Teil 7:

Verteilte Datenbanken, SQL-99, deduktive Datenbanken

Auf Grund des großen Erfolges von relationalen Datenbanken einerseits und offensichtlichen Beschränkungen andererseits haben sich diverse Initiativen gebildet, die versuchen, das relationale Modell weiterzuentwickeln und für die Bewältigung zukünftiger Anforderungen aufzurüsten. Man spricht in diesem Zusammenhang generell von *postrelationalen* Systemen, zu denen vor allem objektorientierte Erweiterungen gehören, die (unter anderem) im SQL-99 Standard vorgeschlagen werden.

Neben der Relationenalgebra hat sich schon frühzeitig auch die Prädikatenlogik erster Stufe als mathematisches Modell für die Verwaltung von Daten angeboten. Dieser Ansatz führte zu so genannten *deduktiven Datenbanksystemen*, denen auch ein Abschnitt gewidmet wird.

Bevor wir uns aber mit alternativen und erweiterten DB-Konzepten beschäftigen, wollen wir noch einen speziellen Implementierungsaspekt näher betrachten, der für die effiziente Abarbeitung von SQL-Kommandos in großen, räumlich verteilten Anwendungen wichtig ist, nämlich *verteilte Datenbanken* (VDBMS).

Verteilte Datenbanken

Nach C.J. Date (1987) ist eine verteilte Datenbank eine logische (virtuelle) Datenbank, deren Teile in einer Menge von unterschiedlichen, echten Datenbanken gespeichert sind.

Die einzelnen beteiligten Datenbankknoten stellen jeweils ein eigenes RDBMS dar mit eigener CPU, Speicher, Benutzer, Administrator etc.

Der Sinn der Verteilung einer Datenbank liegt in der Leistungssteigerung, indem man die Daten nach Möglichkeit dort abspeichert, wo sie am häufigsten benötigt werden und für den Zugriff ein schnelles lokales Netzt verwendet und indem man ein hohes Maß an Parallelität durch Verwendung mehrerer RDBMS bekommt. Gleichzeitig erhält man einen einzigen logischen und konsistenten Gesamtdatenbestand und muss sich nicht selbst um den Abgleich der verwendeten Teile kümmern.

In einem großen multinationale Unternehmen könnte zum Beispiel die Mitarbeitertabelle auf verschiedene Niederlassungen aufgeteilt sein, so dass jeweils die Daten der Mitarbeiter einer bestimmten Niederlassung im lokalen Knoten der Niederlassung gespeichert sind. Die Gesamtinformation über alle Mitarbeiter bildet aber nur *eine einzige logische Tabelle*, die mit normalen SQL-Befehlen auch als Ganzes verarbeitet werden kann. Für die Verarbeitung der gesamten Tabelle müssen die Befehle vom VDBMS auf die einzelnen Knoten aufgeteilt werden und die Teilergebnisse müssen zu einem Gesamtergebnis zusammengesetzt werden. Dieser Mechanismus wird von einer verteilten Datenbank automatisch und für eine Anwendung unsichtbar bereitgestellt. Befehle, die nur die lokalen Daten eines Knotens betreffen, können lokal und mit voller Geschwindigkeit ausgeführt werden. Wenn lokale Befehle die Mehrheit bilden, ist in Summe ein deutlicher Geschwindigkeitsvorteil erzielbar, der praktisch linear mit der Anzahl der Knoten wächst.

Die obige Art der Verteilung von Datenbankinhalten wird als *horizontale Verteilung* bezeichnet. Man kann sich die Verteilungsgrenzen als horizontale Trennlinien in einer Tabelle vorstellen. Üblicherweise werden bestimmte Werte oder Wertebereiche einer bestimmten Spalte für die Konfiguration der Verteilung der Tupel an einen bestimmten Knoten verwendet. In einer Mitarbeitertabelle könnte ein Attribut 'Niederlassung' für die Konfiguration der Verteilung der Mitarbeiterdaten verwendet werden.

Daneben ist auch eine *vertikale Verteilung* denkbar, wobei bestimmte Attribute einer Tabelle auf einem bestimmten Knoten gespeichert sind. Die Verteilungsgrenze verläuft hier als vertikale Trennlinie in einer Tabelle.

