Datenbankperformance mit Microsoft Access

Grundlagen für optimales Tabellen- und Abfrage-Design

Referent: Michael Zimmermann

Microsoft Most Valuable Professional

Organisation der Datenbankentwickler dbdev.org

Access-Stammtisch Rhein-Main

SZ DataWare www.SZWeb.de

Inhalt

Einführungsbeispiel Suche

Mit Datenbankperformance und SQL hat das Beispiel scheinbar gar nichts zu tun. Um jedoch zu verstehen, worum es bei effizientem Performance-Tuning überhaupt geht und wie es eigentlich funktioniert, ist es ausgesprochen lehrreich.

Und da es nebenbei sehr schön demonstriert, wieso ein Index eigentlich so wichtig ist, hat es eigentlich doch schon wieder mit Datenbanken zu tun.

Naive oder lineare Suche

Dieser einfachste Suchalgorithmus durchläuft einfach alle Elemente einer Liste, bis das gewünschte gefunden ist. Im Mittel werden daher bei n Elementen n/2 Suchschritte benötigt.

```
Public Function NaiveSuche( _
                Arr() As String,
                Search As String) _
            As Long

Dim i As Long

Dim k As Long, g As Long

k = LBound(Arr)
g = UBound(Arr)

For i = k To g
    If Arr(i) = Search Then Exit For
Next i

If i > g Then i = 0

NaiveSuche = i

End Function
```

Naive Suche nach **R**:

A	C	F	I	J	K	L	M	O	P	**R**	S	T	V	X	Z
1	2	3	4	5	6	7	8	9	10	**11**					

Der Algorithmus ist auf allen Datenmengen anwendbar.

Bei einer Verdoppelung der Datenmenge verdoppelt sich auch die durchschnittlich benötigte Anzahl der Suchschritte.

Für 65536 Elemente braucht der Naive Algorithmus im Mittel 32768 Schritte

Binäre Suche

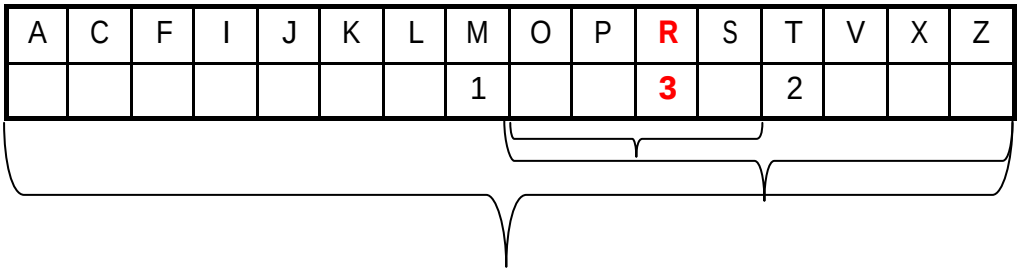
Dieser Algorithmus ist ein sogenannter Divide-et-Impera-Algorithmus und zur Suche erheblich effizienter, insbesondere bei großen Datenmengen.

Er nimmt an, daß das gesuchte Element in der Mitte der Liste liegt, und prüft das Vorliegen ab. Wenn – erwartungsgemäß – dieses Element doch kein Treffer ist, wird verglichen, ob das gewünschte Element größer oder kleiner als die Mitte ist. Wenn es beispielsweise kleiner ist, kann die größere Hälfte für die weitere Suche entfallen.

Die Hälfte, in der jetzt zu suchen ist, wird jetzt nach dem gleichen Schema wieder halbiert und so weiter.

```
Public Function BinaereSuche( _
                Arr() As String, _
                Search As String) _
            As Long
'Voraussetzung: Array muß sortiert sein

Dim i As Long, j As Long

Dim k As Long, g As Long, m As Long

k = LBound(Arr)
g = UBound(Arr)

Do
    m = (g + k) \ 2
    If Search > Arr(m) Then k = m + 1 Else g = m - 1
Loop Until Arr(m) = Search Or k > g

If Arr(m) <> Search Then m = 0

BinaereSuche = m

End Function
```

Binäre Suche nach **R**:



Kein Licht ohne Schatten: Der Algorithmus ist *nur auf sortierten Datenmengen* anwendbar, da die Annahme, ein Element aufgrund eines Vergleichs in einer bestimmten Hälfte zu finden, sonst keinen Boden hat.

Bei einer Verdoppelung der Datenmenge kommt jedoch nur ein (1!) Suchschritt hinzu.

Für 65536 Elemente braucht der Binäre Algorithmus maximal 16, im Mittel 8(!) Schritte.

Da viele Datenbankmechanismen (Sortierung , Selektion oder Join seien hier genannt) das Vorliegen einer Vorsortierung nutzen können bzw. müssen, erklärt sich hoffentlich anschaulich die überragende Bedeutung der Indizes, die, wie wir noch sehen werden, nichts anderes als eine Sortierinformation sind, für die Datenbank-Performance.

Moral:

Performance-Tuning basiert i. d. R. nicht auf „geheimen“ High-Speed-Befehlen, sondern auf Mathematik: Wie viele Bytes werden bewegt, wie viele Schritte sind erforderlich, welche Zusatzkosten entstehen für RAM-Verwaltung, Prozessorbelastung, Plattenzugriffe.

Im Beispiel oben ist der bessere Algorithmus für *einen* Durchlauf sogar spürbar langsamer als sein Konkurrent, da er mehr Vergleiche durchführt, in der Schleife rechnet und die langsame Do-Loop-Konstruktion statt des schnelleren For-Next benutzt. Der Vorteil beruht allein auf dem effizienteren logischen Prinzip.

Solche und ähnliche Algorithmen werden von Datenbanksystemen intern eingesetzt, um z. B. Anforderungen durch eine WHERE-Klausel o. ä. zu erfüllen.

Meßmethoden

QueryPerformanceCounter

```
Private Declare Function QueryPerformanceCounter _
            Lib "kernel32" ( _
                x As Currency) _
            As Boolean

Private Declare Function QueryPerformanceFrequency _
            Lib "kernel32" ( _
                x As Currency) _
            As Boolean
```

Diese API-Funktion ist die von Windows zur Verfügung gestellte Meßmethode mit der höchsten zeitlichen Auflösung, die systemabhängig bis in den Nanosekundenbereich reicht.

In der Beispiel-DB findet sich eine Kapselung der API-Funktion in eine im Code bequem verwendbare Routine, die auch die systematischen Tücken, die beim Messen grundsätzlich zu beachten sind, umschifft.

Prozeduraufrufzähler

```
Public Declare Sub Sleep _
            Lib "kernel32" ( _
                ByVal dwMilliseconds As Long)

Public Function CallCounter( _
                Optional Dummy As Variant, _
                Optional SleepTime As Long, _
                Optional Reset As Boolean) _
            As Long

Static z As Long
z = z + 1
Sleep SleepTime
If Reset Then z = 0
CallCounter = z
End Function
```

Die Funktion dient dazu, festzustellen, wie Funktionsaufrufe innerhalb von Abfragen von Jet gehandhabt werden. Durch den Sleep-Parameter kann man die Auswirkungen länger dauernder Aufrufe auf den Aufbau der Bildschirmanzeige des Resultsets beobachten.

JetShowPlan

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0\
Engines\Debug]
"JETSHOWPLAN"="ON"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0\
Engines\Debug]
"JETSHOWPLAN"="OFF"
```

Durch diesen Registry-Schlüssel wird eine Protokollausgabe von Jet erzeugt. Man kann dort nachvollziehen, wie eine Abfrage von Jet im Detail ausgeführt wurde.

Die Ausgabe entsteht in Form einer Textdatei, die den Namen **showplan.out** hat. Sie befindet sich im allgemeinen im Ordner Eigene Dateien oder im Ordner, in dem sich die DB befindet. Sollte sich die Datei nicht gleich finden lassen, läßt man einfach eine Windows-Suche mit dem Dateinamen laufen.

Da neue Einträge am Ende angefügt werden, und die Ausgabe bei jedem Ausführen einer Abfrage erfolgt, wird die Datei mit der Zeit sehr groß. Man sollte also den Showplan nur bei Bedarf einschalten oder die Ausgabedateien gelegentlich löschen.

Zur Bedeutung der Ausgaben gibt es leider keine einfach zugängliche Dokumentation, was den Wert erheblich einschränkt. Dennoch kann man ein paar grundlegende Dinge festhalten.

Gute Ergebnisse

- Using Rushmore
- Using Index 'Indexname'
- Using temporary Index
- Semi Join
- Merging Indexes (Merge Join)

Ein optimales Ergebnis bei Bedingungen mit Operatoren ist der Eintrag Using Rushmore. Die Rushmore-Technologie ist eine besonders effiziente Art, die Informationen eines Index zu nutzen.

Zufriedenstellend sind Angaben wie Inner Join, Outer Join, Semi Join, Merge Join zu den zur Join-Berechnung verwendeten Algorithmen.

Einige Beispielausgaben:

```
01) Inner Join table 'tDetail' to table 'tMaster'
      using index 'tMaster!PrimaryKey'
      join expression "D.fiMAster=M.idMaster"
02) Sort result of '01)'
03) Semi Join table 'tMaster' to result of '02)'
      using temporary index
      join expression "H.dtMasterWert=D.dtDetailWert"

01) Restrict rows of table tMaster
      using index 'dtMasterWert'
      for expression "tMaster.dtMasterWert>=0"

01) Outer Join table 'tMaster' to table 'tDetail'
      merging indexes
      join expression
"tMaster.idMaster=tDetail.fiMaster"

01) Restrict rows of table tblDatentypen
      using rushmore
      for expression "(tblDatentypen.id<10000) OR
(tblDatentypen.id>10000)"

01) Inner Join table 'tblMasterZuDatentypen' to table
'tblDatentypen'
      using index 'tblDatentypen!fiZahl'
      join expression
"tblMasterZuDatentypen.id=tblDatentypen.fiZahl"

01) Outer Join table 'tblDatentypen' to table
'tblMasterZuDatentypen'
      merging indexes
      join expression
"tblDatentypen.fiZahl=tblMasterZuDatentypen.id"

01) Semi Join table 'tdatKundeZuBestellung' to table
'tdynBestellungenLongSchlüssel'
      using index
'tdynBestellungenLongSchlüssel!tdatKundeZuBestellungt
dynBestellungenLongSchlüssel'
      join expression "K.idKd=B.fiKd"
```

Schlechte Ergebnisse

- Scan Table
- Restrict … by scanning … testing expression
- Using X-Prod join

Ein schlechtes Ergebnis ist insbesondere „using Xprod join“, da der Kreuzprodukt-Algorithmus der mit Abstand langsamste Join-Algorithmus ist. Er ist aber unvermeidlich, wenn die Join-Felder

nicht indiziert sind oder ungünstige Kriterien eine Indexnutzung verhindern.

```
01) Scan table 'tblDatentypen'
    Using index 'PrimaryKey'

01) Restrict rows of table tblDatentypen
      by scanning
      testing expression "Not tblDatentypen.id=10000"

