Dr. Edgar Huckert
Mail: edgar.huckert@huckert.com
V1.0 01-2008

Bewältigung großer Treffermengen in Datenbankclients mit C#

1. Das Problem

Immer wieder wird mir von Problemen bei Client-Programmen berichtet, die SELECT Anfragen mit großen Treffermengen absetzen: die Wartezeiten bis zum Erscheinen der ersten Treffern sind unerträglich sobald ein bestimmte Trefferzahl überschritten wird. Oft berichten die Endbenutzer davon, dass die Anwendung sich "aufgehängt" habe - in Wirklichkeit dauert es nur sehr lange, bis die Treffer geladen wurden. Wenn die Zahl der Treffer gering ist, ist auch die Reaktionszeit meist in Ordnung.

Der Grund für dieses Verhalten liegt meist in einem zu einfachen Design ("naives Design") der Logik für das Ausführen eines SELECT Statements (darauf konzentrieren wir uns hier - ähnliche Probleme können allerdings auch bei langlaufenden INSERTS, UPDATE oder DELETE Statements auftreten). Sieht man sich die problematischen Anwendungen an, so zeigen sich diese beiden Probleme:

- die Datenbankanwendung arbeitet **synchron**: die Anzeige wird erst bedient, wenn das SELECT fertig ist und alle Treffer abgeliefert hat.

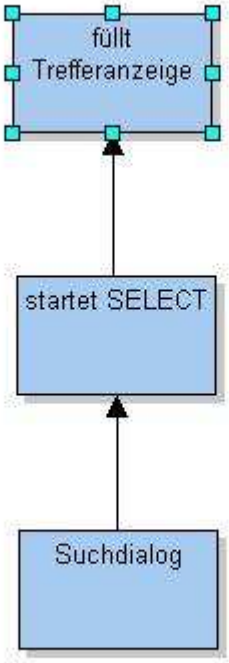
- die Treffer werden **ausschließlich im Hauptspeicher** vorgehalten. In C#- und Java-Anwendungen schlägt der Garbage Collector völlig unvorherbar zu - das Zeitverhalten ist nicht kontrollierbar. Zudem tendieren Garbage Kollektoren bisweilen dazu, sehr viel Rechenzeit zu verbrauchen, wenn der verfügbare Speicher knapp wird.

Grob gesagt hat also die weit verbreitete einfache Lösung zwei Grundfehler:

- sie arbeitet rein sequentiell

- die Zahl der Treffer ist von der Größe des verfügbaren Speichers abhängig

Die übliche naiv-sequentielle Lösung kann so veranschaulicht werden:



Ein darauf aufbauender, etwas komplexerer Ansatz könnte so aussehen:

- Absetzen eines SELECTs
- Anzeige beim Eintreffen der ersten Trefferportion
- Einfüllen der weiteren Treffer im Hintergrund in den Hauptspeicher

Diese etwas komplexere Lösung löst teilweise das **psychologische Problem** („warum warte ich so lange?"): der Benutzer sieht, dass sich etwas getan hat - zumindest bei der ersten Trefferportion. Allerdings wird das technische Hauptproblem - alle Treffer werden in den Hauptspeicher geladen - dadurch nicht gelöst. Die Zahl der möglichen Treffer ist i.A. geringer als in meiner Lösung. Vor allem aber: solche Client-Anwendungen verhalten sich **unkollegial** gegenüber anderen Anwendungen auf dem gleichen PC, da sie den gesamten Hauptspeicher auffressen. Bei rein sequentiellem Ansatz - kein Einsatz von Threads - muss der Benutzer außerdem warten, bis das SELECT beendet ist.

2. Die Lösung

Schon zu Zeiten der IBM-Host Programmierung mit 3270 Terminals oder später zur Zeit der ersten intelligenten Terminals auf Basis von 8-Bit Computern musste das wesentliche Problem gelöst werden: es passte nicht alles in den Hauptspeicher (Memory). Deshalb war es völlig natürlich, nur Portionen der anfallenden Daten in den Hauptspeicher zu laden und weitere Portionen bei Bedarf nachzuladen.

