Apache Solr の新しい内容

Solr 1.3 の新機能と機能改善を活用する

Grant Ingersoll
Member, Technical Staff
Lucid Imagination

2008年 11月 04日

連載「Apache Solr でもっと賢く検索する」が公開された後に、Apache Solr には多くの新機能とパフォーマンス改善が加えられました。この記事では Solr と Lucene のコミッターである Grant Ingersoll が、Distributed Search から簡単なデータベースのインポート、そして組み込みスペルチェックや新しい拡張 API など、Solr 1.3 に盛り沢山に加えられた改善内容を詳しく説明します。

Apache Solr は主に HTTP をベースとしたオープンソースの検索サーバーで、Apache Lucene を基礎に構築されました。私が developerWorks の読者に Solr を紹介したのは 2007年の連載「Apache Solr でもっと賢く検索する」でしたが、最近リリースされた Solr 1.3 ではその連載で紹介した内容に加え、数々の新機能と機能強化が盛り込まれており、その内容についてフォローアップするには Solr 1.3 がリリースされた今が絶好のタイミングです。

Solr にはエンタープライズに対応した機能が豊富に備わっています。例えば、簡単な構成と管理、複数のクライアント言語のバインディング、索引の複製、キャッシング、統計、ロギングなどの機能です。これらの機能に加え、Apache Lucene 2.3 バージョンでの大々的なパフォーマンス向上を基に構築された Solr の1.3 リリースには、後方互換性を持つ新たなプラグ・アンド・プレイ・コンポーネント・アーキテクチャーも備わっています。そしてこの新しいアーキテクチャーがきっかけとなり、Solr を一層強化する新規コンポーネントが作成されています。例えば、1.3 リリースには以下の機能を目的としたコンポーネントが含まれます。

- 「もしかして」を考慮したスペルチェック機能
- 「類似する」 Document の検索機能
- 編集者の入力に基づいて検索結果の優先順を変更する機能 (有料プレイスメントとしても知られています)

さらに、既存の機能 (クエリー解析、検索、ファセット分類、デバッグなど) もコンポーネント化されたため、これらのコンポーネントをチェーニングすることによってカスタマイズした `SolrRequestHandler` を作成できるようになりました。そして最後に多くの企業にとって重要な点として、Solr ではデータベースのコンテンツに直接索引を付けることも、極めて大規模なシステ

ムをサポートするために Distributed Search という手段によってスケールアウトすることも可能になっています。

この記事では Solr について簡単に復習するための内容も記載しますが、前提としているのは、読者が Solr の基本概念を十分理解していることです。基本概念には、schema.xml と solrconfig.xml、索引付けおよび検索の基本、そして Solr で SolrRequestHandler が実行する内容などが含まれます。これらの基本概念についての理解が十分でない場合は、連載「Apache Solr でもっと賢く検索する」を読み返してください。また、この記事の「参考文献」も役立ちます。

この記事ではまず Solr の概念について簡単に復習した後、最新リリースを入手してインストールする方法を、前のバージョンからアップグレードする際の注意事項と併せて説明します。続いて、以前のバージョンからの重要な改善内容をいくつか取り上げ、最後に Solr の新機能について説明して締めくくります。

Solr の概念についての復習

概念上、Solr は大まかに以下の 4 つの領域に分けられます。

- スキーマ (schema.xml)
- 構成 (solrconfig.xml)
- 索引付け
- 検索

スキーマを理解するには、Solr から一歩離れて Lucene の Document という概念を理解する必要があります。`Document` は 1 つ以上の `Field` で構成され、`Field` は名前、コンテンツ、そしてコンテンツの処理方法に関するメタデータで構成されます。コンテンツを検索可能にする手段は分析です。分析プロセスでは、`Tokenizer` (入力ストリームを単語 (トークン) に分割する処理) とゼロ個以上の `TokenFilter` (トークンを変更 (語幹など) または削除する処理) とをチェーニングさせた処理が行われます。この分析プロセスをコードなしで簡単に構成できるようにするのが Solr スキーマです。Solr スキーマはより強い型付けを行うため、`Field` を `String`、`int`、`float`、あるいはその他のプリミティブ型やカスタム型に指定することもできます。

構成の面では、solrconfig.xml ファイルで Solr に索引付け、強調表示、ファセット分類、検索、その他の要求を処理する方法を指定するとともに、キャッシングの処理方法を指定する属性や、Lucene による索引の管理方法を指定する属性を指定します。構成がスキーマに依存することはありますが、スキーマが構成に依存することは決してありません。

索引付けと検索はどちらも、Solr サーバーに送信される HTTP リクエストによって行われます。索引付けを行うには、各 `Field` とそのコンテンツを記述する XML 文書を `POST` 操作で渡せばよいだけです。リスト 1 に、そうした文書の例として、apache-solr-1.3.0/example/exampledocs/ ディレクトリーに置かれたサンプル hd.xml 文書を記載します。

リスト 1. サンプル XML 文書

```
<add>
<doc>
  <field name="id">SP2514N</field>
  <field name="name">Samsung SpinPoint P120 SP2514N -
  hard drive - 250 GB - ATA-133</field>
  <field name="manu">Samsung Electronics Co. Ltd.</field>
  <field name="cat">electronics</field>
  <field name="cat">hard drive</field>
  <field name="features">7200RPM, 8MB cache, IDE Ultra ATA-133</field>
  <field name="features">NoiseGuard, SilentSeek technology, Fluid
  Dynamic Bearing (FDB) motor</field>
  <field name="price">92</field>
  <field name="popularity">6</field>
  <field name="inStock">true</field>
</doc>
</add>
```

検索も同じく、以下のようなHTTP `GET` を送信するだけで簡単に行えます。

```
http://localhost:8983/solr/select?indent=on&version=2.2&q=ipod&start=0&rows=10
      &fl=*%2Cscore&qt=standard&wt=standard
```

上記の例では、`ipod` というクエリーの実行を依頼し、10 件の検索結果を要求しています。クエリーで使用できるさまざまなオプションについての詳しい情報は、Solr Wiki に記載されています(「参考文献」を参照)。ちなみに Solr には現在、SolrJ というクライアントが付属していますが、このクライアントが、簡単に使える一連の Java™ クラスの中に HTTP リクエストの詳細すべてを隠します。SolrJ については、後のセクションで説明します。

より広範な Solr の設計コンテキストを理解するうえでは、ここまでで説明した Solr の概念の簡単な復習で十分なはずです。

Solr 1.3 のインストール方法

Solr を使用してこの記事の例を実行するには、以下のソフトウェアがインストールされている必要があります。

- Java 1.5 以降
- Web ブラウザー。これは、管理ページを表示するために使用します。私が使っているのは Firefox ですが、最近のほとんどのブラウザーで表示できるはずです
- サンプル `DataImportHandler` を実行するためのデータベースとその JDBC ドライバー。サンプルには PostgreSQL を使用しています。MySQL などでも動作するはずですが、その場合は、私が作成した SQL をお使いのデータベースに合わせて変更する必要があるかもしれません。
- サーブレット・コンテナー。この記事では、Solr にパッケージされている Jetty を使用するため、サーブレット・コンテナーを別途入手する必要はありません。一方、Tomcat やその他のコンテナーを使う場合でも、Solr は十分動作するはずです。

