

Apache Lucene および Solr 4 による次世代の検索とアナリティクス

検索エンジン・テクノロジーを利用して、高速かつ効率的でスケーラブルなデータ駆動型アプリケーションを構築する

Grant Ingersoll

CTO
LucidWorks

2013年 12月 05日

Apache Lucene と Solr は、組織が簡単かつ劇的にデータ・アクセス能力を高めることを可能にする、極めて能力の高いオープンソースの検索テクノロジーです。Lucene および Solr のバージョン 4.x では、データ駆動型アプリケーションに、今まで以上に簡単にスケーラブルな検索機能を追加できるようになっています。この記事では、Lucene と Solr のコミッターである Grant Ingersoll が、関連度、分散検索、ファセット処理に関係する、最新の Lucene と Solr の機能を紹介します。高速かつ効率的でスケーラブルな次世代のデータ駆動型アプリケーションを構築するために、これらの機能を利用する方法を学んでください。

私が初めて Lucene と Solr に関する developerWorks の記事を書いたのは、6 年前のことです（「[参考文献](#)」を参照）。この 6 年の間に、Lucene と Solr は信頼できるテクノロジーとしてそれぞれの地位（Lucene は Java API の基盤としての地位、Solr は検索サービスとしての地位）を確立しました。その証拠に、Apple iTunes、Netflix、Wikipedia をはじめとする多数の検索ベースのアプリケーションがこの 2 つのテクノロジーによって駆動されています。また、Lucene と Solr は IBM が開発した質問応答システム Watson の実現にも一役買っています。

長年にわたり、Lucene と Solr は、主にテキスト・ベースの検索に焦点を絞って使用されてきました。その一方で、ビッグ・データという新たな興味深いトレンドが生まれ、それとともに分散コンピューティングおよび大規模アナリティクスに新たな（刷新された）焦点が当てられるようになってきました。さらに、ビッグ・データには多くの場合、リアルタイムでの大規模な情報アクセスも必要となります。こうした推移のなか、Lucene と Solr のコミュニティは岐路に立たされていることを自覚しました。Twitter 界のすべてに対してインデクシングする（「[参考文献](#)」を参照）などといったビッグ・データ・アプリケーションからの高い要求が出てくるなかで、Lucene のコアとなっている基盤は古びてきたからです。しかも、Solr には分散インデクシングのネイティブ・サポートが欠けていたことから、Solr を使用する IT 組織が検索インフラストラクチャーをコスト効果の高い方法でスケーリングするのがますます難しくなってきました。

コミュニティは、Lucene および Solr の基盤 (および、場合によってはパブリック API) の全面的な見直しに着手しました。私たちの焦点は、スケーラビリティの容易な実現、ほぼリアルタイムのインデクシングと検索の実現、さまざまな NoSQL 機能の実現 — いずれもコア・エンジンの能力を活かして行うこと — に移されました。この全面的な見直しが、Apache Lucene および Solr の 4.x のリリースに結集されています。これらのバージョンが目指しているのは、まさに、次世代の大規模データ駆動型アクセスとアナリティクスの問題の解決です。

この記事では、Apache Lucene および Solr のバージョン 4.x の見どころを順に紹介し、その一部でサンプル・コードを記載しますが、まずは「検索エンジンを利用して検索の枠を超える」という概念を実証する、実際のアプリケーションを体験することから始めます。この記事を最大限活用するには、Lucene と Solr の基本 (特に Solr リクエスト) を十分に理解する必要があります。この前提に当てはまらない場合は、Lucene および Solr の基本を学べるサイトや書籍へのリンクを「[参考文献](#)」から参照してください。

クイック・スタート: 検索およびアナリティクスの実際

検索エンジンは、テキストを検索するためだけのものだと思っていたら、それは違います！基本的には検索エンジンとは、迅速かつ効率的にデータをフィルタリングして、類似度に関する何らかの概念 (Lucene と Solr では柔軟に定義されている概念) に従ってデータにランクを付けするものに他なりません。また、検索エンジンは、最近のデータ・アプリケーションに特徴的な、疎なデータや曖昧なデータを効果的に処理します。Lucene と Solr には、数値の高速処理や (この後、話題にする) 複雑な地理空間の質問への応答をはじめ、さまざまな機能が備わっています。これらの機能は、検索アプリケーションと従来のデータベース・アプリケーション (さらには NoSQL アプリケーション) との間の区別を曖昧にします。

例えば、Lucene および Solr には現在、以下の特長があります。

- 結合オプションとグループ化オプションを数種類サポートします。
- オプションで列指向ストレージを使用できます。
- テキストや、列挙データ、数値データを扱ういくつかの方法が用意されています。
- ユーザーが複合データ型や、ストレージ機能、ランキング機能、アナリティクス機能を独自に定義できます。

検索エンジンは、あらゆるデータ問題を解決する特効薬ではありませんが、これまで Lucene と Solr の主な用途がテキスト検索であったからと言って、現在あるいは将来のデータに対する要求に対処するために Lucene や Solr を使用できないわけではありません。検索エンジンの用途については、おなじみの (検索) ボックスで使用することにまったく捉われずに検討することをお勧めします。

検索エンジンが検索のみにとどまらず、どのような用途にまで適用できるかを実際に示すために、このセクションの残りでは、航空関連のデータを Solr に取り込むアプリケーションを紹介します。このアプリケーションはデータ (その大部分はテキストではありません) を照会し、D3 JavaScript ライブラリー (「[参考文献](#)」を参照) を使用してそのデータを処理して表示します。照会するデータ・セットは、米国運輸省運輸統計局の RITA (調査・革新技術庁) のデータと OpenFlights のデータです。これらのデータには、特定の期間における全フライトの出発空港、到着空港、時間遅延、遅延原因、航空会社情報などの詳細が含まれます。このデータを照会するア

アプリケーションを使用して、特定の空港間での遅延、特定の空港での輸送量の増加など、さまざまなことを分析できます。

まずはアプリケーションを稼働させる方法を説明してから、アプリケーションのインターフェースの一部について見ていきます。以降の説明を通して、アプリケーションはさまざまな方法で Solr に問い合わせを行うことによってデータを操作することに留意してください。

セットアップ

始めに、以下の前提条件を満たす環境を用意する必要があります。

- Lucene および Solr
- Java 6 またはそれ以降のバージョンの Java
- 最近の Web ブラウザー (私は Google Chrome と Firefox でテストしました)
- 4GB のディスク領域 (すべてのフライト・データを使用するのでなければ、必要な領域はこれよりも小さくなります)
- *nix 上でのターミナルからの bash (または同様の) シェルによるアクセス。Windows の場合は、Cygwin が必要です。私は OS X で bash シェルを使用した環境でのみテストを行いました。
- wget。サンプル・コード・パッケージに含まれるダウンロード・スクリプトを使用してデータをダウンロードする場合に必要です。フライト・データのダウンロードは、スクリプトを使用せずに手作業で行うこともできます。
- Apache Ant 1.8+。サンプル Java コードを実行する場合、コンパイルとパッケージ化のために必要です。

Lucene、Solr、`wget`、Ant の各ダウンロード・サイトへのリンクについては、「[参考文献](#)」を参照してください。

前提条件を満たす環境が揃ったら、以下の手順に従ってアプリケーションを稼働させます。

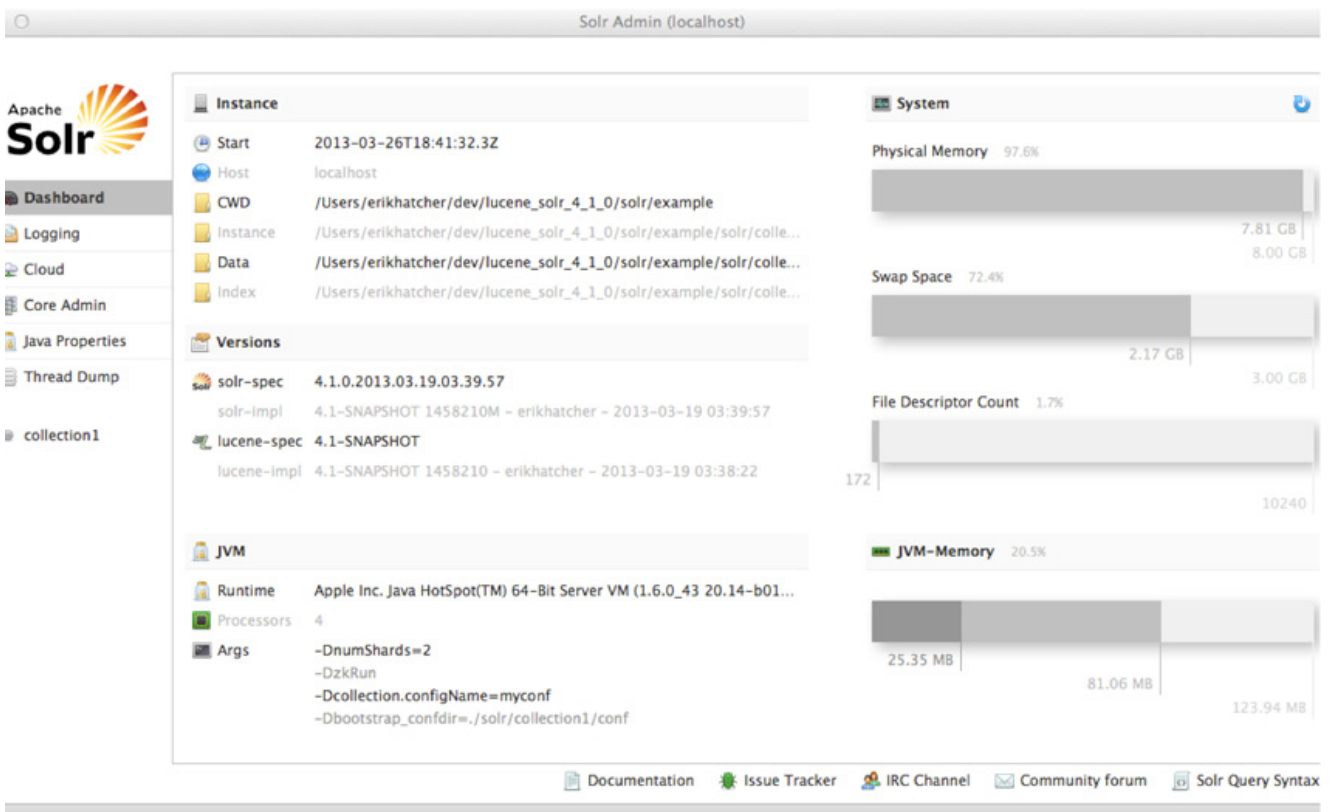
1. この記事のサンプル・コードが含まれる zip ファイルを[ダウンロード](#)して、任意のディレクトリに解凍します。ここでは、このディレクトリを `$SOLR_AIR` と呼びます。
2. コマンドラインで、カレント・ディレクトリを `$SOLR_AIR` に変更します。

```
cd $SOLR_AIR
```
3. Solr を起動します。

```
./bin/start-solr.sh
```
4. 以下のスクリプトを実行します。このスクリプトにより、データのモデル化に必要なフィールドが作成されます。

```
./bin/setup.sh
```
5. ブラウザーで `http://localhost:8983/solr/#/` にアクセスして、新しくなった Solr Admin UI を表示します。図 1 に例を示します。