01) Inner Join table 'tdynBestellungenLongSchlüssel'
to table 'tdatKundeZuBestellung'
      using X-Prod join
      then test expression "K1.idKd<Bn.fiKd"
```

Anmerkungen zu den Join-Typen

Neben der bekannten *logischen* Kategorisierung von Joins wie Inner Join, Left/Right Outer Join, Full Join, Semi Join, Natural Join gibt es auch ähnlich benannte Algorithmen, die beschreiben, mit welchen Anweisungen ein Join durchzuführen ist. Mit diesen hat man in der Regel nichts zu tun, da sie ja Bestandteil des DBMS sind, ebenso wie die Auswahl des anhand der Datenkonstellation bestgeeigneten Algorithmus.

Join-Algorithmen ähneln einer Mischung aus Such- und Sortier-algorithmen, und wie dort gibt es mehr oder weniger effiziente Vorgehensweisen, wobei die beste Wahl noch von der Datenlage (Indizierung, statistische Verteilung) abhängt.

Übliche Algorithmen sind z. B.

- Nested-Loop Join
- Nested-Block Join
- Sort-Merge Join
- Semi Join
- Hash Join

Das schlimmste Ergebnis, das produziert werden kann, ist ein *X prod join* in der Ausgabe. Es bedeutet, daß das kartesische Produkt aller Datensätze gebildet wurde, um dann alle auszu-sortieren, die nicht der Join-Bedingung entsprechen. Dieses Er-gebnis bekommt man zum Beispiel beim Join über nichtindizierte Felder oder bei Theta-Joins, also solchen, die als Vergleichsoperator für tabellenübergreifende Felder etwas anderes als = benutzen.

Wenn es geht, sollte man solche Konstrukte vermeiden.

Einen Einfluß auf die Wahl des Algorithmus, außer durch Indizierung den Boden zu bereiten, hat man sowieso nicht; es ist also müßig, sich hier allzu viele Gedanken zu machen.

Abfragemessung über Recordset:

Wenn man sich nicht mit der Stoppuhr an den Bildschirm setzen will, ist es erforderlich, die Ausführung einer Abfrage per VBA zu kontrollieren. Hierzu wird man ein auf der Abfrage basierendes Recordset erstellen. Um dabei sinnvolle Ergebnisse zu erhalten, sind einige Eigenheiten zu beachten.

Die Zeit für das reine Erstellen des Recordet-Objekts ist nicht aussagekräftig, daher ist ein MoveLast auszuführen.

Um den Einfluß von Berechnungen zu beurteilen, ist zusätzlich ein satzweises Lesen des berechneten Feldwertes nötig, da Jet Werte erst bei Bedarf errechnet.

Für sehr präzise Ergebnisse sollte immer auch der Overhead einer Messung herausgerechnet werden, also die Zeit, die für die reine Verwaltung eines Recordsets benötigt wird. Die im folgenden aufgeführten Beispiele sind in der Auswirkung so drastisch (mehrere hundert bis zehntausend Prozent), daß man von solchen Feinheiten absehen kann.

Für eigene Experimente sei noch darauf hingewiesen, daß die bei mit Meßtechniken weniger Erfahrenen beliebte Methode, den Prüfcode in einer Schleife mehrmals ablaufen zu lassen, um eine bessere zeitliche Auflösung zu bekommen, für Abfragemessungen nicht sinnvoll einsetzbar ist, da Geschwindigkeitsunterschiede hier in der Regel auf Indexnutzung basieren und daher eine nichtlineare Zeitkomplexität aufweisen.

Eine Messung über hunderttausend Datensätze läßt sich im allgemeinen nicht durch eine Messung über hundert Datensätze, die in einer Schleife tausendmal ausgeführt wird, simulieren.

Bedeutung der Prozentangaben

Bei vielen Beispielen sind Angaben, um wieviel Prozent die Geschwindigkeit bei Optimierung steigt. Diese Angaben sind kein konstantes Verhältnis, sondern hängen von Struktur und Menge der Daten ab. Das liegt daran, daß die vom DB-System eingesetzten Algorithmen im allgemeinen keinen proportionalen Zeitzuwachs aufweisen.

Eine Vorgehensweise, die bei wenigen Daten nur anderthalb mal so schnell ist wie die zum Vergleich herangezogene Methode, kann bei größeren Datenmengen durchaus zehn- oder hundertmal so schnell sein. Die im Text angegebenen Werte sind in genau dieser Form außer mit dem Datenbestand, mit dem sie erzeugt worden sind, nicht exakt reproduzierbar. Man verstehe sie daher als Hinweis, bei welcher Konstruktion die bessere Laufzeit zu erwarten ist.

Meßplattform

Alle Tests wurden mit Access XP auf Windows XP Professional vorgenommen.

Die meisten Ratschläge gelten im wesentlichen – zum Teil sogar verstärkt –, für alle Access-Versionen, bei anderen Versionen kann es aber in den Auswirkungen zu mehr oder weniger starken Unterschieden kommen.

Etliche Ratschläge, die einfach auf allgemeinen logischen Grundsätzen basieren, sind sogar in ähnlicher Weise auf andere DB-Systeme übertragbar, hier sind die Abweichungen aber naturgemäß groß, da andere Systeme andere Abfrageoptimierer haben und andere SQL-Syntaxvarianten unterstützen.

Konzeption

BE auf eigene Partition

Um Verluste durch Fragmentierung auf Dateisystemebene zu reduzieren, kann man das Backend exklusiv auf eine eigene ausreichend große Partition legen. Eine Fragmentierung findet so kaum noch statt.

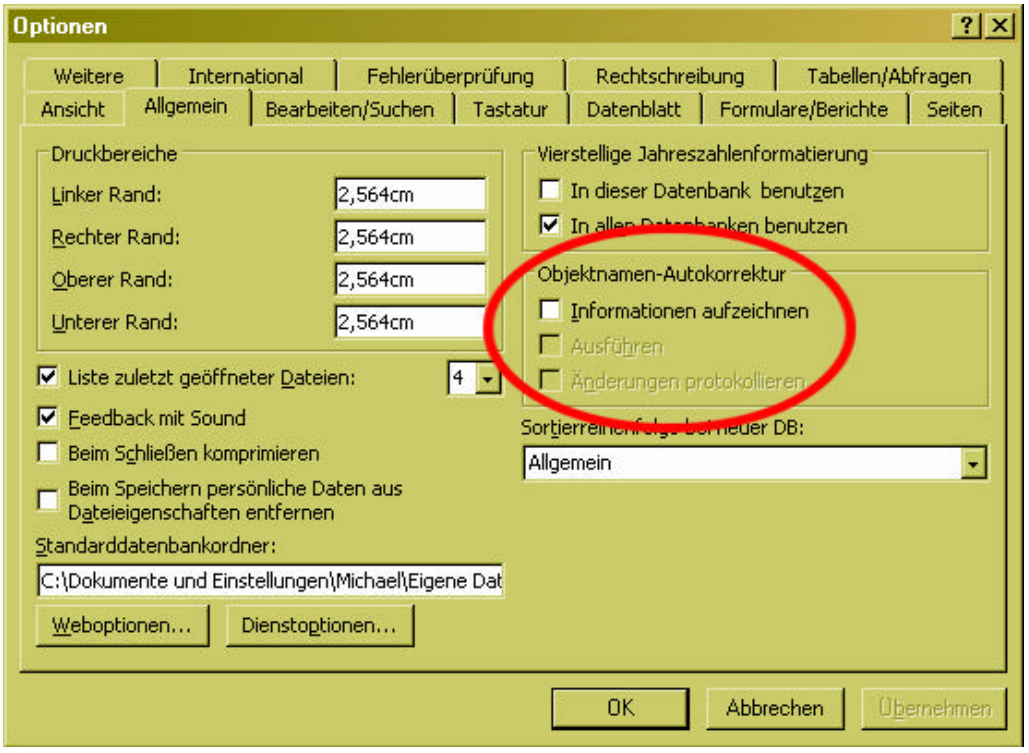
Man bedenke bei der Partitionsplanung folgende Eckdaten: Eine mdb kann maximal 2 GB groß werden, es wird Platz in der Größenordnung der Originaldatei für die temporäre Datei bei der Kompression benötigt und es wird ein wenig Platz für die Erstellung der Locking-Datei (ldb) benötigt.

Kurzer Name, kurzer Pfad

Lange Namen und lange Pfade können zu spürbarem Geschwindigkeitsverlust führen.

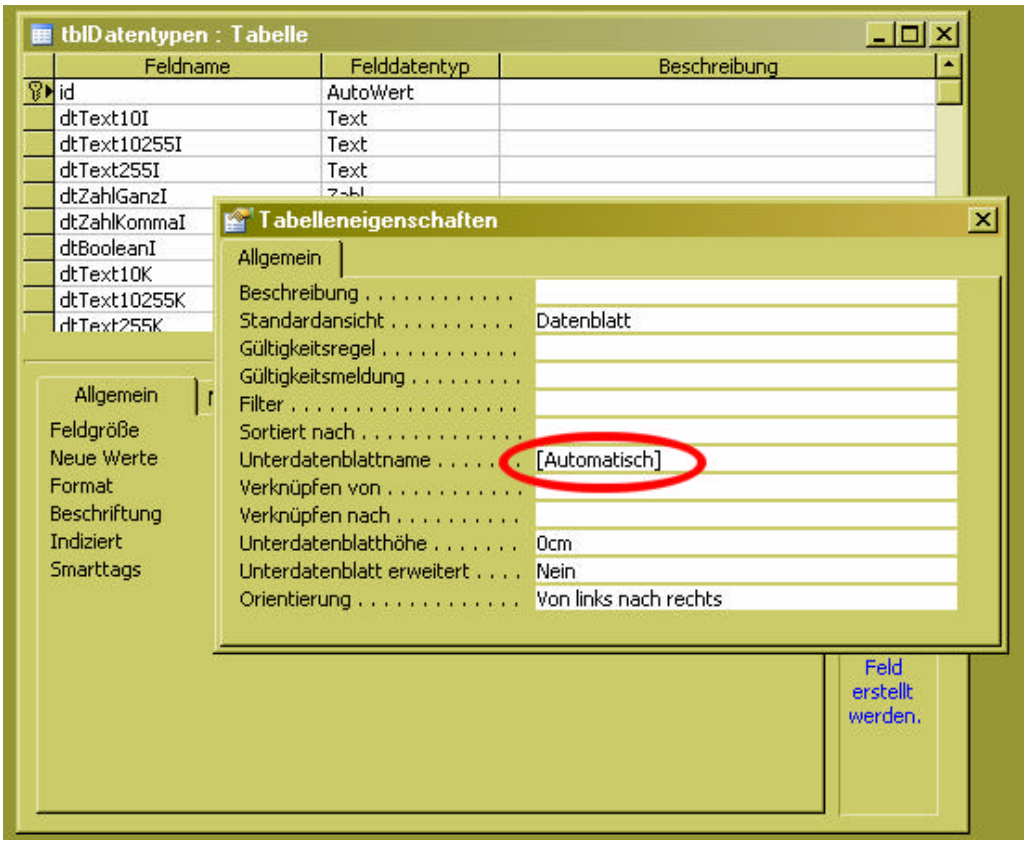
Objektnamen-Autokorrektur aus

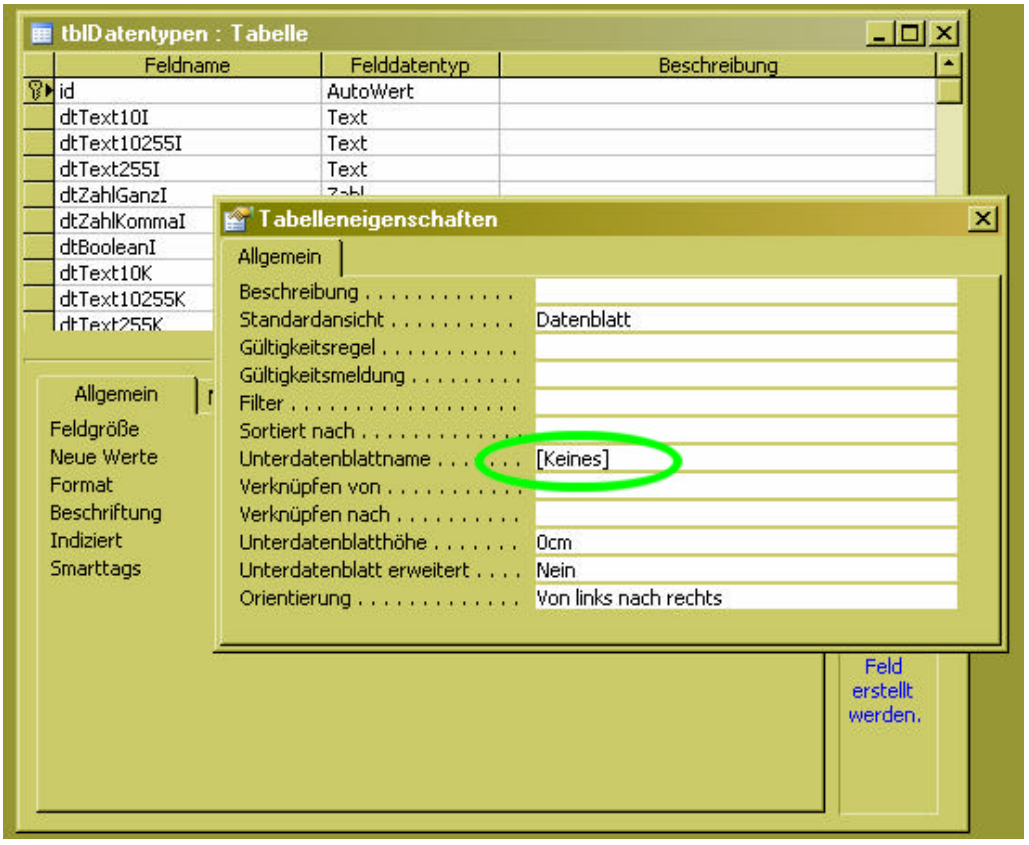
Unter Extras, Optionen, Allgemein die Objektnamen-Autokorrektur komplett deaktivieren.



Unterdatenblätter ausschalten

Im Tabellenentwurf, Eigenschaften Unterdatenblattname von *Automatisch* auf *Kein* stellen.





DB regelmäßig komprimieren

Da hierbei auch die intern geführten Statistiken an den aktuellen Datenbestand angepaßt werden, bringt diese Maßnahme mehr, als von einem reinen Verkleinern der Datei zu erwarten wäre.

Permanentes* Recordset

Wenn man ein Recordset oder ein *gebundenes* unsichtbares Formular während der gesamten DB-Sitzung geöffnet hält, wird das Löschen und Wiedererstellen bei Bedarf der Locking-Datei unterbunden. Dadurch werden z. B. Formulare schneller geöffnet.

*Es heißt *nicht*, wie schon gelegentlich gehört, *persistent*. Das würde bedeuten, daß das Recordset auf einen Datenträger gespeichert würde.

Tabellenentwurf

1 NF einhalten

Das Zerlegen von Feldern ist deutlich weniger performant als das Konkatenieren. Was für Sortierung, Filterung und Gruppierung verwendet werden soll, muß sowieso in einem eigenen Feld abgelegt werden, um es indizieren zu können.

Es ist also günstiger, Vorname und Nachname abzulegen und daraus bei Bedarf Vorname & ' ' & Nachname AS VollerName zu erzeugen, als ein Feld VollerName abzulegen und daraus Vorname und Nachname zu extrahieren.

Jede Informationseinheit, die bedeutungstragend ist, also isoliert gebraucht wird, gehört in ein eigenes Feld.

Long-Schlüssel für Join

Der Datentyp Long Integer wird von allen Datentypen am schnellsten verarbeitet.

Kategorien auslagern

Kategorien werden oft für Filterungen oder Gruppierungen verwendet. Hierfür genügt im Code eine schnell verarbeitbare Ganzzahl statt eines Textes.

Da die Steuerung über Formulare erfolgt, kann man dennoch dem Benutzer in einem Kombifeld anschauliche Texte anbieten, während im Hintergrund mit Zahlen gearbeitet wird.

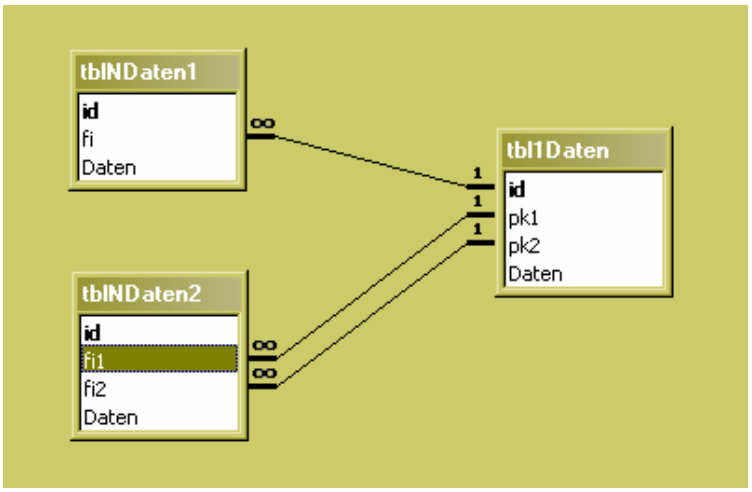
Join-Vermeidung

... durch assoziative (Parallel-) Schlüssel statt Denormalisierung

Es kann in Einzelfällen sinnvoll sein, AutoWert-Schlüssel mit assoziativen Schlüsseln, auch wenn dadurch Mehrfelderschlüssel entstehen, zu ergänzen. Dadurch sind die n-seitigen Nutzdaten als Fremdschlüssel bereits in der 1-Tabelle enthalten. Joins, die sonst erforderlich wären, um die Daten aus der N-Tabelle hereinzuholen, entfallen dadurch.

Wenn auf der N-Seite noch weitere Daten außer den Schlüsselfeldern vorliegen, sind im Gegenzug die dann erforderlichen Mehrfelder-Joins natürlich langsamer als die über ein AutoWert-

Long-Feld. Man muß also im einzelnen abwägen, was insgesamt günstiger ist.



Selbstverständlich kann man in Beziehungen zu weiteren Tabellen nach wie vor den AutoWert-Schlüssel benutzen.

Die Tabelle tblNDaten1 könnte über fi-id beliebige Daten aus tbl1Daten beziehen, zum Beispiel auch pk1 oder pk2, während die Tabelle tblNDaten2, von der wir annehmen, daß sie nur die Daten pk1 und pk1 aus tbl1Daten benötigt, gar keinen Join braucht, da die Daten als fi1 und fi2 bereits in ihr selbst enthalten sind. Die Beziehung ist dennoch nötig, um die referentielle Integrität sicherzustellen.

Da die Relationale Theorie sowohl Fremdschlüsselredundanz, da sonst auch gar keine mehrwertigen Beziehungen möglich wären, als auch das Erstellen von Beziehungen über Parallelschlüssel ausdrücklich erlaubt, ist das Vorgehen kein Verstoß gegen Entwurfsvorschriften aus E-R-Modellierung und Normalisierung.

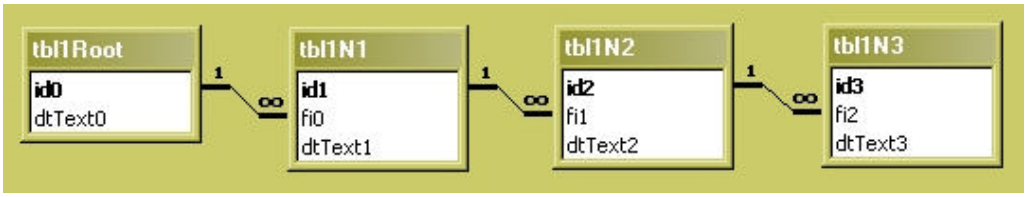
Natürlich darf man *nicht* zu *derselben* Tabelle zweimal *dieselbe* Beziehung – einmal über AutoWert und einmal über einen Parallelschlüssel – aufbauen.

Mehr-Ebenen-Joins

... mit assoziativen Schlüsseln statt AutoWert

Ein ähnlicher Trick läßt sich in einem zugegebenermaßen seltenen Fall anwenden. Wenn in einer mehrstufigen 1:n-Hierarchie häufig *Joins unter Überspringung mittlerer Ebenen* erforderlich sind, ist ein alternativer Aufbau zum gewohnten Long-Autowert-Beziehungsschema günstiger.

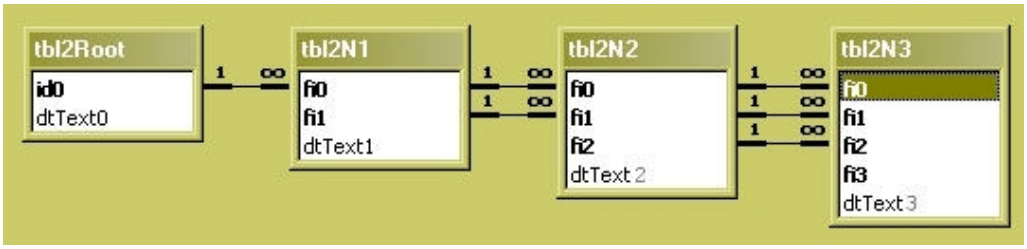
Normaler Aufbau:



Abfrage auf dtText0, dtText3:

```
SELECT
     tbl1Root.dtText0, tbl1N3.dtText3
FROM
     (
          (
               tbl1Root INNER JOIN tbl1N1
               ON tbl1Root.id0 = tbl1N1.fi0
          )
          INNER JOIN tbl1N2
          ON tbl1N1.id1 = tbl1N2.fi1
     )
     INNER JOIN tbl1N3
     ON tbl1N2.id2 = tbl1N3.fi2
```

Alternativer Aufbau:



Die Felder fi1, fi2, fi3 sind dabei vom Typ gruppierte Numerierung, also z. B.

tbl2N1	
Fi0	**fi1**
12	1
12	2
12	3
15	1
15	2
15	3
15	4
15	5
...	...
...	...

tbl2N2		
fi0	**fi1**	**fi2**
12	1	1
12	1	2
12	2	1
12	2	2
12	2	3
12	2	4
12	3	1
12	3	2
...	...	...
...	...	...

tbl2N3			
fi0	**fi1**	**fi2**	**fi3**
12	1	1	1
12	1	1	2
12	1	1	3
12	1	2	1
12	1	2	2
12	1	2	3
12	1	2	4
12	2	1	1
...	...	...	...
...	...	...	...

Abfrage auf dtText0, dtText3:

```
SELECT
     dtText0, dtText3
FROM
     tbl2Root INNER JOIN tbl2N3
     ON tbl2Root.id0 = tbl2N3.fi0
```

Beim alternativen Aufbau werden die Schlüsselwerte jeder Muttertabelle durch alle darunterliegenden Ebenen vererbt und jeweils um eine weitere Numerierungsebene ergänzt. Dadurch kann ein direkter, ebenenübergreifender Join vorgenommen werden.

Diese Variante bietet natürlich nur Vorteile, wenn die Anforderung solcher Joins auch öfter auftritt. Als *grundsätzliche* Alternative ist das Vorgehen *nicht* empfehlenswert.

Berechnungen archivieren

Das Speichern von Berechnungsergebnissen ist zunächst ein Entwurfsfehler und damit verboten. Das Problem hierbei ist die Möglichkeit, widersprüchliche Daten abzulegen und die DB damit in einen inkonsistenten Zustand zu bringen. Argumente der Art, man passe durch entsprechenden Code auf, daß das nicht passiert, ziehen nicht: Erstens werden beim Aufpassen die meisten Kinder gezeugt und zweitens handelt es sich datenbanktheoretisch nicht bloß um ein Vergehen, sondern um ein Verbrechen, d. h. die reine Möglichkeit, auch ohne daß etwas passiert, ist bereits strafbar.

Kalkulation (Zeilenberechnung)

Eine Kalkulation ist eine Berechnung mit Feldinhalten innerhalb eines Datensatzes, also z. B. Netto * (1 + MWSt) AS Brutto

Solche Berechnungen sind nicht indexabhängig, da sie keine Informationen über die Reihenfolge von Daten in verschiedenen Datensätzen verarbeiten.

Diese Berechnungen sind meist sowieso sehr schnell, daher wäre eine Speicherung der Berechnungsergebnisse sinnlos. Natürlich kann man auch hier durch extrem aufwendige Kalkulationen Einbrüche erleiden, weswegen es zu diesem Thema auch noch einige Tips geben wird, im Vergleich zum Rechenaufwand bei Aggregaten sind diese Zeitverluste aber marginal.

Aggregation (Spaltenberechnung)

Eine Aggregation ist eine Berechnung oder Operation über mehrere Datensätze eines Feldes. Die meisten Aggregationen sind indexabhängig.

Bei großen Datenmengen können Aggregierungen sehr zeitintensiv werden. Das Speichern von Aggregaten ist evtl. sinnvoll, wenn die

Eingangsdaten für die Berechnung invariant sind, sich nach Eingabe also sicher nicht mehr ändern. Man sollte hieran strenge Maßstäbe anlegen:

Ein Geburtsdatum beispielsweise ist keineswegs, wie man naiv meinen könnte, per se ein invariantes Datum. Das ist es nur, wenn es per Code mit entsprechender Fehlerabsicherung eingetragen wurde. Bei händischer Eingabe durch Benutzer muß jederzeit mit Eingabefehlern gerechnet werden. Wenn diese irgendwann bemerkt werden, ändert sich dann tatsächlich das Geburtsdatum einer Person – nicht im wahren Leben, aber in der Modellwelt der Datenbank. Bei darauf basierenden berechneten, gespeicherten Werten schlägt die Änderungsanomalie mit voller Härte zu.

Sinnvoll speicherbare Aggregatberechnungsergebnisse sind zum Beispiel Temporaldaten wie Anzahl/Summe am/bis zum <Datum>. Da solche Werte vom System errechnet werden, sind sie als fehlerfrei anzusehen und daher unproblematisch, weil sicher invariant.

Nach diesem Prinzip werden typischerweise u. a. Kontoführungstabellen oder Belegbuchungstabellen (Rechnungen) aufgebaut.

Wenn alle Archivbedingungen erfüllt sind, liegt hierdurch auch kein NF-Verstoß bzw. keine echte Redundanz vor.

Speziell sollte für eine Archivtabelle gelten:

- Es werden keine Datensätze gelöscht → keine Löschanomalie möglich
- Es werden keine Datensätze geändert → keine Änderungsanomalie möglich *(Daher Fehlerbereinigung über Stornobuchungen)*
- Neue Datensätze werden vom System generiert und angefügt → Einfügeanomalie möglich, aber unwahrscheinlich, abhängig von Code-Qualität

Datum	**KdNr**	**Name**	**Anschrift**	**Betrag**
01.03.2005	1	Beispiel GmbH	Beispielweg 12	300
05.04.2005	1	Beispiel GmbH	Beispielweg 12	200
03.07.2005	1	Beispiel GmbH	Beispielweg 12	400
12.08.2005	1	Beispiel GmbH	Musterallee 7	500

Die Wiederholungen in den Anschriften sind im Gegensatz zu dynamischen Tabellen unschädlich, insbesondere durch den Wegfall der Änderungsanomalie, die in der Praxis die größten Probleme verursacht.

Wenn ein nachträgliches Ändern von Daten statthaft sein soll, ist auf das Archivprinzip zu verzichten und statt dessen eine Historisierung mit Gültigkeitszeiträumen vorzunehmen.

In ähnlicher Weise kann man auch Aggregate wie auflaufende Summen zu einem bestimmten Datum abspeichern mit dem Vorteil, die jeweils letzte Summe statt durch eine aufwendige Aggregation seit Beginn der Geschichtsschreibung durch ein schlichtes „Letzte Summe plus aktuelle Vorgänge“ zu berechnen.

Auf keinen Fall sollte man dieses Archivierungsprinzip anwenden, wenn die Eingangsdaten variabel sind: Integrität geht vor Performance.

Richtig indizieren

Das ist die eigentliche Kernvoraussetzung für die Verbesserung der Abfragegeschwindigkeit. Daher ist diesem Thema das ganze nächste Kapitel gewidmet.

Indexplanung

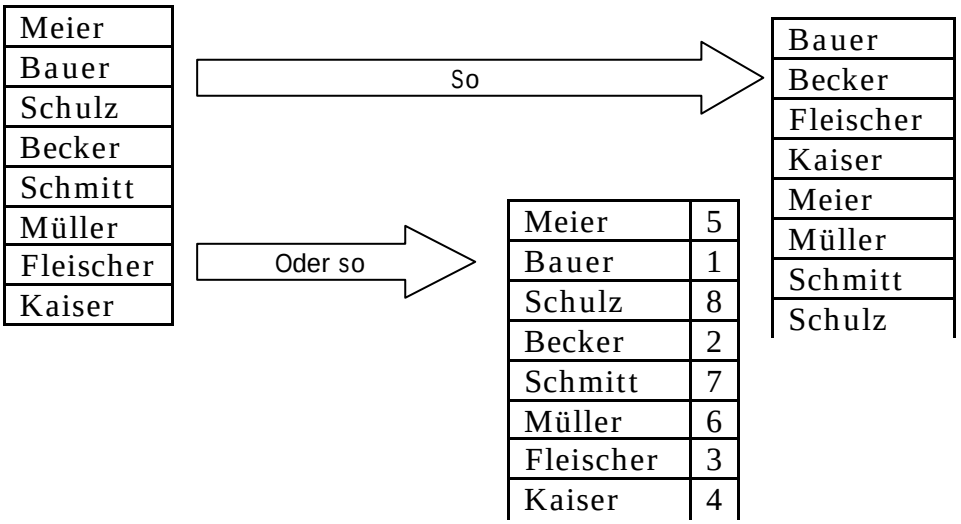
Was ist ein Index?

Ganz vereinfacht ausgedrückt, handelt es sich bei einem Index um eine Sortierung der Daten, also um eine Information, an welcher Position sich bestimmte Einträge befinden. Es gibt prinzipiell zwei Möglichkeiten, eine solche Information zu erzeugen:

- Eine tatsächliche physische Sortierung der Datensätze in der gewünschten Reihenfolge
- Ein Verzeichnis, das zu einem Datensatz seine Positionsnummer bezogen auf eine bestimmte Sortierung speichert.

Die erste Art Index ist besonders effizient, kann aber logischerweise nur für ein Kriterium verwendet werden. Diese Art der Indizierung wird als *Clustered Index* bezeichnet. (In Wahrheit liegen die Dinge etwas komplizierter, es kommt beim Clustering auf die Zuordnung der Daten zu sogenannten Speicherseiten an, auf denen sie Gruppen (Cluster) bilden, aber die tatsächliche physische Reihenfolge der Daten spielt hier eine entscheidende Rolle, um zu verhindern, daß logisch benachbarte Daten auf viele Speicherseiten verteilt sind, die dann vom System alle angefordert werden müßten.)

Der zweite Indextyp, der Non-Clusterded Index, kann beliebig oft verwendet werden, hierbei ist aber nicht sichergestellt – und bei mehreren Kriterien auch logisch ausgeschlossen –, daß benachbarte Daten sicher auf gleichen oder benachbarten Speicherseiten stehen.

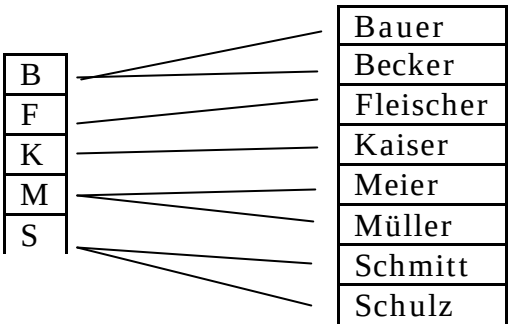


Ein Code-Beispiel zur Erläuterung mit VBA-Arrays:

