Komfortables Job Scheduling auf Basis des Oracle Scheduler

Peter Hawelka und Dieter Pfingsten
pdv Technische Automation + Systeme GmbH
Hamburg

Schlüsselworte:

Job Scheduler, Job-Scheduling, dbms_Scheduler, Batch, Steams Advanced Queueing

Motivation und Übersicht

Trotz umfangreicher online-Verarbeitung innerhalb von Benutzerdialogen bleibt die Job-Verarbeitung in datenintensiven Applikationen eine wichtige Komponente auch in modernen komplexen IT-Systemen. Die Job-Ablaufsteuerung ist dabei das Instrument zum kontrollierten Ablauf der Hintergrundprozesse einzeln oder logisch verkettet. Der Fehlerbehandlung und dem Wiederaufsetzen von Jobs kommt dabei eine besondere Bedeutung zu.

In unserem Vortrag geben wir einen kurzen Überblick, wie wir in einem konkreten Projektumfeld die Oracle-Scheduling-Funktionen der Datenbank für den Aufbau eines leistungsstarken, einfach zu bedienenden Job Scheduler genutzt haben und welche Erweiterungen und Zusatzmodule notwendig waren um die Anforderungen an einen modernen Job Scheduler umzusetzen. Insbesondere stellen wir Ihnen die mit dem Oracle ADF-Framework erstellte Benutzeroberfläche vor und gehen kurz auf die Möglichkeiten ein, die ADF 11g mit seinen Ajax Komponenten zur Erstellung einer modernen Web-Oberfläche bietet, wie sie sonst nur mittels klassischer Rich-Clients umgesetzt werden konnten.

Der hier vorgestellte Job Scheduler soll in einer komplexen, bereits bestehenden, Oracle-Datenbankanwendung mit Oracle Forms-Clients und einer Internet gestützten B2B-Komponente, sämtliche anfallenden Abläufe zentral steuern und automatisieren. Die abzubildenden Arbeitsabläufe ergeben sich aus den Geschäftsprozessen eines großen Lotterieveranstalters in Hamburg und seinen unabhängigen Vertriebsorganisationen. In der Zentrale werden Gewinnziehungen durchgeführt und die Gewinninformationen und Spielberechtigungen mit den Vertriebsorganisationen ausgetauscht. Kommunikationswege sind der klassische DFÜ-Datenaustausch und die internetgestützte Bereitstellung von B2B-Diensten für die dezentralen Vertriebsorganisationen.

Die Motivation, das bisher im Einsatz befindliche Job Scheduling Produkt durch ein nur auf Oracle-Funktionen basierendes System zu ersetzen war eine geplante Migration der Oracle-Datenbanken in eine virtualisierte Real Application Cluster (RAC) Umgebung. Vorzugsweise sollte in dieser Umgebung auf Third Party - Herstellerprodukte verzichtet werden, soweit dieses wirtschaftlich sinnvoll umgesetzt werden kann. Hieraus ergaben sich teilweise spezielle Anforderungen an die Job-Scheduler-Funktionen. So sollte die Migration der bestehenden Arbeitsabläufe (Workflows), die in speziellen Datenbankskripten hinterlegt sind, einfach und kostengünstig möglich sein. Die grundlegenden Eigenschaften des Oracle Schedulers mussten daher angepasst werden. Um dieses einfach zu erreichen, entschlossen wir uns, eine weitere API-Schicht über die Oracle Scheduler API zu legen. Diese API stellt nur die Funktionen bereit, die benötigt werden und verbirgt sämtliche zusätzliche Komplexität.

Zur Kontrolle des weitgehend automatisch- und ereignisgesteuerten Ablaufs von mehreren hundert Jobs täglich wird ein leistungsfähiger Job-Browser benötigt, der sich vorzugsweise als Web-Anwendung in die bestehende IV-Infrastruktur integriert. Hauptmerkmal des Job-Browsers sind die Darstellung von hierarchischen Job-Ablaufketten und die Möglichkeit, Störungen im Ablauf der Jobketten frühzeitig zu erkennen und bearbeiten zu können. Werkzeuge hierfür sind die Darstellung von Ablaufprotokollen, die Filterung von Jobs mittels dynamischer Jobfilter und die Möglichkeit der administrativen Ressourcensteuerung mittels Job-Queues. Es muss beispielsweise möglich sein Wartungsfenster administrativ auch kurzfristig so einzuplanen, dass der Betriebsablauf so wenig wie möglich gestört wird.