図 1. Solr の UI



6. ターミナルで `bin/download-data.sh` スクリプトの内容を表示し、RITA と OpenFlights から何をダウンロードするのか、その詳細を調べます。このスクリプトを実行するか、手作業による方法のいずれかで、データ・セットをダウンロードします。

```
./bin/download-data.sh
```

帯域幅によっては、ダウンロードにかなりの時間がかかる場合があります。

7. ダウンロードが完了したら、データの一部またはすべてに対してインデクシングを行います。

すべてのデータに対してインデクシングする場合は、以下のスクリプトを実行します

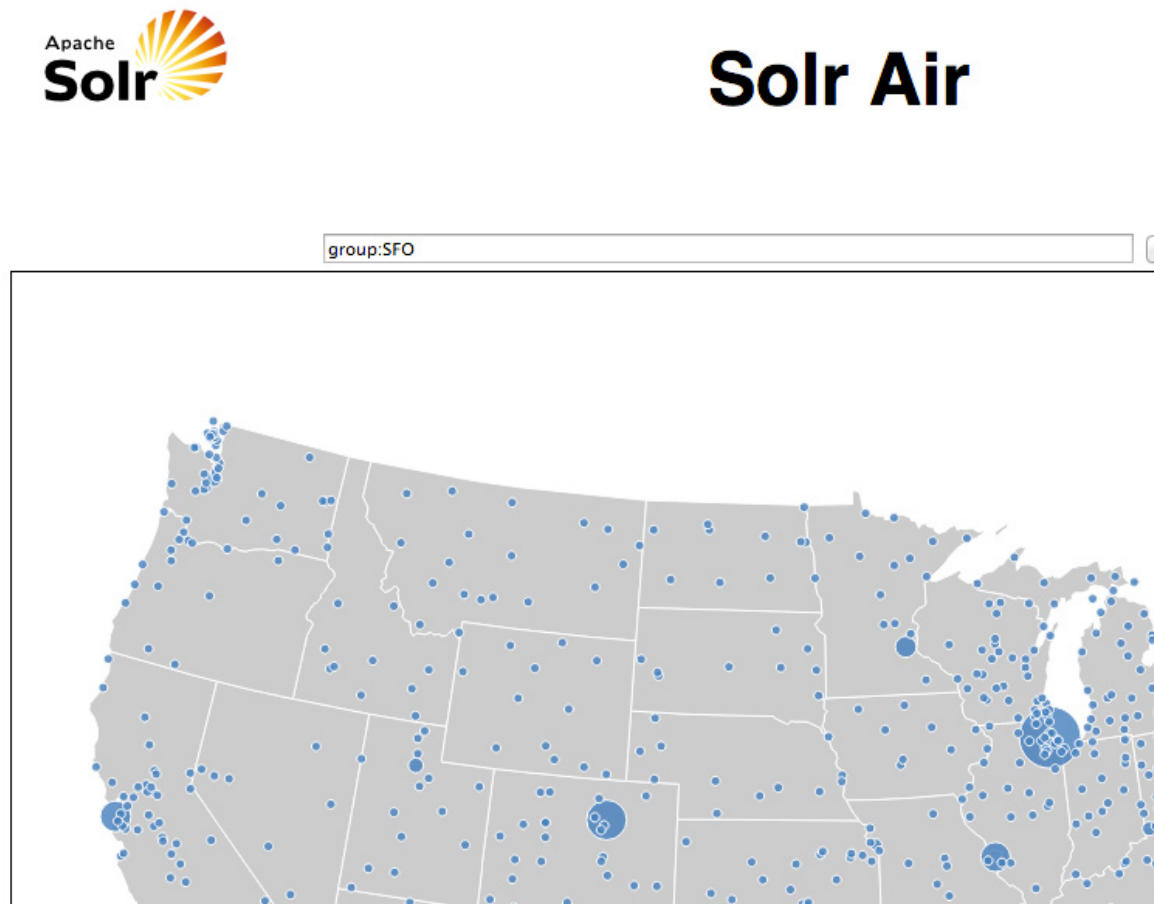
```
bin/index.sh
```

特定の 1 年間のデータに対してインデクシングするには、1987 から 2008 までのいずれかの値を年として指定します。以下はその一例です。

```
bin/index.sh 1987
```

8. インデクシングが完了した後 (マシンによっては、かなりの時間がかかる場合があります)、ブラウザーで `http://localhost:8983/solr/collection1/travel` にアクセスします。図 2 に示すような UI が表示されるはずです。

図 2. Solr Air の UI



データの探索

Solr Air アプリケーションが稼働状態になったら、データと UI に目を通して、どのような質問をすることができるのかを感じ取ってください。ブラウザーには、中心となるインターフェース・ポイントとして地図と検索ボックスの 2 つが表示されます。この地図は、D3 の素晴らしい Airport (空港) サンプル ([参考文献](#)) を参照) を基に作成したものです。コードを修正して、D3 サンプルに付属の CSV ファイルからではなく、直接 Solr からすべての空港情報を読み取るように拡張しました。また、各空港に関する統計計算も行っており、特定の空港にマウスを重ねると統計値が表示されるようにしています。

これから検索ボックスを使用して、高度な検索とアナリティクスの両方を行うアプリケーションを構築する際に役立つ、いくつかの重要な機能を披露します。コードの内容を目で追って理解するには、`solr/collection1/conf/velocity/map.vm` ファイルを参照してください。

ここでは、以下の内容に主な焦点を合わせます。

- ピボット・ファセット
- 統計機能

- グループ化
- Lucene と Solr の拡張地理空間サポート

これらのどれもが、例えば特定の空港でのフライト到着の平均遅延時間や、2つの空港を結ぶフライトの最も一般的な遅延時間 (航空会社ごとの遅延時間、あるいは特定の出発空港とその近辺にあるすべての空港との間での遅延時間) などを調べるのに役立ちます。このアプリケーションでは、Solr で長年にわたって使われているファセット処理機能が組み合わされた、Solr の統計機能を使用して、まず始めに空港を表す「ドット」の地図を描画し、さらに総フライト数、平均遅延時間、最小遅延時間、最大遅延時間などの基本情報を生成します (この機能だけでも、不良データを見つけたり、そこまではできないにしても極端な外れ値を見つけたりする上での素晴らしい方法になります)。こうした内容を実際に行うために (そして、Solr に D3 を統合するのがいかに簡単であるかを示すために)、次の処理を行う軽量の JavaScript コードを実装しました。

1. クエリーを構文解析します (本番品質のアプリケーションの場合は、おそらく、クエリーの構文解析の大半をサーバー・サイドで実行するはずです。あるいは、Solr クエリー・パーサー・プラグインとして、これらの処理を行う可能性もあります)。
2. 各種の Solr リクエストを作成します。
3. 結果を表示します。

リクエストには、以下のタイプがあります。

- 3 文字の空港コード (RDU、SFO など) を指定して検索する。
- 経路 (SFO TO ATL、RDU TO ATL など) を指定して検索する (経由地はサポートしていません)。
- 検索ボックスが空の状態のときに検索ボタンをクリックすると、すべてのフライトに関する各種の統計を表示する。
- near 演算子を使用して (near:SFO、near:SFO TO ATL など)、近隣の空港を検索する。
- 生じ得る遅延時間を、さまざまな飛行距離 (500 マイル未満、500 から 1000 マイル、1000 から 2000 マイル、2000 マイル以上) で調べる (likely:SFO など)。
- Solr の `/travel` リクエスト・ハンドラーにフィードするための任意の Solr クエリー (&q=AirportCity:Francisco など)。

上記の最初の 3 つのリクエスト・タイプは、いずれも同じタイプのリクエストのバリエーションです。これらのバリエーションは、Solr のピボット・ファセット処理機能を利用して、例えば各航空会社のフライト番号ごとに各経路 (SFO TO ATL など) の最も一般的な到着遅延時間を明らかにします。near オプションは、新しい Lucene と Solr の空間機能を利用して、大幅に強化された空間計算 (例えば、複雑な多角形の交差計算など) を実行します。likely オプションでは、Solr のグループ化機能を利用して、ある出発空港から特定の距離の範囲にあって、30 分を超える到着遅延があった空港を示します。これらのリクエスト・タイプは、いずれも少量の D3 JavaScript コードで処理した表示情報によって地図を補完します。一方、上記の最後にあるリクエスト・タイプは、関連する JSON をそのまま返すように作成してあるため、皆さんが独自にデータを調査することができます。このリクエスト・タイプを皆さんが所有するアプリケーションで使用するとしたら、当然、そのアプリケーションに固有の方法でレスポンスを使用する必要があります。