```
Public Sub DemoIndex1()
Dim t() As String
Dim i As Long, n As Long

n = 12
ReDim t(1 To n)

t(1) = "Januar"
t(2) = "Februar"
t(3) = "März"
t(4) = "April"
t(5) = "Mai"
t(6) = "Juni"
t(7) = "Juli"
t(8) = "August"
t(9) = "September"
t(10) = "Oktober"
t(11) = "November"
t(12) = "Dezember"

'Brutale Methode: Reihenfolge anpassen
t(1) = "April"
t(2) = "August"
t(3) = "Dezember"
t(4) = "Februar"
t(5) = "Januar"
t(6) = "Juli"
t(7) = "Juni"
t(8) = "Mai"
t(9) = "März"
t(10) = "November"
t(11) = "Oktober"
t(12) = "September"

For i = 1 To n
    Debug.Print t(i)
Next i

End Sub

Public Sub DemoIndex2()
Dim t() As String
Dim z() As Long
Dim i As Long, n As Long

n = 12

ReDim z(1 To n)
ReDim t(1 To n)

t(1) = "Januar"
t(2) = "Februar"
```

```
t(3) = "März"
t(4) = "April"
t(5) = "Mai"
t(6) = "Juni"
t(7) = "Juli"
t(8) = "August"
t(9) = "September"
t(10) = "Oktober"
t(11) = "November"
t(12) = "Dezember"

'Die Information über die gewünschte alphabetische
'Sortierung steckt im Index-Array statt im Daten-
'Array
z(1) = 4
z(2) = 8
z(3) = 12
z(4) = 2
z(5) = 1
z(6) = 7
z(7) = 6
z(8) = 5
z(9) = 3
z(10) = 11
z(11) = 10
z(12) = 9

For i = 1 To n
    Debug.Print t(i), t(z(i))
Next i

't(i): Originalreihenfolge zum Vergleich; bleibt
'erhalten - t(z(i)): Nutzreihenfolge für Ausgabe

z(12) = 4
z(11) = 8
z(10) = 12
z(9) = 2
z(8) = 1
z(7) = 7
z(6) = 6
z(5) = 5
z(4) = 3
z(3) = 11
z(2) = 10
z(1) = 9

For i = 1 To n
    Debug.Print t(i), t(z(i))
Next i

End Sub
```

Wir haben hier natürlich nur an der Oberfläche gekratzt. Die tatsächlich eingesetzten Indextypen weisen in der Regel eine effizientere Baumstruktur auf:



Da man als Datenbankentwickler hierauf aber keinen Einfluß hat, soll das Thema nicht weiter vertieft werden

Was nützt ein Index?

Alle lesenden Zugriffe oder Berechnungen, die eine definierte Reihenfolge von Daten ausnutzen können, profitieren von einem Index, also Sortierungen, Filterungen, Minimums- und Maximums-Bestimmung u. ä.

Was schadet ein Index?

Schreibende Zugriffe werden durch Indizes verlangsamt, da die Indizes bei Eingabe neuer und Änderung vorhandener Daten mit dem neuen Bestand aktualisiert werden müssen.

Es ist also *keine* gute Idee, möglichst viele Felder zu indizieren. Im Hinblick darauf, daß bei den meisten Datenbanken die Lesezugriffe einen weitaus größeren Anteil am Datenbankgeschehen haben als die Schreibzugriffe, sollten aber die sinnvollen Indizierungen auf jeden Fall umgesetzt werden.

Welche Felder indizieren?

Welche Felder soll man nun indizieren? Grundsätzlich profitieren alle Vorgänge, bei denen es irgendwie auf die Reihenfolge der Daten ankommt, von einem Index. Es sind also alle

- Sortierfelder (ORDER BY)
- Kriterienfelder (WHERE)
- Gruppierfelder (GROUP BY)
- Referenzfelder (JOIN)

zu indizieren. Auch Felder, die in einem SELECT DISTINCT vorkommen, profitieren von einem Index.

Ein Index ist um so effizienter, je weniger Wertwiederholungen im indizierten Feld vorkommen. Ein Index auf *Firmenname* bringt mehr als einer auf *Anrede*, da Firmennamen sich kaum wiederholen, also auf ganz wenige Datensätze eingeschränkt wird, während die Information „Herr“ von einer Million Datensätzen immer noch circa 500.000 übrigläßt. Das heißt nicht, daß ein Index auf solchen Feldern nichts bringt – auf einem stärker distinktiven Feld bringt er aber unter Umständen erheblich mehr.

Mehrfelderindizes richtig planen

Ein Index über mehrere Felder unterstützt nicht nur eine Sortierung nach der Kombination aller beteiligten Felder, sondern auch von Teilmengen.

Angenommen, ein Index geht in dieser Reihenfolge über 4 Felder {A, B, C, D}. Von diesem Index profitieren auch Sortierungen nach den Teilmengen {A, B, C}, {A, B}, {A}. Wenn in diesen Reihenfolgen sortiert werden soll, ist ein zusätzlicher Index unnötig. Sortierungen nach {B, C, D}, {C, D}, {B, C}, {B}, {C}, {D} oder {B, A} würden hingegen nicht profitieren und bräuchten gegebenenfalls einen eigenen Index.

Die Regel dürfte klar sein: Feldkombinationen, die einen Mehrfelderindex ausnutzen können, sind alle, die ausgehend vom Indexkopf Folgefelder ohne Unterbrechung beinhalten.

Überlappende Indizes

Es kann sinnvoll sein, in solchen Fällen überlappende Mehrfelderindizes anzulegen. Um die sechs Sortierungen {A, B, C, D}, {A, B, C}, {A, B}, {A}, {C,B}, {C} zu unterstützen, wären zwei Indizes erforderlich: {A, B, C, D} und {C,B}.

Achtung! Man verwechsle die Feldreihenfolge in einem Mehrfelderindex nicht mit der Sortierreihenfolge (aufsteigend/absteigend) in einem Feld.

Clustered Index in Access

Im Gegensatz zum SQL-Server läßt sich in einer Access-Tabelle der Clustered Index, also die physische Sortierung nach Speicherseiten („gruppenbildender Index“, grob vereinfacht: die tatsächliche Datensatzreihenfolge), nicht einfach festlegen. Access benutzt vielmehr grundsätzlich den *Primärschlüssel* als Clustered Index. Die Clustered-Eigenschaft des Index-Objektes hilft uns hier leider nicht weiter.

Im Hinblick auf die allseits beliebten AutoWert-Felder liegt hier einiges Potential in der Regel brach. Die dadurch optimal unterstützte Datensatzreihenfolge ist die Eingabereihenfolge, die fast immer uninteressant ist.

Da andererseits AutoWerte als unsemantische (referentielle) Selektoren sehr fehlerrobust sind und als Long-Integer-Zahlen sehr schnell verarbeitet werden, möchte man aber ungern darauf verzichten, Beziehungen über sie herzustellen.

Freundlicherweise erlaubt sowohl die relationale Theorie im allgemeinen wie auch Access im besonderen, Beziehungen über jeden Kandidatschlüssel herzustellen. Die folgenden Beispiele zeigen darauf basierend, wie man in Access sowohl zu einem sinnvollen Clustered Index als auch zu einer Beziehung über AutoWerte kommt.

Beispiele zur Indizierung

Beispiel Stammdaten

Hier z. B. Personentabelle

- Felder
  - idPrs Autowert
  - dtAnrede Text
  - dtVorname Text
  - dtNachname Text
- Verwendung
  - idPrs Referenzfeld für Beziehungen, Joins
  - dtNachname Hauptsortierfeld
  - dtVorname Nachsortierfeld

Indizierung 1

- PrimaryKey (Unique, 3 Felder)
  dtNachname ASC, dtVorname ASC, idPrs ASC
- ReferenceKey (Unique, Required)
  idPrs ASC
- Kein Index auf Anrede, da kaum distinktiv

Ideal für Sortierung nach {Nachname}, {Nachname, Vorname}

Sortierung nach Vorname alleine profitiert kaum

Beziehungen werden über den Parallelschlüssel ReferenceKey/idPrs hergestellt.

Indizierung 2

- PrimaryKey (Unique, 3 Felder)
  dtNachname ASC, dtVorname ASC, idPrs ASC
- ReferenceKey (Unique, Required)
  idPrs ASC
- OrderKey1 (Ambique)
  dtVorname ASC
- **Kein** Index auf Anrede, da kaum distinktiv

Ideal für Sortierung nach (Nachname, Vorname), aber auch nur oder zuerst Vorname

Sortierung nach Vorname profitiert vom tabellierten (non-clustered) Index OrderKey1

Beispiel inhaltliche/benutzerdefinierte Sortierung

Hier z. B. Katalog-/Nachschlagetabelle, Kategorien o. ä.

- Felder
  - idKat (AutoWert)
  - dtKatBezeichnung (Text)
  - irKat (Long-Zahl)
- Verwendung
  - idKat: Referenzfeld
  - dtKatBezeichnung: Informelles Feld
  - irKat: Hauptsortierfeld

Indizierung

- PrimaryKey (Unique, 1 Feld)
  irKat ASC
- ReferenceKey (Unique)
  idKat ASC
- **Kein** Index auf KatBezeichnung, da nicht relevant

Ideal für Sortierung nach (irKat), also benutzerdefinierte/inhaltliche Reihenfolge

Warum zusätzlich Referenzschlüssel idKat?

Bei Reihenfolgenänderungen/Einfügen müssen keine Fremdschlüssel angepaßt werden – der große Vorteil des unsemantischen Schlüssels.

Beispiel Bewegungsdaten

Hier: Bestellungen

- Felder
    - idBest (AutoWert)
    - dtBestDatum (Datum)

Verwendung

idBest: Primärschlüssel, Referenzfeld,

dtBestDatum: Nachsortierfeld, Kriterienfeld

Indizierung

PrimaryKey & ReferenceKey (Unique, 1 Feld)
idBest ASC

Kein Index
dtBestDatum, wenn informell *

OrderKey1(Ambique, ASC(!))
dtBestDatum, wenn relevant *

Auch wenn man absteigend wird sortieren wollen, ist trotzdem ein aufsteigender Index anzulegen. Absteigende Indizes werden von Access nicht richtig genutzt.

* *Daten sind informell, wenn sie nur der Information des Benutzers dienen; sie sind relevant, wenn sie vom Db-System verarbeitet werden, also zum Sortieren, als Kriterien, zum Verknüpfen etc. benutzt werden.*

Indexbeschränkungen

Memo-Felder:

In Memo-Feldern sind nur die ersten 255 Zeichen indizierbar. Da Memo-Felder auch sonst einige Tücken haben und ihre Verwendung in 90% aller Fälle, spätestens aber wenn sie mit dem Wunsch nach einer Textsuche im Memo gepaart wird, ein Indikator für Entwurfsfehler im DB-Design ist, sollte man sich bei vermeintlicher Notwendigkeit für Memo-Felder dreimal fragen, ob das sein muß. Nur wenn die Antwort dreimal Ja lautet, sollte man es einsetzen.

Anzahl der Indizes:

In einer Tabelle können maximal 32 Indizes verwaltet werden. Die *unsichtbaren automatischen Fremdschlüsselindizes* (siehe übernächster Punkt) zählen hierbei mit.

Bis zum Fanatismus überzeugten Normalisierungsanhängern geht hier das Herz auf: Wer seine Datenbanken aus Hunderten Tabellen mit je nur einer Handvoll Feldern strickt, wird mit dieser Beschränkung keine Konflikte bekommen.

Anzahl der Felder

Ein Index kann über maximal 10 Felder gehen. Im Gegensatz zur 32-Indizes-Grenze ist das recht großzügig bemessen. Für einen notwendigen Mehrfelderindex über mehr als 10 Felder werden sich kaum sinnvolle Beispiele konstruieren lassen.

Automatischer Fremdschlüsselindex

Das Fremdschlüsselfeld in einer Beziehung (n-Seite) **muß** unbedingt indiziert sein. Diese Indizierung ist so wichtig, daß wir sie **nicht** setzen. Klingt verrückt? Nun, aufgrund der Bedeutung dieses Index setzt Access ihn beim Anlegen einer Beziehung selbst – ohne Aufhebens und unsichtbar. Ein bereits bestehender Index auf diesem Feld wird hierbei *nicht* genutzt. Der Index wird auch in der Indextabelle (Tabellenentwurf, Index) *nicht* angezeigt, er existiert aber. Mit der Indexes-Auflistung von DAO.TableDef läßt er sich anzeigen, das ist aber das einzige Indiz für seine Existenz.

