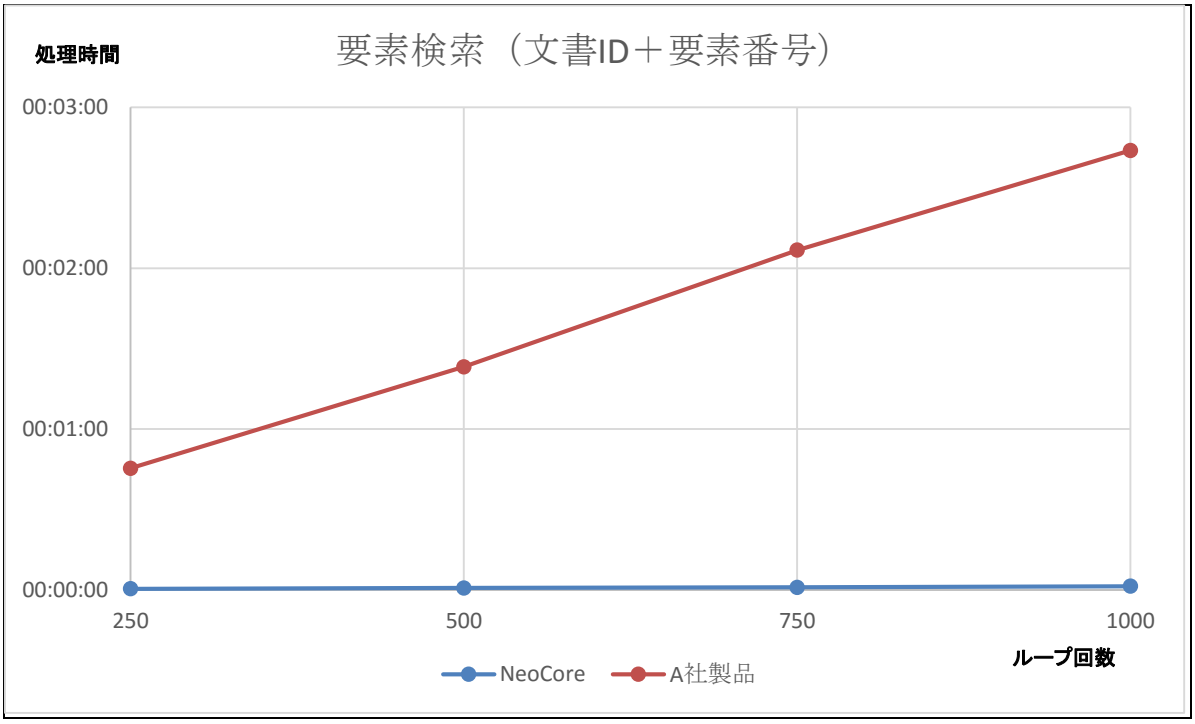
NeoCore:約1分 52 秒

製品 D はほぼ線形といえる時間の伸びだが、NeoCore は次第に傾きが急になっている。

製品 D では最低限のインデックス設定に留めているが、NeoCore はインデックス設計を考慮しなくても良いように、フルオートインデックス機能が自動でインデックスを生成するため、データ登録時の負荷が大きいと思われる。したがってインデックス設計によっては製品 D の方が NeoCore よりも登録時間が長くなる可能性も考えられる。

– 検索(文書 ID＋要素番号)時の処理時間比較

文書 ID(/tree/@id)と文書内の item 要素番号(//item/@num)を指定して item 要素を検索するベンチマーク。



【実行手順の概要:】

・以下の検索処理をそれぞれ 250 回、500 回、750 回、1000 回ループし、各実行時間を計測。
検索対象データ件数 ： 10,000 件

・乱数で検索対象要素(@id, @num)を決定する。

・検索する。
NeoCore ： /ND/tree[@id='@id']//item[@num='@num']
製品 D ： XQUERY にて検索(/tree[@id='@id']//item[@num='@num'])