Das dbms_scheduler Package der Oracle-Datenbank: was kann es und was kann es nicht?

Mit der Oracle Datenbank Version 10g wird eine weitgehend überarbeitete API für die in die Datenbank eingebauten Scheduling-Funktionen seitens Oracle bereitgestellt. Diese API, das dbms_scheduler Package, ersetzt das alte dbms_jobs Package und stellt aufgrund seiner vielen Erweiterungen ein stabiles Fundament für einen Job Scheduler dar.
Allerdings ist die Bedienung der API aufgrund ihrer Vielfältigkeit nicht ganz einfach und damit die Umsetzung für einen Job Scheduler, so wie er in unserem Projekt benötigt wird, relativ aufwendig. Des Weiteren fehlen Funktionen, die eine einfache Hierarchisierung der Jobabläufe ermöglichen. Die Umsetzung von Job-Queues, so wie diese benötigt werden, kann nicht mit der zur Verfügung stehenden API alleine umgesetzt werden. Ebenso erweist sich das im Enterprise Manager bereitgestellte Web-Interface für die Anforderungen des Kunden als nicht brauchbar. Vielmehr wird eine speziell auf das Job Scheduling zugeschnittene Arbeitsoberfläche benötigt, die einfach und intuitiv bedienbar ist und alle notwendigen Arbeitsfunktionen in Bezug auf die administrative Jobsteuerung und Überwachung bereitstellt.

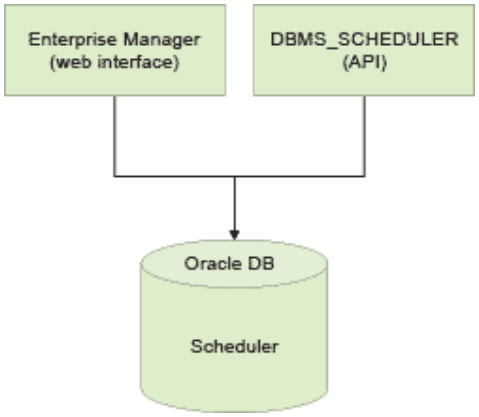

Abb. 1: Schematische Darstellung der Bedienung des Oracle dbms_scheduler

Die grundlegende Struktur des Oracle-dbms_scheduler Package kann man sich vereinfacht so vorstellen: Aufgaben werden in Form von Programmen als Datenbankobjekt gespeichert, dem Programm zugeordnet wird sein „Schedule“, der ebenfalls in der Datenbank gespeichert wird. Ein „Schedule“ enthält die Information, wann und wie ein Job die im Programm definierten Aufgaben, ausführen soll. Der Job kann auch als Instanz eines Programms angesehen werden. Für jedes Programm können Argumente definiert werden, über die dem Job Informationen zur Laufzeit

mitgegeben werden. Schematisch lassen sich die Beziehungen eines Jobs des Oracle Scheduler wie in der Abbildung 2 gezeigt, darstellen.

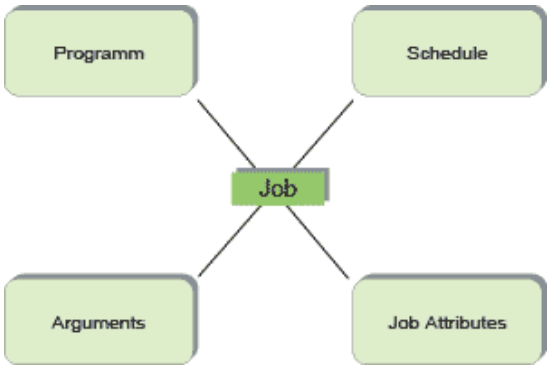

Abb. 2: Darstellung eines Oracle-dbms_scheduler-Jobs mit seinen Beziehungen

Wie die Aufgabe, die ein Job ausführen soll, formuliert werden muss, wird durch den Parameter „program_type“ im Befehl „create_program“ definiert. Ein einfacher „program_type“, um bestehende PL/SQL-Prozeduren auszuführen, ist die „STORED_PROCEDURE“. Das bedeutet, im Programm wird lediglich die „program_action“ mit der Referenz auf ein weiteres Datenbankobjekt, der „Stored Procedure“ hinterlegt. Die Stored Procedure kann nun beliebigen PLSQL-Code enthalten, der die auszuführende Aufgabe hinreichend beschreibt und mit dem Job wird diese Aufgabe dann ausgeführt.