Solr を初めて使用する場合

前提条件のソフトウェアがインストールされている状態で、Solr バージョン 1.3.0 を Apache Mirrors Web サイトからダウンロードし、任意のディレクトリーに解凍します。すると、apache-

solr-1.3.0という名前のディレクトリーが作成されます。次に、端末 (コマンド・プロンプト) で以下のステップを実行してください。

1. `cd apache-solr-1.3.0/example` を実行します (Windows® では / の代わりに `\` を使用してください)。
2. `java -jar start.jar` を実行します。
   ログ出力に以下の行が表示されるまで待ちます。この行は、サーバーが起動したことを示します。
   ```
   2008-10-01 09:57:06.336::INFO:  Started SocketConnector @ 0.0.0.0:8983
   Oct 1, 2008 9:57:06 AM org.apache.solr.core.SolrCore registerSearcher
   INFO: [] Registered new searcher Searcher@d642fd main
   ```
3. Web ブラウザーで http://localhost:8983/solr にアクセスします。すると Solr のウェルカム・ページが表示されます。
4. 別の端末で、`cd apache-solr-1.3.0/example/exampledocs` を実行します。
   `java -jar post.jar *.xml` を実行します。これによって一連の文書が自動的に Solr に追加されます。
5. ブラウザーで、管理ページ (http://localhost:8983/solr/admin/form.jsp) からクエリーを試します。
   `ipod` を検索すると、図 1 の結果になります (一部省略)。

   図 1. 検索結果の例

これで Solr 1.3 は稼働状態となり、作業に取り掛かれるようになりました。この記事では、apache-solr-1.3.0/example/solr/conf ディレクトリーにあるサンプル・ファイル、solrconfig.xml および schema.xml を使用し、この両ファイルに変更を加えていきますが、その前に Solr 1.3 にアップグレードする際の問題を取り上げ、それからこの最新リリースでの機能強化について説明します。アップグレードしているのでなければ、「機能強化」のセクションをスキップして構いません。

Solr をアップグレードする場合

Solr 1.3.0 には以前の Solr リリースとの後方互換性がありますが、アップグレードする際に注意しなければならない点がいくつかあります。第一の注意点として、Solr の複製を使用している場合には、必ず最初に作業ノードをアップグレードしてからマスター・ノードをアップグレードしてください。

Solr の複製

Solr の複製では、いずれも Solr を実行する 1 つ以上の作業ノードが、索引のローカル・コピーをマスター・ノードで行われた変更に同期させます。複製により、パフォーマンスを損なうことなく Solr を拡張し、大量のクエリーを行うアプリケーションのニーズを満たすことができます。Solr ではこのプロセスを極めて効率的に管理することが可能です。詳細は、「参考文献」を参照してください。

第二の注意点として、このバージョンの Solr には新しいバージョンの Lucene が含まれています。これは事実上、Solr が内部の Lucene ファイル・フォーマットをアップグレードするため、以前のバージョンの Solr は新しいバージョンを読み取れない可能性があることを意味します。後でダウングレードが必要になった場合に備え、アップグレードを行う前に索引をバックアップしておくのが賢明です。

第三の注意点として、Solr 1.3 には Dr. Martin Porter の新しいバージョンの Snowball ステミング (語幹抽出) メカニズムも含まれています。このメカニズムを使って語幹を抽出する場合、(実際には起こりそうにないものの) 以前と同じ方法では単語の語幹が抽出されない可能性があります。これに対する最も安全な対策は、クエリー時の分析と索引分析での不一致がなくなるように、コンテンツの索引を付け直すことです。

一部のユーザーに関連する上記の問題を別とすれば、Solr 1.3 は以前のバージョンに簡単に置き換わるはずです。この記事の本題に入る準備が整ったところで、まずは Solr の既存の機能に対して強化された機能について説明します。

機能強化

Solr 1.1 と 1.2 はそのままの状態で十分有効に機能しましたが、(極めて単純なソフトウェアを除くすべてのソフトウェアの常として)、改善の余地がありました。Solr 1.3 にはサーバーの安定性とパフォーマンスを目的とした数多くのバグ・フィックスと改善が盛り込まれています。

パフォーマンスの向上

最新リリースの特徴として何よりもまず挙げられる点は、Lucene ライブラリーのアップグレードです。この最新の Lucene バージョンには、さまざまなところでパフォーマンスの改善が加えられています。私がテストしたところ、索引付けの速度が 5 倍向上しているだけでなく、その他の処理速度も 2 倍から 8 倍向上しているとレポートされました。嬉しいことに、索引付けの高速化はすべての Solr ユーザーに適用されます。さらに、パフォーマンス向上のほとんどには、構成の変更が必要ありません。

一方、solrconfig.xml で簡単に行える 1 つの構成変更によって、索引付けの際に使用するメモリー量をより適切にアプリケーションから制御することができます。バージョン 1.1 と 1.2 では、Solr

は文書のサイズをまったく考慮せずに、メモリー内の文書の数に従って、索引が付けられた文書をディスクに書き出します。そのため、メモリーが非効率的に使用される結果となることがよくありました。小さい文書の場合には、使用可能なメモリーとは関係なく文書が頻繁にフラッシュされることになったり、逆に多くのメモリーを必要とする大きな文書の場合には、フラッシュの頻度が不十分になったりすることがあるからです。しかしバージョン 1.3 では、solrconfig.xml の `<indexDefaults>` セクションにある `<ramBufferSizeMB>` オプションのおかげで、メモリー内の文書の数を指定するのではなく、文書をメモリー内にバッファリングするために使えるメモリーの量を指定できるようになっています。

その他の拡張の要点

Solr 1.3では、今まで以上に Solr の拡張、そして拡張機能の構成と再調整を簡単に行えるようになっています。これまで、新しい機能を実装するには `SolrRequestHandler` を作成しなければなりませんでした。この手法の問題は、他の `SolrRequestHandler` が持つ機能を再利用するのが容易ではないことです。例えば、より優れた方法でファセット分類を実行したい一方で、既存の照会機能と強調表示機能は維持したいというような場合です。このような問題に対処するため、Solr プロジェクトでは各種の `SolrRequestHandler` (`StandardRequestHandler` や `DismaxRequestHandler` など) を `SearchComponent` というコンポーネントにリファクタリングするという考えを思い付きました。これらの `SearchComponent` コンポーネントをチェーニングすることで、新しい `SolrRequestHandler` にするという方法です。今では、新しい `SearchComponent` の機能だけに専念し、その他すべての機能を最適に拡張、再利用、あるいは複製する方法については心配する必要はなくなりました。

一方、既存の `SolrRequestHandler` も今まで通り、すべてそのまま機能するので心配には及びません。ただし、このハンドラーは今では `SearchComponent` のラッパーに過ぎず、実際の作業を引き受けるのは `SearchComponent` になります。表 1 に、新たなコンポーネントである `SearchComponent` をいくつか具体的に示します (表 1 に記載したコンポーネントのうち、`MoreLikeThisComponent` と `SpellCheckComponent` の 2 つについては、この後でさらに詳しく説明します。「参考文献」に記載した `SearchComponent` のリンクも参照してください)。

表 1. よく使用される `SearchComponent`

名前	説明およびクエリー・サンプル
`QueryComponent`	クエリーの実行を Lucene に依頼し、`Document` のリストを返します。 `http://localhost:8983/solr/select?&q=iPod&start=0&rows=10`
`FacetComponent`	結果セットのファセットを決定します。 `http://localhost:8983/solr/select?&q=iPod&start=0&rows=10&facet=true&facet.field=inStock`
`MoreLikeThisComponent`	検索結果のそれぞれについて、同様の (つまり、「類似する」) 文書を見つけ、これらの結果も返します。 `http://localhost:8983/solr/select?&q=iPod&start=0&rows=10&mlt=true&mlt.fl=features&mlt.count=1`
`HighlightComponent`	検索結果のテキスト内で、クエリーで照会した語がある場所を強調表示します。 `http://localhost:8983/solr/select?&q=iPod&start=0&rows=10&hl=true&hl.fl=name`
`DebugComponent`	クエリーがどのように解析されたかに関する情報と、各文書に付けられたスコアの理由についての詳細を返します。