Neben der redundanzfreien Zerlegung der Daten auf mehrere Knoten ist vor allem bei read-only-Zugriff eine *Replikation* von Tabellen oder von Teilen davon auf mehrere Knoten denkbar. Die Daten können dann lokal von mehreren Knoten aus gelesen werden. Bei Änderungen der Daten ist allerdings ein erhöhter Aufwand für den Abgleich aller Kopien erforderlich.

Die 12+1 VDBMS-Regeln von Date

Ähnlich zu den 12+1 Grundregeln für RDBMS von Codd wurden von Date 12+1 Grundregeln für VDBMS definiert, die im Folgenden angeführt sind. Konkrete VDBMS-Produkte erfüllen aber meist nur einen Teil der Forderungen.

Regel 0: Fundamentalprinzip. Für ein Anwendungsprogramm (oder einen Benutzer) sieht das VDBMS genauso aus wie ein nicht verteiltes RDBMS.

Regel 1: Lokale Autonomie. Lokale Daten (Logins, Schema, Teil der Daten, Transaktionslog, Locks) werden lokal verwaltet und lokale Operationen werden lokal ausgeführt. Ein Knoten ist von anderen Knoten unabhängig bei der Bearbeitung lokaler Befehle.

Regel 2: Kein zentraler Knoten. Es gibt keinen ausgezeichneten Master-Knoten, auf dem zentral Daten gespeichert sind oder der für die Ausführung von Befehlen benötigt wird. Alle Knoten sind gleichberechtigt, was zur Vermeidung von Engpässen führt.

Regel 3: Unterbrechungsfreier Betrieb. Es dürfen keine Abschaltzeiten fest eingeplant werden. Die Wartung eines VDBMS muss im laufenden Betrieb möglich sein, ebenso die Wartung der Anwendersoftware. Durch Wartungstätigkeiten eines Knotens dürfen andere Knoten nicht beeinträchtigt werden.

Regel 4: Standortunabhängigkeit. Anwendungsprogramme (oder Benutzer) müssen nicht wissen, wo die Daten physisch gespeichert sind. Ein Anwendungsprogramm kann prinzipiell auf jedem Knoten ausgeführt werden. In einem Data-Dictionary ist gespeichert wo die Daten physisch gespeichert sind. Dieses Data-Dictionary ist selbst auf alle Knoten repliziert.

Regel 5: Fragmentierungsunabhängigkeit. Ein VDBMS muss die Aufteilung einer Relation in Fragmente unterstützen. Die Fragmente können durch eine horizontale (Restriktion) oder vertikale (Projektion) Zerlegung einer Tabelle entstehen. Anwendungsprogramme (oder Benutzer) müssen keine Kenntnis über die Art der Fragmentierung besitzen.

Regel 6: Replikationsunabhängigkeit. Ein VDBMS muss die Replikation (Kopie) einer Tabelle oder eines Fragmentes auf mehreren Knoten unterstützen. Anwendungsprogramme (oder Benutzer) müssen keine Kenntnis über die Replikation besitzen.

Da Änderungen im Zusammenhang mit Replikaten teure Operationen sind, geht man oft von einem periodischen Abgleich der Daten aus, der z.B. einmal pro Tag stattfindet. In der Zwischenzeit sind die Daten aber nicht konsistent. Es liegt an der Anwendung, ob das akzeptabel ist oder nicht.

Ein einfacher Ansatz für die periodische Replikation ist die Übertragung des Transaktionslogs, das alle Änderungen eines Knotens seit der letzten Replikation enthalten muss. Damit werden aber mehrfache Änderungen des gleichen Datensatzes mehrfach übertragen. Bei sehr häufigen Änderungen ist dieses Verfahren daher nicht geeignet, bei wenig Änderungen aber sehr wohl.

Regel 7: Verteilte Befehle. Ein VDBMS erlaubt verteilte Abfrage- und Änderungsbefehle. Das VDBMS sorgt für die Zerlegung der Befehle in Teiloperationen, die an die betroffenen Knoten gesandt werden.

Zur effizienten Ausführung verteilter Befehle muss der Query-Optimizer die Art der Netzwerkverbindung (Bandbreite, Latenz) berücksichtigen.

Regel 8: Verteilte Transaktionsverwaltung. Ein VDBMS stellt Transaktionen unter Bewahrung der ACID-Eigenschaften zur Verfügung.

