Insy-Noticen

Florian Pelz

17. September 2013

Capitel 0-2: DBS-aufbau

Ein DBS ist was?

DBS = DB + DBVS, d.h. ein datenbanksystem ist eine datenbank (die die daten speichert) zusammen mit einem datenbankverwaltungssystem (database management system)

Ein KIS ist was?

Kooperatives informationssystem: DBS + AWS (anwendungssystem) = KIS

Arten von datenbanksystemen

- **RDBS**: relationales datenbanksystem
- **OODBS**: objectorientiertes datenbanksystem
- **ORDBS**: object-relationales datenbanksystem
- **hierarchisches** DBS
- **netzwerkartiges** DBS nach dem CODASYL-standard (d.h. mit in einem graph angeordneten daten, kanten = referencen)
- DBS für **un- und semi-structurierte daten** (d.h. nicht und nur teils maschinenverständliche daten); im gegensatz zu **structurierten** (maschinenverständlichen) daten und **multimedia-daten** wie bilder, videos und tondateien sehen wir als weitere classe von daten – Wir sollten alle vier classen von daten kennen.

Stapelorientierte informationssysteme verarbeiten informationen ohne benutzerinteraction, während **dialogorientierte** informationssysteme auf nutzeranfragen reagieren.

Was für eigenschaften sollte ein gutes informationssystem im allgemeinen erfüllen?

- Redundancen und inconsistencen sollten vermieden werden.
- Die anfragen sollten flexibel gestellt werden können.
- Es sollte große datenmengen und mehrere benutzer unterstützen, aber auch eine zugriffscontrolle und bestimmte voraussetzungen an datenänderungen durchsetzen können.
- Zudem sollte es gut mit verschiedensten fehlern (absturz, speicherfehler, brand im datencentrum, &c.) umgehen und so eine hohe erreichbarkeit und zuverlässigkeit aufweisen.
- Die daten sollten unabhängig von den auf sie zugreifenden anwendungen (dem anwendungssystem) verwaltet und gespeichert werden (gecapselt, nach dem geheimnisprincip /information hiding). Den begriff der **datenunabhängigkeit** sollen wir in diesem zusammenhang kennen.
- Zugriffsberechtigungen sollten eingeschränkt werden können (datenschutz).

Integritätsbedingungen

Integritätsbedingungen (constraints) sind zusicherungen, welche eigenschaften die daten im informationssystem haben müssen. Sie werden so gewählt, dass die daten in der miniwelt des informationssystems nicht durch nutzerfehler von der realen welt zu sehr abweichen können und z.b. ein geburtsjahr eines kunden als 70 statt 1970 gespeichert wird. Integritätsbedingungen kann man sich als einen vertrag mit dem informationssystem vorstellen, der einem bestimmte garantien gibt.

ACID

Ein vorgang der verarbeitung einer oder mehrerer anfragen oder anderer befehle im DBS heißt **transaction**, wenn für ihn die **ACID**-criterien gelten:

- **Atomicity**: Transactionen werden ganz oder gar nicht durchgeführt.
- **Consistency**: Vor und nach ablauf einer transaction werden alle integritätsbedingungen im DBS erfüllt.
- **Isolation**: Parallel ausgeführte transactionen beeinflussen sich nicht beide gegenseitig; sie verhalten sich so als ob sie nacheinander durchgeführt würden.
- **Durability**: Die wirkung als erfolgreich durchgeführt gemeldeter transactionen bleibt auch bei systemfehlern erhalten (z.b. bei einem rechnerabsturz).

Wenn nicht alle aspecte von ACID benötigt werden, kann ein ACID nicht vollständig umsetzendes system efficienter und besser geeignet sein. Für die clausur sollen wir die bedeutung von ACID können und beispiele für die auswirkungen geben können, außerdem sollen wir verletzungen erkennen können (siehe übungsblätter).

Dreischichtenmodell

Es gibt das

- **Datensystem** für die optimierung und übersetzung von benutzeranfragen auf relationen (tabellen) von datensätzen (tupel aus mehreren attributen einer entität) in befehle ans
- **Zugriffssystem** für den efficienten zugriff auf die sätze durch für geeignete datenstructuren optimierte anfragen auf die im hauptspeicher (HSP) gepufferten (gecachten) seiten (pages) oder durch das
- **Speichersystem** für die seitenzugriffe auf externspeichermedien und für die externspeicherverwaltung (externspeicher sind z.b. festplatten).

weitere Componenten eines DBS

Metadaten beschreiben die structur der daten in den datenbanken. Neben daten-, zugriffs- und speichersystem hat ein DBS auch eine metadatenverwaltung für vom datenbankadministrator specificierte metadaten, statt die metadaten fest vorzugeben.