	`http://localhost:8983/solr/select?&q=iPod&start=0&rows=10&debugQuery=true`
`SpellCheckComponent`	入力されたクエリーをスペルチェックし、索引のコンテンツに基づいて、考えられる代わりのスペルを提案します。 `http://localhost:8983/solr/spellCheckCompRH?&q=iPood&start=0&rows=10&spellcheck=true&spellcheck.build=true`

デフォルトでは、すべての `SolrRequestHandler` に`QueryComponent`、`FacetComponent`、`MoreLikeThisComponent`、`HighlightComponent`、および `DebugComponent` が備わっています。独自のコンポーネントを追加する方法は、以下のとおりです。

1. `SearchComponent` クラスを継承します。
2. このコードを Solr が使用できるようにします (「参考文献」に記載されている Solr Plugins Wiki ページのリンクを参照してください。)。
3. コードを solrconfig.xml 内で構成します。

一例として、`com.grantingersoll.MyGreatComponent` という名前の `SearchComponent` を作成し、これを Solr が使用できるようにした後、今度はこのコンポーネントを `SolrRequestHandler` に挿入して、このコンポーネントを使ってクエリーを実行できるようにするとします。この場合、まずは Solr がクラスをインスタンス化する方法を認識できるように、このコンポーネントを宣言する必要があります (リスト 2 を参照)。

リスト 2. コンポーネントの宣言

```
  <searchComponent name="myGreatComp" class="com.grantingersoll.MyGreatComponent"/>
```

次に、このコンポーネントをどの `SolrRequestHandler` に追加するかを Solr に指示します。この場合、以下の 3 つのいずれかの方法を使えます。

- すべての `SearchComponent` を明示的に宣言する方法 (リスト 3 を参照)

  ### リスト 3. すべての `SearchComponent` を明示的に宣言する方法

  ```
  <requestHandler name="/greatHandler" class="solr.SearchHandler">
      <arr name="components">
        <str>query</str>
        <str>facet</str>
        <str>myGreatComp</str>
        <str>highlight</str>
        <str>debug</str>
      </arr>
  </requestHandler>
  ```
- 既存のチェーンの前にコンポーネントを追加する方法 (リスト 4 を参照)

  ### リスト 4. 既存のチェーンの前にコンポーネントを追加する方法

  ```
  <requestHandler name="/greatHandler" class="solr.SearchHandler">
      <arr name="first-components">
        <str>myGreatComp</str>
      </arr>
  </requestHandler>
  ```
- 既存のチェーンの後にコンポーネントを追加する方法 (リスト 5 を参照)

リスト 5. 既存のチェーンの後にコンポーネントを追加する方法

```
<requestHandler name="/greatHandler" class="solr.SearchHandler">
    <arr name="last-components">
      <str>myGreatComp</str>
    </arr>
</requestHandler>
```

DebugComponent に関する注意事項

first-components または last-components の方法を使用すると、DebugComponent は常にチェーンの最終コンポーネントになります。そのため、DebugComponent がレポートする値 (クエリーの結果など) をコンポーネントが変更するときには、これらの方法が特に役立ちます。

SearchComponent へのリファクタリングと同様の方法で、SolrRequestHandler からクエリーの解析を分離することもできるようになりました。そのため DismaxQParser はどの SolrRequestHandler でも使用することができます。それには、以下のように defType パラメーターを渡します。 defType parameter. For example:

```
http://localhost:8983/solr/select?&q=iPod&start=0&rows=10&defType=dismax&qf=name
```

上記では、標準 Lucene クエリー・パーサーの代わりに Dismax クエリー・パーサーを使ってクエリーを解析します。

あるいは、QParser と QParserPlugin を継承して Solr から使用できるようにし、それから solrconfig.xml で構成することによって独自のクエリー・パーサーを作成することもできます。例えば com.grantingersoll.MyGreatQParser と com.grantingersoll.MyGreatQParserPlugin を作成し、Solr から使用できるようにしてから solrconfig.xml で以下のように構成するとします。

```
<queryParser name="greatParser" class="com.grantingersoll.MyGreatQParserPlugin"/>
```

この新しいパーサーを使ってクエリーを解析するには、defType=greatParser というキーと値のペアをクエリー・リクエストに追加します。

この他にも、最新の Solr リリースには多くの改善が加えられています。詳細を調べるには、まずは「参考文献」に記載したリリース・ノートのリンクにアクセスしてください。この記事では、続いて Solr の新機能についての話題に移ります。

新機能

Solr 1.3 が実現する一連の強力な機能は、この検索サーバーを今まで以上に魅力あるものにしています。この記事の残りでは、新しい Solr の機能を紹介し、これらの機能をアプリケーションに組み込む方法を説明します。その方法を実演するために、ここでは RSS フィードをそのフィードに対する評価と組み合わせる単純なアプリケーションを作成します。この評価はデータベースに保管されます。RSS フィードを取得するのは、私の Lucene ブログの RSS フィードからです。この単純なアプリケーションを例に、以下の機能を使用する方法を説明します。

- DataImportHandler
- MoreLikeThisComponent

- `QueryElevationComponent` (「編集による結果プレイスメント」と呼びます)
- SolrJ
- Distributed search (設定の詳細は抜きにして、そのアーキテクチャーについて説明します)

