Kapitel 6

Structured Query Language (SQL)

Structured Query Language (strukturierte Abfragesprache) – oder SQL ist die Verkehrssprache der verschiedenen relationalen Datenbank-Managementsysteme. Visual Basic und Microsoft Access verwenden beide ausschließlich SQL, um Abfragen an Desktop-, Client-/Server- sowie Mainframe-Datenbanken zu richten. Access verfügt über ein grafisches »Query by Example-« (QBE-) Abfragewerkzeug – die Abfrage-Entwurfsansicht –, um automatisch SQL-Anweisungen von Jet (Access) zu schreiben. Mit Access können ausgeklügelte Anwendungen entwickelt werden, ohne je einen Blick auf eine SQL-Anweisung im SQL-Fenster von Access geworfen zu haben. Visual Studio 6.0 integriert einen Satz von Visual Data Tools (VDT), die Fähigkeiten zur Datenbank-Diagrammgestaltung sowie zur grafischen SQL-Abfrageentwicklung für SQL Server (6.x und 7.0) und Oracle-Datenbanken (7.3 und 8.0.3+) beinhalten. Diese Werkzeuge sind grafisch nicht so ausgearbeitet und so leicht zu bedienen wie die entsprechenden Access-Tools, jedoch sind sie völlig ausreichend zur Entwicklung und Veränderung von Client-/Server-Datenbankschemata.

Hinweis

Mit der Erwähnung von Microsoft Access ohne Versionsnummer meinen wir Access 2.0 sowie beide Access-Versionen für Windows 95, Access 7.0 (Access 95) und 8.0 (Access 97). Bezüglich Jet-SQL ergeben sich zwischen diesen drei Versionen keine großen Unterschiede. Gegenüber Access 1.x wurde in Access 2.0 eine ganze Anzahl reservierter Wörter dem SQL-Vokabular hinzugefügt. Die reservierten Wörter `TOP n` *bzw.* `TOP n PERCENT` *von Jet-SQL, eingeführt mit der Version Jet 3.0, werden im Abschnitt »Jet SQL und ANSI SQL-92« näher behandelt; Unterabfragen sind das Thema im Abschnitt »Eingebettete Abfragen und Unterabfragen« später in diesem Kapitel. SQL-Anweisungen, um Tabellen zu Jet-Datenbanken hinzuzufügen bzw. um Jet-Tabellen um Felder und Indizes zu ergänzen, sind zwei der Themen im Kapitel 7 »Kreuztabellen- und Aktionsabfragen ausführen«. Microsoft nennt den SQL-Dialekt von Access* Microsoft Jet Database Engine SQL*; das vorliegende Buch verwendet die kürzere Form dieses Begriffs:* Jet-SQL. *Zwischen* Jet-SQL *und* Transact-SQL *des Microsoft SQL Servers bestehen jedoch erhebliche Unterschiede; diese werden im entsprechenden Abschnitt des aktuellen Kapitels beschrieben.*

Der erste Teil des Kapitels führt in die vom American National Standards Institute (ANSI) standardisierte SQL-Version – dort verzeichnet unter der Kennziffer *X.3.135-1992* und besser bekannt als *SQL-92* – ein. Die Begriffe SQL-89 bzw. SQL-92 weisen auf das standardisierte ANSI SQL hin, nicht auf den Access-Dialekt Jet-SQL. ANSI SQL-92 wurde von der International Standards Organization (ISO) – einer Abteilung der United Nations (UN) mit Hauptsitz in Genf – sowie von der International Electrotechnical Commision (IEC) in dem gemeinsamen Standard ISO/IEC 9075:1992 anerkannt; dort ebenfalls geführt unter dem Begriff *Database Language SQL*. Ein davon abweichender ANSI-Standard (Kennziffer

X.3.168-1989) definiert die *Database Language Embedded SQL*. SQL-92 ist eine durch und durch standardisierte Sprache – viel mehr als xBASE, für die bisher kein unabhängiger Standard existiert.

Die heutigen relationalen Client-/Server-Datenbanken unterstützen SQL-89 sowie viele der neuen reservierten Wörter von SQL-92. Die meisten RDBMS-Hersteller fügen ihre eigenen reservierten Wörter hinzu, wodurch proprietäre SQL-Dialekte geschaffen werden (wie bei *Transact-SQL* von SQL Server). Ein Basiswissen über ANSI SQL ist notwendig, einerseits für die Nutzung der SQL-Pass-through-Verfahren mit der Jet 3.5-Datenbankengine, andererseits für die Anwendung der in Visual Basic 6.0 neuen ActiveX Data Objects (ADO) und ODBCDirect sowie der Remote Data Objects (RDO) und den Remote Data Controls (RDC) der Enterprise Edition. Die Kombination aus RDO/RDC wurde in Visual Basic 6.0 ersetzt durch ADO und das neue DataGrid-Steuerelement.

Der Abschnitt »Jet SQL und ANSI SQL-92« später in diesem Kapitel untersucht die Unterschiede zwischen SQL-92 und Jet-SQL. Wenn Sie mit ANSI SQL vertraut sind – gleichgültig ob SQL-89 oder SQL-92 –, möchten Sie vielleicht direkt zu diesem Abschnitt springen, der die besonderen Spielarten von SQL durch Einsatz der Jet 3.5x-Datenbankengine behandelt. Obwohl das vorliegende Kapitel die grundlegende SQL-Syntax für Daten verändernde Abfragen (in Access und in diesem Buch *Aktionsabfragen* genannt) sowie für Kreuztabellenabfragen von Jet-SQL behandelt, werden Beispiele für den Gebrauch dieser Art von Abfragen in Kapitel 7 und 9 näher erläutert.

Hinweis

Das ANSI-ISO SQL-Committee hat bereits ein Arbeitspapier zur nächsten SQL-Version – vorläufig SQL3 *genannt – erarbeitet. Auf der Madrider Zusammenkunft im Januar 1996 gaben die Verantwortlichen des Komitees bekannt, dass der abschließende Standard von SQL3 nicht vor 1999 erscheinen wird. SQL-89 und SQL-92 befassen sich lediglich mit Zeichen-(Character-)basierten Informationen; SQL3 hingegen setzt sich mit der Speicherung und dem Abruf von Objekten in relationalen Datenbankstrukturen auseinander.*

Das Arbeitspapier von 1996 listet folgende »signifikante neue Eigenschaften« von SQL3 auf:

- *Unterstützung von aktiven »Regeln« (Trigger genannt)*
- *Unterstützung für abstrakte Datentypen (z.B. Objekte)*
- *Unterstützung für mehrfache* `NULL`*-Zustände*
- *Unterstützung für* `PENDANT`*-referentielle Integrität*

- *Eine rekursive `UNION`-Operation für Abfrageausdrücke*
- *Unterstützung für Aufzählungs- und boolesche Datentypen*
- *Unterstützung für `SENSITIVE` Cursor*

Die meisten relationalen Datenbank-Managementsysteme (darunter auch der SQL Server) bieten gegenwärtig Unterstützung für Trigger. Trigger werden in aller Regel dafür eingesetzt, referentielle Integrität und Datenkonsistenz zu gewährleisten. Die Eigenschaft der Unterstützung abstrakter Datentypen (Objekterweiterungen) für Objekt-relationale Datenbanken ist verantwortlich für die Verzögerung des normalerweise im dreijährigen Rhythmus stattfindenden Updatezykluss von SQL. Die RDBMS-Anbieter haben der Sprache SQL ihre eigenen proprietären Objekterweiterungen hinzugefügt. Aufgrund der Verzögerung beim Erscheinen der SQL3-Spezifikation haben die dort vorgesehenen standardisierten Objekterweiterungen möglicherweise nicht die Chance, sich durchzusetzen. Das jüngste offizielle, über 600 Seiten starke Arbeitspapier über ANSI-IEC SQL3 kann man unter `http://epoch.cs.berkeley.edu:8000/sequoia/schema/STANDARDS/SQL3/sql3part2.txt` einsehen.

6.1 Die Wurzeln von SQL

Das relationale Datenbankmodell von Dr. E. F. Codd aus dem Jahre 1970 (vorgestellt in Kapitel 5) war eine theoretische Beschreibung dessen, wie relationale Datenbanken entworfen werden, und nicht darüber, wie sie zu benutzen sind. Um Tabellen zu erstellen bzw. Felder zu bestimmen, die in den Tabellen enthalten sein sollen, um Beziehungen zwischen Tabellen aufzubauen oder die Daten in der Datenbank zu manipulieren, bedarf es einer Datenbank-Programmiersprache. Die erste von Dr. Codd und seiner Mannschaft in den San Jose Labors von IBM definierte Sprache hieß Structured English Query Language (SEQUEL); diese war für eine im Prototyp-Stadium befindliche Datenbank (IBM nannte sie »System R«) entwickelt worden. Die zweite Version von SEQUEL – genannt SEQUEL/2 – wurde später umbenannt zu SQL. Technisch gesehen ist SQL der Name einer Datenmanipulationssprache von IBM, nicht jedoch eine Abkürzung für den Begriff *Structured Query Language*.

Der folgende Abschnitt beschreibt einerseits die Unterschiede zwischen SQL und den prozeduralen Sprachen, die üblicherweise für die Computerprogrammierung eingesetzt werden, andererseits die Art und Weise, wie Anwendungen SQL verwenden im Zusammenspiel mit Desktop-, Client-/Server- und Mainframe-Datenbanken.

6.1.1 Elemente einer SQL-Anweisung

Dieses Buch hat im bisherigen Verlauf von dem Begriff *Abfrage* ausführlichen Gebrauch gemacht, ohne diesen jedoch zu erklären. Da Visual Basic 6.0 SQL zur Ausführung aller Abfragen verwendet, wird der Begriff *Abfrage* in diesem Buch definiert als ein Ausdruck (in einem beliebigen SQL-Dialekt) zur Bezeichnung einer Operation, die von einem Datenbank-Managementsystem ausgeführt werden kann. Eine Abfrage enthält normalerweise zumindest die folgenden drei Elemente:

- Ein *Wort* (z.B. `SELECT`), das den Typ der Abfrage festlegt
- Ein *Prädikat-Objekt*, das einen oder mehrere Feldnamen von einem oder mehreren Tabellenobjekten festlegt (wie `*`, das alle Felder einer Tabelle bestimmt)
- Eine *Präpositions-Klausel*, die das oder die Datenbankobjekt(e) spezifiziert, auf die das *Wort* ausgeführt wird (wie z.B. `FROM TableName`)

Die einfachste mögliche SQL-Abfrage lautet: `SELECT * FROM TableName`; sie gibt den gesamten Inhalt der in `TableName` angegebenen Tabelle als Ergebnismenge zurück. Abfragen werden in diesem Buch klassifiziert als *Auswahlabfragen*, die Daten (Abfrageergebnismengen) zurückliefern, und *Aktionsabfragen*, die Daten in einer Datenbank modifizieren, ohne irgendwelche Daten zurückzugeben.

IBMs Originalversion von SQL, als SEQUEL eingeführt, wies relativ wenige reservierte Wörter und eine einfache Syntax auf. Über Jahre hinweg haben die Herausgeber von Datenbank-Managementsystemen der Sprache SQL neue reservierte Wörter hinzugefügt. Viele dieser reservierten Wörter aus einzelnen proprietären SQL-Versionen haben ihren Weg in den ANSI SQL-Standard gefunden. Vertreiber von SQL-Datenbanken, die ihre Unterstützung für den ANSI-Standard bekunden, haben die Möglichkeit, der Sprache eigene reservierte Wörter hinzuzufügen, solange die hinzugefügten reservierten Wörter keine Probleme bei der Verwendung der durch ANSI spezifizierten reservierten Wörter verursachen. Transact-SQL – die von den Microsoft- bzw. Sybase-Versionen des SQL Servers verwendete Sprache – besitzt beispielsweise weit mehr reservierte Wörter als das konventionelle ANSI SQL. Transact-SQL schließt auch reservierte Wörter zur bedingten Ausführung sowie zur Schleifenbildung innerhalb von SQL-Anweisungen mit ein. (Die reservierten Wörter `CASE`, `NULLIF` und `COALESCE` von SQL-92 sind – um eine bedingte Ausführung zu erreichen – im Vergleich dazu eher primitiv.) Jet-SQL beinhaltet die Anweisungen `TRANSFORM` und `PIVOT`, um Kreuztabellenabfragen auszuführen (ein sehr sinnvolles Konstrukt, das ebenfalls in ANSI SQL nicht vorgesehen ist). Es ist zwar durchaus möglich, mithilfe von ANSI SQL Kreuztabellenabfragen zu generieren, jedoch besitzt eine solche Anweisung eine sehr komplexe Syntax.

Eine weitere Erläuterung der Details der SQL-Syntax erscheint nach dem folgenden Abschnitt, der zunächst die grundlegenden Charakteristiken der SQL-Sprache sowie die Möglichkeiten, SQL mit konventionellem Visual Basic Sourcecode zu kombinieren, beschreibt.

6.1.2 Unterschiede zwischen SQL und prozeduralen Programmiersprachen

Alle Dialekte von SQL gehören zur Gattung der »Fourth-Generation Languages« (4GLs). Der Begriff *Fourth Generation* (vierte Generation) stammt aus der folgenden Beschreibung über die Generationen in der Entwicklung von Sprachen zur Kontrolle von Computer-Operationen:

- *First-Generation* Languages (1GLs) – Sprachen der ersten Generation – verlangten vom Programmierer, in der Binärsprache der Computerhardware (Objekt- oder Maschinencode genannt) zu programmieren. (In diesem Fall ist der Computer das Objekt.) In den frühen Tagen der Mini- und Microcomputer wurde z.B. ein Computer durch Setzen einer Reihe von Schaltern, die ihrerseits Instruktionen (Befehle) direkt an den Prozessor (CPU) des Computers sandten, gestartet bzw. gebootet. Nachdem der Computer hochgefahren war, konnten binärcodierte Instruktionen von einem Lochstreifenleser geladen werden. 1GLs stellten eine Programmierung auf einem sehr mühsamen Weg dar. Die ersten Computer-Betriebssysteme waren direkt in Maschinencode geschrieben und wurden von Lochstreifen oder Lochkarten eingelesen.

- *Second-Generation* Languages (2GLs) – Sprachen der zweiten Generation – brachten den Programmierungsprozess durch Verwendung von Assemblersprachen weit voran, indem sie die Notwendigkeit, jedes einzelne Bit von Prozessorinstruktionen selbst zu setzen, eliminierten. Assemblersprachen erlaubten dem Programmierer, einfache alphabetische Codes -Mnemonics genannt (denn diese Codes waren leichter zu merken als binäre Instruktionen) – zu verwenden; zusätzlich wurden einzelne oder mehrere Prozessorinstruktionen des bisherigen, reichlich undurchschaubaren Objektcodes durch oktale oder hexadezimale Werte ersetzt. Nachdem ein Assemblerprogramm fertig entwickelt war, wurde der Assemblercode durch Compilierung in Objektcode-Instruktionen umgesetzt, die für den Prozessor lesbar und damit ausführbar waren. Microsofts MASM ist beispielsweise ein populärer Assemblersprachen-Compiler für Intel-80x86-Prozessoren. Assemblersprachen sind auch heute noch weit verbreitet, besonders wenn es darum geht, die Ausführungsgeschwindigkeit zu steigern oder die Computerhardware direkt anzusprechen.

- *Third-Generation* Languages (3GLs) – Sprachen der dritten Generation –, repräsentiert durch die frühen Versionen von FORTRAN (FORmula TRANslator) und BASIC (Beginner's All-Purpose Symbolic Instruction Code), erlaubten dem Programmierer, Assemblercode durch einfache Anweisungen – häufig in einer strukturierten Version der englischen Sprache – zu ersetzen. 3GLs werden *prozedurale Sprachen* genannt, denn die einzelnen Anweisungen einer 3GL werden in so genannten *Prozeduren* zusammengefasst, die der Computer in genau der Reihenfolge ausführt, die der Programmierer im Quellcode festgelegt hat. Theoretisch sollte eine prozedurale Sprache unabhängig vom

Prozessortyp sein, für den der Quellcode letztendlich kompiliert wird. Jedoch erreichen derzeit nur wenige Sprachen tatsächlich das Ziel der Plattformunabhängigkeit. Die meisten, darunter auch Visual Basic 6.0 und Visual C++, wurden für den 80x86-Prozessor entwickelt und optimiert. Die Sprache Java verspricht Plattformunabhängigkeit, jedoch ist andererseits der Datenbankzugriff via JDBC (Java DataBase Connector) gegenwärtig nicht so hoch entwickelt und ausgereift wie in Windows-basierten Anwendungen, die Jet oder ODBC verwenden.

- *Fourth-Generation* Languages (4GLs) – Sprachen der vierten Generation – werden häufig als nichtprozedurale Sprachen bezeichnet. Der Quellcode, den der Programmierer entwickelt, teilt dem Computer lediglich das Ergebnis mit, das man erhalten möchte – nicht jedoch den Weg, auf dem dieses Ziel zu erreichen ist. SQL wird gemeinhin als 4GL bezeichnet, denn eine beliebige SQL-Anweisung teilt dem Database Manager (DBM) lediglich mit, welche Daten dieser bereitstellen und zurücksenden soll, nicht jedoch, auf welchem Weg er eine solche Meisterleistung vollbringen kann.

Hinweis

Ob SQL tatsächlich eine 4GL darstellt, wird kontrovers diskutiert, denn die einzelnen SQL-Anweisungen werden streng genommen von 3GLs – in manchen Fällen sogar von 2GLs – ausgeführt, die direkt mit den in der Datenbank gespeicherten Daten umgehen. Diese Datenbanksprachen sind gleichfalls dafür zuständig, die angeforderten Daten in einem Format zurückzusenden, das von der Anwendung verarbeitet werden kann.

Ganz unabhängig von der Diskussion, ob SQL eine 4GL darstellt, ist es wichtig, einige weitere Unterschiede zwischen SQL und 3GLs zu kennen. Die wichtigsten Unterschiede sind:

- SQL ist eine *mengenorientierte* Sprache, wohingegen die meisten 3GLs als *feldorientierte* Sprachen bezeichnet werden können. SQL gibt Mengen von Daten in einem logischen, tabellarischen Format zurück. Die aufgrund einer Abfrage zurückgegebenen Datenmengen sind von den Daten in der Datenbank abhängig und man kann die Anzahl der zurückgegebenen Datenzeilen (als Mitglieder-Datenmenge) kaum vorhersagen. Die Anzahl der Mitglieder der Datenmenge kann – jedesmal, wenn die Abfrage erneut ausgeführt wird – unterschiedlich sein; meist unterscheidet sie sich in Multiuser-Umgebungen bereits sofort nach ihrer Ausführung. 3GLs können nur eine festgelegte Anzahl von tabellarischen Datenelementen zur gleichen Zeit bearbeiten; diese Grenze wird durch die Dimension des Arrays – durch vorherige Zuweisung an eine zweidimensionale Array-Variable – bestimmt. Zu diesem Zweck muss die Anwendung im Vorhinein bereits wissen, wie viele Zeilen und Spalten in der Er-

gebnismenge der SQL-Abfrage enthalten sind, um die zurückgegebenen Daten Zeile für Zeile bzw. Spalte für Spalte zu bearbeiten. Das `Recordset`-Objekt in Visual Basic 6.0 erledigt diese Übertragung automatisch.