Oracles Job-Chains versus Job-Hierachie?
Oracle verwaltet die Jobs mit seinen Laufzeitinformationen in zentralen System-Tabellen und stellt mit seinen ALL_SCHEDULER-Views Sichten auf die Programme und Jobs zur Verfügung.

Es zeigte sich jedoch, dass für ein Job-Scheduling wesentliche Mechanismen fehlen oder nur schwer zu realisieren sind. Insbesondere lassen sich hierarchische Beziehungen von Jobs, das heißt die dynamische Aufrufgeschichte, wie sie zur Laufzeit ausgeführt wurde, nicht einfach abbilden. Mit dem im Oracle Scheduler bekannten Job-Chains lassen sich nur statische Ablaufketten abbilden. Dieses reicht für unseren Anwendungsfall, wie oben kurz begründet, jedoch nicht aus. Folgende zusätzliche Informationen zu den Jobs müssen verwaltet werden:

- Erweiterte Job-Zustandsinformationen (Job-Status)
- Erweiterte Parameter zum automatischen Löschen von bereits ausgeführten Jobs, den Keep-Parametern, die zeitlich oder mengenmäßig definiert werden sollen
- Erweiterte Protokollierungsarten und Speicherung aller Protokolle in der Datenbank
- Eindeutige Referenz zur Jobinstanz. Job Namen sind einzigartig mit einem Ablauf eines Jobs verbunden.
- Ablauf von synchronen und asynchronen Jobs
- Anwendungsberechtigungen müssen auch für Jobs gelten
- Job-Ressourcenverwaltung mittels Job-Queues

Aufbau eines effektiven Job-Scheduler: unser Lösungsansatz

Aus der Analyse des zuvor durch ein Drittprodukt bereit gestellten Job-Schedulings ergaben sich die wesentlichen Anforderungen an die zu implementierenden Job-Scheduling-Funktionen, die hier kurz zusammengefasst sind:

- Jobketten müssen hierarchisch verwaltet werden können, d.h. eine einfache Eltern-Kind-Beziehung der Jobs muss repräsentiert werden. Diese Hierarchien müssen vom Job-Browser entsprechend dargestellt werden können, wobei Hierarchiebäume nur bei Bedarf aufgeklappt dargestellt werden sollen.
- Jobs werden immer als einmalige Jobinstanz gesehen, damit sämtliche Laufzeitprotokolle und Ausführungsprotokolle eindeutig einem Ablauf zugeordnet werden können und so lange wie gewünscht erhalten bleiben.
- Ein Mechanismus zum automatischen Löschen nicht mehr benötigter Jobinformationen beendeter Jobs muss implementiert werden, wobei beispielsweise fehlerhaft beendete Jobs zur Fehlerbearbeitung erhalten bleiben sollen, also nicht automatisch gelöscht werden dürfen.
- Die Job-Ressourcen-Steuerung soll durch Job-Queues erfolgen, so dass Jobs deklarativ zu Gruppen zusammengefasst werden können und hierdurch eine zentrale Ressourcen-Steuerung und Administration erfolgen kann.
- Differenzierte Anzeige von Jobzuständen, wie beispielsweise der Status, den wartende Jobs aufgrund der Job-Queue Verfügbarkeit annehmen.

Hieraus wurde eine Softwarestruktur erarbeitet, die einerseits das Oracle Dbms_Scheduler-Package mit den Oracle Job-Daten und dem Qracle Streams AQ als Basisfunktionalität nutzt, die notwendigen Erweiterungen an Metadaten und Funktionen mittels einer übergeordneten Job-View und einer funktionalen API-Schicht (Job-Scheduler-API) verbindet, so dass eine einfache, leicht bedienbare API und Datenschnittstelle (Job-View) für die Anwendungsprogrammierung bereit steht.