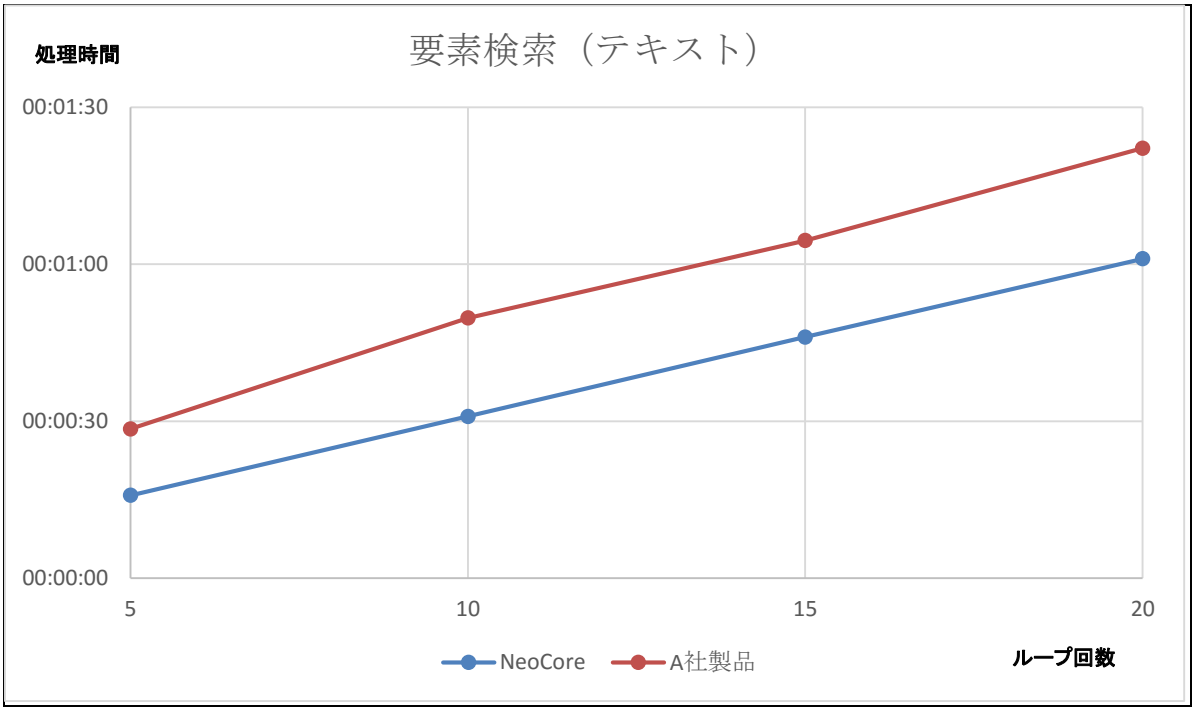
【結果】

圧倒的に NeoCore が速い結果となった。

●ループ回数 1000 回
製品 D:約 2 分 44 秒
NeoCore:約 2 秒

– 検索(テキスト)時の処理時間比較

/tree//item 要素のテキストノードを完全一致検索するベンチマーク。



【実行手順の概要:】

・以下の検索処理をそれぞれ 5 回、10 回、15 回、20 回ループし、各実行時間を計測。
検索対象データ件数 ： 10,000 件

・乱数で検索文字列(「001」～「999」)を決定する。

・検索する。
NeoCore ： /ND/tree//item[. = 検索文字列]
製品 D ： XQUERY にて検索(/tree//item[. = 検索文字列])

