Anleitung für pit-FM Anwendungsentwickler

Geschwindigkeit optimieren

#### Metadaten des Dokuments

*Erzeugt: 12.10.2012*

*Bearbeiter: Hartmut Fischer*

*Status: erster Entwurf*

*Anregungen, Wünsche, Hinweise auf Fehler usw. sind ausdrücklich erwünscht.*

Einleitung

Nachdem ein Workflow bzw. eine Funktion implementiert wurde und korrekt läuft, sollte das Augenmerk auch auf die Ausführungsgeschwindigkeit gelegt werden und einige Test mit genügend großen Datenmengen durchgeführt werden.

Normalerweise achtet man bei der Implementierung primär auf die Korrektheit und nicht so sehr auf die Geschwindigkeit. Ist im ersten Schritt auch das richtige Vorgehen, aber der zweite Schritt, die Optimierung, darf nicht auf der Strecke bleiben.

In Version 16.0.0.x wurden einige existierende Funktionen optimiert und neue Funktionen implementiert, die die Geschwindigkeit bei bestimmten Operationen im Kern erhöhen und die es dem Anwendungsentwickler erlauben, die Performance seiner Klassenformeln zu steigern.

Bei Klassenformeln leidet die Performance oft durch die hohe Anzahl an Datenbankzugriffe.

Jeder Zugriff teilt sich in 3 Phasen auf, jede Phase benötigt mehr oder weniger viel Zeit

1. Übermitteln der SQL-Anweisung an die DB.
2. Parsen und Ausführen der Anweisung durch die DB
3. Senden des Ergebnisses an pit-FM

Die für jede Phase benötigte Zeit hängt von verschiedenen Faktoren ab:

1. Diese Phase ist von der Anbindung an die DB abhängig, d.h. ein lokaler Server (wird beim Entwickeln häufiger verwendet) ist sehr schnell erreichbar, ein über eine langsame Leitung angebundener Server (wie er dann beim Kunden leider manchmal vorliegt) benötigt Zeiten bis zu mehreren 1/10-Sekunden. Falls damit zu rechnen ist, kann man auf seinem lokalen System eine langsame Anbindung simulieren: **Der Eintrag DEBUG\_DELAYONOPEN=100 in Sektion [DBMS …] in Datei db.ini verzögert das Öffnen eines Cursors um 100 Millisekunden (=1/10 Sekunde).**
2. Um das Parsen zu beschleunigen gibt es noch einige Möglichkeiten, die wir demnächst im Kern implementieren werden. Dies ist für Anwendungsentwickler transparent, d.h. sie brauchen nicht zu ändern, nur die Binaries auszutauschen. Die Ausführungsgeschwindigkeit (nach dem Parsen) ist sehr stark von den SQL-Anweisungen selbst und dem Optimierer der DB abhängig. Sehr langsam laufende Anfragen habe ihre Ursache manchmal in der nicht optimal eingerichteten Datenbank (keine/falsche Statistik lässt den Optimizer schlechte Strategien wählen, fehlende Indizes, zu wenig Arbeitsspeicher usw.). Ein „Tuning“ der Datenbank kann zu einer extremen Geschwindigkeitssteigerung (manchmal um Faktor 100) führen. Dies ist jedoch ein eigenes Thema und soll in diesem Dokument nicht behandelt werden. Evtl. ist jemand wie Ronny bereit, ein kleines Dokument dazu zu schreiben.
3. Große Ergebnismengen benötigen Zeit zur Übertragung. Sie sollten vermieden werden (z.B. durch Verwendung von Views).

Testen

Ein extrem wichtiges Hilfsmittel ist das SQL-Logging.

Hier werden alle (lesenden) Zugriffe auf die DB protokolliert.

Nur hier hat man die Möglichkeit zu überwachen, wie viele Anfragen an die DB gestellt werden und wie lange diese zur Ausführung und zum Übertragen des Ergebnisses benötigen.