早速、独自にクエリーを試してみてください。例えば、SFO TO ATL を検索すると、図 3 のような結果が表示されます。

図 3. SFO TO ATL の検索結果の画面

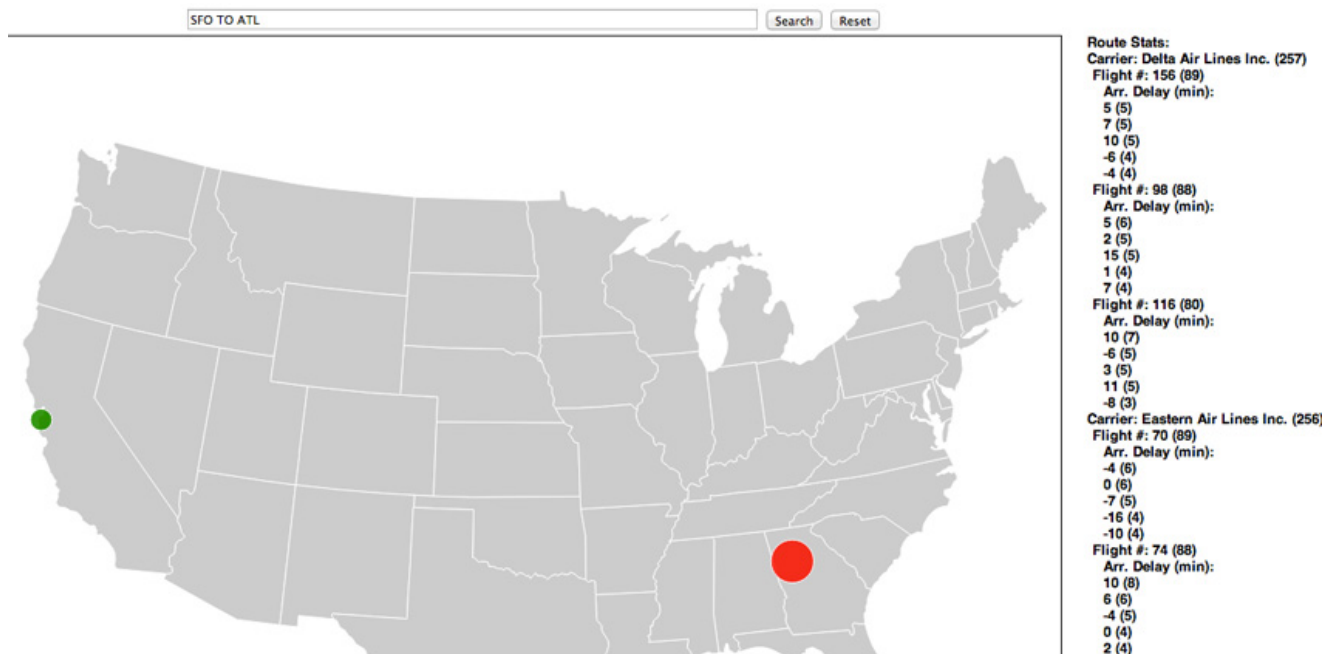


図 3 では、左側の地図で 2 つの空港が強調表示されています。右側の「Route Stats (経路統計)」リストには、各航空会社のフライトごとに、最も一般的な到着遅延時間が示されます (この例では、1987 年のデータだけをロードしました)。このリストから、例えばデルタ航空の 156 便は、アトランタに 5 分遅延して到着したことが 5 回あり、定刻より 6 分早く到着したことが 4 回あったことがわかります。

背後で実行された Solr リクエストは、ブラウザーのコンソールでも (例えば Mac 上の Chrome では、「View (表示)」->「Developer (開発者)」->「Javascript Console (Javascript コンソール)」の順に選択)、Solr ログでも確認することができます。私が使用した SFO-TO-ATL リクエストは以下のとおりです (ここでは見やすくするために、3 行に分割しています)。

```
/solr/collection1/travel?&wt=json&facet=true&facet.limit=5&fq=Origin:SFO
AND Dest:ATL&q=*&facet.pivot=UniqueCarrier,FlightNum,ArrDelay&
f.UniqueCarrier.facet.limit=10&f.FlightNum.facet.limit=10
```

このリクエストで重要な役目を果たすのは、`facet.pivot` パラメーターです。`facet.pivot` は、航空会社名 (UniqueCarrier) から、フライト番号 (FlightNum)、そして到着遅延 (ArrDelay) まで、収集する情報を指定し、それによって収集された情報を基にネストされた構造を生成します。それが、図 3 の「Route Stats (経路統計)」に表示されている構造です。

`near` クエリー (例えば、`near:JFK`) を試してみると、図 4 のような結果になります。

図 4. JFK 近隣の空港を示す画面の例



near クエリーを実行するための Solr リクエストは、Solr の新しい空間機能を利用します。この新しい機能については後ほど詳しく説明しますが、リクエスト自体を見れば、おそらくこの新しい機能が持つ能力の片鱗が垣間見られるはずです (リクエスト全体の内容は、ここでは記事のフォーマット上の理由で省略しています)。

```
...  
&fq=source:Airports&q=AirportLocationJTS:"IsWithin(Circle(40.639751,-73.778925 d=3))"  
...
```

ご想像のとおり、このリクエストが検索するのは、緯度 40.639751、経度 -73.778925 を中心とした半径 3 度 (1 度は約 111 キロメートル) の円の中にあるすべての空港です。

ここまで読んで、Lucene および Solr によるアプリケーションは興味深い方法でデータ (数値、テキスト、その他) を細かく分析できるという強い印象を持ったことでしょう。しかも、Lucene と Solr はどちらもオープンソースであり、商用に適したライセンスで使えるため、独自のカスタマイズを自由に加えることができます。さらに便利なことに、Lucene および Solr のバージョン 4.x では、独自のアイデアや機能を挿入してもコード全体を見直す必要がない箇所が増えています。これから紹介する Lucene 4 (この記事が執筆している時点でのバージョン 4.4) の見どころと Solr 4 の見どころを読むにあたっては、この点を念頭に置いてください。

Lucene 4: 次世代の検索およびアナリティクスの基盤

Lucene 4 での大変貌

Lucene 4 では、パフォーマンスと柔軟性を向上させるために、Lucene の基盤部分のコードをほぼ全面的に書き直しています。それと同時にこのリリースでは、Lucene の新しいランダム化ユニット・テスト・フレームワークとパフォーマンス関連の厳格なコミュニティ標準が作成されたことによって、コミュニティによるソフトウェア開発の方法が大きく様変わりしています。例えば、ランダム化テスト・フレームワーク (誰もが使用できるようにパッケージ化された成果物として提供されています) により、プロジェクトでは可変の部分 (JVM、ロケール、入力内容とクエリー、ストレージ・フォーマット、スコア計算式、その他多数) の相互作用を簡単にテストできるようになっています (Lucene を使用しないとしても、このテスト・フレームワークは皆さん独自のプロジェクトで役立つはずです)。

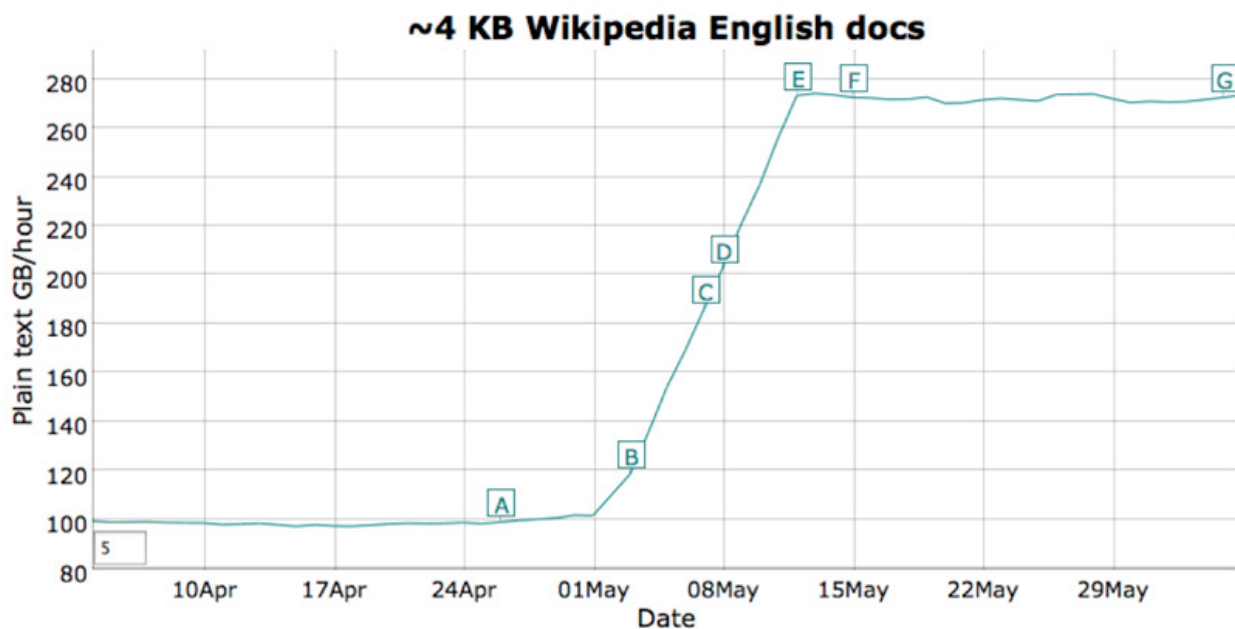
Lucene では、速度とメモリー、柔軟性、データ構造、ファセット処理といったカテゴリーでも重要な追加および変更が行われています (Lucene の変更内容に関するあらゆる詳細を調べるには、すべての Lucene ディストリビューションに付属している CHANGES.txt ファイルを読んでください)。

速度とメモリー

これまでのバージョンでも、Lucene は十分に高速だと一般に認められています (特に、同等の汎用検索ライブラリーと比較した場合)、Lucene 4 でのさまざまな強化によって、これまでのバージョンに比べて著しく高速になっている操作が数多くあります。

図 5 に示すグラフは、Lucene のインデクシングのパフォーマンスを 1 時間あたりのギガバイト数で測定したものです (Lucene コミッターである Mike McCandless 氏が約 2 カ月にわたって夜間に行った Lucene ベンチマーク・テストのグラフです。「[参考文献](#)」を参照)。図 5 には、5 月前半 (年不明) に劇的にパフォーマンスが改善されたことが示されています。

図 5. Lucene のインデクシングのパフォーマンス



前世代の Lucene ではありません

Lucene 4 では、エンジンのため — 延いては新しく興味深いさまざまなことをユーザーが行えるようにするため — に API の変更と機能強化が大々的になされています。しかし、前のバージョンの Lucene からアップグレードするにはかなりの作業が必要になる可能性があります (IndexWriter や IndexReader などのクラスは、前のバージョンから概ね変わっていませんが、例えばターム・ベクトルにアクセスする方法は大幅に変更されています)。これは、下位レベルの API (つまり「エキスパート」API) を 1 つでも使用しているとしたら尚のことです。状況に応じてアップグレードを計画してください。

図 5 に示されているパフォーマンスの向上は、Lucene がインデックス構造を構築する方法や、インデックス構造の構築時に並行処理を行う方法に関して一連の変更 (さらには JVM の変更やソリッド・ステート・ドライブの使用など、その他いくつかの変更) が行われた結果として実現されたものです。これらの変更では、Lucene がインデックスをディスクに書き込む間の同期処理をなくすことに重点が置かれました。詳細については (この記事では説明しませんので)、「[参考文献](#)」から Mike McCandless 氏のブログ投稿へのリンクを参照してください。

Lucene 4 では、インデクシングのパフォーマンスが全体的に改善されただけでなく、ほぼリアルタイムでの (Near Real Time: NRT) インデクシング操作も実行することができます。NRT 操作は、