Die hier vorgestellte Lösung setzt auf die folgenden Techniken:

- Aufteilung der Aufgabe in mindestens **zwei parallele Threads:** SELECT Operation und Anzeige

- Zwischen den beiden Threads gibt es einen **Datenaustausch per Callback**-Funktion.

- Treffer werden **nicht komplett in den Hauptspeicher** geladen, sondern in eine Datei auf der Clientseite. Im Hauptspeicher befindet sich immer nur die angezeigte Menge der Treffer.

- Für das schnelle Auffinden der Treffer in der Trefferdatei wird eine einfache speicherbasierte **Trefferverwaltung** (Datei-Offsets der Portionen) eingerichtet.

Die Treffer für die Anzeige werden aus der erzeugten Trefferdatei geladen, nicht aus dem Hauptspeicher. Sobald diese Trefferdatei komplett vorliegt, kann die Datenbankverbindung geschlossen werden.

Gegenüber der Hauptspeicher-basierten Lösung können weitaus mehr Treffer angezeigt werden. Dateien können üblicherweise sehr viel größer sein als der Hauptspeicher. Zudem verhält sich meine Lösung sehr viel **kollegialer** gegenüber den anderen Anwendungen auf dem Client-PC: sie klaut nicht allen verfügbaren Hauptspeicher. Hauptspeicher wird nur benötigt für die Treffer in der Trefferanzeige (derzeit ca. 25 Zeilen) und für ein wenig Trefferverwaltung (Dateioffsets der Portionen).

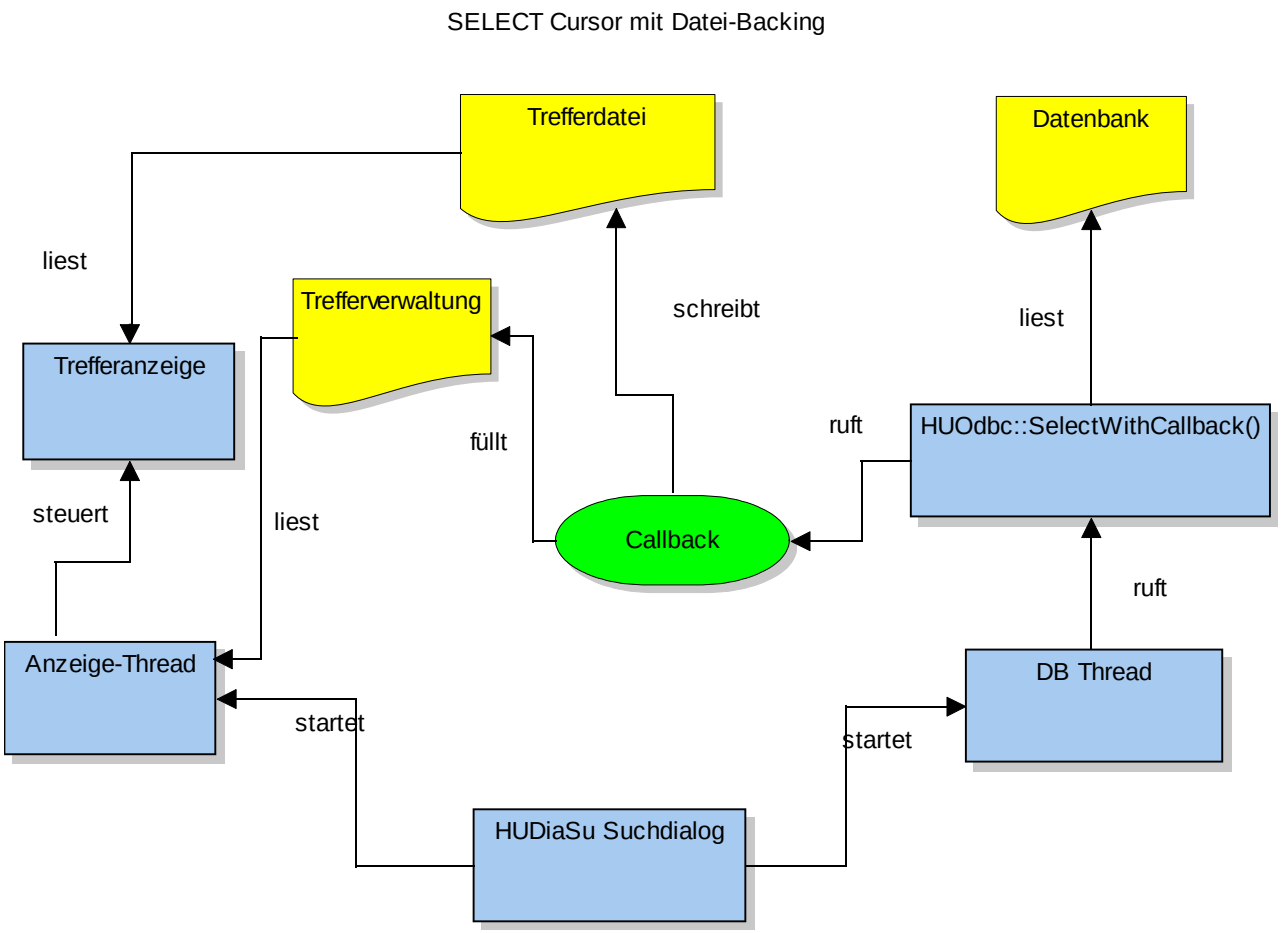
Der Benutzer kann sofort nach dem Eintreffen der ersten Portion damit anfangen, die Treffer zu analysieren. Er kann weitere Daten anzeigen (Scrollen) während im Hintergrund die Trefferdatei aufgefüllt wird. Es empfiehlt sich aus psychologischen Gründen den laufenden Hintergrundbetrieb irgenwie anzuzeigen z.B. über eine Anzeige der Zahl der bereits vorliegenden Treffer oder über eine tachoartige Fortschrittsanzeige.