**Es ist ein Muss für jeden Anwendungsentwickler, über das SQL-Logging die Anzahl und Zeiten der Datenbankzugriffe zu überprüfen.**

Oft ist man sehr überrascht über die Menge der Anfragen. Und hier liegt ein wichtiger Ansatz zur Optimierung. Viele Projekte laufen langsam aufgrund Tausender DB-Anfragen, bei denen jede einzelne schnell beantwortet wird, in der Summe aber erheblich Zeit benötigen.

Der Kern und die pit-FM Programmiersprache stellen ein (relativ) einfaches, aber mächtiges Klassen- und Objekt-Modell zur Verfügung, so dass ein Anwendungsentwickler auf einer abstrakteren Ebene als der von der DB-Schnittstelle zur Verfügung gestellten entwickeln kann. Dadurch sieht man oft nicht sofort, was alles und wie oft eine DB-Anfrage ausgelöst wird. Auf die wichtigsten Punkte wird in den folgenden Kapiteln eingegangen.

Die Strategie zur Geschwindigkeitsoptimierung lautet:

* Unnötige DB-Zugriffe vermeiden.
* Views zur Ermittlung von Summen usw. verwenden.
* Prefetching nutzen. D.h. Entitys, auf die später zugegriffen werden soll mit möglichst wenigen DB-Zugriffen auf einmal in den Cache laden.

DB-Zugriffe vermeiden

## Entity-IDs anstelle von Entitys abfragen: das Schlüsselwort „raw“

Beispiel: Der Ausdruck if (:ref\_A == :ref\_B) wird zwei DB-Anfragen auslösen, falls :ref\_A und :ref\_B noch nicht im Cache liegen und nicht NULL sind. Grund ist, dass hier Referenz-Attribute (die nach dem Lesen aus der Datenbank Entity-IDs enthalten) dereferenziert werden, d.h. es wird das zurückgegeben, wohin sie zeigen. Das ist immer ein Entity oder NULL. Für die obige Bedingung würde es jedoch genügen nur die IDs zu vergleichen. Man muss dem Compiler also klar machen, dass man auf die ID anstelle des referenzierten Entity zugreifen will.

Dazu gibt es das Schlüsselwort „raw“. Der Ausdruck if (raw :ref\_A == raw :ref\_B) vergleicht nur die IDs und lädt nicht die Entitys, das Ergebnis ist identisch aber zwei DB-Anfragen wurden vermieden.

**Regel:**

**Beim Vergleich von Referenz-Attributen mit anderen Referenz-Attributen oder NULL immer nur die IDs vergleichen**

ref\_A op ref\_B 🡪 raw ref\_A op raw ref\_B

ref\_A op NULL 🡪 raw ref\_A op NULL

Bei einem Vergleich mit einem bestimmten Entity, dessen ID bekannt und ist gibt es auch hier eine optimierte Variante.

Anstelle von

if ( :ref\_A == FindEntity( „State“, „00123456:00001234“ ))

sollte der Ausdruck

if ( raw :ref\_A == #00123456:00001234)

verwendet werden.

#00123456:00001234 ist der literale Ausdruck für die ID.

Das letzte Beispiel hinkt etwas, da wir die Verwendung von fixen IDs im Code nicht empfehlen können. Aber folgendes Programmfragment könnte sinnvoll sein.

var entityid $id;

var string $strId;

$strId = …; // als String gemerkte Id holen

$id = StrToEntityId( $strId ); // String in Id konvertieren

if ( raw :ref\_A == $id )

Views verwenden

Manchmal ist es günstig, anstelle die Entitys einzulesen und Summen usw. selbst zu berechnen, dies der DB zu überlassen. Die Standard-Technik bei pit-FM dazu ist das Erstellen eines Views, der genau diese Berechnungen durchführt. Dieser wird als eigene Klasse in pit-FM definiert, die Anbindung an das „Haupt-Entity“ geschieht über ein temporäres Referenz-Attribut (sollte jeder Anpassungsentwickler kennen).

Anzahl DB-Zugriffe verringern

## Prefetching

Definition aus Wikipedia:

Als **Prefetching** (englisch, dt. ‚vorheriges Abrufen (von Befehlen)‘) bezeichnet man in der Informatik das heuristische Laden von Speicherinhalten aufwärts in der Speicherhierarchie, bevor ein Bedarf evident geworden ist, um so im Falle des tatsächlich eintretenden Bedarfs eine höhere Zugriffsgeschwindigkeit zu erzielen.