検索エンジンが変更をインデックスに反映するのにかかる時間を大幅に短縮することが可能です。NRT 操作を使用するには、アプリケーションで Lucene の `IndexWriter` と `IndexReader` との間の調整を取る必要があります。リスト 1 (ダウンロード・パッケージに含まれる `src/main/java/IndexingExamples.java` ファイルからの抜粋) には、この 2 つの相互操作が示されています。

リスト 1. Lucene での NRT 検索の例

```
...
doc = new HashSet<IndexableField>();
index(writer, doc);
//Get a searcher
IndexSearcher searcher = new IndexSearcher(DirectoryReader.open(directory));
printResults(searcher);
//Now, index one more doc
doc.add(new StringField("id", "id_" + 100, Field.Store.YES));
doc.add(new TextField("body", "This is document 100.", Field.Store.YES));
writer.addDocument(doc);
//The results are still 100
printResults(searcher);
//Don't commit; just open a new searcher directly from the writer
searcher = new IndexSearcher(DirectoryReader.open(writer, false));
//The results now reflect the new document that was added
printResults(searcher);
...
```

リスト 1 では、最初に一連のドキュメントに対してインデクシングを行って `Directory` にコミットしてから、`Directory` を検索します (ここまでは、Lucene の従来の手法です)。これに続き、もう 1 つのドキュメントに対してインデクシングを行うときに NRT が関わってきます。Lucene はドキュメントを完全にコミットするのではなく、`IndexWriter` から新しい `IndexSearcher` を作成して検索を実行します。このサンプル・コードを実行するには、`$SOLR_AIR` ディレクトリーから以下のコマンドを順に実行してください。

1. `ant compile`
2. `cd build/classes`
3. `java -cp ../../lib/*:. IndexingExamples`

注: この記事のサンプル・コードのいくつかは、`IndexingExamples.java` に集められています。この後のリスト 2 とリスト 4 のサンプル・コードも、上記と同じコマンド・シーケンスを使用して実行することができます。

画面に出力される内容は以下のとおりです。

```
...
Num docs: 100
Num docs: 100
Num docs: 101
...
```

Lucene 4 では、より高度なデータ構造 (「[有限状態オートマトンおよびその他の特長](#)」で詳細を説明します) を活かして、メモリーにも改善が加えられています。これらの改善は、Lucene のメモリー・フットプリントを減らすだけでなく、ワイルドカードや正規表現を使用したクエリーの速度を飛躍的に向上させます。さらに、大量に割り当てたバイト配列を管理しやすくするために、コード・ベースでは Java `String` オブジェクトを扱わないようになりました (今では Lucene 内部

の至るところで `BytesRef` クラスを目にします)。その結果、`String` のオーバーヘッドがなくなり、Java ヒープ上のオブジェクト数をより適切に制御できるようになったため、STW (Stop-The-World) 型のガーベッジ・コレクションが発生する可能性が低くなっています。

柔軟性の向上は、その一部がパフォーマンスと保管の改善にもつながっています。それは、アプリケーションで使用しているデータ型により適したデータ構造を選択できるようになったためです。例えば次のセクションでわかるように、Lucene では、(密度が高く、圧縮率が低い) 一意のキーにはある特定のインデクシング/保管方法を選択する一方で、テキストについては、テキストのまばらさによく適したまったく別の方法でインデクシング/保管することができます。

柔軟性

Lucene の優れた性質やパフォーマンスをとことんまで引き出そうとする開発者 (および研究者) にとって、Lucene 4.x で柔軟性が高くなったことによって開けた可能性は、宝の山のようなものです。柔軟性を高めるために、Lucene では明確に定義された新しい 2 つのプラグイン・ポイントを利用できるようになっています。どちらのプラグイン・ポイントも、すでに Lucene の開発方法と使用方法の両方に大きな影響を与えています。

セグメントについて

Lucene のセグメントとは、インデックス全体の中のサブセットのことです。多くの点で、セグメントは自己完結型のミニインデックスとなっています。Lucene は、検索時におけるインデックスの利用と書き込み速度とのバランスを取るために、セグメントを使用してインデックスを作成します。セグメントはインデクシング時に一度だけ書き込まれるファイルであり、書き込み時にコミットを実行するたびに新しいセグメントが作成されます。バックグラウンドでは、Lucene が読み取りパフォーマンスの改善とシステム・オーバーヘッドの削減を目的に、デフォルトで定期的に複数の小さなセグメントを大きなセグメントにマージします。このプロセスはユーザーが完全に制御することができます。

新しいプラグイン・ポイントのうちの 1 つは、Lucene の**セグメント**のエンコードとデコードをユーザーが詳細に制御できるように設計されています。エンコード/デコード機能を定義するのは、`Codec` クラスです。`Codec` では、ポスティング・リスト (つまり、転置インデックス) のフォーマット、Lucene ストレージ、ブースト係数 (別名、ノルム)、その他多数のものを制御することができます。

アプリケーションによっては、独自の `Codec` を実装した方がよい場合もありますが、それよりも、インデックスに含まれるドキュメント・フィールドのサブセットで使用されている `Codec` を変更した方がよい場合の方が遥かに多いです。この点については、アプリケーションに入力するデータの種類について考えると理解しやすくなります。例えば、識別用フィールド (例えば、主キー) は通常、一意です。ある 1 つのドキュメントで主キーが出現するのは一度だけなので、主キーについては、記事のテキスト本文をエンコードする方法とは異なる方法でエンコードした方がよい場合があります。そのような場合には、実際に `Codec` を変更するのではなく、`Codec` が委譲する下位クラスのうちの 1 つを変更します。

デモンストレーションするために、私のお気に入りの `Codec` である `SimpleTextCodec` を使用したサンプル・コードを記載します。`SimpleTextCodec` はその名前のとおり、インデックスを単純なテキストでエンコードする `Codec` です (`SimpleTextCodec` が作成され、この `Codec` が Lucene の包括的なテスト・フレームワークをパスしているという事実が、Lucene の柔軟性が向上している証拠です)。 `SimpleTextCodec` は、本番環境で使用するには大きすぎる上に処理に時間がかかりすぎ

るとは言え、Lucene インデックスが内部でどのようなになっているのかを確認するにはうってつけの手段です。これが、私が好んで SimpleTextCodec を使用している理由です。リスト 2 のサンプル・コードでは、Codec インスタンスを SimpleTextCodec に変更しています。

リスト 2. Lucene において Codec インスタンスを変更する例

```
...
conf.setCodec(new SimpleTextCodec());
File simpleText = new File("simpletext");
directory = new SimpleFSDirectory(simpleText);
//Let's write to disk so that we can see what it looks like
writer = new IndexWriter(directory, conf);
index(writer, doc); //index the same docs as before
...
```

リスト 2 のコードを実行すると、ローカルに build/classes/simpletext ディレクトリーが作成されます。Codec の動作を確認するには、カレント・ディレクトリーを build/classes/simpletext に変更し、.cfs ファイルをテキスト・エディターで開いてください。.cfs ファイルは、リスト 3 に記載するスニペットのとおり、まさに従来のプレーン・テキストであることがわかります。

リスト 3. _0.cfs プレーン・テキスト・インデックスファイルの抜粋

```
...
term id_97
  doc 97
term id_98
  doc 98
term id_99
  doc 99
END
doc 0
  numfields 4
  field 0
    name id
    type string
    value id_100
  field 1
    name body
    type string
    value This is document 100.
...
```

極めて大量のインデックスとクエリーを扱うようになるまでは、あるいはベア・メタルをいじるのが大好きな研究者や検索エンジンの達人にとっては、Codec を変更しても功を奏さないことがほとんどです。そのような場合は、Codec を変更する前に、実際のデータを使用して各種の利用可能な Codec を徹底的にテストしてください。Solr のユーザーは、単純な構成項目に変更を加えることで、Codec の機能を設定および変更することができます。詳細については、「[Apache Solr Reference Guide](#)」を参照してください（「[参考文献](#)」を参照）。

もう 1 つの新しくて重要なプラグイン・ポイントは、Lucene のスコアリング・モデルを完全にプラグブルにします。したがって、一部のユーザーがあまりにも単純だと不平をもらしている Lucene のデフォルト・スコアリング・モデルを使用しなくてもよくなります。必要に応じて、代替りとなるスコアリング・モデル (BM25 (Best Match 25) や DFR (Divergence from Randomness) など) を使用することも（「[参考文献](#)」を参照）、独自のスコアリング・モデルを作成することもできます。独自のスコアリング・モデルを作成することになる状況としては、例え

ば自分が作成した複数のドキュメントで分子や遺伝子について取り上げている場合に、分子や遺伝子の出現回数を素早くランク付けしたいけれども、それらのタームの出現回数やそれらのタームが含まれているドキュメントの数に関するスコアリング・モデルが適用できないケースなどです。あるいは、研究論文で読んだ新しいスコアリング・モデルが自分のコンテンツでどれだけ有効に機能するかを確かめるために、そのモデルを試してみたいという場合にも、独自のスコアリング・モデルを使用することになります。理由は何であれ、スコアリング・モデルを変更するには、インデクシングの時点で `IndexWriterConfig.setSimilarity(Similarity)` メソッドを使用し、検索する時点で `IndexSearcher.setSimilarity(Similarity)` メソッドを使用して、モデルを変更する必要があります。リスト 4 では `Similarity` を変更する具体例として、最初にデフォルトの `Similarity` を使用したクエリーを実行し、次に Lucene の `BM25Similarity` を使用したクエリーを実行してインデクシングし直しています。

リスト 4. Lucene における `Similarity` の変更

```
conf = new IndexWriterConfig(Version.LUCENE_44, analyzer);
directory = new RAMDirectory();
writer = new IndexWriter(directory, conf);
index(writer, DOC_BODIES);
writer.close();
searcher = new IndexSearcher(DirectoryReader.open(directory));
System.out.println("Lucene default scoring:");
TermQuery query = new TermQuery(new Term("body", "snow"));
printResults(searcher, query, 10);

BM25Similarity bm25Similarity = new BM25Similarity();
conf.setSimilarity(bm25Similarity);
Directory bm25Directory = new RAMDirectory();
writer = new IndexWriter(bm25Directory, conf);
index(writer, DOC_BODIES);
writer.close();
searcher = new IndexSearcher(DirectoryReader.open(bm25Directory));
searcher.setSimilarity(bm25Similarity);
System.out.println("Lucene BM25 scoring:");
printResults(searcher, query, 10);
```