この例に従うには、サンプル・アプリケーションをダウンロードして、以下の手順を実行してください。

1. sample.zip を apache-solr-1.3.0/example/ ディレクトリーにコピーします。
2. `unzip sample.zip` を実行します。
3. `java -Dsolr.solr.home=solr-dw -jar start.jar` を実行して、Solr を起動 (または再起動) します。
4. データベース管理者として、`solr_dw` という名前のデータベース・ユーザーを作成します。その方法については、お使いのデータベースの説明書を参照してください。この例では、PostgreSQL を使用して `create user solr_dw;` を実行しました。
5. `create database solr_dw with OWNER = solr_dw;` を実行して、このユーザーのデータベース、`solr_dw` を作成します。
6. コマンドラインから src/sql/create.sql ステートメント、`psql -U solr_dw -f create.sql solr_dw` を実行します。この例での出力は、以下のとおりです。

```
 gsi@localhost>psql -U solr_dw -f create.sql solr_dw
psql:create.sql:1: ERROR:  table "feeds" does not exist
psql:create.sql:2: NOTICE:  CREATE TABLE / PRIMARY KEY will create \
  implicit index "feeds_pkey" for table "feeds"
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

データベースやその他のソースからデータをインポートする

大量の構造化データと非構造化データが混在する今の時代では、データベースや XML/HTML ファイル、またはその他のデータ・ソースからデータをインポートして、そのデータを検索可能にすることは当たり前のニーズです。以前はデータベース、ファイル・システム、または RSS フィードへ接続するには独自のカスタム・コードを作成しなければなりませんでしたが、今は Solr の `DataImportHandler` (DIH) がこの不足を補い、データベース (JDBC を使用)、RSS フィード、Web ページ、そしてファイルからのインポートを可能にしています。DIH は apache-1.3.0/contrib/dataimporthandler に含まれており、apache-1.3.0/dist/apache-solr-dataimporthandler-1.3.0.jar に JAR ファイルとして配布されています。

`DataImportHandler` に関する注意事項

DataImportHandler はファイルや Web の クローラーではないため、そのままの状態ではバイナリー・ファイル形式 (MS Office、Adobe PDF、その他の独自仕様のフォーマットなど) のコンテンツ抽出をサポートしません。この記事では DIH について事細かに説明するスペースはないので、詳細については「参考文献」を参照してください。

DIH の概念的な構成要素としては、以下のように分類できます。

- `DataSource`: コンテンツの取得元となるデータベース、Web ページ、RSS フィード、または XML ファイルです。

- 文書/エンティティーの宣言: `DataSource` のコンテンツと Solr スキーマとの間のマッピングを指定します。
- インポート: フル・インポート、または変更されたエンティティーのみの差分インポートを行う Solr のコマンドです。
- `EntityProcessor`: マッピングを行うコード。Solr にはあらかじめ 4 つの実装が用意されています。
- `FileListEntityProcessor`: ディレクトリーを繰り返し処理してファイルをインポートします。
- `SqlEntityProcessor`: データベースに接続してレコードをインポートします。
- `CachedSqlEntityProcessor`: キャッシングを `SqlEntityProcessor` に追加します。
- `XPathEntityProcessor`: XPath ステートメントを使用して、XML ファイルからコンテンツを抽出します。
- `Transformer`: オプション。インポートされたコンテンツを変換してから Solr に追加するユーザー定義コードです。例えば、`DateFormatTransformer` では日付の正規化を行うことができます。
- 変数置換: プレースホルダーの変数を実行時の値に置換します。

まず始めに、DIH を Solr に関連付ける `SolrRequestHandler` をセットアップする必要があります。solr-dw/rss/conf/solrconfig.xml ファイルでの構成は、リスト 6 のようになります。

リスト 6. DIH と Solr との関連付け

```
<requestHandler name="/dataimport"
  class="org.apache.solr.handler.dataimport.DataImportHandler">
<lst name="defaults">
  <str name="config">rss-data-config.xml</str>
</lst>
</requestHandler>
```

この構成では、`DataImportHandler` インスタンスには http://localhost:8983/solr/rss/dataimport でアクセスできること、そしてこのインスタンスは (solr_dw/rss/conf ディレクトリーに配置されている) rss-data-config.xml という名前の構成ファイルを使用してセットアップ情報を取得することを指定しています。ここまでは至って簡単です。

rss-data-config.xml ファイルの中身を見てみると、このファイルでは `DataSources`、エンティティー、`Transformer` のすべてが宣言されて、使用されています。この例で最初に示されている XML タグ (ルート要素の後) が、2 つの `DataSource` 宣言です (リスト 7 を参照)。

リスト 7. `DataSource` の宣言

```
<dataSource name="ratings" driver="org.postgresql.Driver"
      url="jdbc:postgresql://localhost:5432/solr_dw" user="solr_dw" />
<dataSource name="rss" type="HttpDataSource" encoding="UTF-8"/>
```

リスト 7 の最初の宣言は、データベースと接続するための `DataSource` をセットアップします。その名前が `ratings` となっている理由は、このデータベースに評価の情報を保管するからです。この例ではデータベース・ユーザーにパスワードを設定しませんでしたが、このタグにはパスワード属性を追加できることに注意してください。JDBC のセットアップについての知識がある読者には、この `DataSource` 宣言はかなり馴染み深いはずです。2 番目の `rss` という名前の `DataSource`

は、HTTP を使用してコンテンツを取得することを宣言しています。この `DataSource` の URL は後で宣言します。

次に説明しておく価値のあるタグは、`<entity>` タグです。このタグで、RSS フィードのコンテンツとデータベースをどのように Solr の `Document` へとマッピングするかを指定します。エンティティーとは、まとめて 1 つの文書として索引が付けられるコンテンツの単位のことです。例えばデータベースでは、エンティティー宣言が、各行を `Document` 内の `Field` に変換する方法を指定します。1 つのエンティティーに 1 つもしくは複数のエンティティーを含め、これらの子エンティティーを `Document` 全体の `Field` 構造にあてはめていくこともできます。

この時点で rss-data-config.xml の注釈付きサンプルを見ると、エンティティーのほとんどの詳細を理解できるはずです。このサンプルでは、メインのエンティティーが RSS フィードからコンテンツを取得し、そのコンテンツをデータベース内の行に相関させて評価を取得します。リスト 8 に、RSS フィードの例を一部省略して記載します。

リスト 8. 省略した RSS フィード

```
<rss version="2.0"
xmlns:content="http://purl.org/rss/1.0/modules/content/"
xmlns:wfw="http://wellformedweb.org/CommentAPI/"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:atom="http://www.w3.org/2005/Atom"
>
<channel>
<title>Grant's Grunts: Lucene Edition</title>
<link>http://lucene.grantingersoll.com</link>
<description>Thoughts on Apache Lucene, Mahout,
    Solr, Tika and Nutch</description>
<pubDate>Wed, 01 Oct 2008 12:36:02 +0000</pubDate>
<item>
  <title>Charlotte JUG >> OCT 15TH - 6PM -
    Search and Text Analysis</title>
  <link>http://lucene.grantingersoll.com/2008/10/01/
    charlotte-jug-%c2%bb-oct-15th-6pm-search-and-text-analysis/</link>
  <pubDate>Wed, 01 Oct 2008 12:36:02 +0000</pubDate>
  <category><![CDATA[Lucene]]></category>
  <category><![CDATA[Solr]]></category>
  <guid isPermaLink="false">http://lucene.grantingersoll.com/?p=112</guid>
  <description><![CDATA[Charlotte JUG >> OCT 15TH - 6PM - Search and Text Analysis
I will be speaking at the Charlotte Java Users Group on Oct. 15th, covering things
like Lucene, Solr, OpenNLP and Mahout, amongst other things.
]]></description>
</item>
</channel>
```

一方、データベース内の行には、フィードに含まれる記事の URL、評価 (私が無作為に作ったものです)、そして変更日が含まれます。あとは、これを Solr にマッピングすればよいだけです。その方法を説明するため、リスト 9 で rss-data-config.xml でのエンティティー宣言を 1 行ごとに解説します (見やすくするために、行番号を付けて改行を入れています)。

リスト 9. エンティティーの宣言

```
1. <entity name="solrFeed"
2.pk="link"
3.url="http://lucene.grantingersoll.com/category/solr/feed"
4.processor="XPathEntityProcessor"
5.forEach="/rss/channel | /rss/channel/item"
```