Regel 9: Hardwareunabhängigkeit. Ein VDBMS muss hardwareunabhängig sein und somit auf verschiedensten Rechnern laufen können.

Regel 10: Betriebssystemunabhängigkeit. Ein VDBMS muss unter verschiedenen Betriebssystemen ausführbar sein.

Regel 11: Netzwerkunabhängigkeit. Ein VDBMS muss verschiedene Netzwerkarchitekturen unterstützen können.

Regel 12: RDBMS-Unabhängigkeit. In einem VDBMS sollen für die einzelnen Knoten verschiedene RDBMS verwendet werden können (heterogen).

2-Phase-Commit

Für die Verwaltung von Transaktionen über mehrere Knoten müssen spezielle Mechanismen verwendet werden um die ACID-Eigenschaften der Transaktion zu garantieren. Meist wird ein so genanntes 2-Phase-Commit-Verfahren eingesetzt. Dabei wird für die Commit-Operation am Ende der Transaktion in einer ersten Phase jeder Knoten aufgefordert, ein Commit vorzubereiten und in der zweiten Phase wird das Commit abgeschlossen. Allfällige Fehler müssen in der Vorbereitungsphase erkannt werden und führen dann in der zweiten Phase zu einem Rollback über alle beteiligten Knoten.

Angenommen, eine verteilte Transaktion betrifft drei Knoten A, B und C, wobei die Transaktion durch einen Transaktionskoordinator TC koordiniert wird, der zB auch auf dem Knoten A liegt. Nehmen wir weiters an, die Knoten führen jeweils einen Teil der Transaktion aus, den wir mit Ax, Bx und Cx bezeichnen. Dann ist der Ablauf der verteilten Transaktion einschließlich des 2-Phase-Commits etwas so:

1. TC startet Transaktion und vergibt eine global eindeutige Transaktionsnummer Tx.
2. TC fordert A auf, die Teiloperation Ax von Tx durchzuführen
3. TC fordert B auf, die Teiloperation Bx von Tx durchzuführen
4. TC fordert C auf, die Teiloperation Cx von Tx durchzuführen
5. TC sendet prepare-to-commit-Befehl (Phase 1) an A, B und C
6. Wenn alle Rückmeldungen positiv sind, sendet TC commit-Befehl (Phase 2) an A, B und C, sonst sendet TC rollback-Befehl an A, B und C.

Transaktionsserver

Durch die für die Verteilung von Datenbanken eingeführten Mechanismen für verteilte Transaktionen könnten prinzipiell auch andere nicht-RDBMS-Systeme an Transaktionen teilnehmen. Die einzige Forderung ist, dass sie sich an die gleichen Konventionen zur Verwaltung der Transaktionen halten, wie zum Beispiel an ein 2-Phase-Commit-Protokoll. Auf diese Art könnte zum Beispiel eine Transaktion nicht nur aus Datenbankoperationen sondern zusätzlich aus dem Ausdrucken eines Belegs oder dem Versenden eines e-Mails bestehen.

In der Tat hat sich die Verwaltung von verteilten Transaktionen als eigenständiger Dienst herauskristallisiert und es werden diverse Produkte dazu angeboten. Der Bekannteste ist wahrscheinlich der MTS (Microsoft Transaction Server), der ein allgemeiner Dienst zur Verwaltung verteilter Transaktionen ist und im speziellen auch für die Verwaltung verteilter Datenbanken im Microsoft SQL Server verwendet wird. Bei anderen Herstellern finden sich ähnliche Strukturen. In der Java-Welt werden beliebige verteilte Transaktionen durch die J2EE (Java-2 Enterprise Edition) Spezifikation abgedeckt, für die eine Reihe von Produkten existiert (Bea WebLogic, IBM WebSphere, etc.).

Die konkrete Verwendung von verteilten Transaktionen erfordert Wissen über zum Teil recht komplexe APIs und zur Konfiguration der beteiligten Komponenten und sprengt den Rahmen dieser Einführungslehrveranstaltung.