- SQL zählt zu den *schwach typisierten* Sprachen, die meisten 3GLs sind dagegen *streng typisiert*. Es ist nicht notwendig, in einer SQL-Anweisung Felddatentypen anzugeben. SQL-Abfragen geben grundsätzlich alle Datentypen zurück, die den Ursprungsfeldern zugewiesen sind; aus den ursprünglichen Datentypen werden dann die Spalten der Ergebnismenge der Abfrage gebildet. Die meisten 3GL-Compilersprachen sind dagegen streng typisiert. COBOL, C, C++, Pascal, Modula2 sowie ADA sind Beispiele streng typisierter Compiler-Programmiersprachen. Streng typisierte Sprachen verlangen, dass die Namen und Datentypen aller Variablen deklariert werden müssen, bevor an diese Variablen Werte zugewiesen werden können. Falls der Datentyp einer Abfragespalte nicht mit dem zur Aufnahme dieser Daten vorgesehenen Datentyp korrespondiert, tritt ein Fehler (manchmal als »Scheinbarer Widerspruch«-Fehler bezeichnet) auf.
  Visual Basic war zu Beginn eine interpretierte und traditionell schwach typisierte Sprache. Die VBA-Version 6.0, die von Visual Basic 6.0 verwendet wird, ist streng typisiert, was manche Entwickler dazu veranlassen wird, Teile ihres bereits bestehenden Codes zu überarbeiten. Um die Notwendigkeit, die Datentypen von Tabellenfeldern im Vorfeld einer Abfrage bereits kennen zu müssen, zu überwinden, stellt Visual Basic den Datentyp `Variant` bereit, der jeden in Datenbanktabellen gebräuchlichen Datentyp akzeptiert.

Sie dürfen sich glücklich schätzen, dass Sie Visual Basic 6.0 einsetzen, um SQL-Anweisungen auszuführen. Denn in den meisten Fällen brauchen Sie sich keine Gedanken darüber zu machen, wie viele Zeilen eine Abfrage zurückgibt oder welche Datentypen in den Spalten der Abfrage-Ergebnismenge auftreten. Das `Recordset`-Objekt – das die Daten abruft – handhabt diese Details automatisch für Sie. Es ist nicht notwendig, Ihre Visual Basic-Anwendung jedes Mal erneut zu kompilieren und zu linken, sobald Sie eine Abfrageanweisung geändert haben; ändern Sie lediglich die Anweisung und starten Sie die Anwendung erneut. Visual Basic »kompiliert« automatisch die getätigten Änderungen in eine Art Pseudo-Code.

6.1.3 Verschiedene ANSI SQL-Arten

Die gegenwärtigen ANSI SQL-Standards kennen vier Methoden, um SQL-Anweisungen auszuführen. Welche Methode Sie einsetzen, hängt von der verwendeten Programmierumgebung ab, wie im Folgenden beschrieben:

- *Interactive SQL* erlaubt, SQL-Anweisungen an einem Kommandozeilen-Prompt einzugeben, vergleichbar dem Punkt-Prompt von dBASE. Wie bereits in Kapitel 1 erwähnt, ist der Einsatz des interaktiven Befehls `LIST` von dBASE der interaktiven SQL `SELECT`-Anweisung sehr ähnlich. Relationale Mainframe- und Client-/Server-Datenbanken stellen ebenfalls interaktive SQL-Fähigkeiten

zur Verfügung. Microsoft SQL Server stellt zu diesem Zweck die Programme »isql« und »Isql_w« bereit. Der Einsatz von interaktivem SQL wird auch *direkter Aufruf* genannt. Interactive SQL wird als *Mengen-Prozess* bezeichnet; wenn Sie eine Abfrage am SQL-Prompt eingeben, erscheint das Ergebnis Ihrer Abfrage auf dem Computermonitor. Relationale Datenbank-Managementsysteme bieten verschiedene Methoden an, um eine bildlauffähige Ansicht der interaktiven Abfrage-Ergebnismengen anzuzeigen.

- *Embedded SQL* erlaubt, SQL-Anweisungen durch Voranstellen eines Schlüsselworts innerhalb einer klassischen Programmiersprache auszuführen, wie in `Exec SQL` in der Programmiersprache C. Üblicherweise deklarieren Sie Variablen, die Sie für den Abruf von Daten einer SQL-Abfrage einzusetzen beabsichtigen, zwischen den Anweisungen `EXEC SQL BEGIN DECLARE SECTION` und `EXEC SQL END DECLARE SECTION`. Hierfür benötigen Sie einen sowohl für das verwendete RDBMS als auch für die Programmiersprache geeigneten Präcompiler. Der Vorteil von Embedded SQL besteht darin, dass Sie ganze Attributklassen an eine einzige Variable in einem Schritt zuweisen können. Der Nachteil ist, dass Sie mit Abfrage-Ergebnismengen in zeilenweiser Manier umgehen müssen anstatt mit einer Mengenoperation (wie bei Interactive SQL).

- *Module SQL* erlaubt, die SQL-Anweisungen unabhängig von Ihrem 3GL-Sourcecode zu kompilieren und die kompilierten Objektmodule anschließend in Ihr ausführbares Programm zu linken. SQL-Module sind den Visual Basic 6.0-Codemodulen ganz ähnlich: Diese Module enthalten Variablendeklarationen und temporäre Tabellen für die Aufnahme von Abfrage-Ergebnismengen; darüber hinaus können Sie Argumentwerte Ihrer 3GL zu Prozedurparametern, die in SQL-Modulen deklariert sind, weiterleiten. Gespeicherte Prozeduren, die vorkompilierte Abfragen auf Datenbankservern ausführen, teilen mit Module SQL viele Merkmale.

- *Dynamic SQL* erlaubt, SQL-Anweisungen zu erstellen, deren Inhalt zum Zeitpunkt der Erstellung noch nicht vorhergesagt werden kann. (Die vorangehenden SQL-Arten werden als *Static SQL* klassifiziert.) Ein Beispiel für Dynamic SQL: Sie möchten eine Visual Basic-Anwendung entwickeln, die Abfragen an eine Vielzahl verschiedener Datenbanken richten können soll. Dynamic SQL erlaubt, Abfragen an Datenbanken in Form von Strings zu richten. Sie senden beispielsweise eine Abfrage an eine Datenbank und erhalten vom Datenbankkatalog ausführliche Informationen über Tabellen und Felder der Tabellen in der Datenbank zurück. Sobald Sie über die Datenbankstruktur Bescheid wissen, können Sie oder der Anwender Ihrer Anwendung eine entsprechend angepasste Abfrage konstruieren, die in der dynamischen Abfrage korrekte Feld- und Tabellennamen ergänzt. Die Implementierung von Jet-SQL in Visual Basic ähnelt einer Kombination aus Dynamic und Static SQL, obwohl die Jet-Datenbankengine das detaillierte Auslesen der Kataloginformationen automatisch vornimmt, sobald Ihre Anwendung ein `Recordset`-Objekt aus der Datenbank erstellt.

Technisch betrachtet sind Static und Dynamic SQL Methoden zur Bindung von SQL-Anweisungen an Datenbank-Anwendungsprogramme. *Bindung* bedeutet in diesem Zusammenhang: Auf welchem Weg SQL-Anweisungen an den Quell- oder Objektcode gebunden oder mit diesem kombiniert werden können, wie bestimmte Werte an SQL-Anweisungen weitergereicht werden und wie Abfrage-Ergebnismengen verarbeitet werden können. Eine dritte Methode, SQL-Anweisungen zu binden, bildet das Call Level Interface (CLI). Das ODBC-API verwendet das von der SQL Access Group (SAG) entwickelte CLI. (SAG ist ein Zusammenschluss von Herausgebern und Anwendern von relationalen Datenbank-Managementsystemen.) Ein CLI nimmt SQL-Anweisungen einer Anwendung in Form von Strings entgegen und leitet sie direkt zur Ausführung an den Server weiter. Der Server benachrichtigt das CLI, sobald die angeforderten Daten bereitgestellt sind, und leitet diese an die Anwendung weiter. Näheres zum ODBC-CLI finden Sie in Kapitel 18.

Falls Sie zu den COBOL- oder C/C++-Programmierern gehören, die regelmäßig Embedded SQL-Anweisungen schreiben, müssen Sie sich – bei der Ausführung einer `SELECT`-Abfrage – an VBs automatische Erstellung von virtuellen Tabellen gewöhnen, im Gegensatz zur Ausführung `CURSOR`-bezogener `FETCH`-Anweisungen, um Abfrage-Ergebniszeilen einzeln nacheinander zu erhalten.

6.2 ANSI SQL-Anweisungen schreiben

ANSI SQL-Anweisungen besitzen ein sehr flexibles Format. Sämtliche BASIC-Dialekte trennen einzelne Anweisungen voneinander durch das Steuerzeichenpaar für »Neue Zeile« (Wagenrücklauf und Zeilenvorschub, engl. *carriage return* und *line feed*); Java, C, C++ und Pascal/Delphi verwenden als Anweisungstrennzeichen das Semikolon. Im Gegensatz zu diesen Sprachen benötigt SQL zwischen den einzelnen Anweisungselementen, aus denen eine komplette SQL-Anweisung gebildet wird, keine Trennzeichen wie das Steuerzeichenpaar »Neue Zeile«, ein Semikolon oder auch nur ein Leerzeichen. (SQL ignoriert in den meisten Fällen Whitespaces, die Steuerzeichen für Tabulator, Neue Zeile, zusätzliche Leerzeichen usw. beinhalten.) Deshalb können Sie – der besseren Lesbarkeit halber – Leerzeichen benutzen, um SQL-Anweisungen zu formatieren. Die SQL-Beispiele in diesem Buch platzieren Gruppen von zusammengehörigen Bezeichnern und reservierte SQL-Wörter in eigene Zeilen, wobei Einzüge verwendet werden, um fortgesetzte Zeilen zu kennzeichnen. Es folgt als Beispiel eine Jet-SQL-Kreuztabellenabfrage mit den beschriebenen Formatierungsmerkmalen:

```
TRANSFORM Sum(CLng([Order Details].UnitPrice*Quantity*
      (1 - Discount)*100)/100) AS ProductAmount
   SELECT Products.ProductName, Orders.CustomerID
   FROM Orders, Products, [Order Details],
      Orders INNER JOIN [Order Details] ON Orders.OrderID =
        [Order Details].OrderID,
      Products INNER JOIN [Order Details] ON Products.ProductID =
```

```
     [Order Details].ProductID
WHERE Year(OrderDate)=1995
GROUP BY Products.ProductName, Orders.CustomerID
ORDER BY Products.ProductName
PIVOT "Qtr " & DatePart("q",OrderDate) In("Qtr 1",
   "Qtr 2","Qtr 3","Qtr 4")
```

Hinweis

Die eckigen Klammern, die den Tabellennamen [Order Details] umfassen, sind in Jet-SQL spezifisch. Die eckigen Klammern gruppieren Tabellen- oder Feldnamen (Literale) mit enthaltenen Leer- oder anderen Interpunktionszeichen, die in den Benennungskonventionen für Tabellen- und Feldnamen von relationalen Datenbank-Managementsystemen nicht erlaubt sind. Transact-SQL von SQL Server 7.0 akzeptiert – zur Identifizierung von Literalwerten – ebenfalls eckige Klammern bzw. doppelte Anführungszeichen. Jet-SQL verwendet die doppelten Anführungszeichen (») gewöhnlich als Ersatz für die einfachen Anführungszeichen oder den Apostroph ('), der als String-Indikatorzeichen in den meisten SQL-Implementierungen gilt. (Ein einfaches Anführungszeichen verhält sich in Jet-SQL im Übrigen einwandfrei.) Das vorangehende Beispiel der SQL-Anweisung für eine Kreuztabellenabfrage basiert auf den Tabellen der Jet 3.0-Beispieldatenbank Nwind.mdb, die mit Visual Basic 6.0 mitgeliefert wird.

Die folgenden Abschnitte beschreiben den formalen grammatikalischen Aufbau von SQL und erklären, wie SQL-Anweisungen kategorisiert werden. Außerdem führen sie Beispiele für eine Vielzahl von ANSI SQL `SELECT`-Abfragen auf.

6.2.1 Kategorien von SQL-Anweisungen

ANSI SQL teilt sich auf in die folgenden sechs Anweisungskategorien (hier absteigend sortiert nach Häufigkeit des Einsatzes):

- *Data Query Language*- (DQL-) Anweisungen – auch Datenabrufanweisungen genannt – empfangen Daten aus Tabellen und entscheiden, wie die Daten der anfordernden Anwendung präsentiert werden. Das reservierte Wort `SELECT` ist das am häufigsten vorkommende Wort in DQL (bzw. in SQL überhaupt). Weitere häufig gebrauchte reservierte Wörter in DQL sind `WHERE`, `ORDER BY`, `GROUP BY` und `HAVING`. Nicht selten werden diese reservierten Wörter der DQL in Zusammenhang mit anderen SQL-Anweisungskategorien verwendet.
- *Data Manipulation Language*- (DML-) Anweisungen beinhalten die reservierten Wörter `INSERT`, `UPDATE` und `DELETE`, wodurch Tabellenzeilen angehängt, modifiziert oder gelöscht werden. Mithilfe der DML-Wörter werden Aktionsabfragen generiert. Manche Veröffentlichungen rechnen die DQL-Anweisungen der DML-Kategorie zu.

- *Transaction Processing Language*- (TPL-) Anweisungen stellen sicher, dass sämtliche Tabellenzeilen, die von einer DML-Anweisung betroffen sind, in einem Zug aktualisiert werden. TPL-Anweisungen beinhalten beispielsweise die Anweisungen `BEGIN TRANSACTION`, `COMMIT` und `ROLLBACK`.
- *Data Control Language*- (DCL-) Anweisungen regeln den Zugriff von einzelnen Anwendern oder Gruppen von Anwendern auf Datenbankobjekte durch Vergabe von Rechten wie `GRANT` und `REVOKE`. Manche relationalen Datenbank-Managementsysteme bieten Zugriffskontrolle (wie `GRANT` oder `REVOKE`) auch auf Feldebene (einzelne Spalten) einer Tabelle.
- *Data Definition Language*- (DDL-) Anweisungen erlauben, neue Tabellen in einer Datenbank anzulegen (`CREATE TABLE`), Indizes zu erstellen (`CREATE INDEX`), Einschränkungen auf Feldwerte festzulegen (`NOT NULL`, `CHECK`, `CONSTRAINT`), Beziehungen zwischen Tabellen zu definieren (`PRIMARY KEY`, `FOREIGN KEY`, `REFERENCES`) sowie Tabellen und Indizes zu löschen (`DROP TABLE`, `DROP INDEX`). DDL beinhaltet darüber hinaus viele reservierte Wörter in Bezug auf den Erhalt von Daten aus dem Datenbankkatalog. DDL-Abfragen werden in diesem Buch als Aktionsabfragen klassifiziert, denn sie geben keine Datensätze zurück.
- *Cursor Control Language*- (CCL-) Anweisungen wie `DECLARE CURSOR`, `FETCH INTO` und `UPDATE WHERE CURRENT` bearbeiten einzelne Datensätze einer oder mehrerer Tabellen.

Ein DBM-Anbieter, der in seinem Produkt Konformität mit dem ANSI SQL-Standard beansprucht, ist nicht verpflichtet, alle Merkmale des SQL-92-Standards zu unterstützen. Tatsächlich gilt es festzuhalten, dass derzeit kein kommerzielles RDBMS auf dem Markt *alle* Interactive SQL-Schlüsselwörter von SQL-92 implementiert. Die Jet 3.5-Datenbankengine beispielsweise unterstützt kein einziges reserviertes Wort der DCL. Statt dessen verwendet sie die genau festgelegten Sicherheitsobjekte der Data Access Objects mit reservierten und Schlüsselwörtern aus Visual Basic. Die Jet 3.5-Datenbankengine benötigt ebenfalls keine Unterstützung für CCL-Anweisungen, da Cursor in Jet nicht durch SQL-Anweisungen manipuliert werden. Bildlaufcursor von Microsoft SQL Server 6+ werden aus ADO, ODBCDirect bzw. RDO heraus unterstützt.

Dieses Buch verwendet die im Anhang C des *Programmer's Reference for the Microsoft ODBC Software Development Kit (SDK)* definierte Terminologie, um die folgenden Hierarchieebenen an syntaktischer (grammatikalischer) Übereinstimmung von SQL festzulegen:

- *Minimal* – die Anweisungsgrammatik qualifiziert den DBM gerade noch als SQL-DBM, keinesfalls jedoch als RDBMS. Ein DBM mit solch minimalen SQL-Eigenschaften ist heutzutage nicht mehr verkaufsfähig.
- *Kern* – vereinigt minimale grammatikalische SQL-Fähigkeiten und grundlegende DDL- und DCL-Anweisungen, zusätzliche DML-Funktionen, weitere Datentypen außer `CHAR`, SQL-Aggregat-Funktionen – wie `SUM()` oder `AVG()` –

mit einer größeren Anzahl erlaubter Ausdrücke zur Auswahl von Datensätzen. Die meisten Desktop-DBM, denen nachträglich SQL-Fähigkeiten hinzugefügt wurden, unterstützen die Kern-Grammatik von SQL oder etwas mehr.

- *Erweitert* – schließt die beiden vorangehenden Ebenen mit ein sowie DML-Outer Joins, komplexere Ausdrücke in DML-Anweisungen, sämtliche ANSI SQL-Datentypen (auch `long varchar` und `long varbinary`), Batch-SQL-Anweisungen sowie Prozeduraufrufe. Die erweiterte SQL-Grammatik kennt zwei verschiedene Konformitätsebenen: 1 und 2. Die Konformität von ODBC-Treibern entsprechend den erweiterten grammatikalischen SQL-Ebenen wird in Kapitel 18 näher beleuchtet.

Hinweis

Ein Cursor *ist ein Zeiger auf eine spezielle Zeile einer Abfrage-Ergebnismenge, entsprechend dem Satzzeiger eines `ADODB.Recordset`-Objekts. Frühe Versionen von relationalen Client-/Server-Datenbanksystemen boten noch keine Cursor an; Frontend-Anwendungen waren daher gezwungen, Abfrage-Ergebnismengen mithilfe von Low-Level-Code zu manipulieren. Die ersten von Client-/Server-Datenbanksystemen angebotenen Cursor waren vom Typ Vorwärts und nicht aktualisierbar. Bildlaufcursor (in beide Richtungen beweglich), die auch editierbar sind, gehören zu den neueren Entwicklungen am Datenbankmarkt. Vorwärtscursor (von Microsoft auch als »Feuerwehrschlauch-Cursor« bezeichnet) bieten eine bessere Performance als bildlauffähige, serverseitige Cursor (vom Typ Statisch, Schlüsselgruppe oder Dynamisch).*

Hinweis

SQL-92 nimmt gleichfalls eine Klassifizierung vor – dort werden drei Ebenen als Entry SQL, Intermediate SQL *und* Full SQL *definiert. Gemäß dieser Einteilung entspricht Transact-SQL (des SQL Server 7.0) den Anforderungen des Entry SQL und enthält darüber hinaus einige wenige Funktionen von Intermediate und Full SQL. Die SQL-92- Konformitätsebenen korrespondieren nicht mit den ODBC-Konformitätsebenen. Jet unterstützt kaskadierende Löschoperationen und Aktualisierungen an verknüpften Tabellen zur automatischen Wahrung der referentiellen Integrität, verwendet zu diesem Zweck jedoch nicht die Prädikate `ON CASCADE DELETE` (des Intermediate SQL-Levels) oder `ON CASCADE UPDATE` (des Full SQL-Levels). Transact-SQL kennt das Schlüsselwort `CASCADE` nicht; um kaskadierende Operationen auszulösen, müssen Sie Transact-SQL-Trigger schreiben. SQL Server 7.0 unterstützt mehrfache Trigger in einem einzigen Ereignis (z.B. `INSERT`, `UPDATE` oder `DELETE`) und optional auch rekursive Trigger, die sich entweder direkt selbst aufrufen oder von anderen Triggern aufgerufen werden.*

6.2.2 Die formale Grammatik von SQL

Die formale Grammatik von SQL ist in der Backus Naur-Form (BNF) repräsentiert, in der die formale Grammatik vieler Computersprachen spezifiziert ist. Hier folgt nun die volle BNF der Bezeichnung, die die Operation »auf einer Datenbank soll eine Abfrage ausgeführt werden« spezifiziert:

```
<action> ::=
SELECT
|DELETE
|INSERT [ <left paren> <privilege column list> <right paren>]
|UPDATE [ <left paren> <privilege column list> <right paren>]
|REFERENCES  [ <left paren> <privilege column list> <right paren>]
|USAGE
...
<privilege column list> ::= <column name list>
...
<column name list> ::= <column name> [{<comma>, <column name>} ...]
```

Bei der Verwendung der BNF wählen Sie zunächst die Klasse aus (<action> im vorangegangenen Beispiel), in der das reservierte Wort enthalten ist. Mitglieder der Klasse werden durch einen vertikalen Strich (|) voneinander getrennt. Optionale Parameter von reservierten Wörtern und Elementen werden in eckigen Klammern ([]), Literalwerte (wie <privilege column list>) in spitzen Klammern (<>) und zu gruppierende Elemente (wie ein Komma, vor dem ein zweiter <column name> steht) in geschweiften Klammern ({}) eingeschlossen. Danach durchsuchen Sie die Liste der Elemente, um die richtige Zusammenstellung eines Elements herauszufinden. Im vorangehenden Beispiel ist das Element <privilege column list> aus der <column name list> zusammengesetzt. Danach überprüfen Sie, ob das Element <column name list> ebenfalls aus weiteren Elementen zusammengesetzt ist (in diesem Fall ein oder mehrere <column name>-Elemente). Insgesamt ist dieser Vorgang äußerst langwierig, besonders wenn die einzelnen Elemente nicht in alphabetischer Reihenfolge sortiert sind.