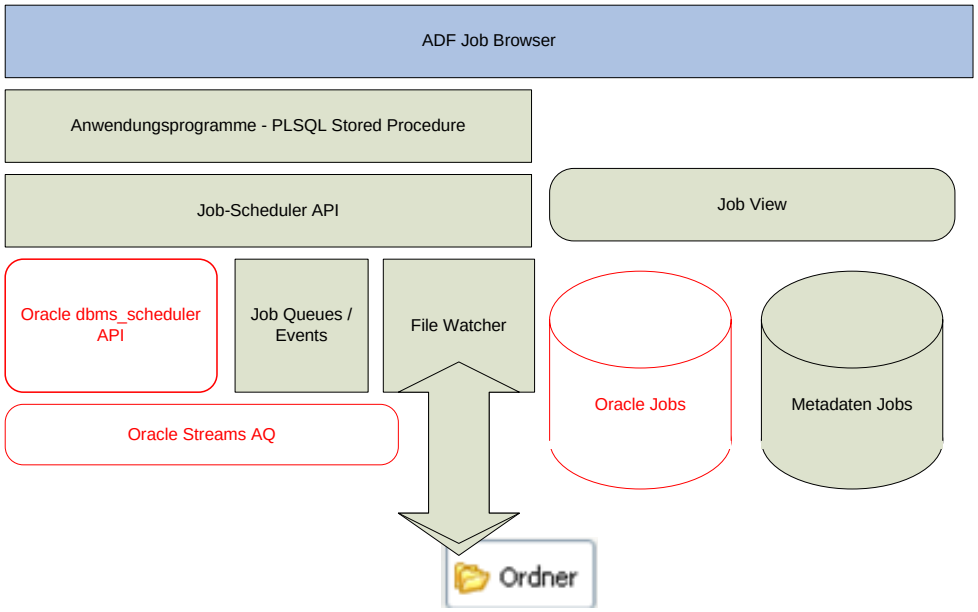

Abb. 3: Schematische Darstellung der Job-Scheduler Software Schichten

Job-Queues realisiert mit Oracle Streams Advanced Queueing (AQ)

Das Oracle Streams AQ ist ein in die Datenbank eingebetteter asynchroner Mechanismus zur Nachrichtenübermittlung, mit dem über die Datenbank Informationen von einem Producer gesendet und von einem oder mehreren Consumern gelesen werden. Für die Nachricht steht eine frei

definierbare Nachrichtenstruktur zur Verfügung. Dieser Mechanismus wird zur Interprozesskommunikation vom Oracle Scheduler genutzt. Es liegt daher nahe, diesen Mechanismus auch für die Implementierung der Job-Queues zu nutzen. In unserem Beispiel geschieht dieses im Prinzip in folgender Weise:

Basis aller Job-Queues ist eine Queue-Table erzeugt mit

```
Dbms_aqadm.create_queue_table(queue_table_name,payload,..)
```

wobei mit payload der Datentyp der Nachricht definiert wird. Es kann ein DB interner Typ, aber auch ein anwendungsspezifischer Typ sein. Anschließend wird eine Message-Queue definiert und angegeben, wer Nachrichten daraus konsumieren darf:

```
Dbms_aqadm.create_queue(queue_name,queue_table_name,..)
Dbms_aqadm.add_subscriber(queue_name,consumer)
```

Um die Anzahl der Message-Queues gering zu halten, haben wir nicht für jede Job-Queue eine eigene Queue eingerichtet, sondern nur eine einzige Queue und die Information über die Zugehörigkeit zur Job-Queue in der Nachricht selbst abgelegt.

Eine Nachricht in der Message-Queue repräsentiert ein „Runtoken“ für eine unserer Job-Queues. Die Anzahl der Runtokens einer Job-Queue definiert, wie viele Jobs, die dieser Job-Queue zugeordnet sind, gleichzeitig laufen können. Ein Job fordert nach dem Start zunächst ein „Runtoken“ der Job-Queue an. Ist eins vorhanden, wird es konsumiert und am Job-Ende wieder eingestellt. Ist keins vorhanden, wartet der Job, bis ein „Runtoken“ von einem anderen Job zurückgestellt wird.

Das Anfordern des Tokens erfolgt über ein PLSQL-Codestück (mit einem `Dbms_aq.dequeue(WAIT)`) vor der eigentlichen Verarbeitungsroutine. Aber wie stellt man sicher, dass der Job nach seinem Ende, also auch im Fehlerfall oder Abbruch durch Runterfahren der Datenbank sein Token zurückgibt?