SQL-99 (alias SQL-3)

Die bei Programmiersprachen stattgefundene Entwicklung hin zu objektorientierten Sprachen hat zu einem beträchtlichen Unterschied der verfügbaren DB-Technologie versus der darauf aufbauenden Programmiersprachen geführt. Objekte, Klassen, Vererbung, Methoden, etc. können nicht direkt in einer relationalen Datenbank modelliert werden. Mit JDBC können zum Beispiel nur jeweils einzelne Attributwerte eines Tupels ausgelesen werden, nicht aber ganze Tupel einschließlich Typinformation.

Zur Verkleinerung dieses Unterschiedes (semantic gap, impedance mismatch) hat man sich intensiv mit dem Thema *objektorientierte Datenbanken* (OODBMS) beschäftigt, die eine direkte Persistierung von Objekten erlauben. OODBMS sind weitgehend unabhängig von relationalen DBs und SQL entwickelt worden. Sie verwenden eigene Mechanismen zur Persistierung, Optimierung, und für Abfragen. Dadurch erlauben sie keine sanfte Evolution sondern erfordern einen radikalen Umstieg auf eine alternative Technologie, die im Allgemeinen aber noch nicht so stabil und effizient ist wie RDBMS. Folglich hielt sich der Erfolg von OODBMS-Ansätzen bisher in Grenzen.

Ein sanfterer, weniger radikaler Ansatz wird mit der Weiterentwicklung von SQL verfolgt, der in einem ANSI-Standard namens SQL-99 (alias SQL-3) vorliegt. Im Wesentlichen handelt es sich dabei um ein RDBMS aber mit einigen Erweiterungen für OO-Programmierung. Der SQL-99 Standard wurde in diversen Aspekten von IBMs DB2 geprägt und daher ist DB2 wahrscheinlich am nächsten bei der Implementierung. Andere RDBMS haben ebenfalls OO-Erweiterungen eingebaut, die aber meist inkompatibel zum Standard sind. Zur Zeit (2003) zeigt sich wenig Bewegung in Richtung einer breiten Unterstützung und Anwendung von SQL-99. Ein wichtiger Schritt in Richtung OO-Erweiterungen bei RDBMS ist durch das Release 9i des Oracle RDBMS erfolgt, der Typerweiterung und dynamische Bindung weitgehend in Anlehnung an den SQL-99 Standard unterstützt.

Strukturierte Datentypen

SQL-99 erlaubt die Definition von strukturierten (zusammengesetzten) Datentypen, die als Tabellentyp oder als Spaltentyp verwendet werden können. Zum Beispiel könnte ein Typ `Adr_typ` definiert werden, der aus `str`, `plz` und `ort` besteht. In den Programmiersprachen C, C++ oder C# würde man von einem *struct* sprechen, in Pascal von einem *RECORD*.

```
CREATE ROW TYPE Adr_typ (str VARCHAR(40), plz INTEGER, ort VARCHAR(40))
```

Um eine Address-Tabelle zu erzeugen kann der Typ verwendet werden.

```
CREATE TABLE Address OF Adr_typ
```

```
CREATE TABLE Kunde (name VARCHAR(40), adr Adr_typ)
```

```
INSERT INTO Kunde VALUES('Testkunde', ('Kärtnerstr. 1', 1010, "Wien"))
```

```
SELECT name, adr.ort FROM Kunde where adr.plz < 2000
```

Die Elemente einer zusammengesetzten Spalte werden bei INSERT immer in runden Klammern eingeschlossen. Zur Selektion eines Bestandteils in SELECT wird die Punktschreibweise verwendet.

Vererbung

SQL-99 erlaubt die Definition von Basistabellen und Erweiterungen davon (Untertabellen). Dieses Konzept entspricht dem Konzept der Klassen und Unterklassen in OO-Sprachen.

```
CREATE TABLE Grosskunde UNDER Kunde (rabattproz DECIMAL(4, 2))
CREATE TABLE Endkunde UNDER Kunde (bonitaet INTEGER)
```

```
INSERT INTO Endkunde VALUES('Test', ('Kärtnerstr. 1', 1010, "Wien"), 1)
```

Die Speicherung kann man sich am besten so vorstellen, wie wir das im Manuskript Teil 5 (Vererbung) eingeführt haben. Allerdings erledigt das DBMS nun die Verwaltung der beteiligten Tabellen und sorgt für die Konsistenz. Außerdem kann das DBMS nun auch gewisse Optimierungen durchführen, weil es mehr Wissen über die Art des Zugriffs hat.