```
6.            dataSource="rss"
7.         transformer="DateFormatTransformer">
8.  <field column="source" xpath="/rss/channel/title"
        commonField="true" />
9.  <field column="source-link" xpath="/rss/channel/link"
        commonField="true" />
10.  <field column="title" xpath="/rss/channel/item/title" />
11.  <field column="link" xpath="/rss/channel/item/link" />
12.  <field column="description"
        xpath="/rss/channel/item/description" />
13.  <field column="category" xpath="/rss/channel/item/category" />
14.  <field column="content" xpath="/rss/channel/item/content" />
15.  <field column="date" xpath="/rss/channel/item/pubDate"
        dateTimeFormat="EEE, dd MMM yyyy HH:mm:ss Z" />
16.  <entity name="rating" pk="feed"
      query="select rating from feeds where feed = '${solrFeed.link}'"
17.   deltaQuery="select rating from feeds where feed = '${solrFeed.link}'
            AND last_modified > '${dataimporter.last_index_time}'"
18.          dataSource="ratings"
19.          >
20.    <field column="rating" name="rating"/>
21.  </entity>
22. </entity>
```

- 行 1: エンティティーの名前 (`solrFeed`)。
- 行 2: Field に設定するオプションの主キー。差分インポートを行う場合にのみ必要です。
- 行 3: 取得する URL。この例では、Solr に関する私のブログ投稿です。
- 行 4: 未処理ソースのコンテンツをマッピングするために使用する `EntityProcessor`。
- 行 5: XML からレコードを取得する方法を指定する XPath 式 (XPath は、XML ファイルに含まれる特定の要素または属性を指定する手段となります。XPath 式について十分に理解していない場合は、「参考文献」を参照してください)。
- 行 6: 使用する `DataSource`。名前で指定します。
- 行 7: ストリングを `java.util.Date` に解析するために使用する `DateFormatTransformer`。
- 行 8: チャネル・タイトル (ブログの名前) を source という名前の Solr スキーマ・フィールドにマッピングします。このマッピングはチャネルごとに 1 回だけ行われるため、この値をすべての Field で使用するように `commonField` 属性を設定します。
- 行 9 ~ 14: RSS フィードのその他のさまざまな部分を Solr の `Field` にマッピングします。
- 行 15: 公開日をマッピングしますが、`DateFormatTransformer` を使用して値を `java.util.Date` オブジェクトとして解析します。
- 行 16 ~ 21: 各記事の評価をデータベースから取得する子エンティティー。
- 行 16: `query` 属性が、実行する SQL を指定します。`${solrFeed.link}` の値は、変数置換によって各記事の URL に置き換わります。
- 行 17: 差分インポートを行うときに実行するクエリー。`${dataimporter.last_index_time}` は DIH によって指定されます。
- 行 18: JDBC の `DataSource` を使用してください。
- 行 20: データベース内の rating 列を rating フィールドにマッピングします。name 属性を指定しない場合、デフォルトで列名が使用されます。

次のステップでは、インポートを実行します。それには、以下の HTTP リクエストを送信します。

上記のリクエストは、索引からすべての文書を削除した後、フル・インポートを行います。繰り返しますが、このリクエストは、まず始めに索引からすべての文書を削除すると警告しておきま

す。http://localhost:8983/solr/rss/dataimport をブラウズすることで、随時、DIH のステータスを調べることができます。この例での出力は、リスト 10 のようになります。

リスト 10. インポート結果

```
<response>
<lst name="responseHeader">
 <int name="status">0</int>
 <int name="QTime">0</int>
</lst>
<lst name="initArgs">
 <lst name="defaults">
  <str name="config">rss-data-config.xml</str>
 </lst>
</lst>
<str name="status">idle</str>
<str name="importResponse"/>
<lst name="statusMessages">
 <str name="Total Requests made to DataSource">11</str>
 <str name="Total Rows Fetched">13</str>
 <str name="Total Documents Skipped">0</str>
 <str name="Full Dump Started">2008-10-03 10:51:07</str>
 <str name="">Indexing completed. Added/Updated: 10 documents.
  Deleted 0 documents.</str>
 <str name="Committed">2008-10-03 10:51:18</str>
 <str name="Optimized">2008-10-03 10:51:18</str>
 <str name="Time taken ">0:0:11.50</str>
</lst>
<str name="WARNING">This response format is experimental.  It is
  likely to change in the future.</str>
</response>
```

差分インポート機能

データベースの操作時には、フル・インポートを行った後に、最後にインポートしてから変更されたレコードのみをインポートすることができます。この機能は差分インポートと呼ばれます。残念ながら、この機能は RSS フィードにはまだ使用できません。使用できるとしたら、そのためのコマンドは以下のようになるでしょう。
`http://localhost:8983/solr/rss/dataimport?command=delta-import`

索引を付ける文書の数は、この例とは異なる場合も考えられます (私がフィードに追加する Solr の記事は他にも考えられるため)。文書に索引が付けられたとことで、索引に対してクエリーを実行できるようになりました。例えば`http://localhost:8983/solr/rss/select/?q=*%3A*&version=2.2&start=0&rows=10&indent=on` と実行すると、索引が付けられた 10 の文書すべてが返されます。

以上の説明で、DIH を使い始められるようになったはずです。詳しく探っていくうちに、変数置換が機能する仕組みや、独自の `Transformer` を作成する方法に興味が湧いてくると思います。これらの内容についての詳細を学ぶには、「参考文献」に記載されている `DataImportHandler` Wiki ページのリンクを参照してください。この記事で次に取り上げるのは、`MoreLikeThisComponent`を使用して同じようなページを検索する方法です。

同様のページを検索する

`MoreLikeThisComponent` と Solr スキーマ

MLT では、フィールドが保管されているか、あるいはフィールドが文書主体の方法で情報を保管する term vector を使用する必要があります。MLT は文書のコンテンツを使用して、その文書で特に重要な用語が何であるのかを判断します。そして、その重要だと判断した

用語と、元のクエリー語とを使って新しいクエリーを作成し、そのクエリーを実行依頼して新たな結果を取得します。このプロセスは、term vector を使うと遙かに効率的になります。term vector を使用するには、schema.xml の <field> 宣言に termVectors="true" を追加するだけです。

Google で検索を行うと、すべての検索結果に「関連ページ」というリンクが含まれていることにおそらく気付くはずです。このリンクをクリックすると別の検索リクエストが出され、元の結果と類似した文書が検索されます。Solr ではこれと同じ機能を、`MoreLikeThisComponent` (MLT) と `MoreLikeThisHandler` で実現しています。MLT は前述したように、標準の `SolrRequestHandler` に組み込まれています。一方、`MoreLikeThisHandler` は MLT を組み込んでいるだけでなく、オプションもいくつか追加されていますが、別個のリクエストを発行しなければなりません。ここでは、使用される可能性が高いことから、MLT に重点を絞って説明します。幸い、必要な設定はありません。そのまますぐに MLT を使ってクエリーを実行することができます。

リクエストには多数の HTTP クエリー・パラメーターを追加できますが、その大部分には賢いデフォルト値があります。そこで、MLT を使い始めるときに知っておかなければならないパラメーターにだけ焦点を絞ることにします。表 2 に、これらのパラメーターを記載します (詳細については、「参考文献」に記載されている Solr Wiki の MLT ページのリンクを参照してください)。

表 2. `MoreLikeThisComponent` のパラメーター

パラメーター	説明	値の範囲
`mlt`	クエリーを実行する際に `MoreLikeThisComponent` のオン/オフを切り替えるブール値。	true | false
`mlt.count`	オプション。結果ごとに検索する同様の文書の数。	> 0
`mlt.fl`	MLT クエリーを作成するために使用するフィールド。	保管されているか、term vector を持つスキーマ内の任意のフィールド
`mlt.maxqt`	オプション。クエリー語の最大数。長い文書にはそれだけ重要な用語も多いため、MTL クエリーがかなり大きくなり、速度の低下や、恐ろしい `TooManyClausesException` の結果をもたらす可能性があります。このパラメーターを指定することで、特に重要な用語だけが保持されるようになります。	> 0

以下のサンプル・クエリーを使用して、返された結果の `moreLikeThis` セクションを調べてみてください。

```
http://localhost:8983/solr/rss/select/?q=*%3A*&start=0&rows=10&mlt=true
  &mlt.fl=description&mlt.count=3
```

