Marktdaten Pre-Processing Engine: Anforderungen

Andreas Mayer
ng.computer.science@googlemail.com

11. Juni 2013

Zusammenfassung

Auf Änderungen im Marktdatenumfeld muss in der Regel zeitnah reagiert werden. Häufig unterliegen Folgeprozsse engen zeitlichen Restriktionen, so dass Verzögerungen nach Möglichkeit zu vermeiden sind. Hierzu muss das zugrundeliegende System diversen Anforderungen genügen. Diese Anforderungen sind der Mittelpunkt dieses Essays.

Dieser Essay trifft die Annahme, dass die Persistenz der Daten durch eine eigene Komponente erfolgt und dass ein fixes Mapping zwischen Spalten und Datenbankentitäten besteht.

1 Einleitung

Das pre-processing von Marktdaten soll möglichst generisch, transparent und einfach erfolgen. Sonderfälle sollen jedoch auch abgebildet werden können. Da die meisten Änderungen relativ einfacher Natur sind, sollte die Konfiguration ohne weitreichende IT-/Informatik-Kenntnisse möglich sein. Hierdurch sollen die Betriebskosten möglichst gering gehalten werden.

Die in diesem Artikel skizziere Lösung befindet sich seit Ende 2009 im produktiven Einsatz ist. Über diese Lösung werden u.a. Daten von Bloomberg, MSCI, J.P. Morgan, Merrill Lynch, Frankfurt Stock Exchange, Tullett, UBS und SIX Swiss Exchange verarbeitet. Einige der vorgestellten Ideen sind bisher nicht Teil der Produktivlösung, sondern befinden sich lediglich auf der

Roadmap.

Das System ist die Grundlage dieses Essays, da es sich als äußerst praxistauglich erwiesen hat.

In den folgenden Kapiteln wird ein grundlegender Anforderungskatalog definiert. Je nach Komplexität des jeweils implementierten Zielsystems können die konkreten Anforderungen variieren. Der Anforderungskatalog beschreibt somit einige, meiner Erfahrung nach, wichtige Aspekte und erhebt keinen Anspruch auf Vollständigkeit.

Spätere Essays werden die Lösung konkretisieren und diverse Aspekte des Systems detailliert beschreiben.

2 Architektur Anforderungen

Dieses Kapitel behandelt Anforderungen die unabhängig von der spezifischen Domäne zu betrachten sind. Diese Anforderungen sollen von der gewählten Softwarearchitektur unterstützt bzw. durch sie ermöglicht werden.

2.1 Flexibles Changemanagement

Die Engine muss in der Lage sein, sich möglichst flexibel an veränderte exogene Bedingungen anpassen zu können. Hierzu gehören bspw. Änderungen an Marktdatenlieferungen. Hierzu ist ein Paradigma zu implementieren, welches kurze Einlernzeiten, bei gegebenen persönlichen Fähigkeiten, garantiert. Änderungen sollen schnell und möglichst von *jedem* Mitarbeiter[1] umgesetzt werden können.

2.2 Klares Design

Die *Code Base* soll eine klare Struktur besitzen und Redundanzen möglichst vermeiden. Dies vereinfacht die Implementierung von automatischen Unit Tests und reduziert die Einarbeitungszeiten für neue Mitarbeiter.

[1] Ich verwende in diesem Essay die maskuline Form des Wortes, beziehe Frauen jedoch ausdrücklich in die Formulierung mit ein.

2.3 Getrennte Versionierung Config/Engine

Da sich die Kern-Funktionalität der Engine, im Vergleich zu den nötigen Änderungen im Tagesgeschäft, relativ selten verändert, wäre eine Versionierung basierend auf einem Source Code Managementsystem (SCM) ein zu großer Overhead. Bei der Erstellung jedes Konfigurationsartefaktes müsste eine neue Version der Engine deployed werden. Die Engine und die dazugehörigen Konfigurationsdaten sollen in getrennten Projekten abgebildet werden. Die Trennung sorgt dafür, dass die Versionierung der Engine unabhängig von den Konfigurationsänderungen erfolgen kann. Hierduch kann die Weiterentwiclung der Engine sich an einer Roadmap orientieren.

Der Prozess zur Verwaltung der Engine kann schwergewichtig sein, wobei der Prozess zur Pflege der Konfigurationen sehr leichtgewichtig ausfallen kann.

2.4 Modularität

In der Regel ist nicht klar welche Features zukünftig benötigt werden. Der Grund hierfür sind die mannigfaltigen Formate, in denen die Anbieter die Daten zur Verfügung stellen. Die Programmstruktur soll es quasi ermöglichen neue Funktionen an einer beliebigen Stelle einzubinden. Dies gewährleistet eine sehr hohe Flexibilität bezogen auf mögliche Erweiterungen.

3 Domänenspezifische Anforderungen

In diesem Kapitel werden konkrete Anforderungen behandelt.

3.1 Chronologische Konsistenz von Konfigurationsartefakten

Die Engine darf, bezogen auf eine zu verarbeitende Entität, keine globale und statische Konfiguration aufweisen. Konfigurationsartefakte müssen immer für einen bestimmten Zeitraum Gültigkeit besitzen. Die letzte gültige Konfiguration gilt ad infinitum.

Beispiel:
Eine Datei verändert sich bspw. am 01.01.2013, so dass eine neue Konfiguration angelegt werden muss. Die Engine muss sicherstellen, dass die Dateien vor dem 01.01.2013 mit der alten Konfiguration verarbeitet werden und erst ab dem Datum der Änderung mit der neuen Version. Hierdurch wird eine historisch korrekte Verarbeitung der Daten sichergestellt.