Microsoft verwendet eine vereinfachte Form der BNF für die von der aktuellen Version der ODBC-API unterstützten Grammatik. Die Syntaxregeln von Jet-SQL verzichten auf die Notwendigkeit, die ::=-Zeichen zu verwenden, um die zugelassenen Ersetzungswerte eines Elements zu bestimmen, und verwenden statt dessen ein tabellarisches Format (siehe Tabelle 6.1). Ellipsen (. . .) in der folgenden Tabelle verdeutlichen, dass dieses Element gesucht werden muss; dieses Element grenzt nicht an das vorangehende Element der Tabelle an.

Element	Syntax
select-statement	SELECT[ALL|DISTINCT|DISTINCTROW] select-list table-expression ...

Tab. 6.1: Ein Ausschnitt aus der Syntax der SELECT-Anweisung in Jet-SQL

Element	Syntax
`select-list`	*|select-sublist[{, select-sublist}...]
`select-sublist`	table-name.*|expression [AS column-alias]|column-name ...
`table-expression`	from-clause|[where-clause]|[group-by-clause]|[having-clause]|[order-by-clause] ...
`from-clause`	FROM table-reference-list
`table-reference-list`	table-reference [{, table-reference}...]
`table-reference`	table-name [AS correlation-name|joined-table] ...
`table-name`	base-table-name|querydef-name|attached-table-name|correlation-name

Tab. 6.1: Ein Ausschnitt aus der Syntax der SELECT-Anweisung in Jet-SQL

Hinweis

> *Der Qualifizierer `DISTINCTROW` und das Element `querydef-name` sind Jet-SQL-spezifisch. `DISTINCTROW` wird im Abschnitt »Theta-Joins und das Schlüsselwort `DISTINCTROW`« später in diesem Kapitel behandelt und in Kapitel 3 wurden Jet-`QueryDef`-Objekte näher erläutert. ADO unterstützt `QueryDef`-Objekte nur sehr begrenzt.*

Nachdem Sie nun alle erlaubten Elementformen in der vorangehenden Tabelle durchgeschaut haben, haben Sie möglicherweise die eigentlichen Schlüsselwörter vergessen, deren Syntax Sie ja verwenden wollen. Die von Microsoft verwendete modifizierte BNF-Form ist ungleich einfacher einzusetzen als das im ANSI SQL-Standard verwendete komplette BNF.

6.2.3 Grammatik einer einfachen SQL SELECT-Anweisung

Hier folgt nun eine wesentlich leichter umsetzbare Syntaxdarstellung einer typischen ANSI SQL-Anweisung, die Bindestriche durch Unterstriche ersetzt:

```
SELECT [ALL|DISTINCT] select_list
   FROM table_names
    [WHERE {search_criteria|join_criteria}
      [{AND|OR search_criteria}]
  [ORDER BY {field_list} [ASC|DESC]]
```

Im Folgenden wird jedes einzelne reservierte SQL-Wort aus der vorangehenden Anweisung erklärt:

- `SELECT` zeigt an, dass die Abfrage Daten aus der Datenbank zurückliefert und sie nicht verändert. Das *`select_list`*-Element enthält die Namen des Felds (der Felder) aus der Tabelle, die im Abfrageergebnis enthalten sein sollen. Mehrfache Felder werden in einer durch Kommas getrennten Liste verzeichnet. Ein Stern (`*`) deutet an, dass die Daten aller Tabellenfelder zurückgegeben werden. Ist mehr als eine Tabelle in die Abfrage eingebunden, verwenden Sie die Syntax *`table_name.field_name`*, wobei der Punkt (`.`) den Tabellennamen vom Feldnamen trennt.
- Der Qualifizierer `ALL` spezifiziert, dass das Abfrageergebnis alle Zeilen der Tabelle, unabhängig von Duplikatwerten, berücksichtigt. `DISTINCT` gibt lediglich Zeilen, die keine doppelten Werte enthalten, zurück; allerdings nimmt diese Abfrage mehr Zeit in Anspruch.
- `FROM` leitet eine Klausel ein, die den/die Tabellennamen beinhaltet, aus dem/denen die Felder stammen, die in der Auswahlliste *`select_list`* angegeben sind. Ist in *`select_list`* mehr als eine Tabelle angegeben, enthält *`table_list`* eine durch Kommas getrennte Liste der verwendeten Tabellennamen.
- `WHERE` leitet eine Klausel ein, die in ANSI SQL zwei verschiedene Zwecke erfüllt: Angabe der Felder, über die Tabellen in der Abfrage verknüpft werden, sowie Begrenzen der zurückgegebenen Datensätze auf solche Datensätze, deren Feldwerte einem bestimmten Kriterium (oder einem Bündel von Kriterien) entsprechen. Die `WHERE`-Klausel muss einen Operator und zwei Operanden – deren erster ein Feldname sein muss – enthalten. (Dieser Feldname muss nicht notwendigerweise auch in *`select_list`* enthalten sein; jedoch muss der Tabellenname, der das entsprechende Feld beinhaltet, im *`table_name`*-Ausdruck enthalten sein.)
- SQL-Operatoren sind `LIKE`, `IS {NULL|NOT NULL}` und `IN` sowie die bekannten arithmetischen Operatoren <, <=, =, =>, > und <>. Wenn Sie den arithmetischen Gleichheitsoperator (=) einsetzen und *`table_name.field_name`*-Werte für beide Operanden angeben, erstellen Sie eine Gleichheitsverbindung (auch *Inner Join* genannt) zwischen den beiden Tabellen anhand der spezifizierten Felder. Ebenso können Sie Links- und Rechts-Inklusionsverknüpfungen anhand der speziellen Operatoren *= bzw. =* erzeugen, falls die verwendete Datenbank *Outer Joins* unterstützt. (Links- und Rechts-Inklusionsverknüpfungen werden beide als *Outer Joins* bezeichnet.) Die verschiedenen Verbindungstypen werden im Abschnitt »Tabellen verbinden« näher erläutert.

Hinweis

Wenn Sie mehr als eine Tabelle in Ihrer Abfrage verwenden, stellen Sie sicher, dass Sie eine Verbindung zwischen den Tabellen mit einer `WHERE Table1.field_name = Table2.field_name`*-Klausel definieren. Wenn Sie die Anweisung, die die Verbindung zwischen den Tabellen definiert, weglassen, wird Ihre Abfrage ein »kartesisches Produkt« aus beiden Tabellen zurückgeben. Als »kartesisches Produkt« bezeichnet man die Kombination aller Felder und aller Zeilen zweier Tabellen. Das Ergebnis ist eine äußerst umfangreiche Ergebnismenge; falls die Quelltabellen selbst bereits einen sehr ausgedehnten Umfang aufweisen, kann es vorkommen, dass Ihr Computer die Operation mit einem Speicherüberlauf abbricht. (Der Begriff »kartesisch« ist abgeleitet vom Namen eines berühmten französischen Mathematikers, René Descartes.) Ein »kartesisches Produkt« ist die letzte »Abfrage vor der Hölle«.*

- `ORDER BY` definiert eine Klausel, die die Sortierreihenfolge der von `SELECT` zurückgegebenen Datensätze bestimmt. Sie geben hier die Felder an, nach denen die Ergebnismenge sortiert werden soll. Mit dem Qualifizierer `DESC` kann eine absteigende Sortierung erzwungen werden; aufsteigende (`ASC`) Reihenfolge ist die Standardeinstellung. Wie in anderen Listenangaben auch: Wenn Sie mehr als einen Tabellennamen einsetzen wollen, verwenden Sie eine durch Kommas getrennte Liste. Falls Sie mit verbundenen Tabellen arbeiten, sollten Sie die spezifische `table_name.field_name`-Syntax einsetzen.

Abhängig vom verwendeten SQL-Dialekt und von der Methode, mit der die SQL-Anweisung zum DBM verschickt wird, müssen Sie die komplette Anweisung mit einem Semikolon abschließen. (Jet-SQL erfordert inzwischen keine Semikola mehr; Anweisungen, die vom ODBC-Treiber direkt zum Datenbankserver gesendet werden, erfordern ebenfalls keinen Abschluss durch ein Semikolon.)

6.2.4 Einsatz der Visual Data-Beispielanwendung, um Abfragen zu untersuchen

Das Add-In Visual Data Manager sowie die Beispielanwendung VisData (...\MSDN98\98VS\1031\Samples\VB98\Visdata\Visdata.vbp) in Visual Basic 6.0 sind erweiterte Versionen der Visual Data Manager-Anwendung, die ursprünglich aus Visual Basic 2.0 stammt. VisData fällt in die Anwendungskategorie »Ad-hoc-Abfragengenerator«. Sie können diese Anwendung dazu verwenden, einfache SQL-Anweisungen zu testen, indem Sie die folgenden Schritten ausführen:

1. Wählen Sie ADD-INS / VISUAL DATA MANAGER, um VisData zu starten.
2. Im Anwendungsfenster von VisData wählen Sie DATEI / DATENBANK ÖFFNEN; die Auswahl von MICROSOFT ACCESS öffnet das Dialogfeld MICROSOFT ACCESS-DATENBANK ÖFFNEN.

3. Im Verzeichnis `...\Vb98` aktivieren Sie den Eintrag `Biblio.mdb` und klicken anschließend auf ÖFFNEN. Biblio.mdb ist eine Jet 3.0-Datenbank. Falls ein Meldungsfenster erscheint und fragt, ob Sie eine System.md?-Datei verwenden wollen, wählen Sie NEIN. Die Eigenschaften, die `TableDef`- und `QueryDef`-Objekte von Biblio.mdb erscheinen nun im Strukturansicht-Steuerelement des Datenbankfensters.

4. Falls das Fenster SQL-ANWEISUNG nicht automatisch mitgeöffnet wird, wählen Sie FENSTER | SQL-ANWEISUNG und anschließend FENSTER / NEBEN-/UNTEREINANDER, um beide MDI-Fenster sichtbar zu machen.

5. Klicken Sie auf die Schaltfläche DBGRID-STEUERELEMENT IN NEUEM FORMULAR VERWENDEN, um die Ergebnismenge in einem datengebundenen Datengitter-Steuerelement (Grid) anzuzeigen; klicken Sie anschließend auf die Schaltfläche RECORDSET VON TYP DYNASET.

6. Im Fenster SQL-ANWEISUNG tippen Sie testhalber (um die Arbeitsweise des Visual Data Manager zu überprüfen) die Anweisung `SELECT * FROM Authors` als einfache Abfrage ein (siehe Abbildung 6.1).

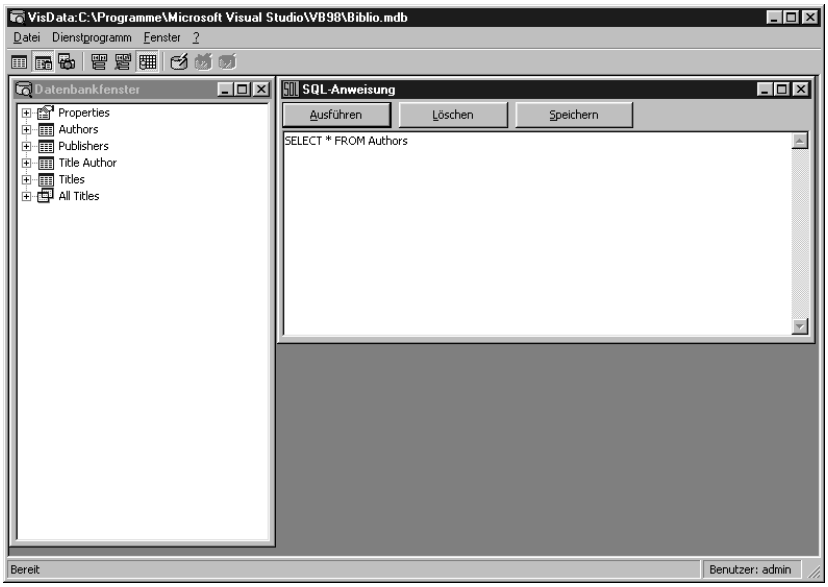

Abb. 6.1: Das Add-In Visual Data Manager führt eine SQL-Testabfrage aus

7. Klicken Sie auf die Schaltfläche AUSFÜHREN des Fensters SQL-ANWEISUNG. Falls ein Meldungsfenster mit der Frage »Handelt es sich um eine SQLPass-Through-Abfrage?« erscheint, klicken Sie auf NEIN. Die Ergebnismenge erscheint im Datengitter-Steuerelement des Fensters SELECT * FROM AUTHORS, wie in Abbildung 6.2 gezeigt.

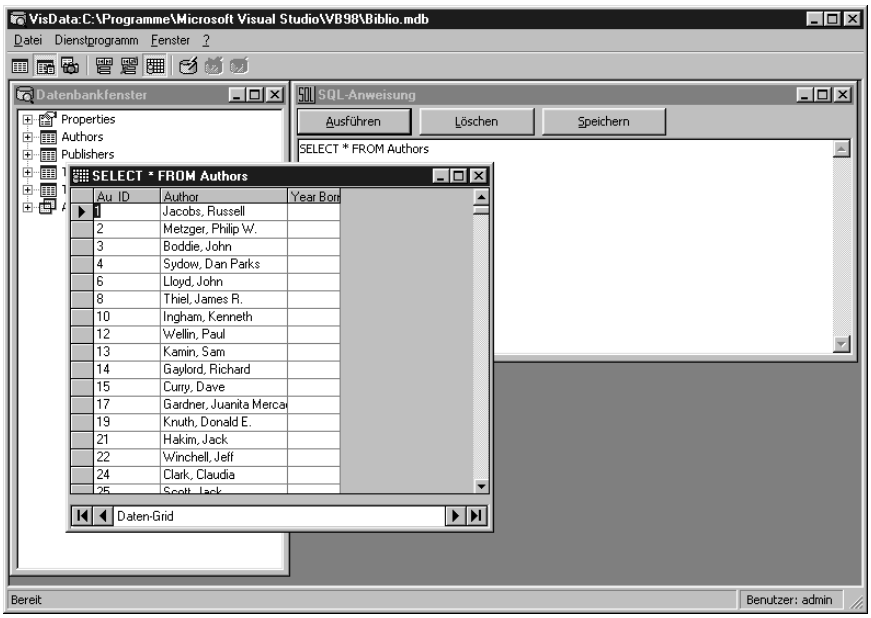

Abb. 6.2: Das Ergebnisfenster des Add-Ins VisData verwendet hier das DBGrid-Steuerelement

8. Schließen Sie das Ergebnisfenster und löschen Sie Ihre SQL-Anweisung durch einen Klick auf die Schaltfläche LÖSCHEN. Schreiben Sie nun folgende SQL-Anweisung `SELECT * FROM Publishers WHERE State = 'NY'` in das Fenster SQL-ANWEISUNG und klicken Sie auf AUSFÜHREN. (Einzelne und doppelte Anführungszeichen sind austauschbar in Jet-SQL.) Wieder lehnen Sie die auftauchende Frage »Handelt es sich um eine SQLPassThrough-Abfrage?« durch einen Klick auf NEIN ab. Das Ergebnisfenster SELECT * FROM PUBLISHERS WHERE STATE = 'NY' erscheint (siehe Abbildung 6.3). Standardmäßig sortiert Jet-SQL das `Recordset` nach dem Primärschlüsselwert (`PubID`).

9. Schließen Sie erneut das Ergebnisfenster und fügen Sie am Ende Ihrer bestehenden Abfrageanweisung den Ausdruck `ORDER BY Zip` hinzu. Führen Sie diese Abfrage aus; das Ergebnis sehen Sie in Abbildung 6.4.

10. Ersetzen Sie das `*` der `SELECT`-Anweisung – wodurch alle Datensätze zurückgegeben werden – durch den Ausdruck `PubID, [Company Name], City`. Dadurch erscheinen nur die drei angegebenen Felder im DBGrid-Ergebnisfenster. Das Ergebnis (gezeigt in Abbildung 6.5) demonstriert, dass Sie in der *`field_names`*-Liste der `SELECT`-Anweisung nicht zwingend die Felder mit angeben müssen, die Sie in der `WHERE`- oder `ORDER BY`-Klausel verwenden.

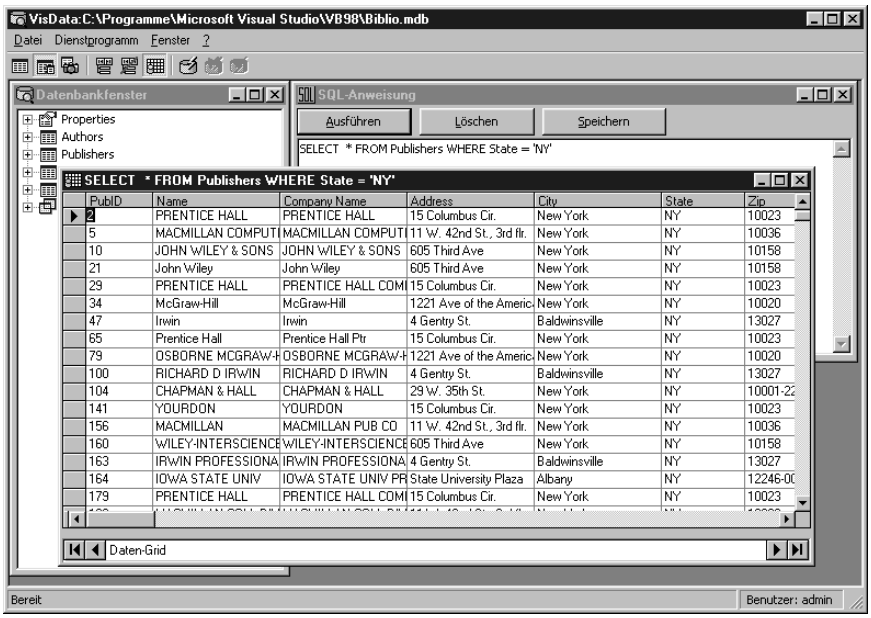

Abb. 6.3: Eine Abfrage, die alle Datensätze von Verlegern (publishers) zurückgibt, die in New York wohnen

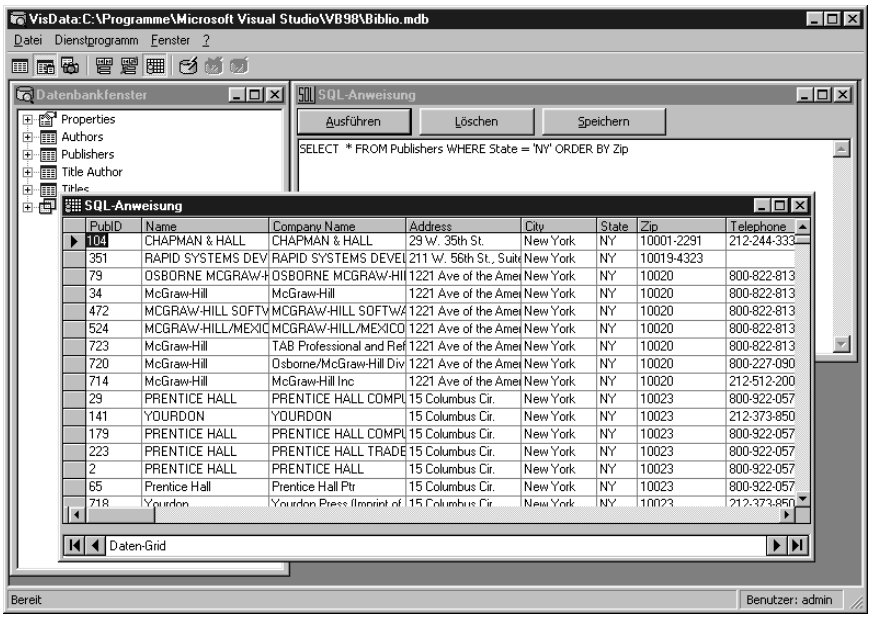

Abb. 6.4: Die Datensätze von Verlegern aus New York, sortiert nach dem Zip-Code

Hinweis

Die eckigen Klammern (`[]`) – wie um den Ausdruck `Company Name` – sind dann notwendig, wenn ein Feldname oder Tabellenname Leerzeichen enthält. Nur Jet-Datenbanken erlauben Leer- und andere Interpunktionszeichen außer dem Unterstrich (_) in Feldnamen. Die Verwendung von Leerzeichen in Feld- oder Tabellennamen (oder bei der Benennung anderer Datenbankobjekte) zählt nicht zu einem guten Datenbank-Programmierstil. In diesem Buch tauchen Leerzeichen in Datenbank-Feld- oder Tabellennamen nur dann auf, wenn sie in Beispieldatenbanken vorkommen, die von anderen erzeugt und uns zur Verfügung gestellt wurden.