```
http://localhost:8983/solr/rss/select/?q=solr&version=2.2&start=0&rows=10
  &indent=on&mlt=true&mlt.fl=description&mlt.fl=title&mlt.count=3
```

次は、「もしかして」(スペルチェック) をアプリケーションに追加する方法を説明します。

スペルの候補を表示する

Lucene と Solr には長いこと、スペルチェック機能がありましたが、これらの機能をシームレスに使えるようになったのは、`SearchComponent` アーキテクチャーが追加されてからのことです。

今では、クエリーを送信して対象用語の検索結果を返させるだけでなく、クエリーに含まれる用語に他のスペルの候補が考えられる場合には、スペルを提案させることもできます。これらの候補を使用して、Google のような「もしかして:」、あるいは Yahoo! のような「~ではありませんか？」という代替案を表示することができます。

組み込みスペルチェックの利点は、索引のトークンに基づいて候補を提案できることです (またそうあるべきです)。つまり、提案されるのは必ずしも辞書の正しいスペルの単語とは限りません。クエリー語のスペル (誤ったスペルも含め) に基づき、似かよったスペルを提案します。例えば、非常に多くの人々が hockey という単語を誤って hockei とスペルしているとします。その場合、hockey を検索するユーザーは、hockei という単語が含まれる文書も見つけなければなりません。文書の著者が正しくスペルできないとしても、hockei が含まれる文書は関連性があるからです。

MLT とは異なり、`SpellCheckComponent` には、solrconfig.xml ファイルと schema.xml ファイルでの構成は必要がありません。何よりもまず必要なのは、スキーマが `Field` とこれに関連する、スペルの辞書として機能するコンテンツが含まれる `FieldType` を宣言することです。原則として、この `FieldType` の分析プロセスは単純なままにして、語幹の抽出やその他のトークン変更などの処理は行わないようにしなければなりません。例えば私のサンプル `FieldType` では、その `<analyzer>` をリスト 11 のように宣言しています。

リスト 11. `<analyzer>` の宣言

```
<fieldType name="textSpell" class="solr.TextField" positionIncrementGap="100" >
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  </analyzer>
</fieldType>
```

この `<analyzer>` は、基本的なトークン化を行い (原則的には空白で分割)、トークンを小文字にしてから重複を削除します。語幹は抽出しません。同期語を拡張することもなく、複雑なところは何もありません。schema.xml の下のほうでは、`textSpell <fieldType>` を使用する、`spell` という名前の `field` を宣言しています。次に、リスト 12 のように `<searchComponent>` を宣言して、必要な部分を solrconfig.xml で組み立てています。

リスト 12. `<searchComponent>` の宣言

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
    <str name="queryAnalyzerFieldType">textSpell</str>
    <lst name="spellchecker">
      <str name="name">default</str>
      <str name="field">spell</str>
      <str name="spellcheckIndexDir">./spellcheckerDefault</str>
    </lst>
</searchComponent>
```

この例では、`queryAnalyzerFieldType` で前に宣言した `textSpell <fieldType>` を関連付けています (前述の `last-components` の方法を使用して、コンポーネントを solrconfig.xml の Dismax および標準 `SolrRequestHandler` 宣言に追加している点に注意してください)。そのため、入力されたクエリーは、スペル用索引との比較で適切に分析されることになります。残りの構成オプションで

は、スペル・チェッカーの名前、スペル用索引の作成に使用するコンテンツが含まれる `Field`、そしてこの索引を保管するディスク上の場所を指定しています。

すべてを構成し終わったら、スペル用索引をビルドする必要があります。そのためには、コンポーネントに以下の HTTP リクエストを送信します。

```
http://localhost:8983/solr/rss/select/?q=foo&spellcheck=true&spellcheck.build=true
```

スペルチェック・ビルドのワークフロー

スペルチェック用索引は、ビルドしてからでないとクエリーを実行することができません。初期ビルドの後、索引を再ビルドする頻度を (アプリケーションによって) スケジュールする必要が出てくる場合があります。また、solrconfig.xml の postCommit イベント・リスナーを使用して、コミット後に再ビルドさせることもできます。再ビルドの頻度は、索引にどれだけ多くの変更を加えるかによって決まります。ただし、初期索引が確立されてから辞書が大幅に変更されることはほとんど考えられないので、再ビルドの頻度はそれほど重要ではありません。

索引がビルドされると、通常どおりのクエリーに `spellcheck=true` パラメーターを追加することで、候補が返されるようになります。リスト 13 は、スペルチェック機能をオンにした場合の例です。

リスト 13. スペルチェックを実演するクエリー

```
http://localhost:8983/solr/rss/select/?q=holr&spellcheck=true
```

リスト 13 のクエリーを実行すると、結果は何も返されないものの、以下の候補が提供されます。

```
<lst name="spellcheck">
 <lst name="suggestions">
  <lst name="holr">
 <int name="numFound">1</int>
 <int name="startOffset">0</int>

 <int name="endOffset">4</int>
 <arr name="suggestion">
  <str>solr</str>
 </arr>
  </lst>
 </lst>
</lst>
```

もう一歩踏み込んで、複数の単語からなるクエリーでスペルチェックを実行することもできます。しかもコンポーネントは、各単語に最適な候補を使用し、それらを並べた新しいクエリーの候補を自動的に作成することも可能です。それには、ミススペルのクエリーと同じく、`spellcheck.collate=true` パラメーターを追加します。

```
http://localhost:8983/solr/rss/select/?q=holr+foo&spellcheck=true&indent=on
&spellcheck.collate=true
```

上記のクエリーがもたらす結果である、`<str name="collation">solr for</str>` は、候補の一部です。ただしこの照合結果は、クエリー語を `AND` 処理するかどうか次第では、実際には結果を返さない場合があることに注意してください。

スペル・チェッカーは、結果として返す候補の数や結果の品質などに関するその他のクエリー・パラメーターを使用することもできます。`SpellCheckComponent` についての詳細は、「参考文献」に記載されている Solr Wiki ページのリンクを参照してください。

次に取り上げるのは、「有料プレイスメント」で自然な結果の順序に上書きする方法です。

編集による結果プレイスメント

検索エンジンはユーザーのクエリーに関連する文書だけを返すのが理想的ですが、実際には編集機能 (他にもっと適切な言葉が見つからないため) によって、特定のクエリーに対しては検索結果の特定の場所に特定の文書が表示されるようにしなければならない場合がよくあります。このようにする理由はさまざまです。ひょっとすると、「配置した」文書が実際に検索結果として最上位にランクされるかもしれませんが、企業が顧客に同様の製品よりも利益率が高い製品を見つけてもらいたいという場合もあれば、サード・パーティーが一連のクエリー語に対して特定のランキングを与えるために費用を支払うこともあり得ます。理由が何であれ、通常の関連性ランキングを使用して、特定のクエリーに対して特定の文書が特定の場所に表示されるようにすることは大抵の場合、容易なことではありません (不可能と言う人もいます)。さらに、検索エンジンがこのような文書の配置を 1 つのクエリーに対してできたとしても、おそらく処理中の他の 50 のクエリーを中断することになるでしょう。こうした認識から、現実世界における検索の基本原則の 1 つが引き出されます。それは、ユーザーがクエリーを入力したからと言って、実際に索引を検索して文書にスコアを付ける必要は必ずしもないということです。検索エンジンの作成を仕事としている者の言葉としては奇妙に聞こえると思いますが、こう考えてみてください。一般的なクエリーをキャッシュに入れて単に結果を検索するだけという方法も (Solr はこれを実行できます)、あるいは前述のいずれかの理由のために結果を「ハードコーディング」するという方法もあり得ます。