```
Sub ShowIndizes()
Dim tbl As DAO.TableDef
Dim i As Long, n As Long
Dim id As DAO.Index
Set tbl = CurrentDb.TableDefs("Tabellenname")
For Each id In tbl.Indexes
    Debug.Print id.Name, id.Fields
Next id
End Sub
```

Der sonst unsichtbare Fremdschlüsselindex ist daran erkennbar, daß sein Name aus den Namen der beteiligten Tabellen besteht (Tabelle1Tabelle2) und seine Fields-Eigenschaft den Namen des Fremdschlüsselfeldes enthält.

Wenn wir das Fremdschlüsselfeld selbst indizieren, liegen auf diesem Feld überflüssigerweise zwei gleichwertige Indizes. Das bringt keinen Gewinn, aber Verluste durch den entstehenden doppelten Verwaltungsaufwand und reduziert unsinnigerweise die mit 32 ohnehin nicht sehr üppig bemessene Indexobergrenze.

Sonstige Join-Felder

Wenn man plant, Joins zu verwenden, die *nicht* über Fremdschlüsselfelder, also Felder, die in Beziehungen vorkommen, gehen – das ist zwar eher ungewöhnlich, aber keineswegs ausgeschlossen –, dann muß man ein solches Join-Feld natürlich selbst indizieren.

Aufsteigend indizieren

Theoretisch ist ein absteigender Index ebenso gut wie ein aufsteigender. Leider hat Access die Unart, absteigende Indizes häufig nicht zu benutzen, zum Beispiel in Kriterien, was sich mit dem JetShowPlan nachweisen läßt.

Absteigende Indizes werden allerdings bei einer absteigenden Sortierung benutzt. *Aber*: Mit einem *auf*steigenden Index wird eine *ab*steigende Sortierung genauso schnell vorgenommen!

Deswegen soll man alle Indizes aufsteigend anlegen, auch wenn man später absteigend nach diesem Feld sortieren will.

Zusammenfassung

Eine vernünftige Indexplanung und Durchführung ist Voraussetzung für alle folgenden Tips zur Abfrageoptimierung. Wer nicht alle erforderlichen Felder sachgerecht indiziert hat, braucht sich über Performance keine weiteren Gedanken zu machen, da dann sowieso Hopfen und Malz verloren ist.

Abfragedesign

Grundlagen

Nur benötigte Felder ausgeben

Felder, die im Abfrageergebnis nicht unbedingt gebraucht werden, sollte man weglassen. Viele spezialisierte Abfragen sind schneller als wenige Universalabfragen. Zudem sind sie ergonomischer, da man einen Benutzer, der eine Telephonnummer eines Kunden benötigt, nicht mit anderen Daten, die er im Moment gar nicht braucht, verwirren soll.

Das heißt natürlich nicht, daß man dem generischen Paradigma untreu werden und auf variable Parameter zugunsten von literalen Kriterien verzichten soll. Erfreulicherweise hinsichtlich Wartbarkeit und Arbeitserleichterung liegt hierin kein schädliches Potential.

Speziell bedeutet das:

- Kein *****
- Bedingungen ausblenden, wenn es keine Oder-Ausdrücke sind

Der Inhalt eines einfachen Bedingungsfeldes ist sowieso in allen DS gleich, also informationslos. Die Kategorie kann ihren Ausdruck im Namen der Abfrage finden oder als Eintrag eines Kombifeldes, über das die Kriterien gesteuert werden.

Nur benötigte Datensätze ausgeben

Was für Felder gesagt wurde, gilt in noch höherem Maße für die Beschränkung der ausgegebenen Datensätze. Daher

- Resultset mit Where einschränken

Das insbesondere, wenn *berechnete Felder* vorkommen. In je weniger Zeilen eine satzabhängige Berechnung durchgeführt wird, desto besser.

Wie man WHERE-Bedingungen ökonomisch aufbaut, wird weiter unten beschrieben.

Abfragen speichern

Eine gespeicherte Abfrage ist deutlich schneller als SQL-Code, der in VBA-Code dynamisch erzeugt wird; in meinen Tests um bis zu 400%

Man sollte daher annehmen, daß eine gespeicherte Abfrage auch als RecordSource eines gebundenen Formulars vorzuziehen ist. Das ist im Prinzip auch richtig. Daß ein in der Datensatzherkunft

eingetragener SQL-String dennoch unkritisch ist, liegt schlicht daran, daß Access hierfür heimlich unsichtbare Systemabfragen erstellt, die noch nicht einmal über Ansicht Ausgeblendete Objekte bzw. Systemobjekte sichtbar werden. Das gleiche gilt für Listen- und Kombifelder.

CurrentDb.QueryDefs oder ein Blick in die Systemtabelle MSysObjects enthüllt aber solche Abfragen, die an ihren Präfixen leicht zu erkennbar sind:

~sq_fFormularName bzw. ***~sq_c***FormularName***~sq_c***ListeName

Bei Einsatz älterer Access-Versionen überzeuge man sich, ob solche Systemabfragen bereits angelegt werden. Wenn nicht, verzichtet man auf SQL-Strings als Datenherkunft und speichert sie selbst als Abfragen.

Nur Indexfelder für mengenorientierte Operationen

Felder, die in einem

- JOIN

oder einer der Klauseln

- ORDER BY
- WHERE
- GROUP BY

Verwendung finden, müssen indiziert sein. Wenn solche Operationen auf nichtindizierten Feldern ausgeführt werden, muß Jet prinzipbedingt zu Sortier- und Join-Algorithmen greifen, deren Laufzeit mit steigender Datenmenge überproportional ansteigt.

Darüber hinaus sind solche Aktionen zu vermeiden auf *berechneten Feldern*, da diese keine Indexnutzung zulassen

Sortierung

Sortierungen sollte man auf das notwendige Mindestmaß beschränken. Wenn eine Ergebnismenge für interne Weiterverarbeitung per Code nicht sortiert sein muß, sollte man darauf ganz verzichten und nicht aus Gewohnheit einfach eine Sortierung festlegen.

Die beste Performance haben Sortierungen auf den wie oben gezeigt herbeigezauberten Clustered Index ohne ORDER BY (500%)

Das ist natürlich nur bei relativ konstanten Daten anwendbar, da neue Datensätze nicht sofort einsortiert werden.

Sortierungen sollen auch ansonsten nur auf Indexfeldern vorgenommen werden, die die betreffende Reihenfolge der Sortierfelder auch unterstützen, wie unter Indexplanung beschrieben wurde.

Viel mehr Gedanken muß sich übers Sortieren nicht machen, da man kaum weitere Gestaltungsmöglichkeiten hat.

Selektion

WHERE-Klauseln gestalten

Vergleichsoperatoren

Unproblematisch sind die folgenden Vergleiche:

- =
- <
- <=
- >
- >=
- Like Text*

Alle diese Operatoren erlauben die Ausnutzung eines bestehenden Index. Zum Like-Operator ist noch anzumerken, daß der Index bis zum Jokerzeichen, aber nicht darüber hinaus genutzt werden kann. Je weiter hinten dieses auftaucht, desto günstiger ist es also.

Problematisch sind:

- <>
- Like *Text

<> wird als Not = ausgeführt (siehe nächster Punkt), und bei Like *... wird in Fortsetzung des oben zu Like Gesagten ein Index überhaupt nicht mehr genutzt.

Wenn solche Filterungen nach hintenliegenden Feldinhaltsteilen systematisch auftauchen, ist die DB falsch normalisiert, und man sollte sich überlegen, wie man das Feld so aufteilen kann, daß Daten als ganze oder führende Feldinhalte auffindbar sind.

NOT vermeiden

Ein Not (Nicht) in einem Kriterium verhindert zuverlässig die Indexnutzung. Hierzu gehört auch die Verwendung von <>.

Solche Ausdrücke sollte man, wenn irgend möglich, mittels entsprechender logischer Gesetze umformen. Man findet weiter unten Beispiele hierzu, nebst den erzielbaren Laufzeitgewinnen.

Es gibt von dieser Regel aber eine wichtige Ausnahme: NOT in Verbindung mit dem EXISTS-Operator bei Unterabfragen ist unschädlich.

OR und AND verwenden

Häufig wird man Kriterien mit den logischen Operatoren Or und And kombinieren. Jet verwendet eine als *Rushmore* bezeichnete Methode, logische Ausdrücke auszuwerten, die besonders effizient ist.

Damit Jet diese Technologie anwenden kann, müssen die Ausdrücke bestimmte Kriterien erfüllen. Man findet unter dem Stichwort Rushmore in der Online-Hilfe einen Artikel dazu, dessen Inhalt hier kurz zusammengefaßt wird.

Ein *einfacher Ausdruck* hat die Form:

```
[Feld] Vergleichsoperator Ausdruck
```

Ein *einfacher optimierbarer Ausdruck* hat die Form

```
[IndiziertesFeld] Vergleichsoperator Ausdruck
```

Also zum Beispiel `[Nachname] = 'Müller'`, wenn Nachname indiziert ist.

Als Operatoren sind zulässig: <, >, =, <=, >=, Between…And, Like, In

In der Access-OH wird auch <> in dieser Liste aufgeführt. Das ist falsch. Mit <> wird nicht nur keine Rushmore-Optimierung durchgeführt, sondern der Index gar nicht mehr benutzt.

Ein *komplexer Ausdruck* ist aus einfachen Ausdrücken mit AND und OR zusammengesetzt. Es ergeben sich für einen zweiwertigen komplexen Ausdruck folgende Kombinationsmöglichkeiten:

- Optimierbarer Ausdruck AND/OR Optimierbarer Ausdruck
- Optimierbarer Ausdruck AND/OR Nichtoptimierbarer Ausdruck
- Nichtoptimierbarer Ausdruck AND/OR Nichtoptimierbarer Ausdruck

Natürlich kann man durch Verknüpfung komplexer Ausdrücke weitere Ausdrücke noch höherer Komplexität aufbauen.

Für das Optimierbarkeitsverhalten eines komplexen Ausdrucks gibt es drei Stufen:

- Vollständig optimierbar
- Teilweise optimierbar
- Nicht optimierbar

Wie sich diese ergeben, kann der folgenden Tabelle entnommen werden.

	Opt	Opt	Opt	NonOpt	NonOpt	NonOpt
AND	Vollständig		Teilweise		Nicht	
OR	Vollständig		Nicht		Nicht	
NOT	Nicht		Nicht		Nicht	

Mit Hilfe logischer Gesetze lassen sich Ausdrücke gegebenenfalls so umformen, daß sie optimierbar werden, zum Beispiel

NOT (x<5 OR y >3) → x>=5 AND y<=3

Der erste Ausdruck ist trotz angenommenem Index auf x und y nicht optimierbar, der zweite Ausdruck ist sogar vollständig optimierbar.

Für solche Umformungen gibt es Heerscharen logischer Theoreme, die praktisch wichtigsten, die jeder Programmierer im Schlaf beherrschen sollte, sind

- Distributivgesetz der Konjunktion und Disjunktion
  A OR (B AND C) = (A OR B) AND (A OR C)
  A AND (B OR C) = (A AND B) OR (A AND C)
- De Morgansche Gesetze
  NOT (A OR B) = NOT A AND NOT B
  NOT (A AND B) = NOT A OR NOT B
- Gesetz von der Doppelnegation
  NOT NOT A = A

Kriterienreihenfolge

Man hört gelegentlich den durchaus plausiblen Tip, man solle die Reihenfolge einschränkender Kriterien so festlegen, daß die größten Einschränkungen zuerst kommen. Das ist im Prinzip richtig, aber dennoch überflüssig.

Zum einen hängt die einschränkende Wirkung von Kriterien vom Datenbestand ab, ist also dem Entwickler im Zweifel nicht bekannt.

Da zum anderen das DB-System interne Statistiken führt, kann es die optimale Kriterienreihenfolge sehr wohl beurteilen und legt diese unabhängig von der Reihenfolge in der Abfrage auch zugrunde.

Man muß sich hierüber also keinen Kopf machen und kann das Denken in diesem Falle Jet überlassen.

OR oder IN

Auch die allgemeine Empfehlung, IN statt OR zu verwenden, ist obsolet, da Jet auch hier unabhängig vom SQL-Code selbständig die bessere Variante mit IN einsetzt.

Vermeidung von OR durch UNION

Gelegentlich kann man über diesen Tip stolpern, allerdings im Zusammenhang mit anderen DB-Systemen. Es ist *keine gute Idee*, dieses Vorgehen für Jet zu adaptieren. Im Gegensatz zu den Systemen, die diesen Tip rechtfertigen, ist Jet – Rushmore sei Dank – durchaus in der Lage, OR-Ausdrücke mit Index zu nutzen und zu optimieren.

Eine Abfrage mit OR ist deutlich schneller als eine ergebnisäquivalente Union-Abfrage.

BETWEEN ... AND oder AND

BETWEEN 1 AND 5 ist (minimal) besser als >=1 AND <=5

WHERE auf berechneten Feldern vermeiden

Bereits einfachste Rechnungen verhindern Indexnutzung. Man kann sich diesen Effekt zum Testen zunutze machen und durch Zahlfeld + 0 oder Textfeld & '' ausprobieren, wie sich ein indiziertes Feld ohne Index verhalten würde.

Als Abhilfe kann man nur versuchen, die Bedingung auf die indizierten(!) Felder, die in die Berechnung eingehen, umzuformen.

Das geht grundsätzlich bei eindeutig umkehrbaren Funktionen.

Gegebenenfalls muß man auch das Tabellendesign entsprechend anpassen.

Angenommen, eine Tabelle enthält Maße von Quadern, deren Länge, Breite, Höhe und Volumen verwaltet werden soll. Zusätzlich ist die Anforderung bekannt, daß das Volumen in WHERE-Klauseln verwendet werden soll, aber nicht Länge, Breite und Höhe.

Statt in der Tabelle (l, b, h) zu speichern und V in einer Abfrage zu berechnen, würde man in der Tabelle zum Beispiel (V, l, b) speichern und dann h in einer Abfrage berechnen.

Bei nicht eindeutig umkehrbaren Funktionen kann es helfen, sie in Monotonie-Intervalle zu zerlegen und daraus eigene Bedingungen zu formen.

Angenommen, es werden die Datensätze gesucht, bei denen das Quadrat eines Feldes x größer als 4 ist. Zunächst ist die Funktion x^2 nicht eindeutig umkehrbar, aber sie läßt sich in die Monotonieintervalle -oo bis 0 und 0 bis oo zerlegen.

Also statt (x * x) > 4 besser x < -2 Or x > 2, womit wir uns, von verhinderter Indexnutzung ausgehend, so erheblich verbessern, daß sogar eine Rushmore-Optimierung möglich wird.

WHERE mit Formularbezug

Der Bezug des Kriterienwertes aus einem Steuerelement in einem Formular ist unproblematisch und in der Regel auch sowieso aus ergonomischen Gründen unumgänglich.

Beispiele zur Optimierung

✘ Schlecht

✔ Gut

Alle Beispiele basieren auf Verwendung indizierter Felder.

Prüfung auf NULL

✘ `IsNull([Feld])`

✔ `[Feld] Is Null`

Geschwindigkeitsvorteil ca. 1500%

Prüfung auf Nicht NULL

✘ `NOT Is Null`

Bei Text

✔ `>= ''` (Achtung! ***Kein*** Leerzeichen zwischen '')

Bei Zahlen

✔ `<0 OR >=0`

Geschwindigkeitsvorteil ca. 700%

Prüfung auf ungleich Zahl

✘ `x<>5 (NOT x=5)`

✔ `x<5 OR x>5`

Geschwindigkeitsvorteil ca. 600%

Prüfung auf Jahr

✘ `Year([Datum]) = 2004`

✔ `#1/1/2004# <= [Datum] AND [Datum] <= #12/31/2004#`

Oder

✔ `[Datum] BETWEEN #1/1/2004# AND #12/31/2004#`

Geschwindigkeitsvorteil ca. 2700%

Prüfung auf 0, Positiv, Negativ bei berechneten Feldern

Felder a, b, x: a * b

✘
```
x=0
x>0
x<0
```

✓
```
a=0 OR b=0
(a>0 AND b>0) OR (a<0 AND b<0)
(a>0 AND b<0) OR (a<0 AND b>0)
```

Geschwindigkeitsvorteil ca. 700%

Prüfung auf nicht in Menge

✗ `NOT IN (3, 4, 5)`

`NOT IN (3, 4, 5)` → `NOT (x=3 OR x=4 OR x=5)`
(Auflösen von IN)

`NOT (x=3 OR x=4 OR x=5)` → `NOT x=3 AND NOT x=4 AND NOT x=5`
(De Morgan)

`NOT x=3 AND NOT x=4 AND NOT x=5` → `x<>3 AND x<>4 AND x<>5`
(Auflösen von Nicht gleich in Ungleich)

`x<>3 AND x<>4 AND x<>5` → `(x<3 OR x>3) AND (x<4 OR x>4) AND (x<5 OR x>5)`
(Auflösen von Ungleich in Größer oder Kleiner)

✓ `(x<3 OR x>3) AND (x<4 OR x>4) AND (x<5 OR x>5)`

Gewinn ca. 500%

Ein weiteres Beispiel zum Ersetzen von NOT IN, bei dem die Kriterienmenge ein Resultset ist, im Kapitel Unterabfragen.

Prüfung auf nicht in Bereich

✗ `NOT BETWEEN 3 AND 12`