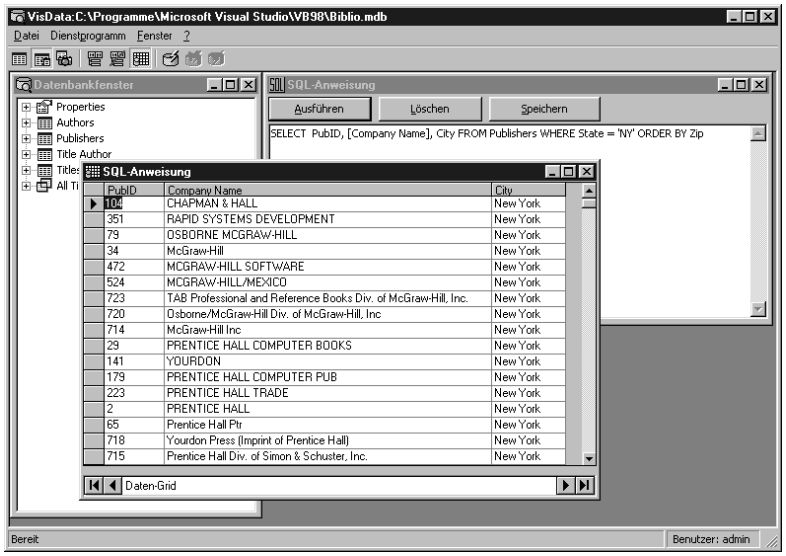

Abb. 6.5: Die Ergebnismenge zeigt nur drei Felder der Verleger-Tabelle

Hinweis

Das DBGrid-Fenster des Visual Data Managers enthält Schaltflächen, die erlauben, die angezeigten Daten zu filtern und zu sortieren, so dass lediglich ausgewählte Datensätze der Ergebnismenge in einer veränderten Reihenfolge angezeigt werden. Ein Filter ist gleichbedeutend mit dem Hinzufügen einer `WHERE field_name where-expression`-Klausel zur SQL-Anweisung. Die Sortier-Schaltfläche entspricht einer `ORDER BY field_names`-Klausel einer SQL-Anweisung. Sie können die Datensätze auch manuell sortieren, indem Sie den Feldnamenbereich im Spaltenkopf des DBGrid-Steuerelements anklicken.

Die VisData-Beispielanwendungen wurden von Microsoft entwickelt, um Features zu demonstrieren, die zur Manipulation und Anzeige von aus Datenbanken stammenden Tabellendaten von Bedeutung sind. VisData bildet eine umfangreiche Quelle von Visual Basic-Codebeispielen; es enthält ein Klassenmodul (`VisDataClass`), mit dessen Hilfe VisData als Visual Basic 6.0-Add-In oder separat als OLE-Automatisierungsserver eingesetzt werden kann. In VisData sind ebenfalls einsatzfähige Codebeispiele enthalten, um die Erstellung und Erscheinungsweise von untergeordneten Formularen in MDI-Anwendungen zu bestimmen bzw. anzupassen.

6.2.5 SQL-Operatoren und -Ausdrücke

Wie bereits weiter vorn erwähnt, stellt SQL die gebräuchlichen arithmetischen Operatoren wie <, <=, =, =>, > und <> zur Verfügung. Diesen Operatoren fügt SQL weitere hinzu, die entweder in Verbindung mit Werten von Textfeldern (`LIKE` oder `IN`) oder zur Behandlung von `NULL`-Werten (`IS NULL` bzw. `IS NOT NULL`) eingesetzt werden. Die Jet 3.51-Datenbankengine unterstützt darüber hinaus den Einsatz vieler nativer VBA-String- sowie nummerischer Funktionen in SQL-Anweisungen. Hierdurch ist es möglich, Spaltenwerte von Ergebnismengen zu berechnen. (Nur wenige dieser VBA-Funktionen sind auch in ANSI SQL vorhanden.)

Hinweis

Access unterstützt den Gebrauch von benutzerdefinierten Funktionen (UDF, User Defined Functions) in SQL-Anweisungen, um Spaltenberechnungen in Abfragen durchzuführen. Visual Basic unterstützt nur native VBA-Funktionen in Form von reservierten Wörtern (wie z.B. `Val()`). Funktionen, die nicht den SQL-eigenen Aggregatfunktionen entsprechen, werden in ANSI SQL als implementationsspezifisch betrachtet. Der Begriff implementationsspezifisch *bedeutet, dass RDBMS-Vertreiber in ihrer Entscheidung frei sind, der eigenen Implementierung von ANSI SQL Funktionen hinzuzufügen, solange die Namen der Funktionen sich nicht mit Originalbezeichnungen von reservierten Wörtern aus dem Sprachschatz von SQL-92 überschneiden.*

Die meisten Operatoren in SQL-Anweisungen sind dyadisch. Dyadische (binäre) Funktionen erfordern zwei Operanden. (Alle arithmetischen Funktionen sowie das reservierte Wort `BETWEEN` sind dyadisch.) Operatoren wie `LIKE`, `IN`, `IS NULL` oder `IS NOT NULL` sind monadisch. Monadische (unäre) Funktionen benötigen lediglich einen Operanden. Ausdrücke mit Vergleichsoperatoren geben `True` oder `False` zurück, jedoch keine Werte. Die folgenden Abschnitte beschreiben detailliert den Einsatz der gebräuchlichen dyadischen bzw. monadischen Operatoren in ANSI SQL.

Dyadische arithmetische Operatoren und Funktionen

Der Gebrauch arithmetischer Operatoren in SQL unterscheidet sich nicht wesentlich von deren Einsatz in Visual Basic oder anderen Computersprachen. Die folgende Aufzählung listet einige Punkte auf, die es beim Einsatz arithmetischer Operatoren und Funktionen in SQL-Anweisungen (besonders in `WHERE`-Klauseln) zu bedenken gilt:

- Die Vergleichsoperatoren = und <> werden bei Text- und nummerischen Felddatentypen eingesetzt. Das »ungleich«-Symbol aus zwei spitzen Klammern (<>) ist äquivalent zur Zeichenkombination `!=` in ANSI SQL. (SQL verwendet das Gleichheitszeichen (=) nicht als Zuweisungsoperator.)
- Die arithmetischen Vergleichsoperatoren <, <=, => und > sind vorwiegend für den Einsatz bei Operanden nummerischen Datentyps gedacht. Wenn Sie diese Vergleichsoperatoren mit Werten aus Textfeldern einsetzen, werden die nummerischen ANSI-Zeichen jedes einzelnen Buchstabens der beiden Felder miteinander von links nach rechts verglichen.

Hinweis

> *Um Werte aus Textfeldern zu vergleichen, die Zahlenwerte enthalten (wie das `Zip`-Feld der `Publishers`-Tabelle der Datenbank Biblio.mdb), umschließen Sie den betreffenden Wert mit einfachen oder doppelten Anführungszeichen. Beispiel:* `SELECT * FROM Publishers WHERE Zip > '12000'`.

- Die verbleibenden Operatoren – +, –, *, / sowie ^ oder ** (die beiden letzten bilden implementationsspezifische Exponentialoperatoren) – sind keine Vergleichsoperatoren. Diese werden lediglich für Berechnungen von Spaltenwerten in Ergebnismengen verwendet (dies ist Thema des nächsten Abschnitts).
- Vermeiden Sie den Einsatz von nativen VBA-Funktionen zur Definition einer berechneten Spalte, wenn Sie SQLPassThrough-Methoden, ADO, ODBCDirect oder RDO einsetzen, um Abfragen direkt auf dem RDBMS-Server auszuführen. Nur wenige Client-/Server-Datenbanken unterstützen gegenwärtig die `VAL()`-Funktion bzw. deren SQL-92-Gegenstück: das Prädikat `CAST`.
- Das Prädikat `BETWEEN` in ANSI SQL bzw. der `BETWEEN`-Operator in Jet-SQL wird bei nummerischen oder Datums- bzw. Zeit-Felddatentypen verwendet. Die Syntax ist *`field_name`* `BETWEEN` *`Value1`* `AND` *`Value2`*. Diese Syntax entspricht dem Ausdruck *`field_name`* `=>` *`Value1`* `OR` *`field_name`* `<=` *`Value2`*. Jet-SQL verlangt, Datums- und Zeitwerte mit Nummernzeichen (#) zu umschließen, z. B. `DateField BETWEEN #1-1-97# AND #31-12-97#`. Datumswerte werden von den meisten SQL-Datenbanken als Strings behandelt. Das Prädikat `NOT BETWEEN` kehrt die Funktion von `BETWEEN` um.

Hinweis

Wo Jet-SQL eine Syntax verwendet, die nicht von ANSI SQL spezifiziert ist (z.B. das #-Zeichen zur Kennzeichnung von Datums- und Zeitfeldern), oder wo in kompletten Jet-SQL-Anweisungen reservierte SQL-Wörter verwendet werden, die gleichfalls in VBA Schlüsselwörter oder reservierte Wörter darstellen, wird für deren Schreibweise (Groß- oder Kleinbuchstaben) die Konvention von VBA verwendet.

Berechnete Spalten

Sie können berechnete Spalten in Ergebnismengen erstellen, indem Sie Felder hinzufügen, die arithmetische SQL-Operatoren verwenden. Sie können auch Funktionen einsetzen, die von der Jet 3.0-Datenbankengine oder Ihrem Client-/Server-RDBMS unterstützt werden. Normalerweise werden berechnete Spalten von Feldern nummerischen Datentyps abgeleitet. Die Datenbank Biblio.mdb verwendet einen nummerischen Datentyp (`long integer Counter`, für Autoincrement-Felder) für das Kennzeichen- (ID-)Feld. Um Berechnungen anzustellen, können Sie das Feld `PubID` oder den Ausdruck `Val(Zip)` zur Grundlage nehmen. Schreiben Sie als Beispiel die Anweisung `SELECT PubID, Name, Zip, (3 * Val(Zip)) AS Zip_Mal_3 FROM Publishers` in das Fenster SQL-ANWEISUNG des Visual Data Managers. (Diese Abfrage ist inhaltlich nicht sehr sinnvoll, sie soll lediglich das oben geschilderte Vorgehen verdeutlichen.) Das Abfrageergebnis sehen Sie in Abbildung 6.6.

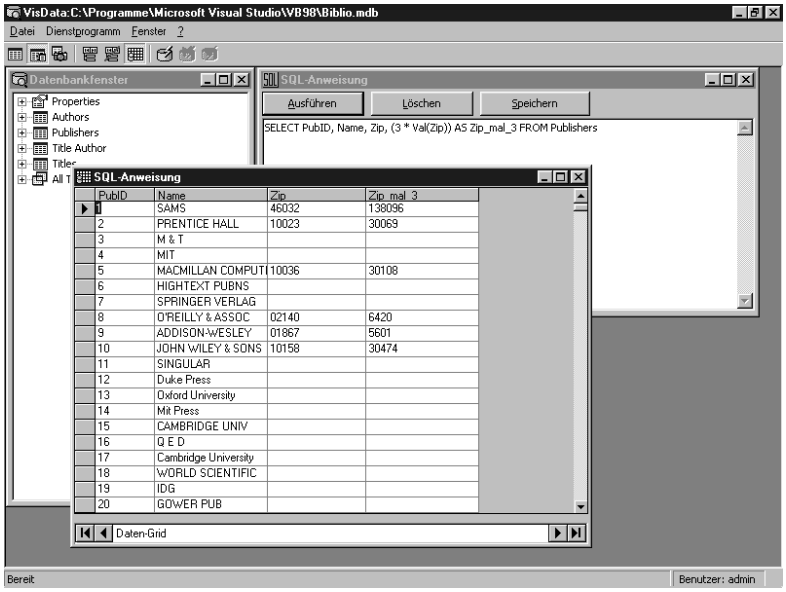

Abb. 6.6: Der Abfrage über die Publishers-Tabelle wurde eine berechnete Spalte hinzugefügt

Der Qualifizierer `AS` bezeichnet einen Alias für den neuen Spaltennamen (*`column_alias`*). In Jet 3.0 ist der Qualifizierer `AS` optional; Sie können statt dessen ein Leerzeichen in die SQL-Anweisung einfügen, wie in `SELECT PubID, Name, Zip, 3 * Val(Zip) Zip_Mal_3 FROM Publishers`. Wenn Sie den Namen für die neue berechnete Spalte nicht angeben, setzt die Jet-Datenbankengine den Ausdruck `Expr1003` als Spaltennamen ein; Jet 1.x stellte den Standard-Alias `Expr1` als Namen für berechnete Spalten zur Verfügung. Der *`column_alias`*, der bei Verwendung von ODBC zur Datenbankverbindung erscheint, ist abhängig von der jeweiligen Implementation. DB2 von IBM beispielsweise stellt keinen Standard-Alias für Spaltennamen mit dem Qualifizierer `AS` zur Verfügung. ODBC-Treiber für DB2 bzw. DB2/2 weisen möglicherweise einer berechneten Spalte den/die Feldnamen zu, von dem/denen die berechneten Werte stammen – oder sie weisen einen willkürlichen Namen zu, wie `Col_1`.

Hinweis

Wenn Sie Leerzeichen im `column_alias` verwenden müssen, sollten Sie den `column_alias` entweder in eckigen Klammern einschließen, wenn Sie die Jet-Datenbankengine verwenden, oder in einfachen Anführungszeichen, wenn Sie ein anderes RDBMS verwenden, das Leerzeichen in Feldnamen unterstützt. Auch wenn Sie Spaltennamen wie `Col 1` entdecken (z.B. innerhalb einer Terminalemulationssitzung bei einer Abfrage an die DB2-Datenbank oder auch an andere Mainframe-Datenbanken), werden diese `column_alias`-Werte vom Abfragewerkzeug auf dem lokalen PC generiert und nicht von der Datenbank selbst.

Monadische Textoperatoren, NULL-Werte und Funktionen

Einer der gebräuchlichsten Operatoren für die WHERE-Klausel – zur Kriterienspezifizierung in Feldern vom Typ Text – ist das ANSI SQL-Prädikat `LIKE`, in Jet-SQL »LIKE-Operator« genannt. (Die Begriffe *Prädikat* und *Operator* sind in diesem Zusammenhang austauschbar.) Das Prädikat `LIKE` erlaubt, nach einem oder mehreren angegebenen Zeichen an beliebiger Stelle im Text zu suchen. Die Tabelle 6.2 stellt die Syntax des ANSI SQL `LIKE`-Prädikats der Syntax des `Like`-Operators in Jet-SQL gegenüber – jeweils eingesetzt in der `WHERE`-Klausel einer SQL-Anweisung.

Zweck	ANSI SQL	Jet SQL	Rückgabe
Übereinstimmung mit beliebigem Text, der diese Zeichen enthält	`LIKE '%am%'`	`Like "*am*"`	ram, rams, damsel, amnesty
Übereinstimmung mit beliebigem Text, der mit diesen Zeichen beginnt	`LIKE 'John%'`	`Like "John*"`	Johnson, Johnsson

Tab. 6.2: Syntax von LIKE (ANSI SQL-Prädikat) und Like (Jet-SQL-Operator).

Zweck	ANSI SQL	Jet SQL	Rückgabe
Übereinstimmung mit beliebigem Text, der mit diesen Zeichen endet	`LIKE '%son'`	`Like "*son"`	Johnson, Anderson
Übereinstimmung mit dem angegebenen Text sowie einem beliebigen folgenden Zeichen	`LIKE 'Glen_'`	`Like "Glen?"`	Glenn, Glens
Übereinstimmung mit dem angegebenen Text sowie einem beliebigen voranstehenden Zeichen	`LIKE '_am'`	`Like "?am"`	dam, Pam, ram
Übereinstimmung mit dem angegebenen Text sowie einem beliebigen voranstehenden Zeichen und einem oder mehreren folgenden Zeichen	`LIKE '_am%'`	`Like "?am*"`	dams, Pam, Ramses

Tab. 6.2: Syntax von LIKE (ANSI SQL-Prädikat) und Like (Jet-SQL-Operator).

Die Prädikate `IS NULL` bzw. `IS NOT NULL` überprüfen, ob ein Wert in das betreffende Feld eingegeben wurde. `IS NULL` gibt `False` und `IS NOT NULL` gibt `True` zurück, wenn das Feld einen Wert – auch einen leeren Zeichenfolgenwert (`""`) oder die Ziffer `0` – beinhaltet. Die VBA-Funktion `IsNull(`*`field_name`*`)` und der SQL-Ausdruck *`field_name`* `IS NULL` sind gleichbedeutend.

Hinweis

*Aus Konsistenzgründen mit dem Standard ANSI SQL verwendet der Jet-OLE DB-Datenprovider (Microsoft.JET.OLEDB.3.51) wieder die ANSI SQL-eigenen Platzhalterzeichen % und _, nicht die Jet-üblichen Zeichen ? und *. Diese Inkonsistenz erfordert leider die Korrektur von bestehendem DAO-Code, um die nun ADO-konforme Verwendung von SQL-Platzhaltern zu gewährleisten.*

Mit den Schlüsselwörtern `Option Compare Binary` (unterscheidet zwischen Groß- und Kleinschreibung) und `Option Compare Text` (unterscheidet *nicht* zwischen Groß- und Kleinschreibung) bestimmen Sie die Schreibweise (ob Buchstaben in Großschrift mit Buchstaben in Kleinschrift übereinstimmen sollen oder nicht) bei einer Suche in einer Visual Basic-Datenbankanwendung, die die Jet 3+-Datenbankengine benutzt. Die Unterscheidungsfähigkeit zwischen Groß- und Kleinschreibung beim Prädikat `LIKE` ist installationsabhängig, wenn Sie Abfragen mit der SQL `PassTrough`-Option an ein beliebiges Client-/Server-Datenbanksystem richten wollen. Der jeweilige Datenbank-Manager konfiguriert die Groß-/Kleinschreibung von Suchoperationen bei der Installation des Datenbanksystems

oder bei Einrichtung einer neuen Datenbank. Bei SQL-PassThrough-Abfragen können Sie die SQL-92-Funktionen UPPER()bzw. LOWER()oder auch die entsprechenden VBA-Funktionen UCase() bzw. LCase() verwenden, um Groß-/Kleinschreibung bei Abfragen unberücksichtigt zu lassen. (Jet 3.0 unterstützt die ANSI SQL-Funktionen UPPER() und LOWER() nicht.)