Im Kontext von pit-FM versteht man unter Prefetching das Laden erst später benötigter Entitys in den Cache, wobei die Möglichkeit ausgenutzt wird mit wenigen DB-Anfragen möglichst viele Entitys zu laden.

*Beispiel:*

 *$el enthält 200 Entitys.*

 *ref\_A und ref\_B verweist jeweils auf ein Entity, diese haben insgesamt 10 verschiedenen Klassen, die von der in ref\_A bzw. ref\_B angegebenen Klasse abweichen.*

*Die Funktionen DoSomething und DoSomethingWithB*

for each $e in $el

{

 DoSomething( $e.ref\_A.Preis);

 DoSomethingWithB( $e.ref\_A.ref\_B );

}

Dieses Codefragment verursacht im schlimmsten Fall um die 800 DB-Zugriffe.

Falls die Entitys in $el noch nicht vollständig im Cache sind, greift hier ein internes Prefetching des Kerns. Beim Zugriff auf das erste Entity aus $el werden automatisch weitere 99 folgende Entitys in den Cache geladen. Dies ist eine Heuristik, da der Kern annimmt, dass die Schleife mehrfach durchlaufen wird und nicht nach dem ersten Entity mit break oder return verlassen wird. Bei 200 Entitys sind es somit zwei DB-Zugriffe für die Entitys aus $el, die wir hier vernachlässigen können.

$e.ref\_A dereferenziert ein Entity, d.h. ein DB-Zugriff pro Entity aus $el, macht zusammen 200.

Da die bei ref\_A angegebene Klasse (z.B. Verwaltungsobjekt) von der Klasse des referenzierten Entity abweicht (z.B. techn. Gerät), muss der Kern die restlichen Daten nachladen. Ein weiterer DB-Zugriff, macht schon 400.

$e.ref\_A.ref\_B dereferenziert zwei Entitys, da aber $e.ref\_A bereits im Cache liegt, gibt’s nur einen weiteren DB-Zugriff für ref\_B plus einen fürs nachladen bei abweichender Klasse, zusammen wieder 400.

Wenn der Kern von den zukünftigen Zugriffen wüsste, könnte er zuvor alle benötigten Entitys mit 3 DB-Zugriffen einlesen plus ein Zugriff für jede Klasse, bei der das eingelesen Objekt von der im Referenz-Attribut angegebenen Klasse abweicht. Im Ergebnis ergeben sich somit ca. 800 DB-Zugriffe ohne Optimierung, 13 DB-Zugriffe mit Optimierung. Da mehr Daten geladen werden dauern die 13 Zugriffe etwas länger, so dass sich in der Praxis eine um ca. Faktor 20-50 erhöhte Geschwindigkeit feststellen lässt.

D.h. anstelle von z.B. 20 Sekunden sind es jetzt weniger als 1 Sekunde !!!

Die Schleife selbst benötigt dann keine weiteren DB-Zugriffe mehr (falls Fkt. DoSomething auch keine weiteren auslöst).

Leider können wir den Compiler nicht so schlau machen, dass er solche Situationen selbst erkennen und die Zugriffe optimieren kann. Aber wir stellen Funktionen bereit, mit denen der Entwickler das Prefetching initiieren kann.

Die wichtigste Funktion ist hier LoadReferencedEntities, verfügbar seit Version 16.0.0.4146. Ihre Argumente sind die EntityListe und alle Referenzen (als Strings). Der Kern sorgt dafür, dass alle Entitys in $el sowie alle über die angegebenen Referenzen erreichbaren Entitys in den Cache geladen werden.