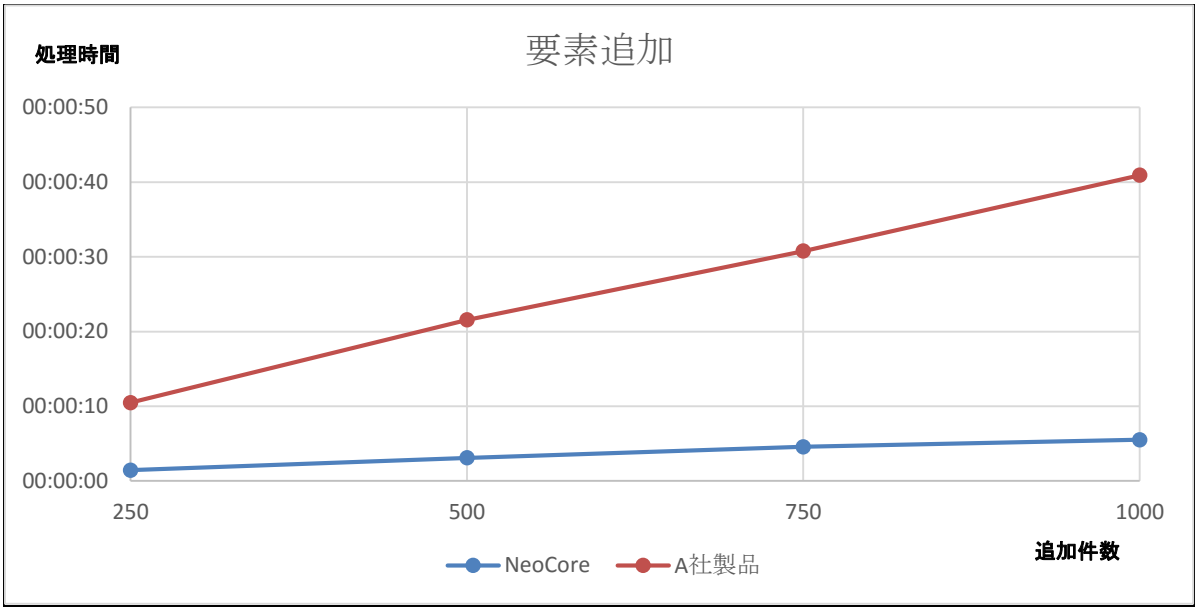
【結果】

1 クエリ当たり秒単位ほどの差が出るなど、NeoCore の方が約 25%～30%速い結果となった。

●ループ回数 20 回
製品 D:約 1 分 22 秒
NeoCore:約 1 分 01 秒

– 要素追加時の処理時間比較

指定された @id、@num の item 要素の後ろに新たな item 要素を追加するベンチマーク。



【実行手順の概要:】

・以下の追加処理をそれぞれ 250 回、500 回、750 回、1000 回ループし、各実行時間を計測。
検索対象データ件数 : 10,000 件

・乱数で追加位置(@id, @num)を決定する。

・追加位置の後ろに item 要素を追加する。

NeoCore :

```
insertXML(″/ND/tree[@id='@id']//item[@num='@num']″, ″<item num='-1'>INSERTED</item>″)
```

製品 D :

```
UPDATE XML_DATA
SET
    DOCUMENT = XMLQUERY('
        transform
        copy $d := $doc
        modify
            do insert <item num=″-1″>INSERTED</item> after $d/tree//item[@num=@num]
        return
            $d
    ' passing DOCUMENT as ″doc″)
WHERE XMLEXISTS('$doc/tree[@id=@id]' passing DOCUMENT as ″doc″)
```

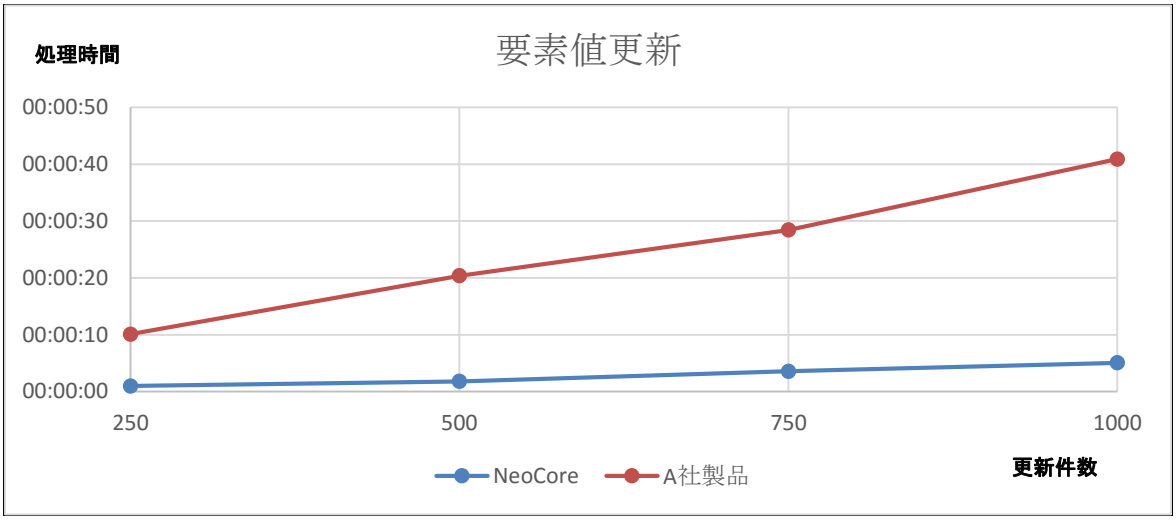
【結果】

NeoCore の方が速い結果となった。

製品 D も 1 件当たりの実行時間は数十ミリ秒ではあるが、NeoCore と比較すると大きな差があることがわかる。

– 要素更新時の処理時間比較

指定された @id、@num の item 要素の要素値(テキストノード)を更新するベンチマーク。



【実行手順の概要:】

・乱数を用いつつも重複の無いような更新対象リスト(@id, @num)を予め作成しておく。
・以下の更新処理をそれぞれ 250 回、500 回、750 回、1000 回ループし、各実行時間を計測。
検索対象データ件数 : 10,000 件
・予め作成しておいたリストから更新対象情報(@id, @num)を取り出す。
・要素値を更新(「MODIFIED_」＋「001」～「999」の乱数値)する。

NeoCore :

```
queryXML("/ND/tree[@id='@id']//item[@num='@num']")
取得した要素の要素値を更新
modifyXML("/ND/tree[@id='@id']//item[@num='@num']", 更新後の要素)
```

製品 D :