Die SQL-92-Funktion POSITION()gibt die Position eines Zeichens von einem untersuchten Feldwert zurück, unter Beachtung folgender Syntax: POSITION(*characters* IN *field_name*). Die äquivalente Jet-SQL-Funktion – sie ist gleichzeitig auch eine VBA-Funktion – ist InStr(*field_name, characters*. Ist in *field_name* kein Wert für *characters* zu finden, gibt die Funktion den Wert 0 zurück.

Die SQL-92-Funktion SUBSTRING()gibt eine bestimmte Anzahl von Zeichen zurück; die Syntax ist SUBSTRING(*field_name* FROM *start_position* FOR *number_of_characters*). Die entsprechende Funktion in Jet-SQL sowie gleichzeitig in VBA heißt Mid(*field_name*, *start_position*[, *number_of_characters*]). Beide Funktionen sind sehr nützlich für die Auswahl und Durchsuchung von Textfeldern.

Transact-SQL des SQL Server bietet die Strukturen IF...ELSE sowie SELECT...CASE...WHEN...THEN für die bedingte Ausführung an. Die VBA-Funktion »inline-if« in Jet-SQL IIF(*LogicalExpression*, *ReturnValueIfTrue*, *ReturnValueIfFalse*) bietet einen Ersatz für die Konstruktion IF...ELSE von SQL Server, die jedoch weniger flexibel ist.

6.2.6 Tabellen verbinden

Wie bereits zuvor erwähnt, können Sie zwei Tabellen in einer Abfrage miteinander verknüpfen (Join), indem Sie die *table_name.field_name*-Operanden mit einem Vergleichsoperator innerhalb der WHERE-Klausel einer SQL-Anweisung versehen. Weitere Tabellen können der Verknüpfung durch Kombination zweier Anweisungen mit dem AND-Operator hinzugefügt werden. In SQL-86 und SQL-89 wurden lediglich WHERE-Verknüpfungen unterstützt. Mithilfe von WHERE-Klauseln können Sie Gleichheitsverknüpfungen, Ungleichheitsverknüpfungen sowie Reflexivverknüpfungen realisieren. Zu den Gleichheitsverknüpfungen zählen normale Gleichheitsverknüpfungen, Links- sowie Rechts-Inklusionsverknüpfungen. Verknüpfungen, die mit dem Gleichheitsoperator erstellt werden, werden als Gleichheitsverknüpfungen bezeichnet.

In SQL-92 wurden das reservierte Wort JOIN sowie dessen Qualifizierer CROSS, NATURAL, INNER, OUTER, FULL, LEFT und RIGHT – die eine Vielzahl an verschiedenen Verbindungen definieren – hinzugefügt. Als dieses Buch entstand, gab es nur wenige relationale Client-/Server-Datenbanksysteme, die JOIN-Anweisungen bereits unterstützten. (Der Microsoft SQL Server beispielsweise beinhaltete in Transact-SQL-Versionen vor 6.5 ebenfalls keine JOIN-Abfrageanweisungen.) Jet-SQL unterstützt INNER, LEFT und RIGHT JOINs mit SQL-92-Syntax unter Verwendung des ON-Prädikats. Weder die USING-Klausel noch die JOIN-Qualifizierer CROSS, NATURAL oder FULL werden von Jet-SQL unterstützt.

Ein `CROSS JOIN` gibt das kartesische Produkt zweier Tabellen zurück. Der Begriff `CROSS` leitete sich her von *cross-product*, einem Synonym für *kartesisches Produkt*. Sie können einen `CROSS JOIN` simulieren, indem Sie in der `WHERE`-Klausel einer `SELECT`-Anweisung die entsprechenden `JOIN`-Komponenten weglassen, wobei `SELECT` die Tabellennamen von mehr als einer Tabelle beinhalten muss. Abbildung 6.7 zeigt das DBGrid-Fenster des Visual Data Managers, das die ersten 18 Zeilen eines 4.540.842 Zeilen umfassenden kartesischen Produkts anzeigt, das durch die SQL-Anweisung `SELECT Publishers.Name, Authors.Author FROM Publishers, Authors` erzeugt wurde. Die Ergebnistabelle enthält 727 Publishers-Datensätze und 6.246 Authors-Datensätze; das ergibt ein kartesisches Produkt von 4.540.842 Zeilen (727 * 6.246 = 4.540.842). Es ist höchst unwahrscheinlich, dass Sie in einer Datenbankanwendung jemals einen `CROSS JOIN` verwenden werden, es sei denn, Sie wenden eine sehr spezielle Einschränkung in der `WHERE`-Klausel auf die Anweisung an.

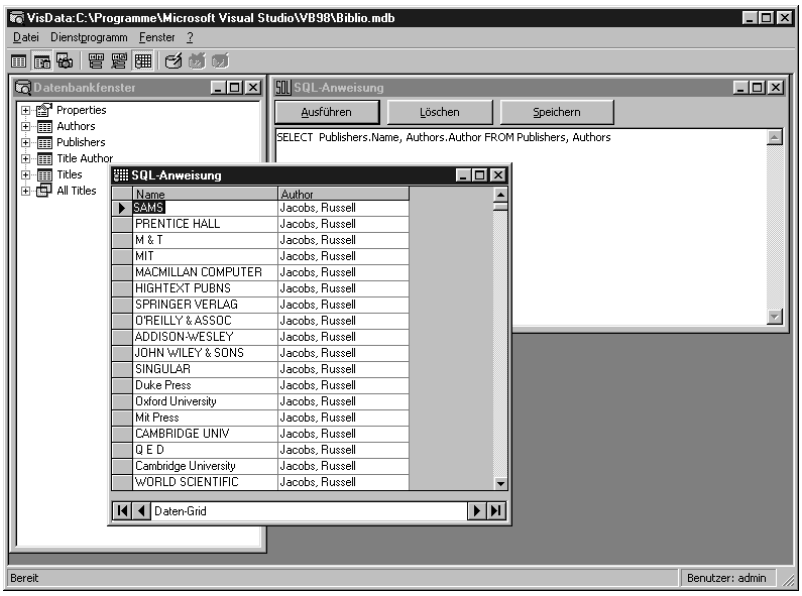

Abb. 6.7: Die ersten achtzehn Zeilen des 4.540.842 Zeilen umfassenden kartesischen Produkts aus den Tabellen Publishers und Authors

Verweis

Wenn Sie die zuvor beschriebene `CROSS JOIN`-Abfrage ausführen wollen, sollten Sie auf jeden Fall ein `Recordset`-Objekt vom Typ Dynaset zugrunde legen und keinesfalls versuchen, die Ergebnistabelle zu sortieren. Ein `Recordset`-Objekt vom Typ Snapshot bzw. die Sortierung der Ergebnistabelle nimmt schnell eine Stunde oder länger in Anspruch – je nach Geschwindigkeit Ihres PCs bzw. Ihrer Festplatte. Eine Ergebnismenge vom Typ Dynaset gibt lediglich eine genügende Anzahl von Zeilen zurück, um die sichtbaren Zeilen des DBGrid-Steuerelements mit Daten zu füllen.

Die gebräuchlichen Arten von `JOIN`s, die Sie mit SQL-89, SQL-92 und Jet-SQL erzeugen können, werden in den folgenden Abschnitten beschrieben.

Hinweis

Alle Verbindungen (`JOIN`s) – mit Ausnahme des `CROSS JOIN` oder kartesischen Produkts – erwarten die Definition identischer Felddatentypen der beiden Verknüpfungsfelder oder aber eine vom verwendeten RDBMS unterstützte Datenkonvertierungsfunktion, die nicht-kompatible Felddatentypen in kompatible Datentypen umwandelt.

Konventionelle Exklusions- bzw. Gleichheitsverknüpfungen

Der am häufigsten eingesetzte Verknüpfungstyp ist die Gleichheitsverknüpfung oder `INNER JOIN`. Sie können eine Gleichheitsverknüpfung innerhalb einer `WHERE`-Klausel nach folgendem Muster erzeugen:

```
SELECT Table1.field_name, ... Table2.field_name ...
   FROM Table1, Table2
   WHERE Table1.field_name = Table2.field_name
```

Die `JOIN`-Syntax von SQL-92 – bei gleichen Resultaten – lautet:

```
SELECT Table1.field_name, ... Table2.field_name ...
   FROM Table1 INNER JOIN Table2
      ON Table1.field_name = Table2.field_name
```

Eine einspaltige Gleichheitsverknüpfung zwischen dem `PubID`-Feld der `Publishers`-Tabelle und dem `PubID`-Feld der `Titles`-Tabelle aus der Datenbank Biblio.mdb sieht folgendermaßen aus:

```
SELECT Publishers.Name, Titles.ISBN, Titles.Title
   FROM Publishers INNER JOIN Titles
      ON Publishers.PubID = Titles.PubID
```

Wenn Sie diese Abfrage mit dem Visual Data Manager ausführen, werden die beiden Tabellen `Publishers` und `Titles` über die Spalte `PubID` beider Felder verknüpft. Das Ergebnis dieser Verbindungsabfrage sehen Sie in Abbildung 6.8.

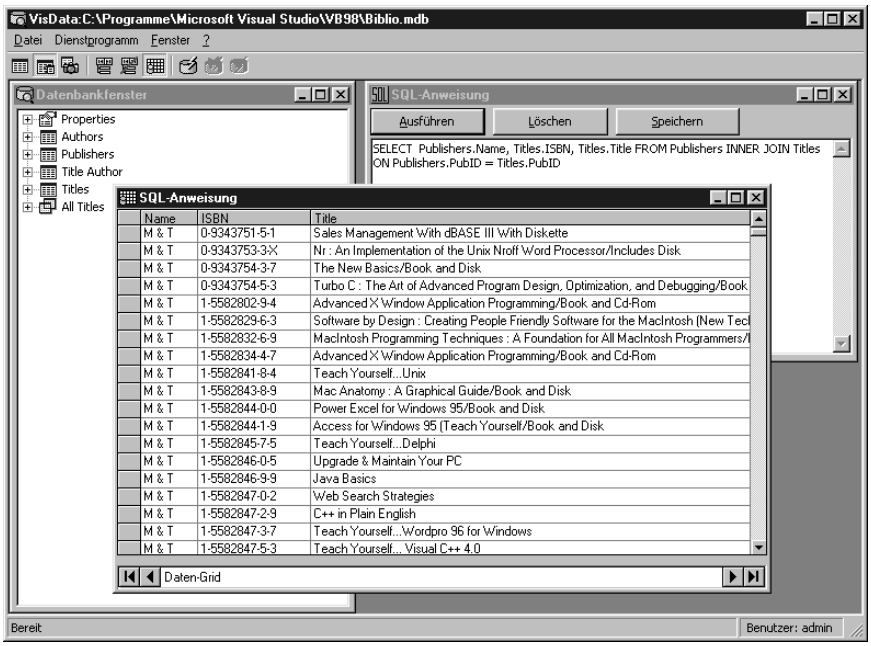

Abb. 6.8: VisData zeigt das Ergebnis einer INNER JOIN-Abfrage über das Feld PubID der Tabellen Publishers und Titles

Hinweis

Der Qualifizierer `INNER` ist optional in SQL-92, in Jet-SQL jedoch erforderlich. Falls Sie ihn weglassen, erscheint (vorausgesetzt Sie verwenden die Jet-Datenbankengine) die Fehlermeldung »Syntax-Fehler in FROM-Klausel«, sobald Sie versuchen, diese Abfrage auszuführen.

Hinweis

Normale (natürliche) Gleichheitsverknüpfungen erzeugen automatisch Verknüpfungen zwischen den identisch bezeichneten Feldern zweier Tabellen; sie benötigen nicht das `ON`-Prädikat in der `JOIN`-Anweisung. Die `NATURAL JOIN`-Anweisung wird von Jet-SQL wie auch von allen anderen kommerziellen Client-/Server-RDBMS-Systemen nicht unterstützt (zumindest bis zum Zeitpunkt, als dieses Buch entstand).

Jet-SQL-Anweisungen, die Sie im grafischen QBE-Entwurfsmodus in Access erstellen, generieren eine erweiterte `JOIN`-Syntax. Access trennt die `JOIN`-Anweisung mit einem Komma von der kompletten `FROM`-Klausel ab und wiederholt anschließend die Tabellennamen in einer eigenen, komplett deklarierten `JOIN`-

Anweisung. Die im folgenden Beispiel aufgeführte Jet-SQL-Syntax ergibt das gleiche Ergebnis wie die zuvor beschriebene ANSI SQL-92-Syntax.

```
SELECT DISTINCTROW Publishers.Name, Titles.ISBN, Titles.Title
   FROM Publishers, Titles,
   Publishers INNER JOIN Titles
      ON Publishers.PubID = Titles.PubID
```

Der Zweck der optionalen Anweisung `DISTINCTROW` in Jet-SQL wird im Abschnitt »Jet-SQL und ANSI SQL-92« später in diesem Kapitel besprochen.

Gleichbedeutend mit den beiden vorangehenden Syntaxbeispielen, die eine `WHERE`-Klausel für die Erzeugung eines `JOIN` verwenden, ist das folgende Beispiel:

```
SELECT Publishers.Name, Titles.ISBN, Titles.Title
   FROM Publishers, Titles
   WHERE Publishers.PubID = Titles.PubID
```

Das Ergebnis ist jedes Mal das gleiche, ob Sie nun die `INNER JOIN`-Anweisung verwenden oder ob Sie die Gleichheitsverknüpfung über eine einfache `WHERE`-Klausel (wie im letzten angeführten Beispiel) erreichen. Da die `WHERE`-Klausel – ohne Angabe von `INNER JOIN` – einfacher ist, verwenden nur wenige Entwickler tatsächlich die `JOIN`-Syntax.

Hinweis

Gleichheitsverknüpfungen geben nur Zeilen zurück, in denen die Feldwerte der verknüpften Felder übereinstimmen. Feldwerte von Datensätzen beider Tabellen, die keine Entsprechungen in den Werten der jeweils anderen Tabelle besitzen, erscheinen nicht in der von einer Gleichheitsverknüpfung zurückgegebenen Ergebnismenge. Werden keine Übereinstimmungen zwischen irgendwelchen Datensätzen gefunden, werden keine Zeilen zurückgegeben. Eine Ergebnismenge ohne Zeilen wird Null-Menge *oder* leere Menge *genannt. Die Fehlermeldung »Kein aktueller Datensatz« erscheint, wenn Sie versuchen, in einer leeren Menge zu navigieren (z.B. mithilfe eines Datensteuerelements, das an die leere Menge gebunden ist).*

Mehrfache Gleichheitsverknüpfungen

Sie können mehrfache Gleichheitsverknüpfungen erzeugen, indem Sie verschiedene Tabellen durch Feldpaare, die kompatible Datentypen aufweisen, miteinander verbinden. Zum Beispiel können Sie aus der Datenbank Biblio.mdb die Tabellen `Publishers`, `Titles`, `Title Author` und `Authors` durch die folgende Jet-SQL-Anweisung miteinander verbinden:

```
SELECT DISTINCTROW Titles.Title, Publishers.Name,
      Titles.ISBN, Authors.Author
   FROM Publishers INNER JOIN
     (Authors INNER JOIN
```

```
    (Titles INNER JOIN [Title Author]
        ON Titles.ISBN = [Title Author].ISBN)
        ON Authors.Au_ID = [Title Author].Au_ID)
        ON Publishers.PubID = Titles.PubID
```

Die gerade angeführte SQL-Anweisung verwendet eingebettete Verbindungen für die Herstellung der benötigten Verknüpfungen, um die Daten Titel, Verleger und Autor für jedes Buch anzuzeigen. Die Verwendung der `JOIN`-Syntax zur Erzeugung mehrfacher Verbindungen wird schnell sehr komplex. Das grafische Entwurfsfenster von Access erlaubt es, solche SQL-Anweisungen weit schneller zu erstellen, als es je durch Selbstentwickeln und -testen von Hand möglich wäre. Die Ergebnismenge dieser Jet-SQL- Abfrage erscheint in Abbildung 6.9.

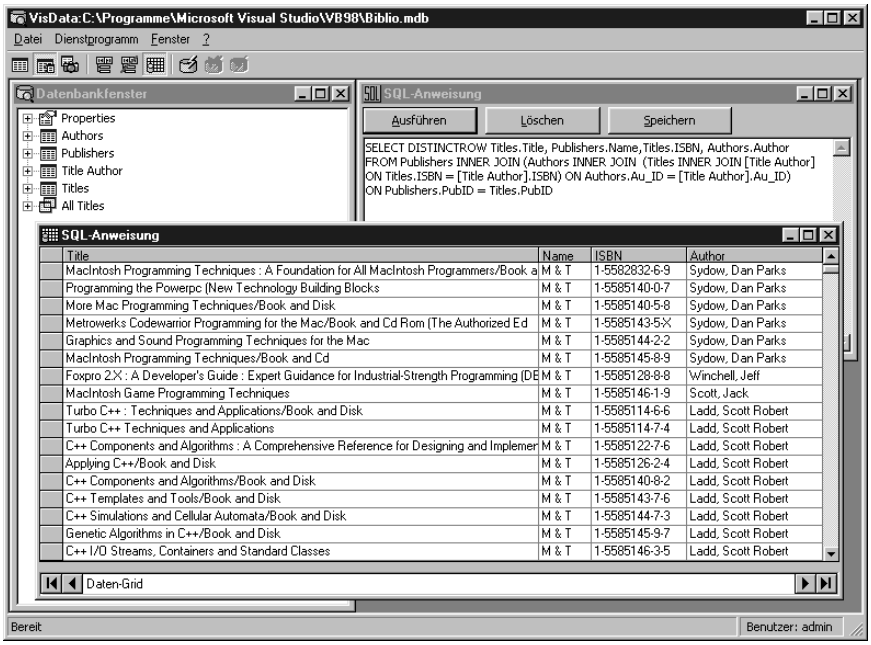

Abb. 6.9: Die Ergebnismenge aus vier verknüpften Tabellen

Äquivalent zur vorstehenden Abfrage nun die ANSI SQL-Anweisung unter Verwendung der `WHERE`-Klausel:

```
SELECT Titles.Title, Publishers.Name,
      Titles.ISBN, Authors.Author
   FROM Publishers, Titles, Authors, [Title Author]
   WHERE Titles.ISBN = [Title Author].ISBN AND
         Authors.Au_ID = [Title Author].Au_ID AND
         Publishers.PubID = Titles.PubID
```

Hinweis

Es gilt als Regel, dass die Verwendung der WHERE-Klausel bei Gleichheitsverknüpfungen einfachere Abfrageanweisungen ergibt im Gegensatz zur Erzeugung durch eine INNER JOIN-Anweisung. Wenn Sie im Rahmen Ihres Projekts jedoch auch OUTER JOINs erstellen müssen – diese sind Thema des nächsten Abschnitts – empfiehlt es sich aus Gründen der Konsistenz, auch in Jet-SQL die INNER JOIN-Syntax zu verwenden.