Dynamische Bindung

SQL-99 erlaubt die Definition von *Methoden* ähnlich wie in objektorientierten Programmiersprachen. Eine Methode ist eine Stored Procedure, die dynamisch an einen Objekttyp gebunden ist. Erst zur Laufzeit wird an Hand des dynamischen Typs eines Objekts die passende Methode ausgewählt (dynamic dispatch) und

aufgerufen. Die Methodensignaturen folgen im SQL-99 Standard nach der CREATE TYPE Anweisung wie in folgendem Beispiel gezeigt:

```
CREATE ROW TYPE Adr_typ (...)
INSTANCE METHOD ToString() RETURNS VARCHAR(256)
```

Die Implementierung der Methoden folgt in SQL-99 separat durch eine `CREATE METHOD` Anweisung.

Hier ist noch ein Beispiel für die Typdeklaration und Vererbung in Oracle 9i, wo Instanzmethoden innerhalb der Typdeklaration aufscheinen und außerdem MEMBER FUNCTION heißen.

```
CREATE TYPE Person_t AS OBJECT (
  name VARCHAR(30),
  dob DATE,
  MEMBER FUNCTION age() RETURN NUMBER,
  MEMBER FUNCTION print() RETURN VARCHAR(40)
) NOT FINAL;

CREATE TYPE Employee_t UNDER Person_t (
  salary NUMBER,
  bonus NUMBER,
  MEMBER FUNCTION wages() RETURN NUMBER,
  OVERRIDING MEMBER FUNCTION print() RETURN VARCHAR(40)
);
```

Tupelreferenzen

Analog zu Referenzen (Zeiger) auf dynamische Datenstrukturen in Programmiersprachen, ist es in SQL-99 möglich, einen Datensatz zu referenzieren. Dazu muss nicht mehr wie bisher eine Fremdschlüsselbeziehung verwendet werden. Der benötigte Surrogat wird vom DBMS verwaltet und ist nicht sichtbar. Damit Zeilenreferenzen möglich sind, muss in der CREATE TABLE-Anweisung eine entsprechende Option angegeben werden. Diese Option führt dazu, dass ein Surrogat eingeführt wird.

```
CREATE TABLE Kunde OF Kunden_typ WITH REF VALUE
```

```
CREATE TABLE Auftrag (artikelName VARCHAR(40), menge INTEGER,
                      kunde REF(Kunden_typ))
```

Weitere Merkmale von SQL-99

- Eine vollständige Programmiersprache für Stored Procedures und Triggers.
- Benutzerdefinierte einfache Datentypen
- Abstrakte Datentypen
- Konstruktoren
- BLOBs, CLOBs
- Rekursion
- verteilte Transaktionen nach dem XA-Standard
- API
- etc.

Der SQL-3 Standard ist etwa dreimal so umfangreich wie der SQL-2 Standard, der noch nicht einmal vollständig implementiert vorliegt. Der Trend geht zur Zeit in Richtung Erweiterung von RDBMS um Fähigkeiten von SQL-99 mit mehr oder weniger starker Anlehnung an die konkrete Syntax von SQL-99. Statt der SQL-99 Programmiersprache kommt teilweise auch Java oder andere 3GL-Sprachen zum Einsatz. Dadurch wird dieser Teil des Standards möglicherweise ganz überflüssig.

Deduktive Datenbanken

Als Alternative zur Relationenalgebra als Basis für eine Datenbank-Abfragesprache kann auch logische Programmierung (ähnlich der Programmiersprache *Prolog*) verwendet werden. Dazu definiert man Prädikate in Form von *Fakten* und *Regeln*.

Die Fakten definieren Relationen und stellen sozusagen explizites Wissen dar. Man spricht auch von der *extensionalen Datenbasis*.

Die Regeln definieren Wissen basierend auf den Fakten und möglicherweise auf anderen Regeln. Die Regeln können sich auch selbst verwenden, was zu rekursiven Regeldefinitionen führt. Die Regeln bilden die so genannte *intentionale Datenbasis*.