Solr はこれを可能にする手段として、`QueryElevationComponent` という謎めいた名前の `SearchComponent` を使用します。このコンポーネントをサンプル・アプリケーションで構成するには、リスト 14 のように宣言します。

リスト 14. `QueryElevationComponent` の宣言

```
<searchComponent name="elevator"
  class="org.apache.solr.handler.component.QueryElevationComponent"
  >
    <!-- pick a fieldType to analyze queries -->
    <str name="queryFieldType">string</str>
    <str name="config-file">elevate.xml</str>
  </searchComponent>
```

`queryFieldType` 属性で指定するのは、送られてきたクエリーを elevation の処理 (検索結果の特定の場所に特定の文書を表示する処理) の対象となるクエリーと突き合わせる方法です。`string FieldType` が指定されている場合は、簡単にするために分析が行われないので、クエリーは完全に一致するストリングでなければなりません。`config-file` 属性は、クエリーとそれに関連する結果が含まれるファイルを指定します。この情報は別個のファイルに保管されるため、外部で編集することができます。このファイルは Solr の conf ディレクトリーまたは data ディレクトリーのいずれかに配置しなければなりません。data ディレクトリーにこのファイルがない場合、Solr が索引を再ロードする必要があるときに必ず再ロードされることになります。

サンプル・アプリケーションでは、elevate.xml を conf ディレクトリーに保管します。このファイルには、クエリー「Charlotte」のエントリーを追加した他、さらに 3 つのエントリーも追加しました (リスト 15 を参照)。

リスト 15. サンプル elevate.xml 構成

```
<query text="Solr">
<doc
 id="http://lucene.grantingersoll.com/2008/06/21/solr-spell-checking-addition/"/>
<doc
  <!-- Line break is for formatting purposes -->
 id="http://lucene.grantingersoll.com/2008/10/01/\
      charlotte-jug-%c2%bb-oct-15th-6pm-search-and-text-analysis/"
         />
<doc
 id="http://lucene.grantingersoll.com/2008/08/27/solr-logo-contest/" exclude="true"/>
</query>
```

リスト 15 では、最初のリンクを常に 2 番目のリンクより上に配置し、3 番目のリンクを結果から完全に除外するように指定しています。その後の結果は、通常の順序です。通常の結果を確認するには (コンポーネントが含まれる場合、elevation がデフォルトで有効になります)、以下のクエリーを実行してください。

```
http://localhost:8983/solr/rss/select/?q=Solr&version=2.2&start=0&rows=10&indent=on
  &fl=link&enableElevation=false
```

elevation を有効にした結果を見るには、以下を試します。

```
http://localhost:8983/solr/rss/select/?q=Solr&version=2.2&start=0&rows=10&indent=on
  &fl=link&enableElevation=true
```

elevation 入力が挿入されていることがわかるはずです。

編集による結果プレイスメントについての説明は以上です。これで、特定の検索に対する検索結果を他の品質を損なうことなく、簡単に変更できるようになったはずです。

SolrJ

連載「Apache Solr でもっと賢く検索する」では、Apache HTTPClient を使用し、Java プラットフォームを介して Solr と通信する単純なクライアントを使いました。バージョン 1.3 の Solr には、HTTP 接続の面倒な詳細も、XML コマンドもすべて隠す、使いやすい Java ベースの API が備わっています。SolrJ と呼ばれるこの新しいクライアントは、Java コードでの Solr の操作をさらに簡単にします。SolrJ の API が明確に定義されたメソッド呼び出しによって、索引付け、検索、ソート、そしてファセット分類の実行を簡易化するからです。

ここでもやはり、単純な例を使うと理解しやすいと思います。サンプル・ダウンロードに、SolrJExample.java という名前の Java ファイルが含まれています (コンパイル方法については、ダウンロードに含まれる README.txt を参照してください)。このファイルでは、いくつかの文書の索引を Solr に設定してから、結果に対してファセット分類を行うクエリーを実行します。このファイルが最初に行うのは、`SolrServer server = new CommonsHttpSolrServer`("http://localhost:8983/solr/rss"); によって Solr インスタンスとの接続を確立することです。これによって、HTTP を介して Solr と対話する `SolrServer` インスタンスが作成されます。次に、索引を付け

る対象とするコンテンツをラップするいくつかの `SolrInputDocument` を作成します (リスト 16 を参照)。

リスト 16. SolrJ を使用した索引付け

```
Collection<SolrInputDocument> docs = new HashSet<SolrInputDocument>();
for (int i = 0; i < 10; i++) {
  SolrInputDocument doc = new SolrInputDocument();
  doc.addField("link", "http://non-existent-url.foo/" + i + ".html");
  doc.addField("source", "Blog #" + i);
  doc.addField("source-link", "http://non-existent-url.foo/index.html");
  doc.addField("subject", "Subject: " + i);
  doc.addField("title", "Title: " + i);
  doc.addField("content", "This is the " + i + "(th|nd|rd) piece of content.");
  doc.addField("category", CATEGORIES[rand.nextInt(CATEGORIES.length)]);
  doc.addField("rating", i);
  //System.out.println("Doc[" + i + "] is " + doc);
  docs.add(doc);
}
```

リスト 16 のループでは、`SolrInputDocument` (その下の見せかけの `Map`) を作成してから、そこに `Fields` を追加しているだけです。これをコレクションに追加し、すべての文書を一度に Solr に送信できるようにします。こうすることによって、索引付けを大幅に高速化し、HTTP でのリクエストの送信に伴うオーバーヘッドを少なくすることができます。次に呼び出す `UpdateResponse response = server.add(docs);` が、文書をシリアライズして Solr に送信するという芸当のすべてを行います。`UpdateResponse` の戻り値には、文書の処理にかかった時間についての情報が含まれます。これらの文書は検索に使えるようにしたいので、続いて `server.commit();` というコミット・コマンドを実行します。

索引付けの後には当然、サーバーに対してクエリーを実行することになります。その方法を、リスト 17 の注釈付きのコードで説明します。

リスト 17. サーバーに対するクエリーの実行

```
//create the query
SolrQuery query = new SolrQuery("content:piece");
//indicate we want facets
query.setFacet(true);
//indicate what field to facet on
query.addFacetField("category");
//we only want facets that have at least one entry
query.setFacetMinCount(1);
//run the query
QueryResponse results = server.query(query);
System.out.println("Query Results: " + results);
//print out the facets
List<FacetField> facets = results.getFacetFields();
for (FacetField facet : facets) {
  System.out.println("Facet:" + facet);
}
```

この単純なクエリーの例では、まず `content:piece` のクエリーによって `SolrQuery` インスタンスをセットアップします。次に、少なくとも 1 つはエントリーを持つすべてのファセットに関するファセット情報を取得するように指示します。そして最後に、`server.query(query)` 呼び出しによってクエリーの実行を依頼し、結果を出力します。これは確かに平凡な例ですが、ここには Solr を操作する際の一連の共通タスクが示されているので、何が可能か (強調表示、ソートなど)

についてより深く考えるきっかけとなるはずです。SolrJ でのクエリーに使用できるオプションについての詳細は、「参考文献」で紹介している SolrJ に関するリンクを参照してください。