```
NOT BETWEEN 3 AND 12
→ NOT (x>=3 AND x<= 12)
→ NOT x>=3 OR NOT x<=12
→ x<3 OR x>12
```

✓ `x<3 OR x>12`

Gewinn ca. 500%

Wertprüfung auf Eingangsvariable verschieben

Felder x, y mit y: x * 0,12 + 24
→ x= (y - 24) / 12 * 100

✗ `y > 100`

✓ `x > 633`

Gewinn ca. 3700%

Redundante Bedingungen ausklammern

✗ (A=3 AND B = 2 AND C = 5) OR (A=3 AND B = 2 AND C = 12)

✓ A=3 AND B = 2 AND (C=5 OR C=12)

Die erste Zeile entsteht recht gerne, wenn Bedingungen mit dem graphischen Abfragedesigner erstellt werden.

Gewinn ca. 20%, A, B, C alle indiziert.

Wenn Nichtindexfelder und Indexfelder gemischt vorliegen, kann die Umstellung dazu führen, daß statt gar keiner Indexnutzung im besten Fall sogar auf Rushmore umgestellt werden kann. Der Vorteil kann also durchaus größer werden.

Prüfung auf Textanfang

✗✗ Left(Feld,1) = 'M'

✗ Feld Like 'M*'

✓ Feld >= 'M' And Feld < 'N'

Gewinn ca. 18900% Version 3 gegen Version 1; 800% Version 3 gegen Version 2

Kalkulation

Das Thema hat mittlerweile weitaus weniger Bedeutung, als in älteren Access-Versionen. Der Jet-Expression-Service ist im Laufe der Zeit spürbar verbessert worden. Dennoch sollen einige Tips zur Optimierung von Rechenausdrücken gegeben werden, wiewohl sie in modernen Versionen von geringerem Interesse sind.

Berechnete Felder

Text vs. Zahl

Gängige Datentypen in absteigender Reihenfolge

- Ganzzahl
- Fließkommazahl/Datum
- Text

Gängige Operatoren in absteigender Reihenfolge

- + - * \ Mod mit (Ganz-)Zahlen
- / und &,sowie + mit Text, Iif (neue Versionen)
- Nz, Iif (alte Versionen), vom Expression Service verfügbar gemachte Funktionen, wie Left, Mid, Right, Year, Month
- Selbstgeschriebene VBA-Funktionen

Ausdrücke umformen

Nachfolgend einige Beispiele, die vor allem in älteren Versionen nützlich sein können. Man prüfe im Einzelfall, was unter den gegebenen Voraussetzungen schneller ist.

Durch eine Anzahl teilen, die auch 0 sein könnte. Dann soll als Rückgabe der NULL-Wert erfolgen:

✘ Wenn(x=0; Null; y/x)

✓ y/(x Or Null)

Diesen Ausdruck nicht verwenden, wenn x auch Werte -1<x<0 Und 0<x<1 annehmen kann.

Negative Zahlen sollen auf 0 gesetzt werden:

✘ Wenn(x<0;0;x)

✓ (x>0) * -x

Negative Zahlen sollen auf NULL gesetzt werden:

✘ Wenn(x<0;Null;x)

✓ ((x>0) oder Null) * -x

Der Wert 1 soll eine 7, der Wert 2 eine 12 ergeben (z. B. Rabattkategorien):

✘ Wenn (x=1;7;Wenn(x=2;12;0))

besser Linearkombination mit Wahrheitswerten:

✓ -(x=1)*7-(x=2)*12

oder, wenn sichergestellt ist, daß x ausschließlich 1 oder 2 sein kann, Polynom:

✓ 5*x+2.

Falls der Definitionsbereich für x auf {0, 1, 2} liegt, dann

✓ -x*(x-8)

Man kann n beliebige Bedingungen immer als n-dimensionale Linearkombination ausdrücken und n Bedingungen vom Typ "=" und abgeschlossener Definitionsmenge als Polynom schlimmsten-falls (n-1)ten Grades.

Leerzeichen automatisch ausfüllen:

✘ Wenn(IstNull(Titel);"";Titel & " ") & Vorname & " " & Nachname

✓ (Titel + " ") & Vorname & " " & Nachname)

In der Verkettung mit "&" verhält sich der Nullwert als neutrales, in der Verkettung mit "+" als dominantes Element

Das Geschlecht ist mit 1 (männlich) und 2 (weiblich) codiert. Daraus sollen die Buchstaben "m" bzw. "w" erzeugt werden:

✘ Wenn(x=1;'m';'w')

✓ Chr(10*x+99)

Das Geschlecht ist mit m und w codiert. Daraus sollen die Kennzahlen 1 bzw. 2 erzeugt werden:

✘ Wenn(x='m';1;2)

✔ (Asc(x)-99)/10

NULL-Wert in Stringfeld in Leere Zeichenfolge konvertieren

✘ Nz(Feld;'')

✔ '' & Feld

Potenzen in Multiplikation auflösen

✘ x^2

✔ x * x

Exp und Log wenn möglich vermeiden

✘ Exp(2 * Log(x))

✔ x * x

Alternative: Berechnung im Formular

Rechenausdrücke in Formularen werden zwar nicht schneller berechnet als in Abfragen, aber wenn die Abfrage als Basis eines Endlosformulars dient, ist der Bildaufbau bei Formularberechnungen erheblich besser.

Zusammenfassung

Die benötigte Zeit für Berechnungen mit Feldwerten innerhalb eines Datensatzes ist bei aktuellen Access-Versionen in der Regel unkritisch. Man kann weitaus kapitalere Böcke schießen, zum Beispiel eine Bedingung in ein solches Feld setzen.

Domänenfunktionen

Statt der Verwendung von Domänenfunktion (DomWert, DomSumme, etc.) sollte man die bezogene Tabelle mittels eines Joins verknüpfen und gegebenenfalls passende SQL-Aggregat-Funktionen einsetzen oder eine Unterabfrage konstruieren. Domänenfunktionen sind aus verschiedenen Gründen normalerweise die langsamste aller Möglichkeiten ein Ergebnis zu erzielen.

Verbund

JOINS gestalten

Bei vielen DB-Systemen liegt hier einiges Optimierungspotential. Bei modernen Jet-Versionen wird jedoch erstaunlich viel vom System automatisch optimiert.

Der grundlegende Ratschlag, einen Join nicht klassisch über WHERE, sondern mit einem echten JOIN aufzubauen, ist für Jet-Abfragen irrelevant.

Klassischer Inner Join

```
SELECT
     A.Feld1, B.Feld2, …
FROM
     A, B
WHERE
     A.id = B.id
```

Expliziter Inner Join

```
SELECT
     A.Feld1, B.Feld2, …
FROM
     A INNER JOIN B
     ON A.id = B.id
```

Beide Syntaxvarianten werden von Jet in der optimalen Form mit INNER JOIN ausgeführt!

Das soll kein Plädoyer für die alte Syntax (erstes Beispiel) sein, zeigt aber, wie weit die Optimierung durch Jet geht.

JOINS mit Bedingungen in einem Datenfeld

Prinzipiell bestünden die zwei Möglichkeiten, Zusatzbedingungen in der Join-Bedingung oder in einer WHERE-Klausel unterzubringen:

```
SELECT …
FROM A INNER JOIN B ON A.id = B.id AND A.Krit = 3
```

versus

```
SELECT …
FROM A INNER JOIN B ON A.id = B.id
WHERE A.Krit = 3
```

oder ganz klassisch

```
SELECT …
FROM A, B
WHERE A.id = B.id AND A.Krit = 3
```

Tatsächlich werden alle drei Varianten ***genau gleich ausgeführt***:

```
01) Restrict rows of table A
      using rushmore
      for expression "A.id=3"
02) Inner Join result of '01)' to table 'B'
      using index 'AidBid'
      join expression "A.id=B.id"
```

Es sei noch einmal daran erinnert, daß das Feld A.Krit natürlich *indiziert* sein muß, um das Ergebnis „using rushmore“ oder wenigstens „using index“ zu erzielen.

Das optimale Vorgehen, erst die einschränkenden Bedingungen je Tabelle auszuwerten, und dann die übrigen Datensätze zu

verknüpfen, wird von Jet in jedem Fall, unabhängig von der Formulierung der Bedingung, durchgeführt.

JOINS mit Bedingungen im Verknüpfungsfeld

Optimierungspotential gibt es hingegen beim Join mit Kriterien im *Verknüpfungsfeld über eine 1:n-Beziehung*; wahrscheinlich der häufigste Fall in einer Datenbank.

```
SELECT …
FROM
     Detail INNER JOIN Master
     ON Detail.fi = Master.id
WHERE
     Detail.fi<10
```

oder

```
SELECT …
FROM
     Detail INNER JOIN Master
     ON Detail.fi = Master.id
WHERE
     Master.id<10
```

Beide Abfragen liefern das gleiche Ergebnis, die zweite, mit der Einschränkung auf der 1-Seite, tut das aber 100 bis 200 % schneller.

Die Ausführungspläne beider Abfragen unterscheiden sich auch durchaus.