Das **Bewegen innerhalb der Trefferdatei (Scrollen)** kann beschleunigt werden durch Zusatzinformationen, die beim Erstellen der Datei ohnehin anfallen. So könnte ein Array (im Beispielsprogramm wird eine *ArrayList* verwendet) hilfreich sein, der die Anfangspositionen der geladenen Portionen innerhalb der Datei festhält. Beim Blättern kann dann der Dateizeiger auf

eine solche Position gestellt werden. Ab dieser Stelle können dann die Trefferzeilen sequentiell aus der Datei gelesen werden. Auch **abrupte Sprünge** ans Ende oder an den Anfang der Treffer können über eine solch einfache Verwaltung der Trefferportionen leicht gelöst werden. Bei sehr großen Treffermenge reicht in einer solchen Verwaltung ein Eintrag für bis zu 10000 Sätze - moderne Platten sind schnell genug um selbst bei solchen Portionen keine nennenswerten Lesezeiten zu erzeugen.

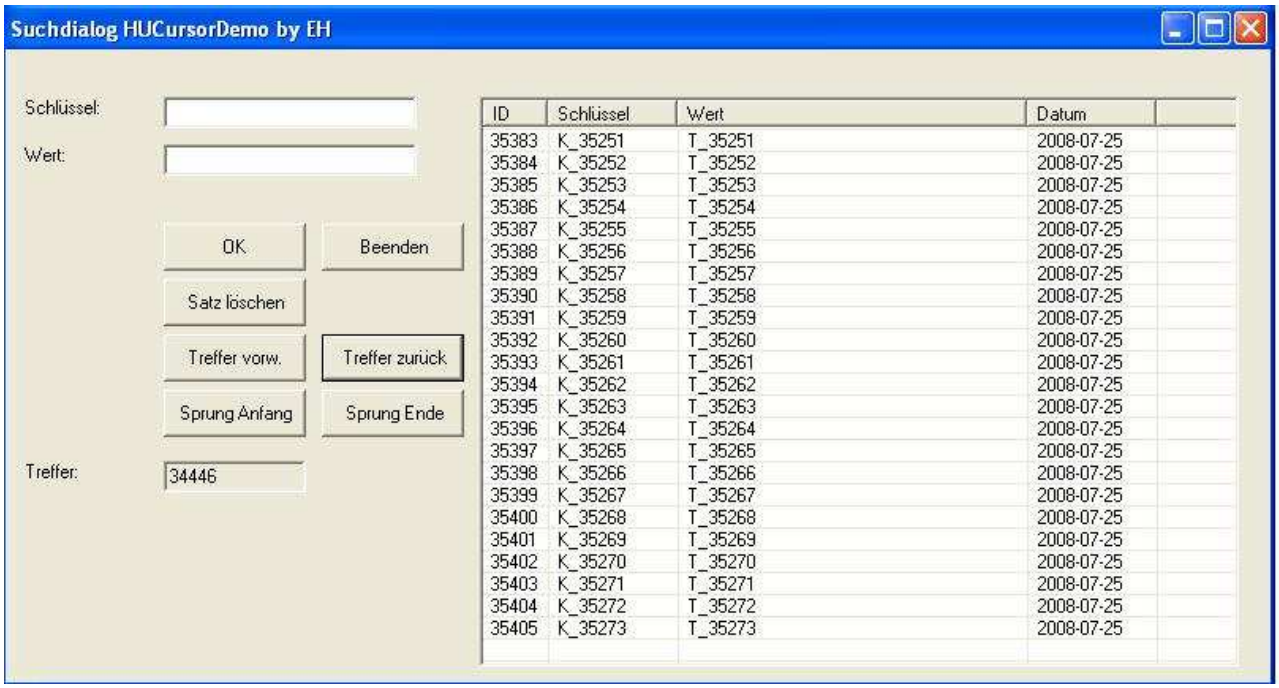
Insgesamt ähnelt die gewählte Lösung stark einem klassischen, dateiorientierten Editor (z.B *vi*). Diese Editoren halten auch nicht alles im Hauptspeicher. Sie laden die jeweils anzuzeigende Portion aus einer Datei nach und können dadurch Dateien bearbeiten, die weitaus größer sind als der Hauptspeicher. Die Analogie zum klassischen Paging in Betriebssystemen ist natürlich auch naheliegend.

Dass die hier demonstrierte Lösung komplexer ist als die naive Methode sieht man schon anhand des komplexeren Diagramms:



Das Bild zeigt nur den Hauptdialog, nicht die Rahmenanwendung.

Der Hauptdialog der Beispielsanwendung sieht graphisch (nein - kein Designpreis dafür!) so aus:



Die unteren vier Knöpfe sind für die Navigation (Scrollen) vorgesehen:

- Treffer vorwärts
- Treffer zurück
- Sprung Anfang ("abrupter Sprung")
- Sprung Ende ("abrupter Sprung")