Distributed Search で索引のサイズをスケーリングする

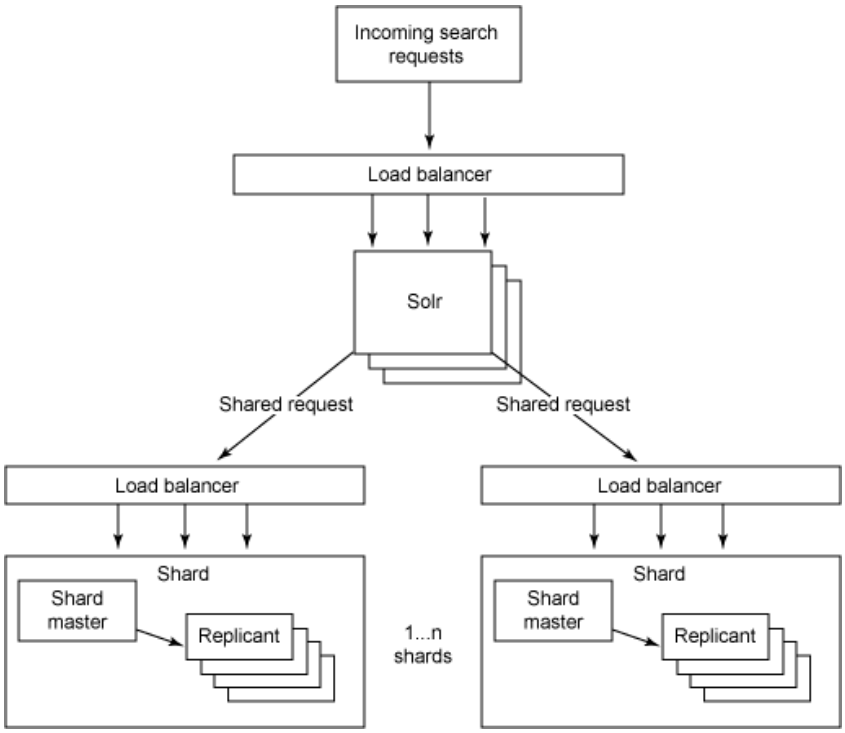
1.3 リリースに至るまでは、Solr は複製によって簡単にクエリー量の増加に対応することはできましたが、1 台のマシンにとっては過大な量の索引に対応するためのスケーリングとなると、(アプリケーションが大部分の作業を行わない限り) 簡単には行きませんでした。例えば Solr では、それぞれに固有の索引が含まれる複数のサーバーをセットアップし、後はアプリケーションに検索を管理させることは可能でしたが、それには相当な量のカスタム・コードが必要になります。そこで 1.3 リリースの Solr で追加されたのが、Distributed Search 機能です。このアプリケーションでは文書を複数のマシン間で分割します。分割された部分は、Solr (および他) では一般的に shard と呼ばれます。shard にはそれぞれに独自の、自己完結した索引が含まれるため、Solr は shard 全体で索引に対するクエリーを調整できるというわけです。残念ながら現時点では、個々の shard に対して索引を付けるために文書を送信するプロセスは、やはりアプリケーションが実行しなければなりませんが、今後の Solr リリースにはこのプロセスも追加される見込みです。当面は、単純なハッシング関数を使用することで、文書の送信先とする shard をその一意の ID を基にして判断することができます。そこで、ここではこの機能で検索を行う側に焦点を絞って説明します。

Solr マシンのサイズ調整

当然のことながら、マシンが保持できる索引のサイズは、マシンの構成 (RAM、CPU、ディスクなど)、クエリーと索引付けの量、文書のサイズ、そして検索パターンによって決まります。ただし通常は、単一のマシンが保持できる文書の数は、数百万から最大約 1 億の間です。

Distributed Search のユーザーは当然、最初にある程度の時間をかけてアーキテクチャーについて検討しなければなりません。少数の shard だけが必要で、複製を考慮に入れないのであれば、マシンごとに索引付けと検索を実行できる shard を 1 つ置けば、比較的簡単に事が運びます。一方、大規模な索引と大量のクエリーを使用する場合には、各 shard の複製が必要になることはほぼ確実です。このようなシステムをセットアップするのによく使われる方法の 1 つは、ロード・バランサーの背後に shard とその複製を配置することです。図 2 に、このアーキテクチャーを示します。

図 2. 分散および複製による Solr アーキテクチャー



Distributed Search に関する注意事項

Solr の Distributed Search 手法には、いくつか欠陥があります。まず、マスター・ノードはフォールト・トレラントではないため、マスター・ノードがダウンすると、システムは新しい文書に索引を付けることも、複製を作成することもできなくなります。しかし、これによって検索が不可能になることはなく、また小規模な分散設定の場合は、shard を手動、あるいはスクリプトと外部モニター・ツールを使って管理できるので大抵は問題になりません。つまり、現行のアーキテクチャーを使って次の Google を作成するのではなく、大量の索引に対応できるようにすべきだということです。もう 1 つの欠陥は、1.3 リリースのすべての `SearchComponent` が分散に対応するわけではないという点です。検索、ファセット分類、デバッグ、強調表示を行うコンポーネントは分散対応ですが、それよりも使用されることが少ないコンポーネントに関しては、現在対応作業が進んでいるところです。Wiki の Distributed Search に関するリンク (「参考文献」を参照) には、この他にもあるそれほど重要ではない、いくつかの注意事項 (驚くことではありません) と Distributed Search についての詳細が記載されています。

図 2 で注目する点は、送られてきた検索リクエスト (Incoming search requests) は、複製されたどの shard に対してでも送ることができるということです。なぜなら、shard はすべて、完全に機能する Solr インスタンスだからです。受信ノードはリクエストを他の shard に送出することもできます。これらのリクエストは一般的な Solr リクエストでしかありません。リクエストを Solr サーバーに送信し、それから `shard` にリクエストを分散させるためには、以下のようにリクエストに shards パラメーターを追加します。

この例では、2 つの Solr サーバーがローカル・ホストで稼働していることを前提としています (つまり、実際に分散されているわけではありません。ここでの説明には有効ですが、読者のセットアップではこの例は機能しないと思います)。2 つのサーバーのうち、メイン・サーバーはポート 8983 で稼働し、もう 1 つはポート 7574 で稼働しています。送られてきたリクエストはポート

8983 のインスタンスに渡され、このインスタンスが shard サーバーにリクエストを送信します。おそらく、アプリケーションではすべての `shard` サーバーの名前をクエリーのたびに渡さなくても済むように、shards パラメーター値を solrconfig.xml にある `SolrRequestHandler` のデフォルト構成の一部として設定することになるはずです。

今後の期待

Solr 1.3 には多くの変更が盛り込まれています。この記事では、スペルチェック、データのインポート、編集によるプレイスメント、Distributed Search など、Solr の多彩な新機能を紹介するとともに、(Solr のベースになっている、高速化された新しいバージョンの Lucene をはじめとする) 強化された Solr の機能についても説明しました。Solr には大幅な変更が加えられた一方、変わっていない点もたくさんあります。つまり、Solr は今でも変わらず支持されている安定した将来性のある検索サーバーで、エンタープライズにすぐにデプロイできるということです。Solr の開発者たちは将来を見据え、すでに文書のクラスタリング、分析オプションの充実、Windows に適した複製、そして重複文書の検出機能の追加作業に取り掛かっています。

ダウンロード

内容	ファイル名	サイズ
Example new feature	j-solr-update.zip	437KB

著者について

Grant Ingersoll

Grant Ingersoll は、Lucid Imagination を創立し、同社の技術スタッフを務めています。専門とするプログラミング分野は、情報検索、マシン・ラーニング、テキスト・カテゴリー化、抽出です。彼は Apache Lucene プロジェクトと Apache Solr プロジェクトのコミッター兼議長で、Apache Mahout マシン・ラーニング・プロジェクトの共同創始者でもあります。