Performance-Probleme vermeiden

Faustregeln

1. Wenn Sie **große Tabellen** haben, **filtern** Sie bitte die erste Ansicht so, dass nur ein kleiner Teil der gesamten Tabelle angezeigt wird, also nur die wirklich notwendigen Datensätze. So kann die Ansicht schneller geladen werden.

2. Achten Sie bitte außerdem darauf, dass **Trigger sich nicht gegenseitig "behindern" oder Zirkelbezüge auslösen**.

3. Verwenden Sie außerdem **`select`-Anweisungen nur, wenn es wirklich notwendig** ist.
   Wenn Verknüpfungen zur durchsuchten Tabelle bestehen, sollten diese unbedingt anstelle von **`select`** genutzt werden.
   Verwenden Sie für Untertabellen die Punktnotation statt **`select`**.
   z.B.
   statt:

   ```
   let me:=this;
   sum((select Rechnungspositionen where Rechnung=i).Gesamt)
   ```

   besser:

   ```
   let me:=this;
   sum((me.Rechnungspositionen [Rechnung=i].Gesamt)
   ```

4. Versuchen Sie, **`select`**-Anweisungen in Funktionsfeldern in Tabellenansichten oder Formularfeldern zu vermeiden.
   Wenn eine **`select`**-Anweisung notwendig ist, verwenden Sie bitte das **`where`** und nicht die eckigen Klammern: [...] für Bedingungen.

5. Das Ergebnis eines benötigten **`select`**-Befehls in einem Skript immer nur einmal ausführen, für den mehrfachen Gebrauch in eine Variable schreiben und dann diese benutzen, um ggf. verschiedene Felder abzurufen.

6. Möglichst keine oder wenige **`select`**-Anweisungen in verschachtelten Schleifen.

7. Besser als Formeln immer wieder “On the fly” berechnen zu lassen, ist es, feste Werte über Trigger in Datenfelder zu schreiben, wenn dies möglich und sinnvoll ist.

8. Zur Suche in Tabellenansichten, wenn man eine Formularansicht als Dashboard nutzt: Der beste Weg wäre es hier, alle Variablen mit **`let`** zu definieren und dann die Suche mit einem **`select`** Befehl durchzuführen.

   Dabei ist es sinnvoll, am Ende die Suche auf z.B. 100 Datensätze zu begrenzen, damit Ninox nicht unnötig viele Datensätze hochlädt. Wichtig ist hier auch das Berücksichtigen des ggf. leeren Suchfeldes mit dem **`not`**-Befehl.

   z.B. so:

   ```
   let mySearch := 'Projekt suchen';
   select Projekte where (not mySearch or Projektname +
   concat('Projekt Mitarbeiter'.'Person auswählen'.(Vorname + " " +
   Nachname)) like mySearch) from 0 to 100
   ```

9. Vermeiden Sie die Speicherung großer Texte in Textfeldern, z.B. große CSV-Dateien beim Transfer oder JSON-Dateien für API-Calls.
   Da Ninox jede Änderung dokumentiert, wird bei großen Texten die Änderungshistorie unverhältnismäßig groß.
   Es ist viel besser, solche Texte mit **`createTextFile()`** als Anhang zu erzeugen und bei Bedarf die Datei mit der API zu öffnen.

Die Performance von Tabellen-Ansichten

Die Performance von Tabellenansichten wird durch drei Faktoren beeinflusst:

- Die Komplexität der Abfrage (Filter),
- die Anzahl der zu ladenden Zeilen,
- die Daten, die pro Zeile geladen werden.

Ninox berechnet Ansichten in folgendem Ablauf:

1. Frage vom Server alle Datensatz-IDs ab, die den Filterkriterien der Ansicht entsprechen.
2. Lade alle Datensätze, die sich unmittelbar aus dieser Filterung ergeben;
   und (im selben Schritt) lade alle Datensätze, die voraussichtlich zur Berechnung der Spalten zusätzlich benötigt werden.
3. Berechne alle Zeilen und Spalten.

Performance-Problem 1: Sehr viele Zeilen

Performance-Probleme treten in der Regel auf, wenn entweder die Anzahl der Zeilen (Ergebnis aus Schritt 1) sehr hoch ist, zum Beispiel 100.000 Zeilen oder mehr, oder wenn im Schritt 2 sehr viele zusätzliche Daten geladen werden.

Ersteres lässt sich lösen, indem Sie die Datensätze stärker filtern – oft benötigen die Anwender nur aktuelle Daten, zum Beispiel die noch offenen Rechnungen, nur aktive Kunden oder Aufgaben, die nicht älter als zwei Wochen sind.

Performance-Problem 2: Sehr viele zusätzliche Datensätze werden geladen

In der Praxis häufiger zu sehen ist, dass berechnete Spalten eingeblendet sind, die Daten aus verknüpften Tabellen heranziehen. Wenn Sie zum Beispiel eine Tabellenansicht aller Kunden erstellen und in dieser Ansicht die Summe der Rechnungsbeträge einblenden, um den Gesamtumsatz pro Kunde anzuzeigen, wird Ninox in Schritt 2 auch alle Rechnungsdatensätze und alle Rechnungspositionen laden, um das Ergebnis berechnen zu können.

Dieses Problem können Sie lösen, indem Sie entsprechende Spalten ausblenden. Falls dies im Einzelfall nicht möglich ist, sollte die Ansicht noch stärker gefiltert werden - oder Sie ersetzen das berechnete Feld durch ein tatsächliches Datenfeld, in dem der Wert durch einen Trigger berechnet und gespeichert wird.