ACID-eigenschaften werden von der transactionsverwaltung sichergestellt.

Eine miniwelt ist

der ausschnitt der realen welt, den ein datenbanksystem modellieren soll.

Anwendungsclassen von informationssystemen

- **OLTP**: On-Line Transaction Processing
- **DW**: Data Warehouse (beinhaltet auch historische daten, nicht nur actuellen zustand der miniwelt; genutzt z.b. für jahresberichte)
- **OLAP**: On-Line Analytical Processing (zur analyse betrieblicher daten)
- **DSS**: Decision Support System

Buzzwords

- **Business Intelligence**: auswerten großer datenbanken für betrieblich relevante informationen
- **Data Mining** /KDD (Knowledge Discovery in Databases): informationssuche aus großen datenmengen
- **CRM**: Customer Relationship Management

Capitel 2: Datenstructuren

Warum (nicht) externspeicher statt dem hauptspeicher in DBS verwenden?

Externspeicher hat als vorteil:

- **persistent**, nicht **flüchtig**
- billiger
- größere speichercapacität

Aber:

- lange zugriffszeit
- evtl. kein wahlfreier zugriff (random access)

Wie speichern wir daten im DBS?

Wir haben eine mehrstufige speicherhierarchie: Im hauptspeicher haben wir einen DB-puffer (d.h. einen cache für die DB), die DB selbst ist dauerhaft im externspeicher gespeichert.

Weniger vereinfacht haben wir vor dem hauptspeicher noch einen cache und nach dem externspeicher noch einen **nearline-archivspeicher**, auf den automatisch vom DBS zugegriffen werden kann (z.B. magnetbänder oder auch festplatten), und einen **offline-archivspeicher** für manuelle backups. In der hierarchie wird die zugriffszeit dabei immer länger, aber die capacität immer größer.

Was ist die **zugriffslücke?**

Der große unterschied in der zugriffszeit zwischen extern- und hauptspeicher (bei einem factor von circa 10^5). Um sie zu füllen könnten neue speichertypen wie z.b. SSDs genutzt werden.

Was ist die inclusionseigenschaft in der speicherhierarchie?

Die eigenschaft jedes mitglieds der hierarchie, alle daten höherer hierarchieebenen gespeichert zu haben.

Speicherzugriffsarten

- **Read-through**: Daten bei bedarf in der hierarchie nach oben durchreichen.
- **Write-through /FORCE**: Daten nach änderung direct in der hierarchie nach unten weiterreichen.
- **Write-back /NOFORCE**: Daten erst bei ersetzung, d.h. wenn nicht mehr genügend speichercapacität für sie vorhanden ist, nach unten weiterreichen.

Können, siehe übungen! Wir können auch vorausschauend schon daten in den speicher laden, von denen wir annehmen, dass wir sie brauchen werden (prefetching).

Ersetzungsstrategien

Wir benutzen ein bestimmtes verfahren, um zu bestimmen, welches datum wir ersetzen, wenn der cache voll ist und wir ein neues in den cache laden müssen.

- **LRU** (**least recently used**) ist eine ersetzungsstrategie, bei der wir stets das am längsten nicht benutzte datum ersetzen, wenn der cache voll ist.
- **FIFO** (**first in, first out**), d.h. wir ersetzen stattdessen einfach das älteste datum, egal wann und wie oft es benutzt wurde.

Können, siehe übungen!

Was sagt Moore's Law über die zukunft von speicher aus?

Circa alle 18 Monate verdoppelt sich das preis-/leistungsverhältnis, d.h. es verhundertfacht sich im jahrzehnt. Speicher wird also immer billiger, nur dessen administration bleibt teuer.

Was für zugriffsverfahren auf den haupt- und externspeicher kennen wir?

Unsere datensätze werden über **schlüssel** (bestimmte attribute des satzes) identificiert. Zum speichern der sätze können wir verschiedene speicherstructuren verwenden, dabei finden wir den ort, an dem der satz gespeichert ist, entweder über einen wiederholten schlüsselvergleich beim durchsuchen der structur oder über eine schlüsseltransformation wie bei hash-tabellen.

Die zugriffskosten messen wir dabei anhand der nötigen seitenzugriffe, da dies normalerweise die teuersten operationen sind.