```
01) Restrict rows of table Details
      using rushmore
      for expression "Detail.fi<10"
02) Inner Join table 'Master' to result of '01)'
      merging indexes
      join expression "Master.id=Detail.fi"
```

```
01) Restrict rows of table Master
      using rushmore
      for expression "Master.id<10"
02) Inner Join result of '01)' to table 'Detail'
      using index 'Detail!fi'
      join expression "Master.id=Detail.fi"
```

Die Ex-post-facto-Rationalisierung fällt in Kenntnis der Ergebnisse leicht: auf der 1-Seite kommt ein bestimmter Wert natürlich seltener (einmal) vor, als auf der n-Seite, wodurch die Beschränkung effektiver ist.

Tabellenreihenfolge

Hier ergab sich ein etwas uneinheitliches Bild.

1) (A × B) × C

```
SELECT …
FROM
     (
     A INNER JOIN B
     ON A.id = B.id
     )INNER JOIN C
     ON A.id = C.id
```

2) (A × C) × B

```
SELECT …
FROM
     (
     A INNER JOIN C
     ON A.id = C.id
     ) INNER JOIN B
     ON A.id = B.id
```

Beide Abfragen liefern *dieselbe Ergebnismenge* und *sehen im graphischen Entwurf genau gleich aus*.

Bei einigen Fällen wurden die Abfragevarianten von Access im SQL-Code auf die gleiche Syntax umgeschrieben, bei anderen blieb die unterschiedliche Klammersetzung erhalten, aber die Ausführungspläne waren identisch und dann gab es auch noch das Ergebnis, daß die Ausführungspläne bei beiden Abfragen verschieden waren und die Laufzeit ebenfalls.

Wann welches Verhalten auftritt, scheint in erster Linie damit zusammenzuhängen, ob zwischen den im Join verwendeten Feldern *Beziehungen mit referentieller Integrität* bestehen.

Eine Änderung der Klammerung, also der Reihenfolge der Joins, ist im graphischen Entwurf nicht möglich, sondern muß im SQL-Code erfolgen.

Als Faustregel für die Reihenfolge der Joins gilt, daß man den Join mit der größeren *Selektivität*, also den, der die kleinere Ergebnismenge liefert, nach innen klammert, und damit zuerst ausführen läßt. Das sind – Vorsicht, Falle! – keineswegs unbedingt die *Tabellen mit den wenigsten Datensätzen*, sondern die *Tabellenpaare mit den wenigsten Übereinstimmungen* im Verknüpfungsfeld.

Das ohne empirischen Test einzuschätzen fällt nicht leicht, da die Selektivität von der Datenverteilung abhängt. Man prüfe also die beiden Abfragen, die sich aufgrund der mittleren Tabelle ergeben, auf die Anzahl der zurückgegebenen Datensätze.

Im Beispiel wären das A × B und A × C, da A mit beiden anderen Tabellen verknüpft wird.

Man beachte, daß Jet beim Vorkommen von Outer Joins unter Umständen nicht alle Reihenfolgen zuläßt.

Wenn alle Joins über Beziehungen mit referentieller Integrität laufen, scheint man sich darauf, daß Jet selbst die optimale Join-Reihenfolge findet, verlassen zu können. Ein Eingriff in den SQL-Code, um die Reihenfolge zu beeinflussen, ist dann obsolet.

Man sollte sicherheitshalber die Ausgabe des Showplan zu Rate ziehen, damit man weiß, was passiert, und nicht auf Vermutungen angewiesen ist

Vergleichsoperatoren

Wenn möglich soll man Equi Joins verwenden, also solche mit = im Verknüpfungsausdruck.

Theta Joins, also Vergleiche wie

- A.id < B.id
- A.id <= B.id
- A.id = B.id *2

führen zu einem Cross Join ohne Indexnutzung.

INNER und OUTER Joins

Outer Joins sind etwas langsamer als Inner Joins. Da sie aber nicht durcheinander ersetzbar sind, wird man den Outer Join in Kauf nehmen, wenn eine Anfrage ihn erfordert. Immerhin ist der Outer Join zur Indexnutzung in der Lage; er rutscht also keinesfalls in die Liga des Cross Join ab.

Falls man jedoch im Einzelfall einen Outer Join benutzt hat, der aufgrund der Datenlage (Eingabepflicht) kein anderes Ergebnis als ein Inner Join liefern kann, sollte man ihn dadurch ersetzen.

JOIN oder Unterabfrage

Die Legende besagt, daß Unterabfragen oft sehr langsam seien und ein Join auf jeden Fall die bessere Wahl. In der Tat trifft es zu, daß es häufig Fälle gibt, in denen beim Vergleich Join vs. Subquery der Join erheblich besser abschneidet. Es gibt allerdings auch Konstellationen, in denen das Gegenteil richtig ist.

Weiter unten wird der Klärung der Frage, wann eine Subquery und wann ein Join Vorteile hat, in einem eigenen Kapitel nachgegangen.

Gruppierung

Indexfelder benutzen

Repetitiones non placent – Wiederholungen gefallen nicht, sagt der Römer. Er sagt aber auch: Qui nocent, docent – Leiden sind Lehren.

Also zum wiederholten Male: Gruppierungen sollte man nur auf indizierten Feldern vornehmen.

WHERE statt HAVING

Mit Where werden Kriterien vor der Gruppierung angewandt, mit Having danach. Rein logisch macht Having also nur Sinn, wenn es auf einem Feld benutzt wird, das erst im Laufe einer Gruppierung durch Aggregation entsteht.

Mit anderen DB-Systemen und auch in älteren Access-Versionen sollte man das beherzigen, in neueren denkt Jet mit. Eine unsinnige Having-Klausel wird bei der Ausführung einfach zu Where übersetzt.

Man kann das Verhalten mit dem ShowPlan überprüfen und seine Abfragen entsprechend gestalten.

Normalerweise kommt Having auf Nichtaggregatfeldern auch nicht absichtlich zustande, sondern durch eine Eigenheit des graphischen Abfragedesigners:

Ein Eintrag in der Kriterienzeile führt bei nichtgruppierten Abfragen zu einer Where-Klausel, bei gruppierten unsinnigerweise aber zu einer Having-Klausel. Die Where-Klausel ist hier hinter der „Funktion“ Bedingung versteckt.

Folgegruppen vermeiden

Angenommen, man nimmt eine Gruppierung nach Kundennummern vor, um Umsatzsummen aus Aufträgen oder ähnliches zu berechnen.

idKd	SumeVonAuftragswert
3	5000
5	3500
12	8000

```
SELECT
     K.idKd,
     Sum(A.Auftragswert) As SumeVonAuftragswert
FROM
     Kunde AS K INNER JOIN Auftrag AS A
     ON K.idKd = A.fiKd
GROUP BY
     K.idKd
```

Graphisch:

Feld:	idKd	Auftragswert		
Tabelle:	K	A		
Funktion:	Gruppierung	Summe		

Nun soll aber noch der Name des Kunden angezeigt werden:

idKd	Name	SumeVonAuftragswert
3	Donaubauer	5000
5	Habermacher	3500
12	Zimmermann	8000

✘
```
SELECT
     K.idKd,
     K.Name,
     Sum(A.Auftragswert) As SumeVonAuftragswert
FROM
     Kunde AS K INNER JOIN Auftrag AS A
     ON K.idKd = A.fiKd
GROUP BY
     K.idKd, K.Name
```

Graphisch:

Feld:	idKd	Name	Auftragswert	
Tabelle:	K	K	A	
Funktion:	Gruppierung	Gruppierung	Summe	

Das dürfte der meist eingeschlagene Weg sein. Da der Name aber anhand der Kundennummer schon eindeutig feststeht, ist eine Folgegruppierung danach nicht nur sinnlos, sondern sogar schädlich, da Gruppierungen einigen Rechenaufwand verursachen.

Besser ist

✓ (✘ Es geht gleich noch besser)
```
SELECT
     K.idKd,
     First(K.Name) As KName,
     Sum(A.Auftragswert) As SumeVonAuftragswert
FROM
     Kunde AS K INNER JOIN Auftrag AS A
     ON K.idKd = A.fiKd
GROUP BY
     K.idKd
```

Graphisch:

Feld:	idKd	Name	Auftragswert	
Tabelle:	K	K	A	
Funktion:	Gruppierung	ErsterWert/Min	Summe	

First gibt einfach nur den zufällig ersten Wert aus. Da nach der Gruppierung zum Beispiel nach idKd = 3 alle folgenden Namenwerte ‚Donaubauer' lauten, spielt es überhaupt keine Rolle, welcher davon genommen wird.

Der erste ist gerade recht, weil er einfach am Anfang steht und nicht gesucht werden muß.

Min gibt den kleinsten Wert aus, bei lauter identischen Werten spielt das also auch keine Rolle.

Man hat in dieser Konstellation also die Qual der Wahl. Das ist nicht zu verallgemeinern – normalerweise haben First und Min *unterschiedliche* Auswirkungen, aber wenn alle zur Auswahl stehenden Werte durch die Vorausgruppierung sowieso gleich sind, tun sie dasselbe.

Welches soll man also nehmen?

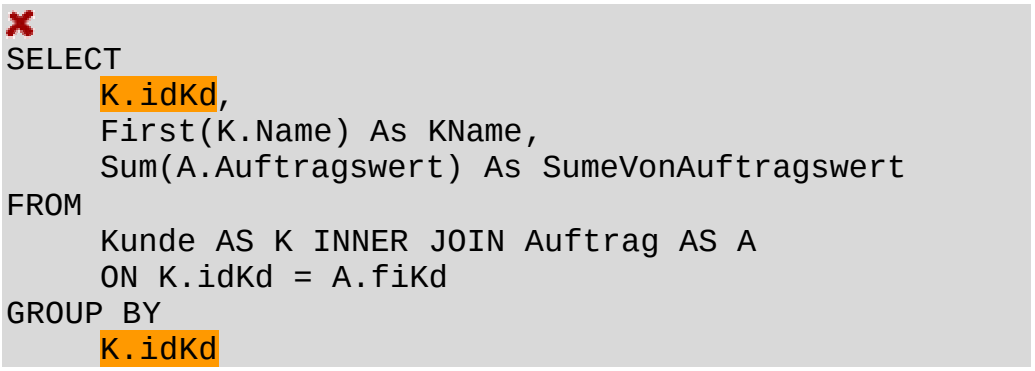
Wenn das Folgefeld, hier Name, *nicht indiziert* ist, ist normalerweise *First* schneller, wenn es *indiziert* ist, ist *Min* schneller.

Der Gewinn von First/Min gegenüber Group By beträgt bei wenigen Daten auf einem Folgefeld etwa 30%, bei großen Datenmengen und mehreren Folgefeldern kann da einiges zusammenkommen.

Ob First oder Min im Einzelfall schneller ist, sollte man am besten ausprobieren.

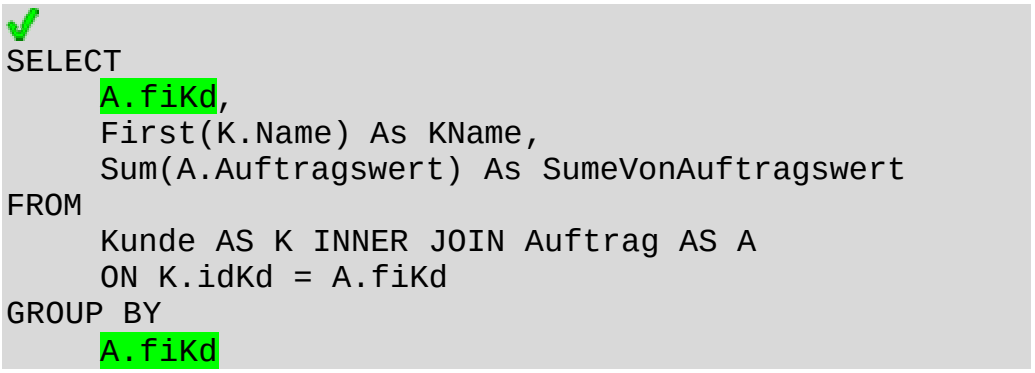
Group By im Join

Das Ergebnis von eben kann man noch einmal deutlich verbessern, wenn man die Gruppierung auf der richtigen Seite der Verknüpfung anlegt. Also nicht wie oben



```
SELECT
     K.idKd,
     First(K.Name) As KName,
     Sum(A.Auftragswert) As SumeVonAuftragswert
FROM
     Kunde AS K INNER JOIN Auftrag AS A
     ON K.idKd = A.fiKd
GROUP BY
     K.idKd
```

sondern



```
SELECT
     A.fiKd,
     First(K.Name) As KName,
     Sum(A.Auftragswert) As SumeVonAuftragswert
FROM
     Kunde AS K INNER JOIN Auftrag AS A
     ON K.idKd = A.fiKd
GROUP BY
     A.fiKd
```

Wenn man die Gruppierung statt auf der 1-Seite auf der n-Seite vornimmt, bringt das ab 40% - 60% aufwärts.

Umgang mit Aggregatfunktionen

Count(*)

Abweichend von der Grundregel ganz zu Anfang „* vermeiden“, gilt bei der Aggregatfunktion Anzahl (Count), daß Count(*) besser als Count([Feld]) ist, da die Count(*)-Syntax speziell für Rushmore optimiert wurde.

„Echte“ Aggregate

… wie Summe, Durchschnitt sowie First/Last profitieren nicht vom Index. Für eine Summe beispielsweise ist ein Index belanglos, da in jedem Fall alle Werte addiert werden müssen, wobei deren Reihenfolge, Kommutativgesetz sei Dank, keine Rolle spielt. Felder, die also nur für solche Aggregate benötigt werden, muß man deswegen nicht indizieren.

„Unechte“ Aggregate

… wie Min, Max, Count profitieren enorm. In einem indizierten Feld reduziert sich der zur Minimumsbestimmung nötige Algorithmus beispielsweise auf das schlichte Lesen des ersten Eintrags. Solche Felder sollen also indiziert werden.

Eindeutige Datensätze

Um aus einem Feld mit Wertwiederholungen eindeutige Datensätze zu bekommen, bieten sich zwei Vorgehensweisen an:

```
SELECT Feld FROM Tabelle GROUP BY Feld
```

oder

```
SELECT DISTINCT Feld FROM Tabelle
```

Der Vorteil von DISTINCT gegenüber GROUP BY liegt hier bei 1400%

UNION-Abfragen

Durch E-R-Modell vermeiden

Rund drei Viertel aller Union-Abfragen entspringen erfahrungsgemäß einem falschen Datenmodell. Oft gesehene Anforderungen sind zum Beispiel:

Man möchte für eine Telephonliste Felder aus einer Kundentabelle und einer Lieferantentabelle untereinander bringen oder Adressen aus einer Tabelle Dozenten und einer Tabelle Studenten für ein Rundschreiben o. ä.

Ein richtiges Modell würde beispielsweise eine Tabelle Firmen führen, die gemeinsame Datenfelder von Kunden und Lieferanten enthält, und die kunden- und lieferantenspezifischen Daten in Spezialisierungstabellen mittels je einer 1-1/0-Beziehung anhängen.

Das, was im fehlerhaften Modell durch die Union-Abfrage erreicht werden soll, liegt im richtigen Modell schon als Tabelle vor.

Natürlich ist ein sauberes Modell noch etwas komplexer, da Kunden auch Personen sein können. Da das hier nicht unser Thema ist, sei auf mein Skript zur Datenbanknormalisierung verwiesen, das den korrekten Tabellenaufbau beschreibt.

Als Geschwindigkeitsvorteile durch diese Maßnahme konnten Gewinne von 1200% - 1500% gemessen werden.

UNION ALL

Bei den Union-Abfragen, die auch in einem ordentlichen Datenmodell auftauchen, sollte man statt Union Select besser Union All Select benutzen.

Dazu muß man wissen, daß in der einfachen Union ein Distinct schon eingebaut ist; mehrfache gleiche Datensätze werden also aus dem Ergebnis eliminiert. Wenn das Auftreten von Dubletten nicht stört oder wegen der Datenlage solche sowieso nicht auftauchen, kann man durch Union All diese Automatik außer Kraft setzen.

Der Geschwindigkeitsvorteil hierdurch bewegt sich in der Größenordnung 80%

Unterabfragen

Funktionsweise

Eigenständige Query als

- Vergleich in WHERE-Klausel
- Herkunft eines einzelnen Feldes, wenn eindeutig
- Datenherkunft im FROM (Derived Table, eigentlich keine UA im engeren Sinn)

Vor- und Nachteile

Die größte Hürde ist zweifellos, daß von graphischen Query-Designern Verwöhnte textorientiert SQL schreiben müssen. Hinzu kommen die für Unterabfragen spezifischen Syntaxbestandteile wie EXISTS und die Mengenprädikate.

Wenn man dieses rein organisatorische Problem dadurch gelöst hat, daß man endlich einmal den guten Vorsatz zum Jahreswechsel, die

komplette SQL-Syntax auswendig zu lernen, eingelöst hat, steht einem sinnvollen Einsatz wenig im Wege.

Man unterscheidet:

- *Korrelierte* UA (**mit** Bezug auf Hauptabfrage)
  Diese werden n-mal ausgeführt → Alle Performance-Regeln in der UA beachten!
- *Unkorrelierte* UA (**ohne** Bezug auf Hauptabfrage)
  Diese werden nur einmal ausgeführt und stellen ihr Ergebnis allen DS der Hauptabfrage zur Verfügung

Manche Anfragen sind nur mit Unterabfragen lösbar.

Umformung in JOINS

Korrelierte Unterabfragen sind in der Regel auch als Join formulierbar, was oft auch die schnellere Alternative ist. Man sollte das jedoch nicht ungeprüft nachbeten, es gibt auch genug Fälle, in denen es anders herum ist. Man vergewissere sich im Einzelfall, welche Lösung am günstigsten ist.

Kriterium in Fremdtabelle

Eine der typischen Anwendungen für Subqueries ist das Filtern eines Resultsets nach einem Kriterium, das aus einer anderen Tabelle bezogen wird.

Beispiel:

- Vorgang (idVrg, VrgBezeichnung, fiPrs)
- Personal (idPrs, PrsName,fiAbt)
- Abteilung (idAbt, AbtName)

mit Vorgang n:1 Personal n:1 Abteilung

Anfrage: Suche alle Vorgänge, die zur Abteilung 3 gehören. Die Information ist als Fremdschlüssel fiAbt in Personal enthalten, allgemein „Filtere Daten aus Tabelle A nach einem Kriterium in einer Tabelle B, die mit A verknüpft ist“

Mit Unterabfrage in der WHERE-Klausel und IN:

```
SELECT
     Vorgang.idVrg, Vorgang.VrgBezeichnung
FROM
     Vorgang
WHERE
     Vorgang.fiPrs IN
     (
     SELECT
          Personal.idPrs
     FROM
          Personal
     WHERE
          fiAbt = 3
)
```

Mit Unterabfrage in der WHERE-Klausel und Mengenprädikat:

```
SELECT
     Vorgang.idVrg, Vorgang.VrgBezeichnung
FROM
     Vorgang
WHERE
     Vorgang.fiPrs = SOME
     (
     SELECT
          Personal.idPrs
     FROM
          Personal
     WHERE
          fiAbt = 3
)
```

Mit Unterabfrage mit EXISTS:

```
SELECT
     Vorgang.idVrg, Vorgang.VrgBezeichnung
FROM
     Vorgang
WHERE EXISTS
     (
     SELECT
          Personal.idPrs
     FROM
          Personal
     WHERE
          Personal.idPrs = Vorgang.fiPrs
     AND
          Personal.fiAbt = 3
     )
```

Diese typischen Subqueries lassen sich als Join so formulieren:

```
SELECT
     Vorgang.idVrg, Vorgang.VrgBezeichnung
FROM
     Personal INNER JOIN Vorgang
     ON Personal.idPrs = Vorgang.fiPrs
WHERE
     Personal.fiAbt = 3
```

Die Join-Variante ist um etwa 200% schneller als die Subqueries.

An dieser Stelle sei auch noch einmal an den *alternativen Schlüsselaufbau* erinnert, bei dem Detailtabellen alle Schlüsselinformationen aus allen übergeordneten Tabellen erben. Damit wäre die Anfrage ohne jeden Join und Subquery lösbar, da die Abteilungsnummer als Schlüsselteil sowieso Bestandteil der Vorgangstabelle wäre.

Größter/Kleinster

Ein weiteres typische Anwendung für eine Subquery: Suche den Datensatz, der im Kriterienfeld den größten (kleinsten) Wert hat.

Beispiel:

- Bestellung (idBst, BstDatum, fiKd, …)

Anfrage: Suche die neuesten Bestellungen

Subquery mit Mengenprädikat:

```
SELECT
     Bestellung.idBst, Bestellung.BstDatum
FROM
     Bestellung
WHERE
     Bestellung.BstDatum >= ALL
          (
          SELECT
               Bestellung.BstDatum
          FROM
               Bestellung
          )
```

Um diese Anfrage ohne Subquery auszudrücken, kann man sich des TOP-Prädikates bedienen:

```
SELECT TOP 1
     Bestellung.idBst, Bestellung.BstDatum
FROM
     Bestellung
ORDER BY
     Bestellung.BstDatum DESC
```

Man beachte, daß TOP 1 keineswegs sicher nur einen DS liefert. Wenn im Beispiel mehrere Bestellungen am letzten Bestelldatum vorgenommen wurden, werden bei der Feldauswahl im Beispiel auch alle diese DS ausgegeben.

Wenn TOP 1 auf einem nicht eindeutigen Feld benutzt werden sollte, um das Maximum als einen Rückgabewert zu ermitteln, wäre ein SELECT DISTINCT TOP 1 erforderlich.

Im vorliegenden Beispiel ist es allerdings gerade erwünscht, daß mehrere DS erscheinen, wenn mehrere den gleichen Höchstwert aufweisen. Die Subqueries verhalten sich genauso.

Subquery mit Aggregat

```
SELECT
     Bestellung.idBst, Bestellung.BstDatum
FROM
     Bestellung
WHERE
     Bestellung.BstDatum =
          (
          SELECT
               MAX(Bestellung.BstDatum)
          FROM
               Bestellung
          )
```

Hier ist die Subquery-Lösung mit >= ALL deutlich am langsamsten, gefolgt von der TOP1-Lösung. Die Subquery mit Aggregatfunktion ist aber der ungeschlagene Sieger.

Diese letzte Abfrage ist um 20.600% besser als TOP1 und um 128.000% besser als die Abfrage mit Mengenprädikat.

Nun noch zwei generische Beispiele, in denen das Pendel ganz erheblich zugunsten der Unterabfrage ausschlägt:

1:n Master mit Detail („Kunden mit Bestellung“)

Inner Join mit Distinct

```
SELECT DISTINCT
     M.id, M.Text
FROM
     Detail AS D INNER JOIN Master AS M
     ON D.fi = M.id
```

Exists mit Subquery

```
SELECT
     M.id, M.dtText
FROM
     Master AS M
WHERE EXISTS
     (
     SELECT
          D.id
     FROM
          Detail As D
     WHERE
          D.fi = M.id
     )
```

Die Subquery zeigte sich beim Testen um 2.300% bis zu 12.700% (!) schneller. Zudem ist sie im Gegensatz zu Select Distinct bearbeitbar.

1:n Master ohne Detail („Kunden ohne Bestellung“)

Outer Join mit Where Is Null

```
SELECT
     M.id, M.Text
FROM
     Master AS M LEFT JOIN Detail AS D
     ON M.id = D.fi
WHERE
     D.fi Is Null
```

Not (!) Exists mit Subquery

```
SELECT
     M.id, M.Text
FROM
     Master AS M
WHERE NOT EXISTS
     (
     SELECT
          D.fi
     FROM
          Detail As D
     WHERE
          D.fi = M.id
     )
```

Vorteil der zweiten Lösung im Test 1.500% bis 17.000% (!!)

Das kann auch als Ersatz dienen, um eine Unterabfrage mit NOT IN zu vermeiden, wie

```
SELECT
     M.id, M.Text
FROM
     Master AS M
WHERE M.id NOT IN
     (
     SELECT
          D.fi
     FROM
          Detail As D
     )
```

Man kann zwar durchaus allgemeine Aussagen treffen, in welchen Konstellationen Lösungen mit oder ohne Subquery vorzuziehen sind. Die betreffenden Regeln sind aber so komplex und mit Ausnahmen, zu deren Auftreten es natürlich auch wieder komplexe Regeln gibt, durchsetzt, daß es einerseits den Rahmen dieses Skriptes sprengt und andererseits schlichtes Testen der Alternativen schneller zum Ziel führt.

Daher nur ein paar unverbindliche Faustregeln, deren Zutreffen man im Einzelfall testen möge.

1) Vergleiche auf

```
>= ALL (SELECT …)
<> ALL (SELECT …)
<= ALL (SELECT …)
```

sind eher ungünstig, da mit allen Ergebnissen verglichen werden muß. Ebenso

```
NOT IN (SELECT …)
```

2) Vergleiche auf

```
>= SOME (SELECT …)
= SOME (SELECT …)
<= SOME (SELECT …)
```

sind günstiger, da nach dem ersten Treffer abgebrochen werden kann. Vergleiche mit

```
IN (SELECT …)
```

verhalten sich ähnlich

3) Vergleiche auf

```
= (SELECT MIN/MAX …)
```

sind günstig, da nur auf einen Wert verglichen wird, der über einem Indexfeld mit den Aggregatfunktionen Min/Max obendrein schnell gefunden wird.

4) Vergleiche mittels

```
EXISTS (SELECT …)
NOT EXISTS (SELECT …)
```

sind meist extrem schnell; letzterer sogar trotz NOT.

SQL oder VBA?

Bisher war ausschließlich von lesenden Abfragen die Rede. Dafür gibt es zwei Gründe: Lesende Vorgänge machen im normalen Datenbankbetrieb den Löwenanteil der Zugriffe aus und der Stoff mußte in einem zweistündigen Vortrag Platz finden.

Zu Schreibzugriffen sollen daher nur einige kleine Anmerkungen gemacht werden.

Wenn man eine Aufgabe wie Datensatzlöschung, Änderungen oder Anfügen mit einer Aktionsabfrage oder einem SQL-Statement lösen kann, ist das fast immer die schnellere Lösung, verglichen mit einem Recordset-Objekt in einer VBA-Prozedur.

Man kann gespeicherte Aktionsabfragen oder auch SQL-Statements mit CurrentDb.Execute ausführen lassen.

Es gibt im wesentlichen zwei Ausnahmen, in denen eine VBA-Lösung deutliche Vorteile bieten kann:

Einfügen vieler Datensätze per Code, die nicht mit einem SELECT erfaßt werden können

Wenn man Daten beispielsweise aus einer Excel-Tabelle einzufügen hat, ist INSERT INTO …SELECT … besser als ein Recordset.AddNew in einer Schleife.

Wenn aber die Daten nicht relational vorliegen oder aus Textimporten stammen, ist die Recordset.AddNew-Methode günstiger, als in einer Schleife immer wieder Execute mit INSERT INTO … VALUES(…) auszuführen.

Aufgaben, die Zwischenspeicherung oder Schleifen erfordern

Man kann mit viel Zauberei mit reinem SQL auf Feldwerte eines durch ein Sortierkriterium festgelegten Vorgängerdatensatzes zugreifen. Wenn man statt dessen eine VBA-Funktion schreibt, die diesen Zugriff kapselt, und sich diesen Wert in der Abfrage ausgeben läßt, geht das erheblich schneller.

Ein bitterböser Tabellenentwurf

Nachdem nun hoffentlich jeder die außerordentliche Bedeutung der Indizes erkannt hat, ist der Boden bereitet, um noch ein wenig über eine Art der Tabellengestaltung herzuziehen, die in letzter Zeit von manch einem gepflegt wird, weil er sich in die vermeintliche Flexibilität dieser Form verliebt hat.

Statt jeder Eigenschaft ein Feld zuzuweisen, wie es die Relationale Theorie zwingend vorschreibt

id	Vorname	Nachname	Gewicht	Haarfarbe
1	Peter	Müller	86	blond

wird ein Aufbau der Form

id	Feldname	Feldwert
1	Vorname	Peter
1	Nachname	Müller
1	Gewicht	86
1	Haarfarbe	blond

vorgenommen.

Es dürfte auf Anhieb klar sein, daß hier sämtliche Basis-anforderungen an ein Tabellenfeld nicht mehr erfüllbar sind.

- Es können keine Eingabepflichten und Domänenfeldregeln festgelegt werden
- Es können keine sinnvollen Datentypen festgelegt werden, performantere Datentypen (Zahlen) müssen notgedrungen als Texte abgespeichert werden
- Felder können nicht sinnvoll indiziert werden

Wem es also nicht genügt, daß diese nichtrelationale Struktur in einer Relationalen Datenbank nichts zu suchen hat, läßt sich hoffentlich dadurch bekehren, daß die letzten beiden Punkte der Performance extrem abträglich sind.