Inklusionsverknüpfungen

Exklusionsverknüpfungen oder INNER JOINs (Gleichheitsverknüpfungen) geben nur Zeilen mit übereinstimmenden Werten zurück. Inklusionsverknüpfungen oder OUTER JOINs geben alle Zeilen einer der Tabellen zurück sowie von der anderen Tabelle nur die Zeilen mit einem übereinstimmenden Wert im Verknüpfungsfeld. Es gibt zwei Arten von Inklusionsverknüpfungen:

- LEFT OUTER JOIN gibt alle Zeilen der Tabelle (bzw. der Ergebnismenge) zurück, die links von der LEFT OUTER JOIN-Anweisung angegeben ist, sowie nur die Zeilen mit übereinstimmenden Werten von der Tabelle, die rechts der JOIN-Anweisung steht. Alternativ können Sie – bei Verwendung von WHERE-Klauseln – den Operator *= anstatt der Anweisung LEFT OUTER JOIN spezifizieren.
- RIGHT OUTER JOIN gibt alle Zeilen der Tabelle (bzw. der Ergebnismenge) zurück, die rechts von der RIGHT OUTER JOIN-Anweisung angegeben ist, sowie nur die Zeilen mit übereinstimmenden Werten von der Tabelle, die links der JOIN-Anweisung steht. Alternativ können Sie – bei Verwendung von WHERE-Klauseln – den Operator =* anstatt der Anweisung RIGHT OUTER JOIN spezifizieren.

Konventionsgemäß werden Inklusionsverknüpfungen in einer 1:n-Form erzeugt. Die Primärtabelle – sie stellt die »1«-Seite der Beziehung dar – erscheint in der Abfrageanweisung links des JOIN-Ausdrucks bzw. des entsprechenden Operators einer WHERE-Klausel. Die verknüpfte Tabelle – sie stellt die »n«-Seite der Beziehung dar – erscheint auf der rechten Seite des JOIN-Ausdrucks bzw. des entsprechenden Operators. Der Einsatz von LEFT OUTER JOIN listet alle Datensätze der Primärtabelle auf, ungeachtet von Übereinstimmungen zu Feldern der verknüpften Tabelle. Die Verwendung von RIGHT OUTER JOINs ist nützlich, um verwaiste Datensätze aufzuspüren. *Verwaist* sind Datensätze in verknüpften Tabellen dann, wenn sie keine zugehörigen Datensätze in der Primärtabelle aufweisen. Verletzungen der referentiellen Integritätsregeln sind häufig die Ursache für verwaiste Datensätze.

Die SQL-92-Syntax für eine Abfrage, die als Ergebnis alle Titel-Datensätze – ohne Rücksicht auf übereinstimmende Werte in der Autorentabelle – auflistet, lautet:

```
SELECT Titles.Title, [Title Author].ISBN, [Title Author].Au_ID
   FROM Titles LEFT OUTER JOIN [Title Author]
      ON Titles.ISBN = [Title Author].ISBN
```

Die gleichlautende Inklusionsabfrage (ebenfalls ANSI SQL-konform), jedoch unter Einsatz der `WHERE`-Klausel:

```
SELECT Titles.Title, [Title Author].ISBN, [Title Author].Au_ID
   FROM Titles
      WHERE Titles.ISBN *= [Title Author].ISBN
```

Jet-SQL verlangt, die spezielle Syntax des vorstehenden Beispiels zu verwenden, und erlaubt nicht den Einsatz des reservierten Worts `OUTER` in einer `JOIN`-Anweisung. Access 97 lässt bei der Eingabe der Abfrageanweisung das reservierte Wort `OUTER` zwar zu, ignoriert es jedoch bei der Ausführung. Hier nun die Jet-SQL-Entsprechung der obigen Anfrageanweisung:

```
SELECT DISTINCTROW Titles.Title, [Title Author].ISBN,
    [Title Author].Au_ID
   FROM Titles LEFT JOIN [Title Author]
      ON Titles.ISBN = [Title Author].ISBN
```

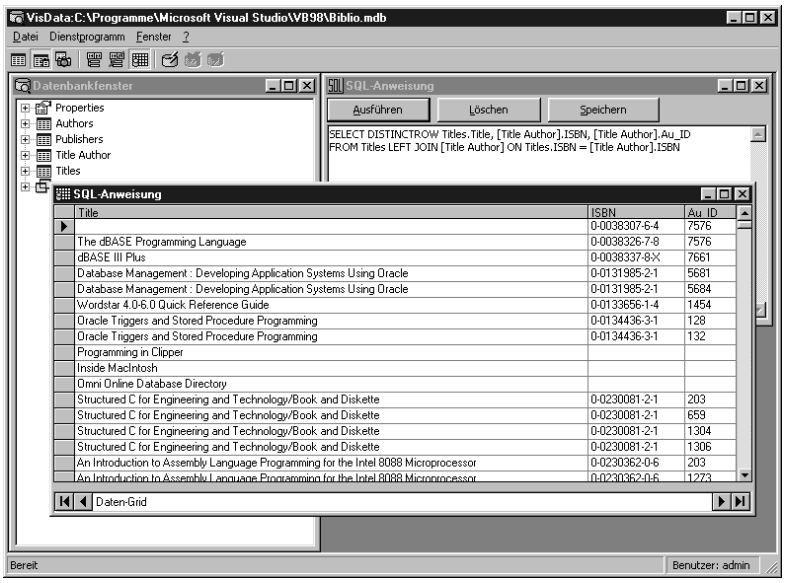

Abb. 6.10: Das Ergebnis einer Abfrage an die Biblio.mdb-Datenbank, in der der Ausdruck INNER JOIN durch LEFT JOIN ersetzt wurde

Das Ergebnis dieser Abfrage an die Biblio.mdb-Datenbank erscheint in Bild 6.10. Sie erhalten in der Ergebnismenge 279 anstatt 241 Zeilen, da durch Verwendung

der Inklusionsverknüpfung `LEFT JOIN` 38 Titel-Datensätze auftauchen, die keine Entsprechung in der Autorentabelle aufweisen.

Hinweis

*Jet-SQL unterstützt nicht die Operatoren *= und =* in `WHERE`-Klauseln. Um in Jet-SQL Inklusionsabfragen zu erstellen, müssen Sie die reservierten Wörter `LEFT JOIN` bzw. `RIGHT JOIN` verwenden. Diese Restriktion gilt jedoch nicht für SQLPassThrough-Abfragen, die an Server gerichtet werden, die die Operatoren *= und =* unterstützen (z.B. SQL Server).*

Hinweis

Wenn Sie versuchen, die Tabelle `Authors` der vorangehenden Abfrage hinzuzufügen, erhalten Sie folgende Fehlermeldung: »Verknüpfungsausdruck nicht unterstützt. Nummer: 3296«. Grund: Die Jet 3.5x-Datenbankengine kann die folgende SQL-Anweisung nicht in einer einzigen Abfrageanweisung ausführen:

```
SELECT DISTINCTROW Titles.Title, Titles.ISBN,
     [Title Author].Au_ID
   FROM Authors INNER JOIN
     (Titles LEFT JOIN [Title Author]
      ON Titles.ISBN = [Title Author].ISBN)
      ON Authors.Au_ID = [Title Author].Au_ID
```

Sie müssen zwei eingebettete Abfragen ausführen: Die erste Abfrage erzeugt die `LEFT JOIN`-Ergebnismenge, die zweite Abfrage verbindet über eine `INNER JOIN` die Tabellen `Authors` mit `Title Author`. Eingebettete Abfragen sind Inhalt des Abschnitts »Eingebettete Abfragen und Unterabfragen« später in diesem Kapitel.

Theta Joins und das Schlüsselwort DISTINCTROW

Sie können Verknüpfungen mit anderen Vergleichsoperatoren außer =, *= und =* erstellen. Joins (Verknüpfungen), die keine Gleichheitsverknüpfungen darstellen, werden *Theta Joins* genannt. Die häufigste Form von Theta Joins ist die Ungleich- (Nicht-gleich-) Verknüpfung mit folgender Syntax: `WHERE table_name.field_name <> table_name.field_name`. Die Biblio.mdb-Datenbank enthält keine Tabellen mit Feldern, die sich für eine Demonstration von nicht-gleichen Joins eignen. Öffnen Sie daher im Visual Data Manager die Datenbank Nwind.mdb, um eine Abfrage auszuführen, die Datensätze in der `Orders`-Tabelle herausfindet, deren Werte im Feld `ShipAddress` sich von den Angaben im Feld `Address` der Tabelle `Customers` unterscheidet. Führen Sie hierzu folgende Abfrage aus:

```
SELECT DISTINCTROW Customers.CompanyName, Customers.Address,
      Orders.ShipAddress
   FROM Customers, Orders,
   Customers INNER JOIN Orders
      ON Customers.CustomerID = Orders.CustomerID
   WHERE Orders.ShipAddress<>Customers.Address
```

Diese Abfrage über Tabellen der Nwind.mdb-Datenbank ergibt eine Ergebnismenge, die die Abbildung 6.11 zeigt.

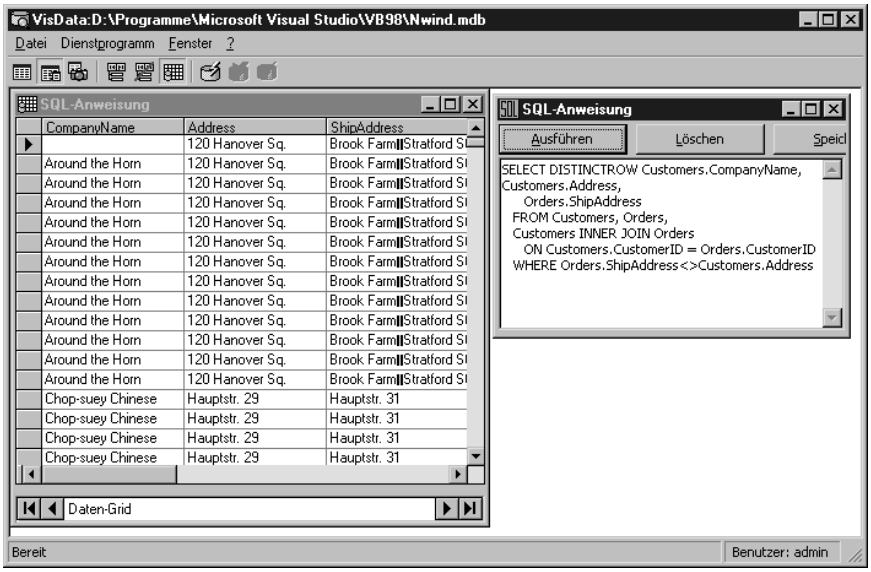

Abb. 6.11: Ein nicht-gleicher Theta Join zeigt Kundenadressen, deren Rechnungs- und Lieferanschrift nicht übereinstimmen

Wenn Sie die gleiche Abfrage ohne den Jet-SQL-Qualifizierer `DISTINCTROW` ausführen, bekommen Sie dasselbe Ergebnis. Ersetzen Sie jedoch das von Jet-SQL stammende `DISTINCTROW` durch den ANSI SQL-Qualifizierer `DISTINCT`, weichen die Ergebnisse merklich voneinander ab, wie in Abbildung 6.12 zu sehen.

Die aus gerade fünf Zeilen bestehende Ergebnismenge (siehe Abbildung 6.12) wurde mit der folgenden Anweisung erstellt, die im Übrigen in ANSI SQL und Jet-SQL gleich ist:

```
SELECT DISTINCT Customers.CompanyName, Customers.Address,
      Orders.ShipAddress
     FROM Customers, Orders
   WHERE Customers.CustomerID = Orders.CustomerID
     AND Orders.ShipAddress<>Customers.Address
```

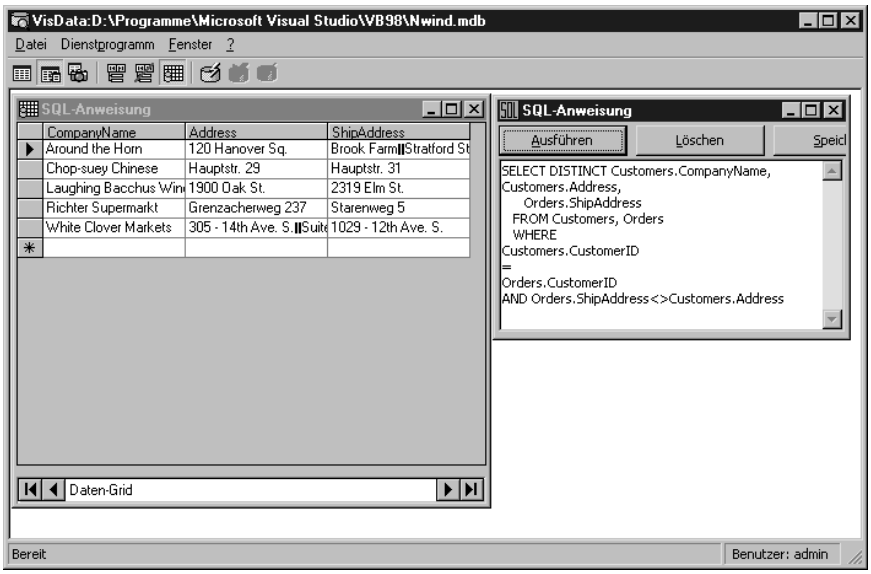

Abb. 6.12: Der Unterschied in der Ergebnismenge (gegenüber Bild 6.11) durch Einsatz des DISTINCT-Qualifizierers

Der `DISTINCT`-Qualifizierer veranlasst die Abfrage, nur Zeilen zurückzugeben, die unterschiedliche Feldwerte in den durch `SELECT` spezifizierten Feldern aufweisen. Im Gegensatz dazu bestimmt der Jet-SQL-Qualifizierer `DISTINCTROW`, dass in der Ergebnismenge alle Zeilen enthalten sind, die unterschiedliche Werte in irgendeinem Feld der beiden Tabellen aufweisen – der Feldvergleich ist in diesem Fall nicht beschränkt auf Felder, die in der `SELECT`-Anweisung angeführt wurden.

Reflexivverknüpfungen und zusammengesetzte Spalten

Eine Reflexivverknüpfung ist eine Verknüpfung zweier Felder derselben Tabelle, die vergleichbare (kompatible) Datentypen aufweisen. Das erste Feld einer Reflexivverknüpfung ist häufig das Primärschlüsselfeld, das andere Feld gewöhnlich ein Fremdschlüsselfeld mit Beziehung zum Primärschlüsselfeld; jedoch ist eine solche Struktur keine Bedingung für eine Reflexivverknüpfung – sie sollte jedoch beachtet werden, will man sinnvolle Ergebnisse erzielen.

Wenn Sie eine Reflexivverknüpfung erstellen, erzeugt das RDBMS eine Kopie der Originaltabelle und verknüpft dann die Tabellenkopie mit der Originaltabelle. In Biblio.mdb gibt es keine Tabellen, deren Felder sich für sinnvolle Reflexivverknüpfungen eignen würden. Die Tabelle `Employees` der Datenbank Nwind.mbd dagegen beinhaltet das Feld `ReportsTo` (Vorgesetzter), das die `EmployeeID` (Personal-Nr.) des Vorgesetzten dieses Mitarbeiters enthält. Es folgt nun die entsprechende Jet-SQL-Anweisung, die in einer Reflexivverknüpfung innerhalb der Tabelle `Employees` anzeigt, welche Mitarbeiter welche Vorgesetzten haben:

```
SELECT Employees.EmployeeID AS EmpID,
      Employees.LastName & ", " & Employees.FirstName AS Employee,
      Employees.ReportsTo AS SupID,
      EmpCopy.LastName & ", " & EmpCopy.FirstName AS Supervisor
   FROM Employees, Employees AS EmpCopy,
      Employees INNER JOIN EmpCopy
      ON Employees.ReportsTo = EmpCopy.EmployeeID
```

Die für Reflexivverknüpfungen notwendige temporäre Tabellenkopie (hier `EmpCopy` genannt) erstellen Sie in der `FROM`-Klausel durch die Anweisung `FROM ... Employees AS EmpCopy`. Jeder Feldname in der Abfrage wird durch den `AS`-Qualifizierer mit einem Aliasnamen versehen. Die Spalten `Employee` und `Supervisor` sind zusammengesetzte Spalten, deren Werte durch Kombination der Felder `LastName` und durch Komma getrennt `FirstName` gebildet werden. Die Ergebnismenge sehen Sie in Abbildung 6.13.

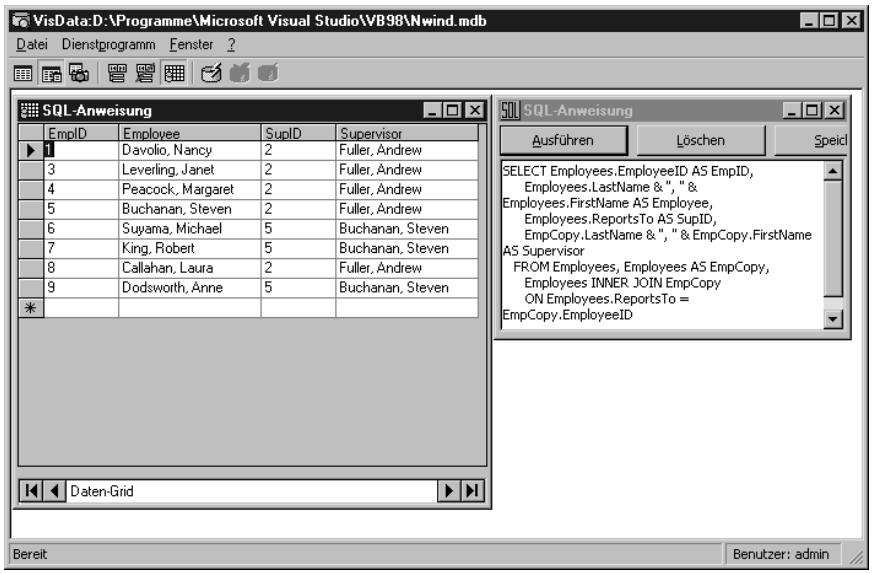

Abb. 6.13: Die Ergebnismenge einer Reflexivverknüpfung innerhalb der Tabelle Employees der Datenbank Nwind.mdb

ANSI SQL unterstützt keine `SELF INNER JOIN`-Verknüpfung, jedoch können Sie eine zum vorangehenden Beispiel äquivalente Fassung erzeugen, die ANSI SQL-konform ist. Sie müssen hierzu lediglich die Anweisung `INNER JOIN ... ON` ersetzen durch die Klausel `WHERE Employees.ReportsTo = EmpCopy.EmployeeID`.

Hinweis

Reflexivverknüpfungen sind eher ungebräuchlich, zumal eine Tabelle in der vierten Normalform kein zu `ReportsTo` entsprechendes Feld enthalten würde. Eine eigene Tabelle würde zu den Werten des Felds `EmployeeID` in Tabellen für Mitarbeiter und Vorgesetzte verknüpft. Allerdings führt das Anlegen einer zusätzlichen Tabelle mit Informationen, die sich auch ohne Mehrdeutigkeiten durchaus in einer einzigen Tabelle unterbringen lassen, zu einer Übernormalisierung. Die Furcht vor Übernormalisierung ist der hauptsächliche Grund dafür, dass viele Datenbankentwickler den Normalisierungsprozess bei der dritten Normalform beenden.

6.2.7 Verschachtelte Abfragen und Unterabfragen

Die Jet-Datenbankengine erlaubt, persistente `QueryDef`-Objekte zu erzeugen, die anstelle von Tabellen in SQL-Anweisungen benutzt werden können. Ein `QueryDef`-Objekt wird deshalb als *persistent* bezeichnet, weil die Abfragedefinition (`QueryDef`) als benanntes `Document`-Objekt in einer Jet-Datenbank gespeichert wird. Ein `QueryDef`-Objekt ist vergleichbar einer von einer SQL-Anweisung erstellten SQL-Sicht. Abfragen, die auf einem `QueryDef`-Objekt anstatt einer Tabelle basieren, werden *verschachtelte* Abfragen genannt. Die Jet-Datenbankengine behandelt `TableDef`- und `QueryDef`-Objekte aus Benennungsgründen als einheitliche Objektklasse; es ist deshalb unmöglich, ein `QueryDef`-Objekt und eine Tabelle mit gleichem Namen in derselben Datenbank zu haben. Wenn Sie eine auf einem `QueryDef`-Objekt basierende Abfrage ausführen, erstellt Jet zunächst auf Grundlage des `QueryDef`-Objekts (manchmal auch *Innere Abfrage* genannt) durch deren Ausführung eine Ergebnismenge in Form eines `Recordset`-Objekts. Danach führt Jet die eigentliche, primäre Abfrage (auch *Äußere Abfrage* genannt) aus, die sich an das zuvor erstellte `Recordset` richtet.