リスト 4 のコードを実行して、その出力を調べてください。確かにスコアが異なっていることがわかるはずです。BM25 手法での結果の方がユーザーの目的とする結果セットを正確に反映しているかどうかは、結局のところ、コードの作成者とそのコードを使用するユーザーの判断に任せられます。皆さんには、実験を行いやすいようにアプリケーションをセットアップして (A/B テストが有用です)、`Similarity` の結果だけでなく、クエリー作成、`Analyzer`、そしてその他多くの項目をさまざまに変えた場合の結果を比較することをお勧めします。

有限状態オートマトンおよびその他の特長

Lucene のデータ構造とアルゴリズムの完全な見直しは、Lucene 4 でのとりわけ興味深い進歩へとつながりました。その進歩とは、以下の 2 つです。

- `DocValue` (カラム・ストライド・フィールド (column stride field) とも呼ばれます)。
- 有限状態オートマトン (FSA) および有限状態トランスデューサー (FST)。この記事では以降、FSA と FST の両方を併せて FSA と呼びます (厳密に言うと、FST は、アクセスする対象がそのノードであることから出力値ですが、その違いは、この記事では重要ではありません)。

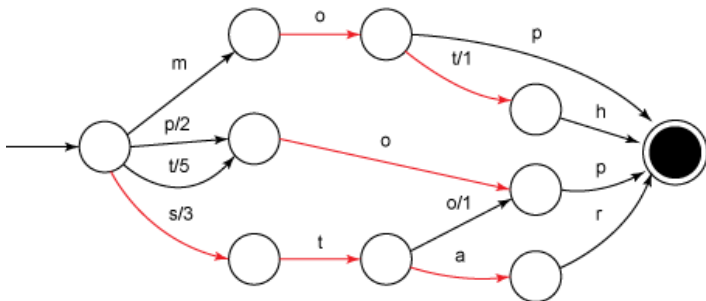
`DocValue` と FSA はいずれも、アプリケーションに影響を及ぼす可能性がある特定のタイプの操作に対して、新しく重要なパフォーマンス上のメリットをもたらします。

DocValue について言うと、多くの場合、アプリケーションは単一のフィールドのすべての値に極めて迅速かつ順番にアクセスする必要があります。あるいは、アプリケーションがインデックスからインメモリ・バージョンを作成するコストを伴わずに、ソートやファセット処理のために値の検索を短時間で行わなければならない場合もあります (非転置として知られるプロセス)。DocValue は、このようなニーズに応えるように設計されています。

ワイルドカードやファジー要素を使用したクエリーを多用するアプリケーションでは、FSA を使用することによって、パフォーマンスの大幅な改善が見られるはずです。現在 Lucene と Solr がクエリーの機能としてサポートしている、自動入力補完機能およびスペルチェック機能には、FSA が使われています。また、Lucene のデフォルト Codec は、内部で FSA を使用してターム辞書 (Lucene が検索中にクエリー・タームを参照するために使用する構造) を保管することによって、ディスクおよびメモリのフットプリントを大幅に削減しています。FSA には言語処理におけるさまざまな用途があるため、他のアプリケーションでも Lucene の FSA 機能が有益であることが判明する可能性があります。

図 6 に、単語 mop、pop、moth、star、stop、top およびそれぞれに関連付けられた重みを使用して <http://examples.mikemccandless.com/fst.py> から生成した FSA を示します。この例から想像できるように、FSA では入力 (例えば moth) からスタートして、この入力を文字 (m-o-t-h) に分割してから、弧を辿っていきます。

図 6. FSA の例



リスト 5 (この記事のサンプル・コードのダウンロードに含まれる FSAExamples.java ファイルからの抜粋) に、Lucene の API を使用して独自の FSA を作成する単純な例を示します。

リスト 5. 単純な Lucene オートマトンの例

```
String[] words = {"hockey", "hawk", "puck", "text", "textual", "anachronism", "anarchy"};
Collection<BytesRef> strings = new ArrayList<BytesRef>();
for (String word : words) {
    strings.add(new BytesRef(word));
}
//build up a simple automaton out of several words
Automaton automaton = BasicAutomata.makeStringUnion(strings);
CharacterRunAutomaton run = new CharacterRunAutomaton(automaton);
System.out.println("Match: " + run.run("hockey"));
System.out.println("Match: " + run.run("ha"));
```

リスト 5 では、さまざまな単語から Automaton を作成して、それを RunAutomaton に取り込みます。その名前が示唆するように、RunAutomaton は指定した入力でオートマトンを実行し、この例の場合は、取り込まれた入力文字列をリスト 5 の最後にある一連の println 文で突き合わせます。

これは平凡な例ですが、これよりも遥かに高度な Lucene API の機能を理解する上での基礎となります。Lucene API の機能 (および DocValue) の詳細については、読者が各自で調べるのに任せします (「[参考文献](#)」の関連するリンクを参照)。

ファセット処理

ファセット処理は基本的に、ユーザーがクエリーに追加するキーワードを推測しなくても簡単に検索結果を絞り込めるようにドキュメントの属性のカウントを生成します。例えば、ユーザーがテレビを購入するためのショッピング・サイトを検索しているとすると、ファセットによって、どのメーカーが何種類の TV モデルを製造しているかがわかります。一方で、検索ベースのビジネス・アナリティクスおよびレポート作成ツールを駆動するためにファセット処理を使用することも多くなっています。より高度なファセット処理機能を使用すれば、ユーザーがファセットを興味深い方法で細かく分析できるようになります。

ファセットは長い間 (バージョン 1.1 以降) Solr の代表的な特徴でしたが、現在は Lucene に独自のファセット処理モジュールが加わり、スタンドアロンの Lucene アプリケーションでこのモジュールを利用できるようになっています。Lucene のファセット処理モジュールは、機能の面では Solr のファセット処理モジュールほどは充実していませんが、いくつかの興味深いトレードオフを提示しています。インデクシング時にファセット処理に関するいくつかの決定を行わなければならないという点で、Lucene のファセット処理モジュールは動的ではありません。その一方、このモジュールは階層型であり、メモリーへの動的なフィールド非転置処理のコストを伴いません。

リスト 6 (サンプル・コードの FacetExamples.java ファイルからの抜粋) に、Lucene の新しいファセット処理機能のいくつかの例を示します。

リスト 6. Lucene のファセット処理の例

```
...
DirectoryTaxonomyWriter taxoWriter =
    new DirectoryTaxonomyWriter(facetDir, IndexWriterConfig.OpenMode.CREATE);
FacetFields facetFields = new FacetFields(taxoWriter);
for (int i = 0; i < DOC_BODIES.length; i++) {
    String docBody = DOC_BODIES[i];
    String category = CATEGORIES[i];
    Document doc = new Document();
    CategoryPath path = new CategoryPath(category, '/');
    //Setup the fields
    facetFields.addFields(doc, Collections.singleton(path)); //just do a single category path
    doc.add(new StringField("id", "id_" + i, Field.Store.YES));
    doc.add(new TextField("body", docBody, Field.Store.YES));
    writer.addDocument(doc);
}
writer.commit();
taxoWriter.commit();
DirectoryReader reader = DirectoryReader.open(dir);
IndexSearcher searcher = new IndexSearcher(reader);
DirectoryTaxonomyReader taxor = new DirectoryTaxonomyReader(taxoWriter);
ArrayList<FacetRequest> facetRequests = new ArrayList<FacetRequest>();
CountFacetRequest home = new CountFacetRequest(new CategoryPath("Home", '/'), 100);
home.setDepth(5);
facetRequests.add(home);
facetRequests.add(new CountFacetRequest(new CategoryPath("Home/Sports", '/'), 10));
facetRequests.add(new CountFacetRequest(new CategoryPath("Home/Weather", '/'), 10));
FacetSearchParams fsp = new FacetSearchParams(facetRequests);

FacetsCollector facetsCollector = FacetsCollector.create(fsp, reader, taxor);
```

```
searcher.search(new MatchAllDocsQuery(), facetsCollector);

for (FacetResult fres : facetsCollector.getFacetResults()) {
    FacetResultNode root = fres.getFacetResultNode();
    printFacet(root, 0);
}
```

リスト 6 で、標準的な Lucene のインデクシングおよび検索の枠を超えている重要な部分は、FacetFields、FacetsCollector、TaxonomyReader、および TaxonomyWriter の各クラスを使用しているところにあります。FacetFields は、ドキュメント内に適切なフィールド・エントリーを作成し、インデクシング時に TaxonomyWriter と連携して動作します。検索時には、TaxonomyReader が FacetsCollector と連携して、カテゴリーごとに該当するカウントを取得します。また、Lucene のファセット処理モジュールは、2 つ目のインデックスを作成することにも注目してください。このインデックスが有効であるためには、メインのインデックスと同期した状態を維持していなければなりません。**リスト 6** のコードを実行するには、前の例で使ったのと同じコマンド・シーケンスを使用しますが、java コマンドの IndexingExamples は FacetExamples に置き換えてください。コードを実行すると、以下の結果になるはずです。

```
Home (0.0)
  Home/Children (3.0)
    Home/Children/Nursery Rhymes (3.0)
  Home/Weather (2.0)

Home/Sports (2.0)
  Home/Sports/Rock Climbing (1.0)
  Home/Sports/Hockey (1.0)
Home/Writing (1.0)
Home/Quotes (1.0)
  Home/Quotes/Yoda (1.0)
Home/Music (1.0)
  Home/Music/Lyrics (1.0)
...
```

この特定の実装には、コストが高くなる可能性のある Home ファセットのカウントを含めていないことに注意してください。このオプションをサポートするには、適切な FacetIndexingParams (この記事では説明しません) を設定します。Lucene のファセット処理モジュールには、他にもここでは取り上げていない機能があります。これらの機能 (および、この記事では触れていないその他の新しい Lucene の機能) については、記事の「[参考文献](#)」で詳しく探ってください。次は、Solr 4.x に話題を移します。

Solr 4: スケーラブルな検索とアナリティクス

API の観点からすると、Solr 4.x のルック・アンド・フィールは前のバージョンからほとんど変わっていません。しかし 4.x では、Solr をこれまでになく使いやすく、そしてスケーラブルにするためにさまざまな強化がなされています。さらに、このバージョンの Solr では新しいタイプの質問に回答できるようになっており、それはいずれも上記で概説した Lucene のさまざまな機能強化を利用することで実現しています。その他の変更は、開発者が Solr を使い始める際のエクスペリエンスを対象としたものです。例えば、まったく新しい「[Apache Solr Reference Guide](#)」(「[参考文献](#)」を参照) は、バージョン 4.4 以降のすべての Solr リリースのドキュメントとして、書籍に匹敵するクオリティーになっています。また、新たに導入された Solr のスキーマレス機能により、あらかじめスキーマを定義することなく簡単かつ迅速に新規データをインデックスに追加できるようになっています。Solr のスキーマレス機能についてはこの後すぐに説明しますが、その