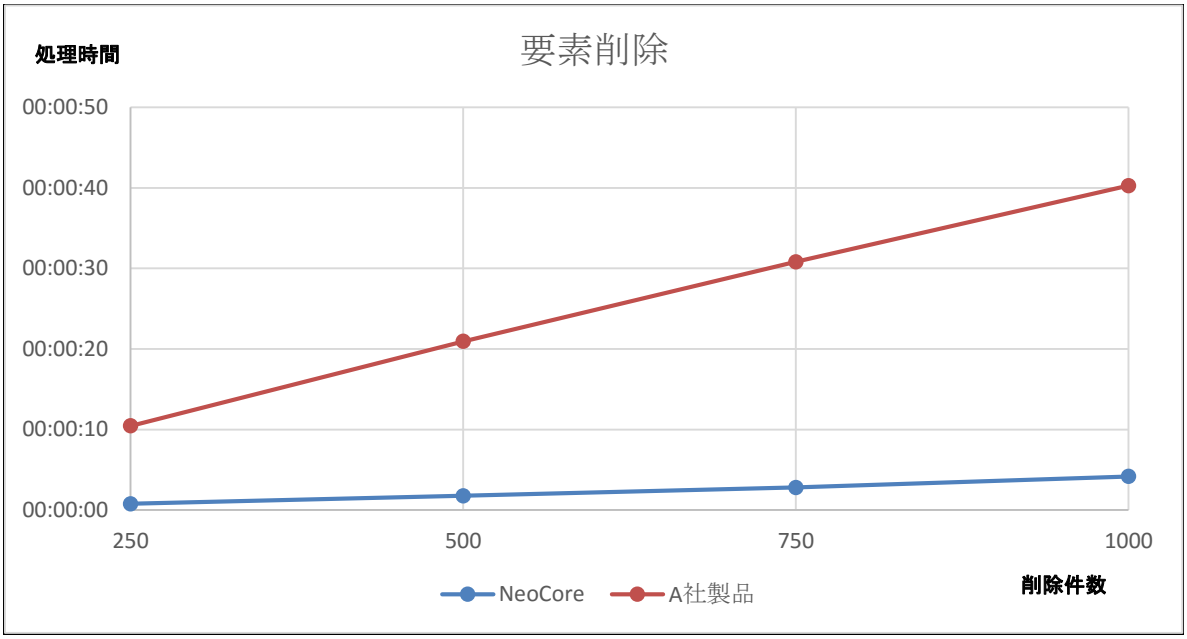
```
UPDATE XML_DATA
SET
    DOCUMENT = XMLQUERY('
        transform
        copy $d := $doc
        modify
            do replace value of $d/tree//item[@num=@num] with "@newValue"
        return
            $d
    ' passing DOCUMENT as "doc")
WHERE XMLEXISTS('$doc/tree[@id=@id]' passing DOCUMENT as "doc")
```

【結果】

要素追加と同様、NeoCore の方が速い結果となった。
製品 D も 1 件当たりの実行時間は数十ミリ秒ではあるが、NeoCore と比較すると大きな差があることがわかる。

– 要素削除時の処理時間比較

指定された @id、@number の item 要素を削除するベンチマーク。



【実行手順の概要:】
・乱数を用いつつも重複の無いような削除対象リスト(@id, @num)を予め作成しておく。
・以下の削除処理をそれぞれ 250 回、500 回、750 回、1000 回ループし、各実行時間を計測。
検索対象データ件数 : 10,000 件
・予め作成しておいたリストから削除対象情報(@id, @num)を取り出す。
・要素を削除する。

NeoCore : deleteXML(″/ND/tree[@id='@id']//item[@num='@num']″)

製品 D :

```
UPDATE XML_DATA
SET
    DOCUMENT = XMLQUERY('
        transform
        copy $d := $doc
        modify
            do delete $d/tree//item[@num=@num]
        return
            $d
    ' passing DOCUMENT as ″doc″)
WHERE XMLEXISTS('$doc/tree[@id=@id]' passing DOCUMENT as ″doc″)
```

【結果】
要素追加や更新と同様、NeoCore の方が速い結果となった。
製品 D も 1 件当たりの実行時間は数十ミリ秒ではあるが、NeoCore と比較すると大きな差があることがわかる。

5. 考察

今回のベンチマークでは、ほぼ全ての実行で NeoCore の処理が速い結果となった。

XML 文書の登録などの、ある程度まとまった量のインデックス情報を一度に処理する必要があるケースでは NeoCore が遅くなるケースもあるが、その分、インデックスを活用したパス検索では非常に高速なレスポンスを得られる事が実証された。

上記の通り、XML データベース「NeoCore」は、構造の異なるスキーマレスの XML データの処理に関しては、RDB の XML 機能よりも高速かつ安定したパフォーマンスが得られると言える。

最後に、NeoCore のパフォーマンスを最大限に生かすためには、XML データを設計する段階で考慮しなければいけない点がいくつか存在する。特に、複雑な構造のデータや大量のデータ処理を行う場合には、事前にサンプルプログラムによる検証を実施する事を推奨する。