Eine verschachtelte Abfrage ist die einzige praktikable Methode, um die Fehlermeldung »Verknüpfungsausdruck nicht unterstützt. Nummer: 3296« zu umgehen, wenn man `INNER` und `OUTER JOIN`s in derselben Abfrage miteinander kombinieren will (siehe hierzu den Abschnitt »Inklusionsverknüpfungen« weiter vorn in diesem Kapitel). Um mit dem Visual Data Manager ein persistentes `QueryDef`-Objekt zu erstellen und anschließend das von `QueryDef` erzeugte `Recordset` mit der `Authors`-Tabelle zu verbinden, vollziehen Sie bitte die folgenden Schritte nach:

1. Löschen Sie im Visual Data Manager die letzte SQL-Anweisung; schreiben Sie für das `QueryDef`-Objekt die folgende SQL-Anweisung in das SQL-Textfenster:

```
SELECT DISTINCTROW Titles.Title, [Title Author].ISBN,
[Title Author].Au_ID
FROM Titles LEFT JOIN [Title Author]
ON Titles.ISBN = [Title Author].ISBN
```

2. Führen Sie die Abfrage aus, um die Syntax zu überprüfen. Klicken Sie auf die Schaltfläche SPEICHERN, um das VisData-Eingabefenster zu öffnen; tragen Sie den Namen `qdfAllTitles` in das Textfeld ein (siehe Abbildung 6.14). Klicken Sie auf OK, um Ihre Abfrage in Biblio.mdb zu speichern und zugleich das Dialogfeld zu schließen. Verneinen Sie eine eventuell auftauchende Meldung mit der Frage, ob es sich um eine SQL-Pass-Through-Abfrage handelt (Klick auf NEIN).

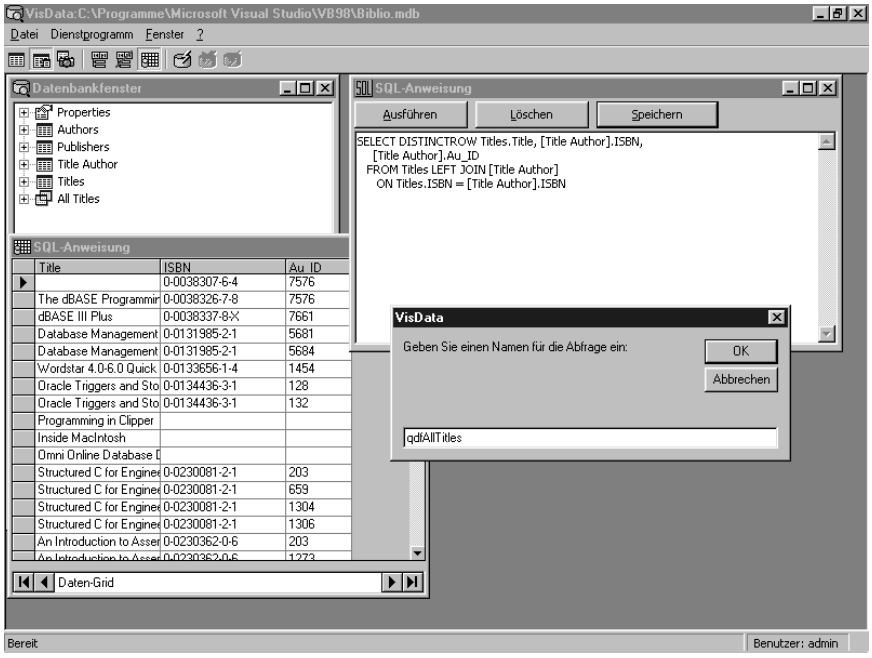

Abb. 6.14: Das Speichern einer Inneren Abfrage als QueryDef-Objekt

3. Löschen Sie das SQL-Anweisungsfenster erneut und geben Sie den folgenden SQL-Anweisungstext ein:

```
SELECT DISTINCTROW qdfAllTitles.Title, qdfAllTitles.ISBN,
Authors.Author
FROM qdfAllTitles LEFT JOIN Authors
ON qdfAllTitles.Au_ID = Authors.Au_ID
```

4. Ein Klick auf die Schaltfläche AUSFÜHREN startet die innere und äußere Abfrage. Die erzeugte Ergebnismenge sehen Sie in Abbildung 6.15.

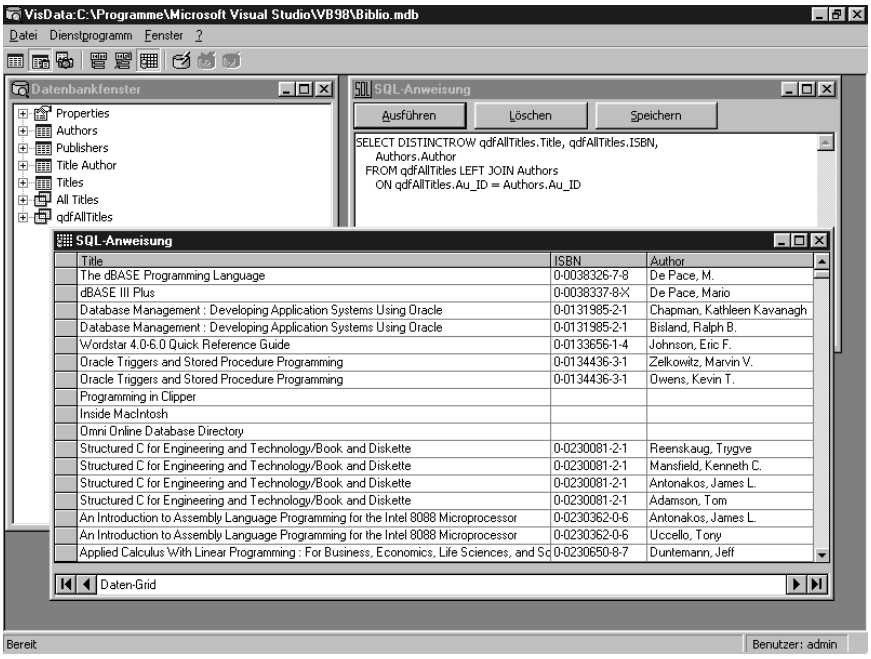

Abb. 6.15: Die Ausführung der äußeren Abfrage bewirkt, dass zunächst die innere Abfrage abgearbeitet wird

Verschachtelte Abfragen und Unterabfragen sind eng verwandt. Mit Erscheinen von Access 2.0 wurde in Jet 2.0 die Fähigkeit zu Unterabfragen eingeführt. Im vorangegangenen Beispiel nimmt das `QueryDef`-Objekt den Platz einer SQL-Unterabfrage ein. Unterabfragen können Sie in der *`field_list`* einer `SELECT`-Anweisung einsetzen oder als ein Kriterium innerhalb einer `WHERE`- oder `HAVING`-Klausel einer Abfrageanweisung. Unterabfragen sind darauf begrenzt, lediglich eine einzige Spalte zurückzuliefern, und kommen in zwei Abwandlungen vor:

- Unterabfragen von Untermengen definieren mithilfe einer `SELECT`-Anweisung eine Ergebnismenge, um die Menge von Datensätzen zu limitieren, an die sich die äußere, primäre Abfrage dann richtet. Das typische Syntaxschema einer äußeren Abfrage, die eine innere Untermengen-Unterabfrage beinhaltet, ist die folgende:

```
SELECT field_list
   FROM table_list
   WHERE field_name [NOT] IN
     (SELECT field_name
        FROM table_name
        WHERE search_criteria)
```

- Das IN-Prädikat ist in diesem Fall dem =-Operator gleichzusetzen, der in konventioneller SQL-Syntax eine Gleichheitsverknüpfung herbeiführt. Der Einsatz des NOT-Operators ist gleichbedeutend mit dem <>-(Theta Join) Operator. Unterabfragen besitzen jedoch keine Fähigkeit zu Links-Inklusionsverknüpfungen (LEFT JOIN oder *=); die vorangegangene verschachtelte (QueryDef-) Abfrage kann mit der Unterabfragen-Methode also nicht nachgebildet werden.
- Vergleichende Unterabfragen haben ebenfalls das Ziel, die Menge der Datensätze für die äußere Abfrage zu begrenzen – hier jedoch mithilfe der Abschätzung nummerischer Werte. Das Syntaxschema einer äußeren Abfrage, die eine vergleichende Unterabfrage enthält, ist das folgende:

```
SELECT field_list
   FROM table_list
   WHERE field_name {<|<=|=|=>|>} {ANY|SOME|ALL}
     (SELECT field_name
        FROM table_name
        WHERE search_criteria)
```

- Das Prädikat ANY oder SOME gibt die Datensätze zurück, für die ein Vergleich mit beliebigen Datensätzen der Unterabfrage True ergibt. Das Prädikat ALL gibt die Datensätze zurück, für die ein Vergleich mit allen Datensätzen der Unterabfrage True ergibt.

Die folgende SQL-Anweisung gibt Datensätze aus der Nwind.mdb-Datenbank für jeden Artikel der Tabelle Products mit einem Stückpreis (Feld UnitPrice) zurück, der gleich oder größer als jeder beliebige Artikelpreis der Tabelle Order Details – abzüglich eines Rabatts von mindestens 25% – ist:

```
SELECT ProductID, ProductName, CategoryID
   FROM Products
   WHERE UnitPrice > ANY
     (SELECT UnitPrice FROM [Order Details]
         WHERE Discount >= .25)
```

Diese Abfrage gibt 76 der 77 möglichen Werte der Tabelle Products zurück (siehe Abbildung 6.16); lediglich Artikel Nr. 33 – Geitost – stimmt mit dem Suchkriterium nicht überein. Wenn Sie nun das Prädikat ANY durch ALL ersetzen, bekommen Sie lediglich ein einziges Produkt in der Ergebnismenge zurück: Côte de Blaye, Artikel-Nr. 38 – ohne Zweifel ein französischer Wein aus einem hervorragenden Jahrgang. Der Preis des Côte de Blaye ist als einziger größer oder gleich dem aller anderen Artikel, auf die ein Rabatt von mind. 25% gewährt wurde.

Unterabfragen können auch in Zusammenhang mit INSERT-, UPDATE- und DELETE-Aktionsabfragen verwendet werden; diese Abfragen werden in Kapitel 7 näher behandelt. Weitere Beispiele für den Gebrauch von Unterabfragen finden Sie in der Visual Basic 6.0 Online-Hilfe unter dem Stichwort *Unterabfragen*. Unterabfragen sind sehr nützlich, jedoch ist es nicht leicht, sie korrekt zu definieren. In

den meisten Fällen kann anstatt einer Unterabfrage eine verschachtelte Abfrage verwendet werden. Verschachtelte Abfragen verbuchen für sich den Vorteil, dass die Jet-Engine das `QueryDef`-Objekt – die innere Abfrage – bereits vorkompiliert, was möglicherweise einen Geschwindigkeitsvorteil ergibt. Sichten des SQL Servers, die `QueryDef`-Objekten, die Zeilen zurückgeben, eng verwandt sind, bieten gewöhnlich Performancevorteile im Vergleich zu Unterabfragen.

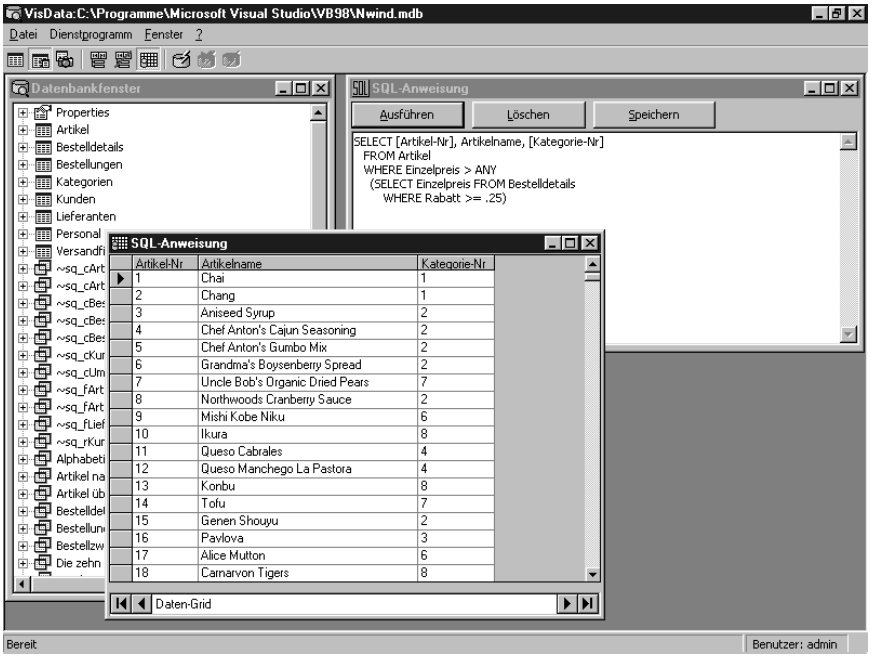

Abb. 6.16: Die Ergebnismenge einer Abfrage mit dem ANY-Operator, die eine Unterabfrage beinhaltet

6.2.8 UNION-Abfragen

`UNION`-Abfragen – eingeführt mit Jet 2.0 – ermöglichen, die Ergebnismengen zweier unabhängiger Abfragen miteinander zu kombinieren. Das Syntaxschema von `UNION`-Abfragen lautet:

```
{TABLE table_name1|SELECT field_list1 FROM table_list1}
   UNION [ALL]
    {TABLE table_name2|SELECT field_list2 FROM table_list2}
```

Die Anzahl der Felder von *`table_name1`* muss der Anzahl von Feldern in *`table_name2`* oder *`field_list2`* genau entsprechen, jedoch müssen die Felddatentypen nicht übereinstimmen. Doppelte Zeilen werden nicht zurückgegeben, es sein denn, Sie setzen das Prädikat `ALL` ein. Es folgt ein Beispiel einer SQL-

Anweisung (abgeleitet von einem Beispiel aus der Online-Hilfe unter dem Stichwort »UNION Operation«), das Datensätze von Kunden und Lieferanten aus Brasilien zurückgibt:

```
SELECT CompanyName, City, SupplierID AS ID
   FROM Suppliers
   WHERE Country = "Brazil"
   UNION SELECT CompanyName, City, CustomerID
      FROM Customers
      WHERE Country = "Brazil"
   ORDER BY City
```

Das Feld `SupplierID` hat den Datentyp `Long`, `CustomerID` jedoch den Typ `Text`, wie aus Abbildung 6.17 hervorgeht. Die Feldnamen der Ergebnismenge inklusive der Aliasnamen richten sich nach den Feldnamen der ersten Abfrage. Sie können eine `ORDER BY`-Klausel am Ende der `UNION`-Abfrage einsetzen, um die Sortierreihenfolge der gesamten Ergebnismenge zu bestimmen.

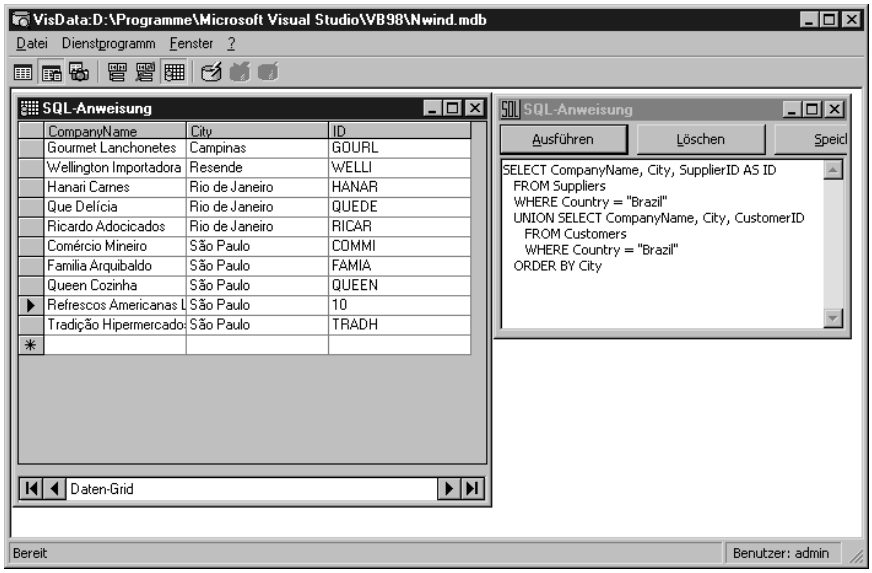

Abb. 6.17: Die Ergebnismenge einer UNION-Abfrage, die Werte von verschiedenen Datentypen im Feld ID vereinigt

6.2.9 SQL-Aggregatfunktionen und die Klauseln GROUP BY und HAVING

ANSI SQL beinhaltet Mengenfunktionen (in diesem Buch als *SQL-Aggregatfunktionen* bezeichnet), die Mengen von Datensätzen beeinflussen. Die Standard-Aggregatfunktionen von SQL-92 sind in der folgenden Liste aufgeführt. Das Funktionsargument *`field_name`* kann ein Feldname sein (falls notwendig mit dem

Tabellenkennzeichner *table_name.*) oder das spezifische Zeichen (*) für »alle Felder«. Hier die Liste:

- `COUNT (`*`field_name`*`)` gibt die Anzahl Zeilen zurück, die in *field_name* Werte ungleich Null (`NOT NULL`) aufweisen. `COUNT(*)` gibt jedoch die Anzahl aller Zeilen der Tabelle oder Abfrage zurück – ohne Rücksicht auf etwaige Nullwerte in *field_name*.
- `MAX (`*`field_name`*`)` gibt den größten Wert in *field_name* innerhalb der Datensatzmenge zurück.
- `MIN (`*`field_name`*`)` gibt den kleinsten Wert in *field_name* innerhalb der Datensatzmenge zurück.
- `SUM (`*`field_name`*`)` gibt die Summe aller Werte in *field_name* innerhalb der Datensatzmenge zurück.
- `AVG (`*`field_name`*`)` gibt das arithmetische Mittel aller Werte in *field_name* innerhalb der Datensatzmenge zurück.

Die SQL-Aggregatfunktionen können mit persistenten und virtuellen Tabellen (wie Ergebnismengen) arbeiten. Das Syntaxschema von Abfragen, die SQL-Aggregatfunktionen einsetzen, ist:

```
SELECT FUNCTION(field_name|*) [AS column_alias]
```

Das nachfolgende Beispiel gibt einen einzigen Datensatz zurück mit dem Wert der von Ihnen gewählten Aggregatfunktion. Sie können die SQL-Aggregatfunktionen mit der Datenbank Biblio.mdb anhand der folgenden Abfrage testen:

```
SELECT COUNT(*) AS Gesamt,
      SUM(PubID) AS [Summe-Total],
      AVG(PubID) AS Durchschnitt,
      MIN(PubID) AS Minimum,
      MAX(PubID) AS Maximum
   FROM Publishers
```

Das Ergebnis dieser Aggregatabfrage finden Sie in Abbildung 6.18.

Datenbanken mit bezeichnendem Inhalt weisen für gewöhnlich Tabellen auf, deren Felder Objektklassifikationen enthalten. Die Datenbank Biblio.mdb enthält keine solche Klassifikation. Dagegen klassifiziert die Tabelle `Products` der Datenbank Nwind.mdb ein auserlesenes Sortiment exotischer Nahrungsmittel in acht Kategorien. Sie müssen die `GROUP BY`-Klausel verwenden, wenn Sie SQL-Aggregatfunktionen auf jede einzelne Kategorie dieser Nahrungsmittel anwenden wollen. Die Klausel `GROUP BY` erzeugt eine virtuelle Tabelle, die (sicher nicht überraschend) *gruppierte Tabelle* genannt wird.