Durch die Möglichkeit der Rekursion geht dieser Ansatz prinzipiell über die Relationenalgebra hinaus, die ja Rekursion nicht ausdrücken kann. Ein typisches Anwendungsbeispiel ist die Beschreibung eines Stammbaums durch eine Eltern-Kind-Beziehung, die die Fakten darstellen. Weitere Verwandtschaftsbeziehungen können als Regeln ausgedrückt werden (siehe unten).

Ein konkrete Abfragesprache für deduktive Datenbanken ist die Sprache *Datalog*, deren wichtigste Merkmale im Folgenden kurz beschrieben werden.

Datalog

Ein Datalog-Programm besteht aus *Fakten* und *Regeln*.

Fakten. Die Fakten definieren Relationen ähnlich wie in relationalen Datenbanken und halten sich an folgende Syntax.

> Faktum = Relationsname "("Attribut {"," Attribut}")" "." .

Beispiele für Fakten eines Familienstammbaums wären etwa

```
parent("Tom", "Bob").
parent("Tom", "Anna").
parent("Joe", "Tom").
```

mit der Bedeutung, dass Tom zwei Kinder namens "Bob" und "Anna" hat und selbst ein Kind von "Joe" ist. Diese Faktenbasis kann schon für einfache Abfragen, verwendet werden. Abfragen (Queries) beginnen immer mit ":-" und enden mit einem ".". Beispiele sind:

```
:- parent("Joe", "Tom").
```

Das deduktive Datenbanksystem sucht in den Fakten nach dem angeführten Ausdruck (entspricht einem Tupel) und gibt *true* oder *false* zurück je nachdem ob eines gefunden wird oder nicht.

Eine andere Möglichkeit der Abfrage entsteht durch Verwendung von *Variablen* in der Query. In Anlehnung an die Konventionen von Prolog benennen wir Variablen im Folgenden mit einem einzelnen Großbuchstaben. In der folgenden Abfrage bezeichnet *X* also eine Variable und die Abfrage bedeutet, dass nach allen möglichen Werten für X gesucht werden soll, die das Prädikat erfüllen.

```
:- parent("Tom", X).
```

Das deduktive Datenbanksystem versucht nun alle möglichen Werte für X zu finden, sodass parent("Tom", X) erfüllt ist und gibt die gefundene Menge als Ergebnis zurück. In obigem Beispiel gibt es alle Kinder von "Tom" zurück, also {"Bob", "Anna"}. Genauso gut könnte man aber auch nach den Eltern von "Bob" fragen oder nach beiden, wobei in letzterem Fall eine Menge von Paaren von X- und Y-Werten zurückgegeben wird, für die parent(X, Y) gilt, also die gesamte Faktenbasis für parent.

```
:- parent(X, "Bob").
:- parent(X, Y).
```

Regeln. In obiger Faktenbasis sind "Bob" und "Anna" offensichtlich Enkel von "Joe", weil "Bob" und "Anna" Kinder von "Tom" sind und "Tom" ist Kind von "Joe". Dieser Zusammenhang lässt sich durch eine allgemeine Regel ausdrücken: X ist ein Großelternteil von Y wenn ein Z existiert sodass X ein Elternteil von

Z ist und Z ein Elternteil von Y. Man führt hier also eine (lokale) Hilfsvariable Z ein, um den Zusammenhang auszudrücken. In Datalog lässt sich die Regel 1:1 ausdrücken.

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Diese Regel besagt, dass X ein Großelternteil von Y ist (oder Y ein Enkel von X), falls X ein Kind Z hat, dass seinerseits ein Kind Y hat. Der linke Teil der Regel ist der Regelkopf (*head*), der hier zwei Variablen X und Y einführt, die innerhalb der rechten Regelseite, dem Rumpf, verwendet werden müssen. Im rechten Teil der Regel, auch *sub-goals* genannt, wird zwischen den Variablen des Kopfs eine Beziehung hergestellt, die aus einer *Und*-Verknüpfung der sub-goals besteht. Jedes sub-goal ist selber wieder ein Prädikat. Diese Form von Regeln nennt man auch *Horn-Klauseln*. Die sub-goals können lokale Variablen innerhalb der Regel einführen. Die sub-goals besagen in obigem Beispiel, dass es ein Z geben muss (also eine Belegung der Variablen Z), für die parent(X, Z) gilt und parent(Z, Y).

Man beachte, dass X, Y und Z *Variablen* sind, keine *Konstanten* wie "Bob" oder "Anna".