Performance-Problem 3: Komplexe Formeln

Verwandt mit vorhergehendem Fall ist die Situation, dass sehr komplexe Formeln in den Spalten der Ansicht hinterlegt sind. Um dies zu illustrieren:

Angenommen, eine Ansicht zeigt 1000 Zeilen. Eine der Spalten vergleicht einen Wert der Zeile mit 1000 Datensätzen aus einer anderen Tabelle. So ergeben sich schnell 1.000.000 Vergleiche, die Ninox berechnen muss, obwohl die Ansicht zunächst überschaubaren Umfang hatte.

Betrachten Sie dabei nicht nur die unmittelbare Formel, die angezeigt wird - eventuell nutzt die Formel weitere Formelfelder und die Komplexität ist über mehrere Ebenen verborgen.

Performance-Problem 4: "Sub-`selects`"

Ninox bietet im Umfang der Script-Sprache auch ein `select` Statement. Die Berechnung von `select` kann Ninox nicht so gut optimieren, wie die Nutzung von Verknüpfungen. `select` sollte in Formeln von Tabellen-Ansichten nicht genutzt werden, da jede Zeile der Ansicht eine separate Abfrage auslöst.

Die Performance von Buttons

Ninox-Scripte, die durch Buttons (oder bei Klick auf ein Formelfeld) ausgelöst werden, laufen im Kontext des Clients und nicht als Transaktion. Das heißt, Ninox führt Befehl für Befehl separat aus, als hätte der Nutzer diese Eingaben getätigt.

Daraus ergeben sich folgende Konsequenzen:

- Die Ausführungszeit kann lange dauern, wenn viele Befehle ausgeführt werden, da Client und Server permanent untereinander Daten austauschen müssen.
- Das Script kann mittendrin abgebrochen werden, zum Beispiel indem man das Browser-Fenster schließt.
- Während der Script-Ausführung können auch Datenänderungen von anderen Nutzern auftreten.

Um dies zu optimieren, können alle Befehle in einer Transaktion geklammert werden mit "`do as transaction`". Die Ausführung erfolgt dann in einer einzigen Transaktion auf dem Server (Web-Version) oder in der Client-Datenbank (native Apps). Mit "`do as server`" erzielt man einen ähnlichen Effekt, erzwingt aber die serverseitige Ausführung auch in den nativen Apps.

In der Regel laufen Scripte innerhalb von "`do as server`" erheblich schneller ab.

Im Gegenzug sind aber die Einschränkungen im nachfolgenden Abschnitt "Performance von Transaktionen" zu beachten.

Beachten Sie auch: Trigger (z.B. bei Änderung von Daten) werden automatisch innerhalb der ändernden Transaktion ausgeführt - analog zu "`do as transaction`".

Die Performance von Transaktionen

Alle Trigger bei Datenerstellung oder Änderung sowie alle Scripts innerhalb von "`do as transaction`" oder "`do as server`" laufen innerhalb einer Transaktion ab, die entweder vollständig ausgeführt wird oder gar nicht, wenn sie abbricht. Eine Transaktion ist schreibend, wenn wenigstens ein Befehl innerhalb des Scripts potentiell Daten ändern könnte (z.B. den Wert eines Datenfeldes ändern).

Ninox führt schreibende Transaktionen seriell aus. Das heißt, Lesezugriffe behindern sich gegenseitig nicht, aber ein schreibender Zugriff kann nachfolgende schreibende Transaktionen blockieren.

Normalerweise bedeutet dies keine Einschränkung, selbst wenn hunderte Nutzer parallel arbeiten. Problematisch wird dieses Verhalten allerdings, wenn eine schreibende Transaktion lange Zeit (mehr als wenige Millisekunden) benötigt.

Vermeiden Sie innerhalb von Transaktionen:

- Die Abfrage externer APIs
- Komplexe Abfragen über viele Daten

Hinweis:
Nutzerinteraktionen – z.B. `dialog(...)`, `openTable(...)` – sollten in keinem Fall innerhalb einer Transaktion verwendet werden. Bei serverseitiger Ausführung (zum Beispiel im Web) funktionieren diese schlicht nicht.

Transaktionsklammern aufbrechen

Ninox bietet Möglichkeiten, den Transaktionskontext aufzubrechen mit der Anweisung “do as deferred”.

Angenommen, bei Änderung eines Rechnungsstatus soll ein externes System informiert werden. Dann können Sie mit `do as deferred` verhindern, dass dieser API-Aufruf, der im Rahmen eines Triggers ablaufen soll, nachfolgende Transaktionen blockiert. Schreiben Sie dazu:

```
do as deferred
      http(...)
end
```

Sie können das Ergebnis aus dem API-Call auch weiterverwenden. **Wichtig** ist aber, dass diese Weiterverwendung selbst wieder einen neuen Transaktionskontext erzeugt, ansonsten wäre schon der `http(...)` Aufruf Teil einer schreibenden Transaktion.
Beispiel:

```
let invoice := this;
do as deferred
      let response := http(...);
      do as deferred
            invoice.Message := text(response.result)
      end
end
```

In diesem Fall ist das erste “`do as deferred`” eine lesende Transaktion, da keine Daten geändert werden. Das geschachtelte “`do as deferred`” ist eine schreibende Transaktion, da ein Feld des Rechnungsdatensatzes “invoice” geändert wird.

Mehr Informationen dazu finden Sie unter:
https://docs.ninox.com/en/script/introduction-to-ninox-script/optimize-performance-of-scripts