Den Ansatz mit Event-basierten Jobs, wie sie in Version 10gR2 eingeführt wurden, verwarfen wir zugunsten einer Lösung, die sich besser in unsere Architektur einpasste. Hierbei nutzt man aus, dass der DB-Scheduler die Job-Events wie JOB_SUCCEEDED, JOB_FAILED etc. auch über eine Message-Queue signalisiert. Über eine Callback-Prozedur auf dieser System-Event-Queue kann man dann in jedem Fall das Job-Ende erkennen und das vorher konsumierte „Runtoken“ wieder einstellen.

```
Dbms_scheduler.add_event_queue_subscriber(..)
Dbms_aq.register( sys.scheduler$_event_queue,
        'plsql://pdv.notify?pr=1',..)
```

Was passiert aber, wenn sehr viele Jobs in einer Job-Queue gleichzeitig gestartet werden, die Anzahl gleichzeitig laufender Jobs stark begrenzt ist?
Dann würden alle Jobs erstmal starten, d.h. auch jeweils einen Prozess erzeugen, und auf Runtoken warten. Damit wäre der Server mit wartenden Jobs ausgelastet.

Um die Anzahl der wartenden Jobs und damit auch der laufenden Prozesse zu minimieren, muss man sicherstellen, dass ein startender Job, der ein „Runtoken“ anfordern will auch eines erhält.

Jobs starten nur, wenn sie auch aktiviert (`Dbms_scheduler.enable(job)`) sind. Das bedeutet, wenn man den Job-Zustand Enabled/Disabled in Verbindung setzt zur Anforderung eines Runtokens, kann man erreichen, dass nur lauffähige Jobs tatsächlich aktiv sind und einen Prozess erzeugen.

Ein einfaches Beispiel soll diese Situation veranschaulichen:
100 Jobs werden gleichzeitig aufgesetzt in einer Job-Queue, die aus Performancegründen den Ablauf von maximal 10 Jobs parallel zulässt. Nach dem Erzeugen der Jobs werden anschließend aber nur 10 tatsächlich auch aktiviert, die restlichen verbleiben inaktiv. Sobald ein Job sich beendet, stellt er sein „Runtoken" wieder in die Job-Queue und prüft die Job-Queue auf Jobs im Zustand inaktiv. Existiert ein Job, wird er aktiviert. Damit weist die Datenbank maximal 10 Prozesse für diese Job-Queue auf.

Die Verarbeitung von Ereignissen – Filewatcher reloaded

Das neue Oracle Scheduler-Objekt des Filewatchers wurde erst mit der Version 11gR2 eingeführt, aber die Notwendigkeit der Überwachung von Verzeichnissen bestand schon vorher. So auch in diesem Projekt, wobei es galt, den Mechanismus zur Erfassung und Verarbeitung von File-Events – ausgelöst durch einen Filewatcher – und anwendungsspezifischen Events zu vereinheitlichen. Beim „Filewatcher" bestand zudem eine Anforderung darin, übertragene Dateien richtig zu erfassen, deren Zeitstempel weit in der Vergangenheit lag (kein „touch" beim Transfer). Zudem sollte bei einem späteren Upgrade auf Version 11gR2 eine einfache Umstellung auf den darin enthaltenen Filewatcher möglich sein. Auch hier bot sich als Lösung an, die Mittel des Advanced Queuing zu nutzen.

Das Auftreten von Events wird über Message-Queues signalisiert. Im Falle des Filewatchers wird über eine Callback-Prozedur der Message-Queue ein Verarbeitungsjob gestartet, der über die Definition des „Filewatchers" das Programm zur Verarbeitung identifiziert.

Ein Beispiel für die Verarbeitung von Ereignissen ist die Synchronisierung von komplexen Job-Ablauffolgen, deren Abbildung durch Oracle Job-Chains nicht oder nur aufwändig realisierbar ist.

In einem einfachen Fall startet ein laufender Job einen weiteren und soll auf dessen Ende warten, bevor er seine Ausführung fortsetzt. Damit er auf das richtige Job-Ende reagieren kann, muss der aufrufende Job dem aufgerufenen mitgeben, welches Synchronisations-Event er erheben soll.

Der Job-Browser als ADF Client – Umsetzung und einige Erfahrungen

Neben einer einfachen API zur Implementierung und Steuerung von Jobs aus der Programmierung heraus, stand in diesem Projekt auch die Realisierung einer intuitiven, modernen und gleichsam leistungsfähigen Benutzeroberfläche im Vordergrund. Diese sollte so schwach wie möglich mit den speziellen Datenbankfunktionen des Oracle Schedulers und seinen Erweiterungen gebunden werden und so leicht austauschbar bleiben.