前に、Solr Air アプリケーションで一部をデモンストレーションした、検索、ファセット処理、関連度といった機能が、Solr 4.x ではどのように強化されたか、その一部を紹介します。

検索、ファセット処理、関連度

Solr 4 の新機能のいくつかは、検索およびファセット処理面と、インデクシング面との両面で次世代のデータ駆動型アプリケーションを構築しやすくすることを目的にしています。表 1 にはこれらの見どころを要約し、ものによってはコマンドおよびコードの例を記載しています。

表 1. Solr 4 のインデクシング、検索、ファセット処理での見どころ

名称	説明	例
ピボット・ファセット処理	親ファセットでフィルタリングして、そのファセットのすべてのサブセットについてのカウントを収集します。詳細は、 Solr Air サンプル・アプリケーション を参照してください。	各種のフィールドでピボットを実行する例: http://localhost:8983/solr/collection1/travel? &wt=json&facet=true&facet.limit=5&fq=&q=*:*&facet.pivot=Origin, Dest, UniqueCarrier, FlightNum, ArrDelay&indent=true
新しい関連度関数クエリー	関数クエリーの中で、さまざまなインデックス・レベルの統計 (あるタームが含まれているドキュメントの数、そのタームの出現回数など) にアクセスします。	返されるすべてのドキュメントに対し、ターム「Origin:SFO」が含まれているドキュメントの数を追加する例: http://localhost:8983/solr/collection1/travel? &wt=json&q=*:*&fl=*,{!func}docfreq('Origin', %20'SFO')&indent=true このコマンドは、新しい DocTransformers 機能も利用することに注意してください。
結合	より複雑なドキュメントの関係を表し、それらのドキュメントを検索時に結合します。将来の Solr リリースでは、さらに複雑な結合のサポートが予定されています。	Airport データ・セットに出現する出発空港コードが含まれるフライトのみを返す例 (結合を行わずにリクエストの結果に照らし合わせます): http://localhost:8983/solr/collection1/travel? &wt=json&indent=true&q={!join%20from=IATA%20to=Origin}*:*
Codec サポート	個々のフィールドに対するインデックスおよびポスティング・フォーマット用の Codec を変更します。	フィールドに SimpleTextCodec を適用する例: <fieldType name="string_simpletext" class="solr.StrField" postingsFormat="SimpleText" />
新規更新プロセッサ	Solr に送信されたドキュメントに対してインデクシングが行われる前に、そのドキュメントに変更を加えるためのコードを、Solr の更新プロセッサ・フレームワークを使用して追加します。	<ul style="list-style-type: none"> フィールドの変形 (フィールドの連結、数値の解析、トリムなど) スクリプトによる処理。JavaScript コードや、JavaScript エンジンでサポートされているその他のコードを使用してドキュメントを処理します。Solr Air サンプル・アプリケーションの update-script.js ファイルを参照してください。 ドキュメントで使用されている言語 (英語や日本語など) を特定することを目的とした言語検出 (厳密には、3.5 でも使用できますが、ここでも取り上げておくだけの価値があります)。
アトミック更新	ドキュメントの変更された部分のみを送信し、残りの部分を Solr に処理させます。	コマンドラインから、cURL を使用してドキュメントの送信元を 243551 から F00 に変更する例: curl http://localhost:8983/solr/update -H 'Content-type:application/json' -d '[{"id":"243551","Origin":{"set":"F00"}}]'

表 1 の最初の 3 つのサンプル・コマンドは、Solr Air デモ・データに対して (Solr Air UI ではなく) ブラウザーのアドレス・フィールドで実行することができます。

関連度関数、結合、codec (およびその他の Solr 4 の新しい機能) についての詳細は、「[参考文献](#)」で Solr Wiki へのリンクや、その他の関連するリンクを参照してください。

スケーリング、NoSQL、NRT

ここ数年での Solr に対する最も重要な変更を 1 つ挙げるとしたら、それはおそらく、複数ノードによるスケーラブルな検索ソリューションを作成するのが極めて簡単になったことです。Solr 4.x

では、Solr をこれまで以上に簡単に、何十億ものレコードを保管してアクセスするための信頼できるメカニズムへとスケールリングすることができます。しかも、今まで Solr を有名にしてきた検索機能とファセット処理機能も引き続き利用することができます。さらに、キャパシティの変更が必要になった場合には、クラスターのバランスを再調整できるだけでなく、オプティミスティック・ロックやコンテンツのアトミック更新を利用することも、まだインデクシングされていないデータをリアルタイムで取得することもできます。Solr に新しく追加されたこれらの分散機能は、まとめて SolrCloud と呼ばれます。

SolrCloud がどのように機能するかと言うと、まず、ドキュメントが (オプションの) 分散モードで実行中の Solr 4 に送信されると、そのドキュメントはハッシュ・メカニズムに従ってクラスター内の (リーダー (leader) と呼ばれる) ノードにルーティングされます。すると、リーダーはそのドキュメントに対してインデクシングを行ってシャード (shard) に格納します。これにより、シャードには 1 つのリーダーによって扱われる単一のインデックスとゼロ以上のレプリカが存在するようになります。

例として、4 台のマシンと 2 つのシャードがあるとします。Solr の起動時に、4 台のマシンのそれぞれが他の 3 台のマシンと通信し、そのうちの 2 台のマシンがリーダーとして選出されます (1 つのシャードにつき 1 つのリーダー)。残りの 2 つのノードは、自動的にいずれかのシャードに属するレプリカになります。何らかの理由でいずれかのリーダーに障害が発生した場合には、レプリカ (この例では唯一のレプリカ) がリーダーの役目を引き継ぐため、システムは引き続き正常に機能することが保証されます。この例から推測できるように、本番システムでは、システム停止に対処できるだけの十分な数のノードが参加する必要があります。

SolrCloud の動作を確認するには、[Solr Air サンプル・アプリケーション](#)で使った start-solr.sh スクリプトを、`-z` フラグを指定して実行し、2 つのノードと 2 つのシャードからなるシステムを起動します。そのためにはまず、*NIX コマンドラインから以下のコマンドを実行して、古いインスタンスをシャットダウンします。

```
kill -9 PROCESS_ID
```

次に、システムを再起動します。

```
bin/start-solr.sh -c -z
```

Apache ZooKeeper

Zookeeper は、リーダーの選出、定足数の設定、およびその他のタスクを実行することで、クラスター内のノードを調整するように設計された分散コーディネーション・システムです。Zookeeperのおかげで、Solr クラスターが「分割脳」シンドロームに悩まされることは決してありません。分割脳シンドロームとは、パーティショニング・イベントの結果、クラスターの一部がクラスターの他の部分とは独立して動作するようになるという事態です。Zookeeper についての詳細は、「[参考文献](#)」を参照してください。

`-c` フラグにより、古いインデックスが消去されます。`-z` フラグは Solr に対し、組み込みバージョンの [Apache Zookeeper](#) を使用して起動するように指示します。

ブラウザーで SolrCloud 管理ページ (<http://localhost:8983/solr/#/~cloud>) にアクセスして、クラスターに 2 つのノードが参加していることを確認してください。これを確認した後、コンテンツに

対して再度インデクシングを行うと、インデックスは両方のノードに分配されます。システムに対するすべてのクエリーも自動的に分配されます。このクラスターですべてのドキュメントと突き合わせを行う検索を実行した場合、2つのノードに対して実行したときと、1つのノードに対して実行したときのヒット件数は同じになるはずです。

start-solr.sh スクリプトは 1 つ目のノードに対し、以下のコマンドで Solr を起動します。

```
java -Dbootstrap_confdir=$SOLR_HOME/solr/collection1/conf
-Dcollection.configName=myconf -DzkRun -DnumShards=2 -jar start.jar
```

このスクリプトは 2 つ目のノードに対し、Zookeeper の場所を指示します。

```
java -Djetty.port=7574 -DzkHost=localhost:9983 -jar start.jar
```

組み込み Zookeeper は、使い始めるには素晴らしいのですが、本番システム用に高可用性と耐障害性を確保するには、クラスター内にスタンドアロンの Zookeeper インスタンス一式をセットアップする必要があります。

SolrCloud 機能をベースに、NRT のサポートと、以下をはじめとする多数の NoSQL ライクな機能が追加されています。

- オプティミスティック・ロック
- アトミック更新
- リアルタイムの取得操作 (特定のドキュメントを、それがコミットされる前に取得)
- トランザクション・ログに支えられた永続性

Solr の分散機能と NoSQL 機能の多く (ドキュメントおよびトランザクション・ログの自動バージョン管理など) は、追加の設定なしですぐに使用できます。それ以外のいくつかの機能を使用する際には、表 2 の説明と例を参考にしてください。

表 2. Solr 4 の分散機能および NoSQL 機能の要約

名称	説明	例
リアルタイムでの取得	インデクシングや分散の状態に関わらず、ID を指定することによってドキュメントを取得します。	ID が 243551 のドキュメントを取得する例: http://localhost:8983/solr/collection1/get?id=243551
シャード分割	インデックスを複数の小さなシャードに分割して、クラスター内の新しいノードにマイグレーションできるようにします。	shard1 を 2 つのシャードに分割する例: http://localhost:8983/solr/admin/collections?action=SPLITSHARD&collection=collection1&shard=shard1
NRT	NRT を使用すると、前のバージョンよりも速かに短時間で新規コンテンツを検索できます。	solrconfig.xml ファイルで <autoSoftCommit> を有効にします。例: <autoSoftCommit> <maxTime>5000</maxTime> </autoSoftCommit>>
ドキュメントのルーティング	どのドキュメントをどのノードで存続させるかを指定します。	ユーザーのデータのすべてが特定のマシン上にあることを確認する必要があります。Joel Bernstein 氏のブログ投稿を読んでください (「 参考文献 」を参照)。
コレクション	Solr の新規コレクション API を使用して、必要に応じてコレクションをプログ	hockey という名前の新規コレクションを作成する例: http://localhost:8983/solr/admin/collections?action=CREATE&name=hockey&numShards=2

ラムによって作成、削除、または更新します。

スキーマレスへの移行

スキーマレスとはマーケティングのための誇大広告なのか？

データ・コレクションにスキーマがないことはめったにありません。「スキーマレス」とは、データが取らなければならない形式をエンジンが指定しなくても、スキーマがどうかをエンジンに「知らせる」データに対し、データ取得エンジンが適切に反応できることから生まれたマーケティング用語です。例えば、Solr は JSON 入力を受け入れ、その JSON に暗黙的に定義されたスキーマに基づいて、コンテンツに対して適切にインデクシングすることができます。スキーマを 2 箇所 (例えば、JSON ドキュメントと Solr) で定義する代わりに 1 箇所 (JSON ドキュメント) で定義するということなので、誰かが Twitter で私に指摘したように、「スキーマレス」と呼ぶよりも、「レススキーマ」(より少ないスキーマ)と呼ぶ方が適切な表現です。