In obigem Beispiel ist das Finden einer Variablenbelegung für Z die Aufgabe des deduktiven Datenbanksystems und das Kernstück der Abfragebearbeitung. Meist geht eine deduktive DB top-down vor und versucht ausgehend von der Query des Benutzers Variablenbelegungen zu finden und über die sub-goals weitere Variablen zu binden. Dabei kann es zu systematischem Ausprobieren einer großen Zahl von Kombinationen kommen, was durch möglichst geschickte Optimierungstechniken nach Möglichkeit vermieden wird.

Die obigen Abfragen hätte man auch in SQL mit einem einzigen SELECT definieren können. In SQL ist es aber nicht möglich, eine Regel zu definieren, die sich über *beliebig viele Ebenen* der Elternbeziehung erstreckt, also zum Beispiel eine Abfrage: "Welche Nachkommen hat X?". Gemeint sind sowohl direkte Nachkommen entsprechend der parent-Beziehung als auch indirekte Nachkommen wie Enkel, Ur-Enkel, Ur-Ur-Enkel etc. In Datalog ist es möglich, eine derartige *rekursive Beziehung* zu definieren. Man muss nur ausdrücken, dass ein Vorfahre (ancestor) entweder ein Elternteil ist oder ein Vorfahre eines Elternteils. Die Oder-Verknüpfung wird durch Einführung einer weiteren Regel mit gleichem Kopf ausgedrückt.

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Die Abfrage zur Bestimmung aller Nachkommen von Joe ist dann

```
:- ancestor("Joe", X).
```

Man könnte aber auch alle Vorfahren von Anna ermitteln mit

```
:- ancestor(X, "Anna").
```

und würde als Ergebnis {"Tom", "Joe"} bekommen.

Restriktionen

Ohne auf die Details der Bearbeitungsstrategie eingehen zu wollen, sollte klar sein, dass nicht jede mögliche Regel auch sinnvoll ist. Ebenso wie bei Rekursion oder Schleifen in Programmiersprachen muss die Terminierung gewährleistet sein. Im Fall von Datalog ist dies zum Unterschied von einer SELECT-Anweisung nicht statisch garantiert, obwohl einige einfache Sonderfälle prinzipiell erkennbar wären, zum Beispiel ist folgende Regel offensichtlich falsch, weil der Kopf und eines der sub-goals identisch sind.

```
ancestor(X, Y) :- parent(X, Z), ancestor(X, Y).
```

Weitere Probleme können durch die Verwendung der Negation oder Ungleichheit entstehen.

Nicht in jedem Fall, aber in sehr vielen, kann so wie oben eine Regel direkt und 1:1 aus dem Wissen über die reale Welt definiert werden. Logisches Programmieren ist verwandt mit menschlicher Logik aber nicht identisch.

Bezug zu SQL

Die Fakten entsprechen 1:1 den Tabellen von SQL.

Die Regeln kann man mit Views assozieren, wobei Views aber nicht rekursiv verwendbar sind. Rekursion müsste man in SQL mit Stored Procedures durch Iteration ausdrücken. Man müsste in SQL aber jeweils eine eigene Stored Procedure für jede Art der Verwendung von Variablen in der Abfrage vorbereiten.

Eine Abfrage entspricht einem SELECT oder einem Aufruf einer Stored Procedure.

Die elementaren relationalen Operatoren lassen sich 1:1 in Datalog ausdrücken. z.B. die Projektion durch Einführung von Variablen in den sub-goals. Beispiel zur Projektion von parent auf eines der beiden Attribute:

```
isParent(X) :- parent(X, Y).
isChild(X) :- parent(Y, X).
```

Die folgenden Abfragen liefern dann die Menge aller Parents bzw. die Menge aller Kinder.

```
:- isParent(X).
:- isChild(X).
```

Ebenso lassen sich das Kreuzprodukt und Joins einfach ausdrücken. Zum Beispiel könnte man eine Relation definieren, die aus drei Attributen besteht, Großelternteil, Elternteil, Kind, was einem EQUI-Join entspricht.

```
grandParentChild(X, Y, Z) :- parent(X, Y), parent(Y, Z).
```

Damit ist das Manuskript zu *Einführung in Datenbanken* abgeschlossen.

Josef Templ, Juni 2005