3.2 Separation von Header und Body

Die Engine muss in der Lage sein, den Header- und Bodyanteil der Daten zu unterscheiden. In der späteren Verarbeitung der Daten soll der Header als Metainformation hinsichtlich der konkreten Struktur der Datensätze genutzt werden können.

Die Header müssen über ein *pre-processing* manipuliert werden können. Hierbei sollen zum einen standardisierte Methoden zum einsatz kommen, andererseits sollen aber auch maßgeschneiderte Prozesse implementiert werden können. Diese Maßnahme ist nötig, da bspw. Header von Bloomberg anders in einen Zeilenvektor konvertiert werden müssen als Header von MSCI. Einige Files enthalten bereits einen *brauchbaren*[2] Header, sodass keine weitere Verarbeitung notwendig ist.

3.3 Dynamische Workflow-Struktur (optional)

Meiner Erfahrung nach bietet sich eine Workflow-Struktur an, in der Komponenten dynamisch zusammengefügt werden können. Dies sollte über die Konfiguration steuerbar sein. Ein statischer Ablauf kann ausreichend sein, es kommt jedoch zu Redundanz und zu einer Verkomplizierung der Komponenten durch Fallunterscheidungen.

Beispiel:
Stellen Sie sich vor, Sie haben eine Filter-Komponente entwickelt und möchten diese in den Ablauf integrieren. Sie haben eine geeignete Stelle gefunden und alles scheint gut zu funktionieren. Der Filter arbeitet ziemlich weit vorne in der Verarbeitungskette und ist somit in der Lage die Rohdaten zu filtern.

[2] Die Verarbeitung von originären Headern ist selbstverständlich möglich. Ich halte es für den einfacherern Ansatz die Header bereits ganz am Anfang zu normalisieren.

Es ist nun möglich, dass sich im weiteren Ablauf Komponenten befinden, die die Originaldaten anreichern oder verändern. Wenn Sie nun einen Filter weiter hinten in der Prozesskette benötigen, müssen Sie stages einführen. Der konfigurierte Filter muss schließlich wissen in welcher stage er die Daten überprüfen soll. In diesem Fall muss der Hauptalgorithmus außerdem um einen zusätzlichen hook erweitert werden.Je nach Komplexität der Verarbeitungskette kann das statische Konstrukt an seine Grenzen stoßen. Die Implementierung der statischen Lösung ist im Gegensatz zu der dynamischen Workflow-Struktur aber einfacher[3].

3.4 Header-Body Synchronisation

Header und Body sollten immer synchron sein. Wenn eine Komponente eine Spalte entfernt oder hinzufügt muss der Header entsprechend angepasst werden. Dies vereinfacht die spätere Konfiguration[4].

Es soll möglichst vermieden werden mit fixen Spaltenindizes zu arbeiten, da dies ein hohes Fehlerpotenzial birgt. Stattdessen sollte mit den konkreten Spaltenüberschriften gearbeitet werden.

Beispiel:
Eine Datei enthält den Wert Yield in der vierten Spalte. Eine Prüfmethode, bspw. `assert(column(4) > 0)` sollte lieber durch `assert(Yield > 0)` ersetzt werden. Die Engine soll die konkreten Spaltenindizes idealerweise selbst ermitteln. Hierdurch wird der durch Verschiebungen von Spalten verursachte operative Aufwand neutralisiert.

3.5 Paketierung

Die Paketierung[5] ermöglicht es eine Version inkl. aller Abhängigkeiten zu deployen. Sollte die Notwendigkeit bestehen, können beliebig komplexe Fallback-

[3]Wenn ich heute eine neue Lösung entwickeln würde, wäre das Workflow-Prinzip von Anfang an ein klarer Basisbestandteil. Im von mir entwickelten System musste der Hauptalgorithmus bereits um diverse, komponentenspezifische hooks erweitert werden, so dass die Struktur darunter gelitten hat.

[4]Komponenten können später anstelle von Spaltenindizes mit konkreten Überschriften konfiguriert werden.

[5]Die Paketierung allein reicht natürlich nicht aus. Ein entsprechender Build- und Test-Prozess ist obligatorisch.

Mechanismen implementiert werden. Die Paketierung begünstigt außerdem einen *Continuous Integration*[6] Ansatz. Fehler, welche auf inkonsistente Artefakte zurückzuführen sind[7], werden hierdurch vermieden.

3.6 Zeilenweise vs. gesamte Verarbeitung

In der Regel kann der Großteil der Daten line-by-line verarbeitet werden. Die Engine muss jedoch auch in der Lage sein, bei Bedarf eine *read-all-process-later*[8] Funktionalität umzusetzen. Es gibt Fälle in denen es nötig ist eine Datei erst komplett in den Speicher zu laden, bevor die gewünschte Verarbeitung durchgeführt werden kann. Hierbei kann es sich bspw. um ein Mapping handeln, welches implizit in den Daten enthalten ist.

[6] http://martinfowler.com/articles/continuousIntegration.html

[7] Dies tritt auf, wenn bspw. eine Funktion konfiguriert ist, die nicht von der Engine interpretiert werden.

[8] Dieses Verhalten kann auch simuliert werden wenn nur eine zeilenweise Verarbeitung implementiert ist. Eine Komponente muss die Zeilen lesen und in einen Cache schreiben. Ein darauf folgendes PostProcessing Modul muss den Cache dann auslesen, verarbeiten und die Daten in der gewünschten Form ausgeben.