私の経験から言うと、システムが実際のデータの型を誤って認識しているために午前 2 時にデバッグ作業に追われるという事態を厭わないというのでない限り、ほぼ必ずと言ってよいほど、本番システムではスキーマレス機能を使用するべきではありません。

Solr のスキーマレス機能は、最初に schema.xml ファイルを定義するというオーバーヘッドなしに、クライアントが迅速にコンテンツを追加できるようにします。Solr は受信したデータを検査し、一連の連鎖する値パーサーにデータを渡していきます。値パーサーはデータの型を推測し、該当するフィールドを自動的に内部スキーマに追加して、そのコンテンツをインデックスに追加します。

通常の本番システム (いくつかの例外はあります) ではスキーマレス機能を使用するべきではありません。それは、値の推測は常に完璧であるとは限らないためです。例えば、Solr が新しいフィールドを初めて認識したときに、そのフィールドを整数として識別したことから、スキーマには整数の `FieldType` を定義したとします。しかし、Solr がそのフィールドで認識するコンテンツの残りが浮動小数点の値で構成されているとしたら、例えば 3 週間経ってからようやくそのフィールドが検索にまったく役に立っていないと気付く羽目になります。

その一方、開発の初期段階や、フォーマットをほとんど何も制御できないコンテンツに対してインデクシングを行う場合には、スキーマレス機能が特別役に立ちます。例えば表 2 には、Solr のコレクション API を使用して新規コレクションを作成する以下の例が記載されています。

```
http://localhost:8983/solr/admin/collections?action=CREATE&name=hockey&numShards=2)
```

コレクションを作成した後は、スキーマレス機能を使用して、そのコレクションにコンテンツを追加することができます。ただし最初に、現在のスキーマを調べてください。スキーマレス機能のサポートを実装する一環として、Solr ではスキーマにアクセスするための REST (Representational State Transfer) API も追加しました。ブラウザー (またはコマンドラインの `cURL`) で `http://localhost:8983/solr/hockey/schema/fields` にアクセスすると、hockey コレクションに対して定義されたすべてのフィールドを確認することができます。表示されるフィールドは、すべて Solr Air サンプル・アプリケーションのものです。create オプションは、私のデフォルト構成を新しいコレクションのベースとして使用するため、スキーマはこれらのフィールドを使用します。この構成は、必要に応じてオーバーライドすることができます (補足: サンプル・コードのダウン

ロードに含まれている setup.sh スクリプトは、新しいスキーマ API を使用してすべてのフィールド定義を自動的に作成します。

スキーマレス機能を使用してコレクションにコンテンツを追加するには、以下のスクリプトを実行します。

```
bin/schemaless-example.sh
```

先ほど作成した hockey コレクションには、以下の JSON が追加されます。

```
[
  {
    "id": "id1",
    "team": "Carolina Hurricanes",
    "description": "The NHL franchise located in Raleigh, NC",
    "cupWins": 1
  }
]
```

この JSON をコレクションに追加する前のスキーマを調べるとわかるように、`team`、`description`、`cupWins` は新しいフィールドです。上記のスクリプトが実行されたとき、Solr はこれらのフィールドのタイプを自動的に推測して、スキーマ内にフィールドを作成しました。確認のために、<http://localhost:8983/solr/hockey/schema/fields> で結果を最新のものに更新してください。今度はフィールドのリストに `team`、`description`、`cupWins` のすべてが定義されていることがわかるはずです。

(地理空間だけではない) 空間の改善

Solr では長年にわたってポイント・ベースの空間検索をサポートしているため、ユーザーはあるポイントから一定の距離内にあるすべてのドキュメントを検索することができます。Solr はこの手法を n 次元の空間でサポートしているものの、ほとんどのユーザーは地理空間検索でこの手法を使用しています (例えば、自分がいる場所の近くにあるすべてのレストランの検索など)。ただし、これまでの Solr には、多角形に対するインデクシングやインデクシングされた多角形内での検索などといった、より複雑な空間機能がサポートされていませんでした。新しい空間パッケージには、以下の見どころがあります。

- Spatial4J ライブラリー (「[参考文献](#)」を参照) による新しいさまざまな空間タイプ (長方形、円、線、任意の多角形など) のサポート、および WKT (Well Known Text) フォーマットのサポート。
- 多値インデックス付きフィールド。複数のポイントを同じ 1 つのフィールドにエンコードするために使用することができます。
- 構成可能な精度。開発者が精度と計算速度との間のバランスをより柔軟に制御することができます。
- コンテンツの高速フィルタリング。
- `Is Within`、`Contains`、`IsDisjointTo` を使用したクエリーのサポート。
- オプションの JTS (Java Topological Suite) サポート (「[参考文献](#)」を参照)。
- Lucene API および成果物。

Solr Air アプリケーションのスキーマには、新しい空間機能を利用するようにセットアップされたいくつかのフィールド・タイプがあります。以下の 2 つのフィールド・タイプは、空港データの緯度と経度に対処するために定義しました。

```
<fieldType name="location_jts" class="solr.SpatialRecursivePrefixTreeFieldType"
distErrPct="0.025" spatialContextFactory=
"com.spatial4j.core.context.jts.JtsSpatialContextFactory"
maxDistErr="0.000009" units="degrees"/>

<fieldType name="location_rpt" class="solr.SpatialRecursivePrefixTreeFieldType"
distErrPct="0.025" geo="true" maxDistErr="0.000009" units="degrees"/>
```

`location_jts` フィールド・タイプでは、オプションの JTS 統合を明示的に使用してポイントを定義している一方、`location_rpt` フィールド・タイプではこのオプションを使用していません。単純な長方形よりも複雑な形状に対してインデクシングを行う場合には、JTS バージョンを使用する必要があります。システムの精度は、フィールドの属性を利用して定義することができます。これらの属性は、インデクシングの際に必要になります。Solr では検索時にデータを効率的に使用できるようにするために、Lucene と Spatial4j によって複数の方法でデータをエンコードするからです。皆さんが作成するアプリケーションでは、そのアプリケーションに固有のデータを使用してテストを実行し、インデックスのサイズ、精度、クエリー時のパフォーマンスに関するトレードオフを決定することになるはずです。

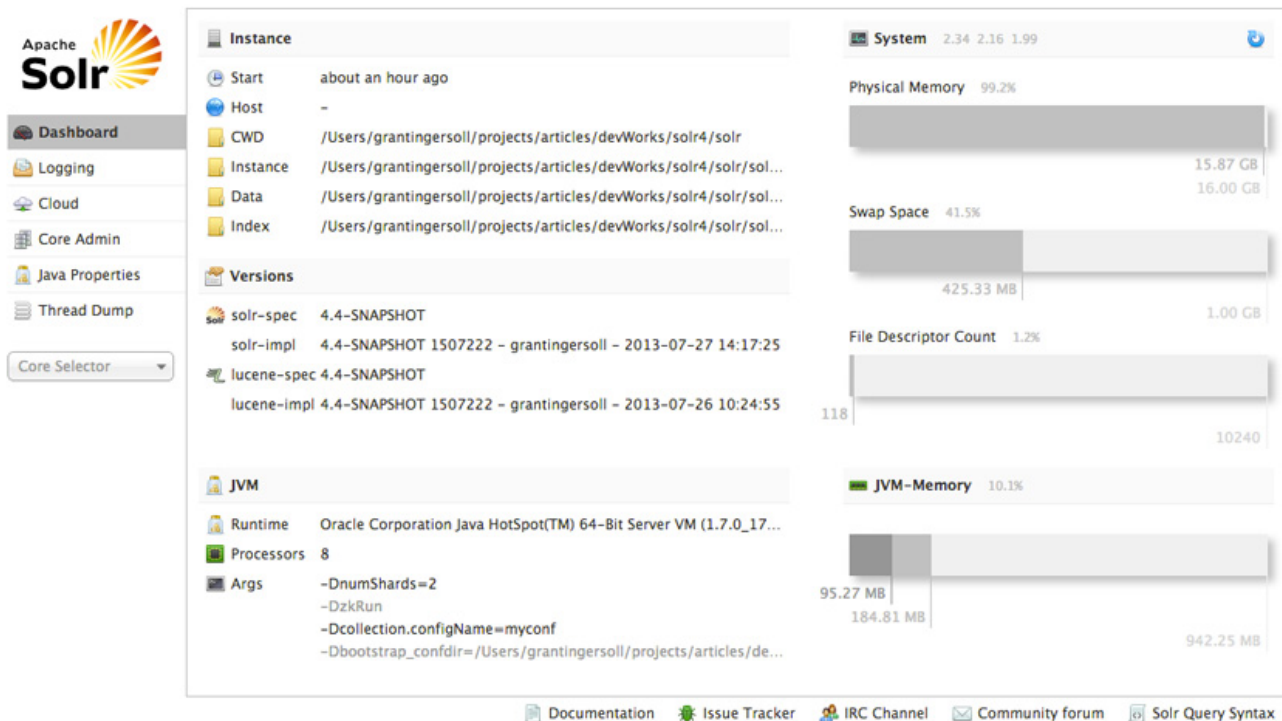
さらに、Solr Air アプリケーションで使用されている `near` クエリーでは、指定された出発空港および到着空港近辺の空港を検索するために、新しい空間クエリー構文 (`Circle` での `IsWithin`) を使用しています。

新しい管理用 UI

Solr に関するセクションを締めくくるに当たり、これまでになくユーザー・フレンドリーな最新の Solr 管理用 UI を紹介しないわけにはいきません。この新しい UI は、ルック・アンド・フィールが整理されているだけでなく、SolrCloud 用の新しい機能や、ドキュメント追加用の新しい機能、その他多数の新機能が追加されています。