Die Trefferanzeige unterhalb der vier Navigationsknöpfe zeigt jeweils die Anfangsposition des gezeigten Ausschnitts an. Während des Ladevorgangs wird diese Anzeige mit der jeweils geladenen Zahl der Treffer gefüllt. Dies ist psychologisch wichtig: es zeigt dem Endanwender, dass im Hintergrund etwas geschieht. Statt dieses schreibgeschützen EDIT Feldes ist auch eine Fortschrittsanzeige denkbar.

Man kann auch die (hier nicht sichtbaren) Scrollbars der Listbox (genauer: *ListView*) für die Navigation umfunktionieren- das ist hier nicht geschehen.

Natürlich gibt es auch **Nachteile** gegenüber den erwähnten naiven Methoden:

- schon das Arbeiten mit zwei parallelen Threads kann dem unerfahrenen Benutzer Problem bereiten
- Die Trefferdatei muss zwischen den Threads **geshart** werden - auch das kommt nicht jeden Tag vor
- Die Behandlung der Cursorbewegungen innerhalb der Trefferanzeige muss jetzt ausprogrammiert werden. In der naiven Methode übernimmt das jeweilige Framework dies.

Übrigens muss eine "naive" Lösung weiterhin verfügbar sein. Bei einer zu

erwartenden kleinen Trefferzahl macht es wenig Sinn, den hier beschriebenen komplexen Ansatz zu verfolgen.

3. Alternative Lösungen

Natürlich kann man sich auch andere Lösungen vorstellen. Eine alternative Lösung könnte so aussehen:

- im SELECT Statement wird eine Teilmenge der Treffer angefordert
- bei jedem Scrollen in der Anzeige wird die nächste Portion per neuem SELECT angefordert

Diese Lösung funktioniert - ist jedoch kaum portabel über unterschiedliche Datenbanksystem hinweg. Soweit ich weiß gibt es **kein portables** SQL Konstrukt, das ohne Kenntnis der Semantik der Daten ein portionsweises Laden erlauben würde. Dennoch hat eine solche alternative Lösung den Vorteil, dass sie **keine parallelen Threads** erfordert und somit einfacher im Design ist als die oben vorgestellte Lösung.