Im gegebenen Beispiel würde folgende Zeile vor der Schleife zu der Geschwindigkeitssteigerung führen: LoadReferencedEntities( $el, „ref\_A“, „ref\_A.ref\_B“ );

**Regel:**

**Wird eine Entityliste durchlaufen und in der Schleife über Referenzattribute auf Entitys zugegriffen (direkt/über OnInit-Trigger temporärer Attribute/über Funktionsaufrufe), muss für diese Entitys vor dem Durchlauf ein Prefetch erfolgen (mittels Fkt. LoadReferencedEntities).**

## Optimierte Funktionen des Kern verwenden

Seit Version 16.0.0.4172 ist die optimierte Delete-Funktion verfügbar, die jetzt auch alle Entitys einer EntityListe optimiert löschen kann.

Das Löschen von von pit-FM erfolgt in zwei Phasen, der Prefetch-Phase und der Delete-Phase.

Die Prefetch-Phase liest die durch den Löschvorgang zu löschende bzw. zu ändernde Entitys ein, die durch die eingestellte Delete-Semantik definiert sind. In dieser Phase wird noch kein OnDeleting-Trigger gefeuert. Das Prefetching hier erfolgt optimiert

In der Delete-Phase werden die Entitys wie bisher (d.h. in gleicher Reihenfolge) durchlaufen und gelöscht bzw. geändert. In dieser Phase müssen keine Entitys mehr eingelesen werden (Ausnahme: durch Trigger aufgerufenen Klassenformeln könnten auf Entitys zugreifen, die noch nicht im Cache liegen). Die OnDeleting-Trigger werden aufgerufen. Falls deren Klassenformeln Entitys löschen, werden diese Delete-Aufrufe sofort und vollständig ausgeführt, wieder jeweils optimiert in 2 Phasen. Es kann also zu verschachtelten Delete-Aufrufe kommen (wie zuvor auch).

Falls eine EntityListe gelöscht werden soll, so kann das Prefetching für alle enthaltenen Entitys parallel erfolgen, d.h. die Anzahl der DB-Zugriffe ist wesentlich kleiner als beim Löschen der einzelnen Entitys.

**Regel:**

**Sind mehrere Entitys gleicher Basisklasse zu löschen, müssen diese als Array von Entitys oder als Entityliste zusammen gelöscht werden.**

Anstelle von

for each $e in $el

{

 Delete( $e);

}

Verwende

Delete( $el );

Regeln für Trigger

## OnInit

**Regel:**

**Niemals DB-Zugriffe auslösen, auch nicht indirekt über Funktionsaufrufe oder verwenden von temporären Attributen, deren OnInit( :Attribute )-Trigger wiederum DB-Zugriffe auslöst.**

Falls diese Regel nicht eingehalten wird, sind alle Optimierungsversuche die auf Prefetching basieren (LoadReferencedEntities, optimiertes Delete usw.) zum Scheitern verurteilt!

Mir fällt im Moment auch keine Situation ein, bei der ein DB-Zugriff nötig wäre. Falls es bei einem Projekt aus irgendeinem Grund anders ist, meldet euch bitte bei mir. Evtl. kann der Kern erweitert werden, um diese Situation beim Prefetching zu berücksichtigen. Dazu müssten wir sie aber erst mal sehen.

Niemals Entitys erzeugen / löschen / ändern\*.

## OnInit( :Attribute)

Niemals Entitys erzeugen / löschen / ändern\*.

Diese Regel ist eigentlich selbstverständlich, da Initialisierungs-Trigger nur lesen sollten, ich habe aber bereits Projekte gesehen, bei denen diese Regel umgangen wurde. Es ist möglich, dass solche Projekte in Zukunft nicht mehr funktionieren (nach Binärupdate, wird zuvor angekündigt und es gibt beim Projektstart eine entsprechende Fehlermeldung).

## Ondisplayfunction

Wenn möglich viele oder langsame DB-Zugriffe vermeiden, dieser Trigger wird von der Oberfläche regelmäßig und oft aufgerufen. Dauert der Aufruf zu lange, reagiert das System evtl. träge auf Benutzerinteraktionen.

Niemals Entitys erzeugen / löschen / ändern\*.

\*) Ändern eines Entity bedeutet hier es so zu ändern, dass ein Speichern in die DB erforderlich ist. Ändern von temporären Attributen oder temporären Entitys, die nicht in der DB gespeichert werden ist erlaubt.