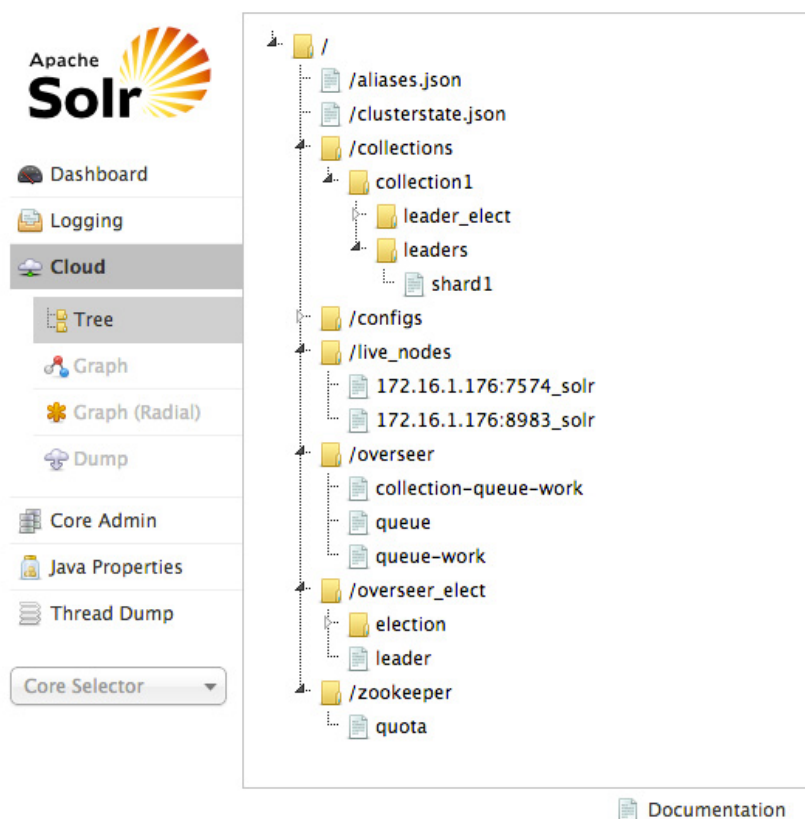
まず手始めに、ブラウザーで `http://localhost:8983/solr/#/` にアクセスしてください。すると、Solr の現在の状態に関する多くの情報 (メモリー使用量、作業ディレクトリーなど) が簡潔に取り込まれたダッシュボードが表示されます (図 7 を参照)。

図 7. Solr のダッシュボードの例



ダッシュボードの左側に表示されている「Cloud (クラウド)」を選択すると、UI に SolrCloud に関する詳細が表示されます。例えば、構成、アクティブ・ノード、およびリーダーの状態に関する詳細情報と併せ、視覚化されたクラスター・トポロジが表示されます。図 8 に一例を示します。少し時間をとって、クラウド UI で可能なすべての操作をひとつおりに行ってください (可能な操作を表示するには、SolrCloud モードで実行していなければなりません)。

図 8. SolrCloud の UI の例



特定のコア/コレクション/インデックスに結び付けられていない UI の領域として最後に取り上げるのは、「Core Admin (コア管理)」の一連の画面です。これらの画面では、コアの追加、削除、リロード、切り替えを含め、コアの管理をポイント・アンド・クリック操作で制御することができます。図 9 に、コア管理の UI を示します。

図 9. コア管理の UI の例

The screenshot displays the Apache Solr Core Admin interface. On the left, a sidebar contains navigation options: Dashboard, Logging, Cloud, Core Admin (highlighted), Java Properties, and Thread Dump. A 'Core Selector' dropdown is positioned below the sidebar. The main panel shows a list of cores under the heading 'collection1'. The selected core's details are shown on the right, including a 'Core' section with fields like startTime, instanceDir, and dataDir, and an 'Index' section with fields like lastModified, version, numDocs, maxDoc, deletedDocs, optimized, current, and directory. The 'optimized' and 'current' fields are marked with red 'X' icons, indicating they are not optimized or current.

Field	Value
startTime:	about 2 hours ago
instanceDir:	/Users/grantingersoll/projects/articles/devWorks/solr4/solr,
dataDir:	/Users/grantingersoll/projects/articles/devWorks/solr4/solr,
lastModified:	about an hour ago
version:	809
numDocs:	1313315
maxDoc:	1320729
deletedDocs:	7414
optimized:	
current:	
directory:	org.apache.lucene.store.NRTCachingDirectory:NRTCachingDi /Users/grantingersoll/projects/articles/devWorks/solr4/solr, lockFactory=org.apache.lucene.store.NativeFSLockFactory@7

「Core (コア)」リストからコアを選択すると、そのコアに固有の情報および統計の概要にアクセスすることができます。図 10 に一例を示します。

図 10. コアの概要の例

The screenshot displays the Apache Solr Admin UI. On the left is a sidebar with the Solr logo and navigation links: Dashboard, Logging, Cloud, Core Admin, Java Properties, Thread Dump, a dropdown menu (currently showing 'collection1'), Overview (selected), Analysis, Config, Dataimport, Documents, Ping, Plugins / Stats, Query, Replication, Schema, and Schema Browser. The main content area is titled 'Statistics' and shows the following information:

- Last Modified: about an hour ago
- Num Docs: 1313315
- Max Doc: 1320729
- Deleted Docs: 7414
- Version: 809
- Segment Count: 22
- Optimized: [optimize now](#)
- Current:

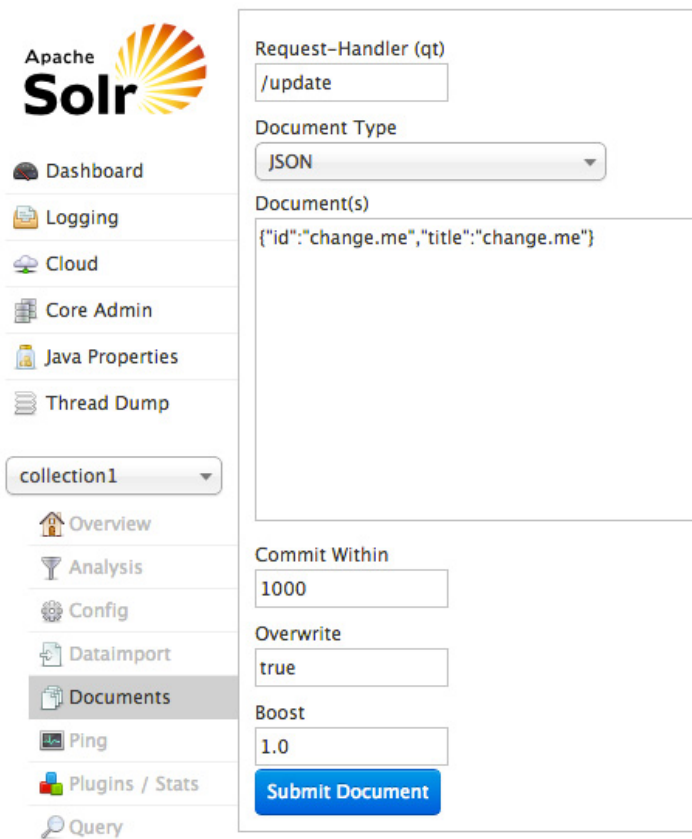
Below the statistics is the 'Replication (Slave)' section, which contains a table:

	Version	Gen	Size
Master (Searching)	1375107529390	26	102.71 MB
Master (Replicable)	1375107529390	26	-
Slave (Searching)	1375107529385	26	108.16 MB

At the bottom is the 'Healthcheck' section, which displays the message: 'Ping request handler is not configured with a healthcheck file.'

コアごとの機能のほとんどは、4.x より前のバージョンの UI の機能と同様ですが (ただし、遥かに良くなっています)、「Documents (ドキュメント)」については例外です。「Documents (ドキュメント)」では、さまざまなフォーマット (JSON、CSV、XML など) のドキュメントを UI から直接コレクションに追加できるようになりました (図 11 を参照)。

図 11. UI からドキュメントを追加する例



Apache Solr

- Dashboard
- Logging
- Cloud
- Core Admin
- Java Properties
- Thread Dump

collection1

- Overview
- Analysis
- Config
- Dataimport
- Documents**
- Ping
- Plugins / Stats
- Query

Request-Handler (qt)
/update

Document Type
JSON

Document(s)
{"id":"change.me","title":"change.me"}

Commit Within
1000

Overwrite
true

Boost
1.0

Submit Document

PDF や Word などのリッチなタイプのドキュメントをアップロードすることもできます。時間を取って、インデックスにいくつかのドキュメントを追加したり、その他のコレクションごとの機能 (「Query (クエリー)」インターフェースや改良された「Analysis (分析)」画面など) をブラウズしたりしてみてください。

今後の展開

次世代の検索エンジン・テクノロジーでは、ユーザーがデータの活用方法を決めることができます。Lucene および Solr 4 でできることをたっぷり紹介したこの記事を読んだことで、アナリティクスと推奨が関係する、テキスト・ベースではない検索問題を検索エンジンがどのように解決するかについて、皆さんの知識が広がったことを願います。

Lucene と Solr は、30 人を超えるコミッターと数百人のコントリビューターに支えられて維持されている大規模なコミュニティのおかげで常に進化を続けています。現在、このコミュニティがアクティブに開発している主要なブランチは、最新の公式リリース 4.x ブランチと、次のメジャー・リリース (5.x) を表すトランク・ブランチの 2 つです。公式リリースのブランチでは、コミュニティは、後方互換性と現在のアプリケーションの容易なアップグレードの両方に重点を置いたインクリメンタル方式での開発に全力で取り組んでいます。トランク・ブランチでは、前のリリースとの互換性の確保という点で、コミュニティの制約はやや緩くなっています。Lucene または Solr の最先端技術を試してみたいという方は、Subversion または Git からトランク・ブランチのコードをチェックアウトしてください (「[参考文献](#)」を参照)。いずれを選択す

るにしても、Lucene および Solr を利用することで、プレーン・テキスト検索を遥かに上回る強力な検索ベースのアナリティクスを実現することができます。

謝辞

David Smiley 氏、Erik Hatcher 氏、Yonik Seeley 氏、Mike McCandless 氏のご協力に感謝いたします。

ダウンロード

内容	ファイル名	サイズ
Sample code	code.zip	60.3MB

著者について

Grant Ingersoll



Grant Ingersoll は、LucidWorks の共同創設者であると同時に CTO でもあり、Lucene コミュニティーのアクティブなメンバー — Lucene および Solr のコミッター、Apache Mahout 機械学習プロジェクトの共同設立者、そして Apache Software Foundation の長期にわたるメンバー — にもなっています。Grant は、シラキュース大学の自然言語処理センターで自然言語処理および情報検索の研究に携わった経験があり、アマースト大学で数学およびコンピューター・サイエンスの学士号を取得し、シラキュース大学でコンピューター・サイエンスの修士号を取得しています。彼は、『Taming Text』(Manning Publications、2013年) の共著者でもあります。Twitter の @gsingers で Grant をフォローしてください。

© Copyright IBM Corporation 2013

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)