Als Schnittstelle zur Datenbank fungiert ein View-Objekt, das alle benötigten Daten aus den speziellen `SYS.DBA_SCHEULER_Views` mit den Erweiterungen zusammenführt. Besonderes Augenmerk galt hier der Performance dieser Schnittstellen-View. Ein weiteres Merkmal für eine effiziente Job-Überwachung und Bearbeitung stellen Filterfunktionen auf die Schnittstellen-View dar.
Die Filter werden als einfacher String in einer Datenbank-Tabelle abgelegt und können über einen Funktionsaufruf als Where-Condition vom Client-Modul direkt angewendet werden. Hierdurch lässt sich eine weitgehende Unabhängigkeit des Job-Client von komplexen Datenbankfunktionen erreichen.

Als primäre Entwicklungsplattform haben wir uns für das Oracle Application Developer Framework (ADF) 11g entschieden (Rich Faces + Business Components).
Ein Prototypansatz mit unterschiedlichen Technologien, wie z.B. Apex 4.0, zeigte, dass die Anforderungen, insbesondere an die hierarchische Darstellung der Jobs, mit ADF am produktivsten umsetzbar sind. Dabei profitiert man von den zahlreichen built-in -Funktionalitäten des ADF. Eine der wesentlichen Komponenten für dieses Projekt ist der sogenannte Tree-Table, da diese den Mittelpunkt der meisten Benutzer-Interaktionen darstellt.

Das AD Framework bietet darüber hinaus samt der Datenbankanbindung die weitestgehende Unterstützung und die modernste Umsetzung, was sich auch in der vorgelagerten Prototyp-Phase positiv gezeigt hat. Mit ADF war es möglich ein realistisches Layoutmodell zu erstellen und anhand dessen konnten die Einzelheiten mit den Anwendern leicht abgestimmt werden. Hierdurch wurde von Anfang an eine hohe Akzeptanz erreicht.
Des Weiteren kam die „User Customization“ von ADF zum Einsatz, die es ohne großen Programmieraufwand erlaubt, dass der Endanwender die Oberfläche nach seinen Bedürfnissen, z.B. welche Spalten angezeigt werden sollen, konfigurieren kann. Durch Einschalten des MDS-Repository werden die Benutzerkonfigurationen automatisch für jeden Benutzer abgelegt.

Der Einsatz von „Regions“ und „Task-Flows“ erlaubt die Umsetzung eines Single-Window-Konzeptes sowie modifizierbarer Wizards für das Einplanen, Ändern und Wiederaufsetzen von Jobs. Die PL/SQL-API wurde über den im JDeveloper enthaltenden JPublisher in Java-Klassen gewandelt und in das Application-Model importiert. Eine Besonderheit liegt darin, dass vor dem Aufruf der API-Funktionen eine Proxy-Connection an die Datenbank durchgeführt wird, um die PL/SQL API unter individueller Benutzerkennung ausführen zu können. Dieses ist insbesondere beim Starten und Wiederaufsetzen von Jobs notwendig, da dieses unter eigener oder sogar unter der Benutzerkennung eines anderen Benutzers geschehen muss. Das Berechtigungskonzept der Anwendung darf in keinem Fall durch die Jobs aufgebrochen werden.

Neben vielen Vorteilen zeigt auch ADF, wie jedes andere Framework, seine Schwächen. So verlässt es sich bei der Authentifikation auf den J2EE-Container, sprich dem Oracle Weblogic-Server. Dieser bietet leider keine Möglichkeit eine Authentifikation gegen die Datenbank-User durchzuführen. So muss man hier einen eigenen Login-Provider schreiben, welches schlecht dokumentiert und bei weitem komplizierter ist, als der einfache JAAS Provider, der im OC4J noch ausreichte.

Viele kleine oder größere Bugs führen schnell zu länglichen Workarounds. So werden z.B. Fehlermeldungen durch die Komponente, die das automatische Auffrischen der Anzeige ermöglichen soll, mit dem Auffrischen ebenfalls automatisch geschlossen. Die Lösung, die dem Entwickler hier bleibt, ist ein tiefgreifender Eingriff in das vorhandene ADF-Messaging-Konzept.
Ein weiteres Beispiel ist die mit Version 11.1.1.3 hinzugekommene Funktion einen „Task-Flow“ als Popup zu definieren, welches den in vorherigen Versionen mühsamen Umweg über die Verwendung von dynamische Regionen und entsprechenden Managed Beans erheblich vereinfacht. Leider führt dieses neue Konzept zu einer spontanen und anhaltenden 100% -igen CPU-Auslastung des Weblogic-Servers, so dass wir auch hier wieder auf die „herkömmliche“ Art der Implementierung ausweichen mussten.
Dieses sind nur einige wenige Beispiele, die den ansonsten positiven Eindruck etwas trüben. Oracle sei angeraten, sich in den nächsten Versionen dringend vermehrt dem Bug-Fixing zu widmen.