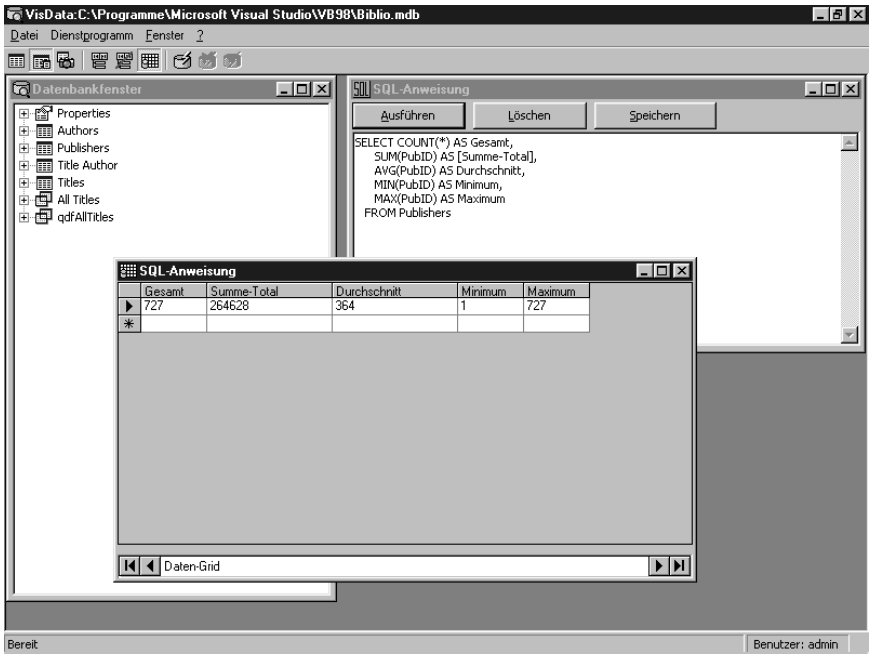

Abb. 6.18: Eine einfache Aggregatabfrage, die Summen-, Durchschnitts-, Minimum- und Maximumwerte einer nummerischen Menge zurückgibt

Die folgende SQL-Aggregatabfrage zählt die Anzahl der Nahrungsmittel jeder Kategorie (bezogen auf das Feld `CategoryID`) der Artikeltabelle aus der Datenbank Nwind.mdb; außerdem werden zwei Summenwerte und ein Durchschnittswert für jede Kategorie berechnet:

```
SELECT Category AS Kategorie,
      COUNT(ProductID) AS Artikel,
      Format(AVG(UnitPrice), "#,##0.00 DM") AS Durchschnittspreis,
      SUM(UnitsInStock) AS [Summe (Lager)],
      SUM(UnitsOnOrder) AS [Summe (Bestellungen)]
   FROM Products
   GROUP BY CategoryID
```

Hinweis

Die vorangehende Abfrage nutzt die VBA-Funktion `Format()`, um die zurückgegebenen Werte für den Durchschnittspreis in konventionellem Währungsformat anzuzeigen. In ANSI SQL ist eine solche Funktion nicht vorhanden.

Das Ergebnis der obigen Abfrage sehen Sie in Abbildung 6.19.

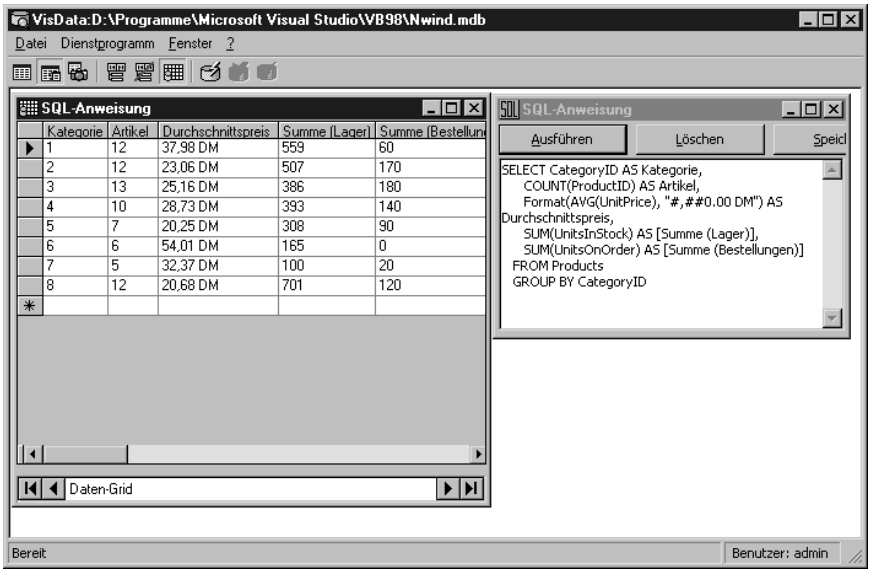

Abb. 6.19: Die Verwendung der Klausel GROUP BY in Zusammenhang mit SQL-Aggregatfunktionen

Wenn Sie nur einzelnen Gruppen (Kategorien) Aggregatkriterien zuweisen wollen, könnten Sie eventuell der Meinung sein, dieses Ziel durch Konstruktion mit einer `WHERE`-Klausel zu erreichen. Jedoch gelten `WHERE`-Klauseln immer für die gesamte Tabelle, nicht für bestimmte Bereiche. SQL bietet zu diesem Zweck die `HAVING`-Klausel an. `HAVING` arbeitet analog einer `WHERE`-Anweisung, jedoch bezogen auf Gruppenebenen. Um also die Anwendung von SQL-Aggregatfunktionen auf die spezielle Menge einer Gruppe zu begrenzen, müssen Sie der SQL-Anweisung die `HAVING`-Klausel mit einem `IN()`-Operator hinzufügen. Das folgende Jet-SQL-Beispiel gibt nur Zeilen der Kategorien 1 (Getränke) und 2 (Gewürze) zurück:

```
SELECT CategoryID AS Kategorie,
      COUNT(ProductID) AS Artikel,
      Format(AVG(UnitPrice), "#,##0.00 DM") AS Durchschnittspreis,
      SUM(UnitsInStock) AS [Summe (Lager)],
      SUM(UnitsOnOrder) AS [Summe (Bestellungen)]
   FROM Products
   GROUP BY CategoryID
   HAVING CategoryID IN(1, 2)
```

6.3 Jet-SQL und ANSI SQL-92

Die vorhergehenden Abschnitte haben viele der syntaktischen Unterschiede zwischen Jet-SQL und ANSI SQL-92 (sowie früheren ANSI SQL-Versionen wie SQL-86 und SQL-89) umrissen. Jet-SQL unterstützt beispielsweise nicht die Datensteuerungssprache (DCL, Data Control Language) von ANSI SQL für das Gewähren bzw. Entziehen von Benutzerberechtigungen (`GRANT` bzw. `REVOKE`) auf Datenbankobjekte. In Access 97 und Visual Basic 6.0 ist es möglich, Berechtigungen für Benutzer und Gruppen zu verändern, indem man den Wert der Eigenschaft `Permission` eines `Container`- oder `Document`-Objekts mithilfe von VBA-Code festlegt. Access 97 kontrolliert die Zugriffsrechte in der Benutzeroberfläche und besitzt zu diesem Zweck einen eigenen Sicherheits-Assistenten. In Visual Basic 6.0 müssen Sie selbst eine Benutzeroberfläche zur Sicherheitskontrolle entwickeln. Jet-SQL unterstützt auch keine Cursor Control Language (CCL); die Jet 3.5-Datenbankengine handhabt alle cursorbezogenen Aktivitäten selbst.

Die folgenden Abschnitte fassen einerseits die bisher beschriebenen einzelnen Unterschiede zwischen Schlüsselwörtern von Jet-SQL und reservierten Wörtern von ANSI SQL zusammen; andererseits beschreiben sie, wie Jet-SQL mit von ANSI SQL definierten Datentypen umgeht.

6.3.1 Reservierte Wörter in ANSI SQL und Schlüsselwörter in Jet-SQL

Reservierte Wörter in ANSI SQL werden traditionell in Großschrift geschrieben. Diese reservierten Wörter sollten nicht als Namen von Objekten (z.B. Tabellen oder Felder) oder als Bezeichner für Parameter und Variablen innerhalb von SQL-Anweisungen verwendet werden. Für die Sprachelemente von Jet-SQL verwendet dieses Buch die Bezeichnung *Schlüsselwörter*, da – mit Ausnahme einiger Jet-SQL-Funktionen – die Jet-SQL-Schlüsselwörter keine reservierten Wörter in ANSI SQL darstellen. Obwohl es möglich ist, Jet-SQL-Schlüsselwörter als Objektnamen zu verwenden, stellt ein solcher Gebrauch keine generell akzeptierte Programmierpraxis dar und zählt nicht zu einem guten Programmierstil.

Die häufig verwendeten reservierten Wörter (inklusive der Funktionen und Symbole) von ANSI SQL, die keine direkte Entsprechung in Jet-SQL-Schlüsselwörtern bzw. -Symbolen besitzen, werden in Tabelle 6.3 aufgeführt. Diese Tabelle berücksichtigt viele der in SQL-89 und SQL-92 neu hinzugekommenen reservierten Wörter nicht, da diese bisher in keiner kommerziellen Client-/Server-RDBMS-Implementation zu finden sind, die zum Erscheinungszeitpunkt dieses Buchs auf dem Markt verfügbar waren.

Reserviertes Wort	Kategorie	Ersatz (in Jet-SQL)
AUTHORIZATION	DCL	In Jet SQL nicht unterstützt.
BEGIN	TPL	BeginTrans-Methode von Visual Basic.
CHECK	DDL	In Jet DDL nicht unterstützt.
CLOSE	DCL	Jet-SQL unterstützt keine DCL-Elemente.
COMMIT	TPL	CommitTrans-Methode von Visual Basic.
CREATE VIEW	DDL	Von der Access DDL nicht unterstützt.
CURRENT	CCL	Jet 3.5 handhabt Cursorkontrolle selbst.
CURSOR	CCL	(Siehe vorhergehende Bemerkung)
DECLARE	CCL	(Siehe vorhergehende Bemerkung)
DROP VIEW	DDL	Von der Access-DDL nicht unterstützt.
FETCH	CCL	Feldname eines Recordset-Objekts.
GRANT	DCL	In Jet SQL nicht unterstützt.
IN Unterabfrage	DQL	Anstelle einer Unterabfrage besser eine Abfrage über ein Abfrage-Dynaset durchführen.
POSITION()	DQL	Funktion InStr() verwenden.
PRIVILEGES	DCL	In Jet SQL nicht unterstützt.
REVOKE	DCL	(Siehe vorhergehende Bemerkung)
ROLLBACK	TPL	Rollback-Methode verwenden.
SUBSTRING()	DQL	Mid()-Funktionen verwenden.
WORK	TPL	Für BeginTrans-Methode nicht notwendig.
*=	DQL	LEFT JOIN verwenden.
=*	DQL	RIGHT JOIN verwenden.
!= (nicht gleich)	DQL	Operator <> anstelle != verwenden.
: (Variablenpräfix)	DQL	PARAMETERS-Anweisung verwenden (falls notwendig).

Tab. 6.3: Gebräuchliche reservierte Wörter von ANSI SQL, die keine direkte Entsprechung in Jet-SQL besitzen

Im Gegensatz dazu führt Tabelle 6.4 Jet-SQL-Schlüsselwörter auf, die keine reservierten Wörter in ANSI SQL darstellen. Viele Jet-SQL-Schlüsselwörter beschreiben Datentypen, die durch Verwendung der db...-Konstanten spezifiziert werden (siehe auch Kapitel 2). Auch Datentypkonvertierung von und nach ANSI SQL ist dort kurz beschrieben.

Jet-SQL	ANSI SQL	Kategorie	Zweck
BINARY	Keine Entsprechung	DDL	Kein offizieller Jet-Datentyp (verwendet für das SID-Feld in System.mdw).
BOOLEAN	Keine Entsprechung	DDL	Logischer Felddatentyp (nur Werte 0 oder -1).
BYTE	Keine Entsprechung Integer (tinyint bei SQL Server).	DDL	Datentyp Asc()/Chr(), 1-Byte
CURRENCY	Keine Entsprechung	DDL	Currency-Datentyp.
DATETIME	Keine Entsprechung	DDL	Date/Time-Felddatentyp (Variant-Untertyp 7).
DISTINCTROW	Keine Entsprechung	DQL	Erzeugt ein aktualisierbares Recordset-Objekt.
DOUBLE	REAL	DDL	Gleitkommazahl mit doppelter Genauigkeit (Double).
IN-Prädikat bei Kreuztabellenabfragen	Keine Entsprechung	DQL	Definiert feste Spaltenköpfe für Kreuztabellenabfragen.
LONG	INT[EGER]	DDL	Integer-Datentyp Long.
LONGBINARY	Keine Entsprechung	DDL	OLE-Objekt-Felddatentyp.
LONGTEXT	Keine Entsprechung	DDL	Memo-Felddatentyp.
(WITH) OWNERACCESS (OPTION)	Keine Entsprechung	DQL	Führt Abfragen mit Zugriffsbeschränkungen aus, die vom Objekteigentümer kontrolliert werden.
PARAMETERS	Keine Entsprechung	DQL	Abfrageparameter, die vom Benutzer oder vom Programm eingegeben werden (sollte im Visual Basic-Code vermieden werden).
PERCENT	Keine Entsprechung	DQL	Verwendung mit TOP.
PIVOT	Keine Entsprechung	DQL	In Kreuztabellenabfragen verwendet.

Tab. 6.4: Jet-SQL-Schlüsselwörter und -Symbole, die keine reservierten Wörter und Symbole in ANSI SQL sind

Jet-SQL	ANSI SQL	Kategorie	Zweck
SHORT	SMALLINT	DDL	Integer-Datentyp, 2 Bytes.
SINGLE	FLOAT	DDL	Gleitkommazahl mit einfacher Genauigkeit (Single).
TEXT	VARCHAR[ACTER]	DDL	Text-Datentyp.
TOP	Keine Entsprechung	DQL	TOP *n* oder TOP *n* PERCENT.
TRANSFORM	Keine Entsprechung	DQL	Bezeichnet eine Kreuztabellen-Abfrage.
? (LIKE-Platzhalter)	_ (Platzhalter)	DQL	Einzelnes Zeichen mit Like.
* (LIKE-Platzhalter)	% (Platzhalter)	DQL	Kein oder mehrere Zeichen.
# (LIKE-Platzhalter)	Keine Entsprechung	DQL	Einzelne Ziffer, von 0 bis 9.
# (Datums-Bezeichner)	Keine Entsprechung	DQL	Beinhaltet Datums-/Zeitwerte.
<> (nicht gleich)	!=	DQL	Visual Basic verwendet ! als Trennzeichen.

Tab. 6.4: Jet-SQL-Schlüsselwörter und -Symbole, die keine reservierten Wörter und Symbole in ANSI SQL sind

Hinweis

Der bit-Datentyp von T-SQL entspricht dem Datentyp BOOLEAN in Jet-SQL. Transact-SQL des SQL Server 7.0 unterstützt inzwischen die reservierten Wörter TOP bzw. TOP nn PERCENT.

Jet-SQL stellt die vier in Tabelle 6.5 aufgelisteten statistischen SQL-Aggregatfunktionen (diese sind in ANSI SQL nicht vorhanden) bereit. Diese vier statistischen Aggregatfunktionen werden in der Microsoft Visual Basic-Dokumentation sowohl groß- als auch kleingeschrieben; in diesem Buch wird durchgängig die Großschreibweise befolgt.

Access-Funktion	Zweck
STDDEV()	Standardabweichung einer repräsentativen Menge einer Population.
STDDEVP()	Standardabweichung einer Population.
VAR()	Statistische Schwankung einer repräsentativen Menge einer Population.
VARP()	Statistische Schwankung einer Population.

Tab. 6.5: Statistische Aggregatfunktionen in Jet-SQL

Tabelle 6.6 führt Jet-SQL-Schlüsselwörter auf, die häufig sowohl in Klein- als auch in Großschreibweise erscheinen, im Gegensatz zur grundsätzlich versal gesetzten SQL-Schriftformatierung in der Microsoft-Dokumentation sowie den Codebeispielen, die mit Visual Basic 6.0 ausgeliefert werden. Jet-SQL-Schlüsselwörter, die gleichzeitig reservierte Wörter in Visual Basic sind, erscheinen in Fettdruck.

Jet-SQL und Visual Basic	ANSI SQL und dieses Buch	Jet-SQL und Visual Basic	ANSI SQL und dieses Buch
And	AND	Max()	MAX()
Avg()	AVG()	Min()	MIN()
Between	BETWEEN	**Not**	NOT
Count()	COUNT()	**Null**	NULL
Is	IS	**Or**	OR
Like	LIKE	Sum()	SUM()

Tab. 6.6: Schreibweisen-Konvention für Jet-SQL-Schlüsselwörter und reservierte Wörter in Visual Basic 6.0

6.3.2 Datentypkonvertierung zwischen ANSI SQL und Jet-SQL

Tabelle 6.7 führt die von ANSI SQL spezifizierten Datentypen sowie die adäquaten Datentypen in Jet-SQL auf, falls solche Entsprechungen existieren. Datentypkategorien in ANSI SQL sind die Vorläufer der SQL-92-Datentypkennzeichner.

ANSI SQL-92	Jet-SQL	Variant Untertyp	Bemerkung
Nummerisch exakt	Number		
INTEGER	Long (Integer)	3	2 Bytes
SMALLINT	Integer	2	4 Bytes

Datentypkonvertierung von und nach ANSI SQL und Jet-SQL

ANSI SQL-92	Jet-SQL	Variant Untertyp	Bemerkung
`NUMERIC[(p[, s])]`	Nicht unterstützt		p = Präzision, s = Skalierung
`DECIMAL[(p[, s])]`	Nicht unterstützt		p = Präzision, s = Skalierung
Ungefähre nummerische Werte	Number		
`REAL`	`Double` (Präzision)	5	8 Bytes
`DOUBLE PRECISION`	Nicht unterstützt		16 Bytes
`FLOAT`	`Single` (Präzision)	4	4 Bytes
Zeichen (Text)	Text		
`CHARACTER[(n)]`	`String` besitzen variable	8	(Text-Felder Länge)
`CHARACTER VARYING`	`String`	8	
Bit-Strings	Nicht unterstützt		
`BIT[(n)]`	Nicht unterstützt		(Binär-Felder Länge)
besitzen			
variable			
`BIT VARYING`	Nicht unterstützt		(Von Microsoft verwendet)
Datums-/Zeitwerte			
`DATE`	Nicht unterstützt		10 Bytes
`TIME`	Nicht unterstützt		8 Bytes (plus Bruchanteil)
`TIMESTAMP`	Date/Time	7	19 Bytes
`TIME WITH TIME ZONE`	Nicht unterstützt		14 Bytes
`TIMESTAMP WITH TIME ZONE`	Nicht unterstützt		25 Bytes
Intervalle (Datums-/Zeitwerte)	Nicht unterstützt		

Datentypkonvertierung von und nach ANSI SQL und Jet-SQL

Viele der in der Jet-SQL-Spalte als »nicht unterstützt« aufgeführten Datentypen werden von OLE DB-Datenprovidern oder ODBC-Treibern in Standard ODBC-Datentypen konvertiert; ODBC-Datentypen sind kompatibel mit Jet-SQL-Datentypen. Bei der Verwendung angehängter Datenbankdateien werden die Datentypen vom jeweiligen – zur Jet-Datenbankengine gehörenden – ISAM-Treiber für dBASE-, FoxPro-, Paradox- und Btrieve-Dateien konvertiert. Datentypkonvertierung via ODBC- oder ISAM-Treiber ist eines der Themen von Kapitel 8.