Eine weitere Lösung könnte eine Dateipufferung auf Serverseite statt auf Clientseite vorsehen. Dies könnte zu einem Cache-ähnlichen Mechanismus auf Serverseite führen (Szenario: viele Clients führen öfters das gleiche SELECT aus und die Daten ändern sich nur selten). Im Normalfall jedoch hat ein Client-programm nicht das Recht, auf Serverseite Dateien anzulegen. Für einen WEB-Ansatz (CGI oder SOA auf Serverseite) könnte diese Lösung aber interessant sein.

Eine interessante alternative Lösung könnte auch darin bestehen, das in Windows verfügbare asynchrone IO (das es in anderen Betriebsystemen wie VMS schon seit Jahrzehnten gibt) auf Datenbankanwendung anzuwenden. Asynchrone Operationen verwenden wie in diesem Beispiel Callback-Methoden. Eine solche Lösung - ich habe sie nicht weiter geprüft - macht wahrscheinlich die beiden Threads überflüssig.

4. Technisches

Das beigefügte Beispielprogramm verwendet im Kern nur drei **Klassen**:

- die Steuerungsklasse **HUCursorDemo**
- die Datenbankklasse **HUOdbc**.
- der Suchdialog **HUDiaSu -** enthält auch den größten Teil der Logik

Die Beispielsanwendung ist **nicht** für beliebige SELECTs konfigurierbar: dazu müsste etwas mehr Aufwand getrieben werden. Immerhin können einige Parameter wie Tabellenname, Name der Spalten etc. in der Deklaration der Klasse geändert werden.

Die Datenbankklasse *HUOdbc* ist "abgestrippt": sie enthält im Wesentlichen nur die hier benötigten Funktionen für SELECT. Eine naive Implementierung (sie ist für geringe Trefferzahlen weiterhin sinnvoll!) ist ebenfalls enthalten.

Die Aufteilung der Threads folgt nicht ganz dem Klassenschema: die **Callback-Methode** *CallbackMethod()* gehört dynamisch zum Datenbank-Thread, obwohl sie in der Klasse HUCursorDemo angesiedelt ist.

In der Klasse *HUDiaSu* muss das **Sharen der Datei** beachtet werden. Da wir mit paralellen Threads arbeiten, die die auf die gleiche Datei zugreifen, muss die Datei explizit für gesharten Zugriff geöffnet werden - sonst sind Exceptions vorprogrammiert. Weiterhin ist darauf zu achten, dass die Trefferdatei auf Byteebene ansteuerbar sein muss. Zu den einfachen Dateioperationen wie *ReadLine()* kommen weniger bekannte Operationen wie *Seek()* hinzu.

In der Steuerungsklasse *HUCursorDemo* können einige Parameter eingestellt werden - z.B. die Größe in Zeilen der anzuzeigenden Portion oder aber die maximal erlaubte Trefferzahl. Auch in dieser Lösung ist es natürlich sinnvoll, die Zahl der Treffer nach oben zu begrenzen.

Diese Beispielsanwendung - und insbesondere die Klasse *HUOdbc* - wurde für Client-Zugriff über **ODBC** geschrieben. Ich sehe keinen Grund, weshalb der gleiche Ansatz mit leichten Modifikationen nicht mit anderen Zugriffskonzepten wie ADO etc. laufen sollte. Getestet habe ich die Anwendung gegen MSAccess/Jet DLL und gegen Oracle. 2.5 Millionen Treffer konnten locker verarbeitet werden.

Der C#-Quellcode für das Beispielsprogramm *HUCursorDemo* kann als ZIP vom gleichen Speicherort wie dieses Dokument heruntergeladen werden. Das Projekt kann **ohne IDE** (Visual C# o.ä.) kompiliert und gestartet werden. Voraussetzung ist das .NET Framework in mindestens der Version 2.0. Die Kompilation geschieht auf der Kommandoebene ("schwarzes Fenster"):

```
msbuild hucursordemo.proj
```