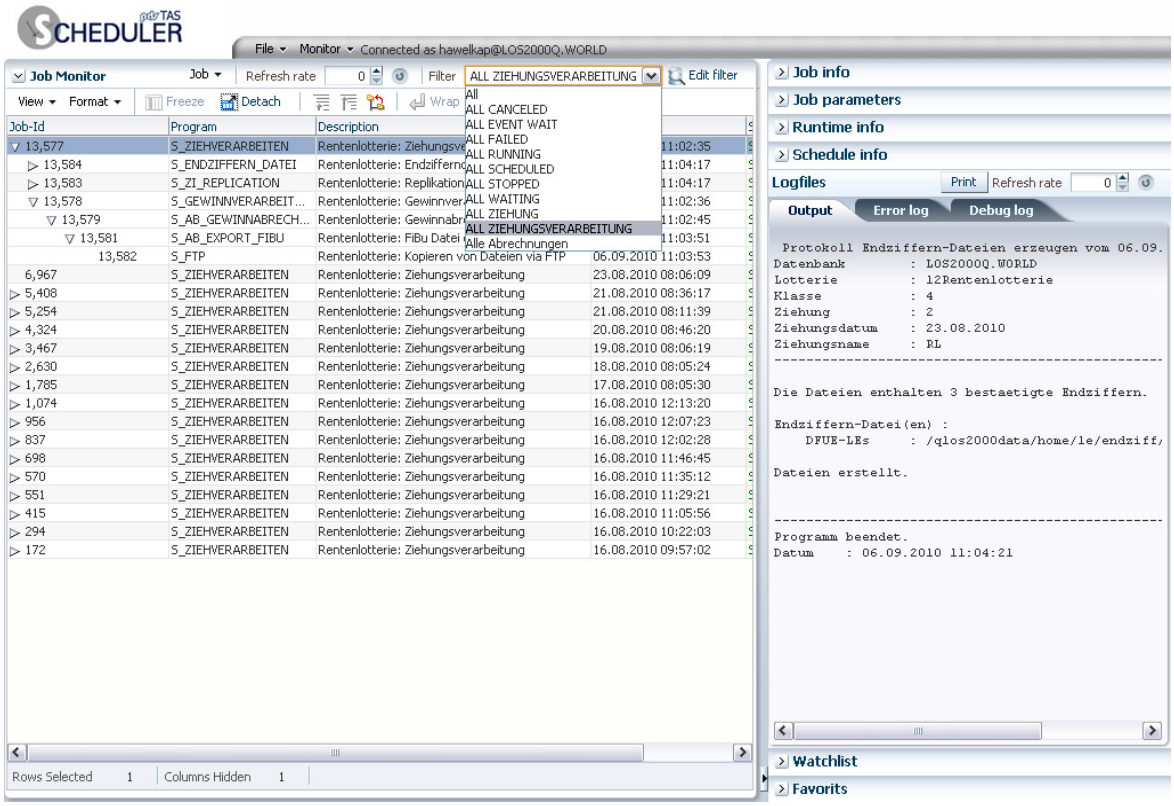

Abb. 4: Jobmonitor der geplanten und ausgeführten Jobs in einer hierarchischen Darstellung

Als Ergebnis bewährt sich der ADF-Job-Browser als performantes Arbeitswerkzeug, mit dem ein schnelles und übersichtliches Arbeiten ermöglicht wird und eine hohe Benutzerakzeptanz erreicht wird. Das vorgestellte Schnittestellenkonzept erlaubt aber auch eine Erstellung des Jobs-Browsers auf einer anderen technologischen Basis, wie APEX, .NET, u.a. wenn dieses in einem speziellen Projektumfeld notwendig oder gewünscht wird.

Kontaktadresse:

Dr. Peter Hawelka
pdv Technische Automation + Systeme GmbH
Dorotheenstraße, 64
D-22301 Hamburg

Telefon: +49 (0) 40 69213-266
Fax: +49 (0) 40 69213-278
E-Mail hawelka@pdv-tas.de
Internet: www.pdv-tas.de