Datenstructuren, bei denen schlüssel verglichen werden, sind die folgenden:

(Es gibt in der clausur aber wahrscheinlich nur aufgaben zu B-bäumen. Kostenberechnungen sollten wir verstehen.)

Sequentielle listen auf externspeichern: Wir speichern die sätze aufeinanderfolgend auf vielleicht auch aufeinanderfolgenden seiten des externspeichers. Die sätze werden entweder ungeordnet oder sortiert gespeichert. Ein vorteil sequentieller listen kann die physische nachbarschaft der sätze und unter umständen auch der seiten sein (bei seiten nennen wir dies **cluster-bildung**), welche allerdings nur vorteilhaft ist, wenn die liste in der reihenfolge geordnet ist, in der wir auf die sätze zugreifen möchten.

Die anzahl der sätze pro seite nennen wir **blockungsfactor** b. Bei n sätzen sind die zugriffskosten, um auf alle seiten zuzugreifen $N_S = \frac{n}{b}$. Im mittel kostet der zugriff auf einen bestimmten schlüssel $N_S = \frac{n}{2b}$, da der schlüssel erwartungsgemäß genau in der mitte der liste liegt. Mit binärsuche können wir mit $N_S = \log_2(\frac{n}{2b})$ zugreifen, dafür müssen die adressen der seiten und nicht bloß der sätze jedoch auch alle aufeinanderfolgen. Das einfügen ist teuer, da alle seiten nach (oder alternativ vor) der neuen verschoben werden müssen – das kostet doppelt, da wir lesen und rückschreiben müssen, also im mittel $N_S = 2 \cdot \frac{n}{2b}$.

Bei sequentiellen listen mit **splitting** können wir keine physisch benachbarten seiten haben, jede seite verweist stattdessen auf ihre vorgänger- und nachfolgerseite, wodurch binäre suche nicht mehr möglich ist. Wenn wir splitting verwenden, wird beim einfügen eines satzes in eine volle seite der liste dafür die hälfte der sätze der seite in eine neue seite verschoben (die seite wird gesplittet) und die referencen werden entsprechend angepaßt. So ist schnelleres einfügen möglich ($N_S = 2 \cdot 3$), aber die suche dauert länger und der speicherbedarf wächst.

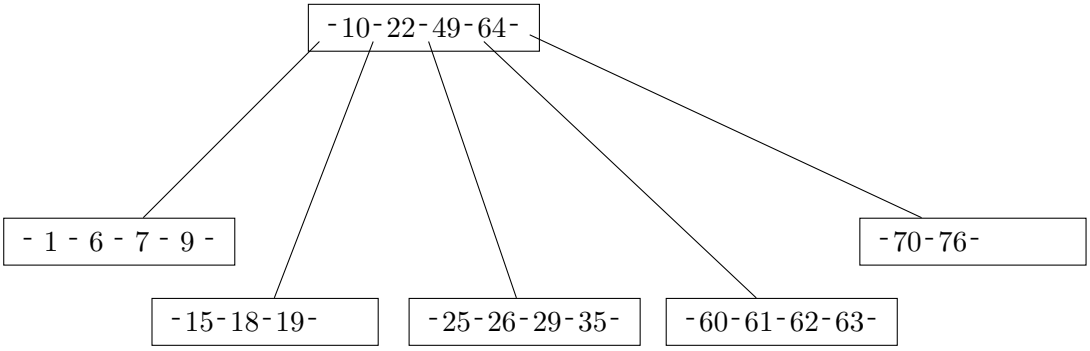
Gekettete listen auf externspeichern haben für jeden satz eine referenz auf den nächsten satz in der liste gespeichert (im beispiel auf den folien ist die verkettung cyclisch – der letzte satz verweist wieder auf den ersten); die in der liste aufeinanderfolgenden seiten müssen somit nicht mehr physisch benachbart sein und es gibt keine clusterbildung. Der zugriff auf alle sätze kostet $N_S = n$, im mittel brauchen wir zum finden einer seite $N_S = \frac{n}{2}$ zugriffe. Zum einfügen brauchen wir dafür nur $N_S \leq 2 \cdot 2$ seitenzugriffe. Ein vorteil ist, dass wir bei geketteten listen unterschiedliche verkettungen der sätze für unterschiedliche sortierreihenfolgen aufbauen können.

Mehrwegbäume wie **binäre suchbäume** sind eine datenstructur, die efficienteres suchen und fast so efficientes einfügen erlauben. Die höhe des baumes bestimmt die such- und einfügezeit. Jede seite können wir als knoten des baumes nutzen. Bei **balancierten** bäumen (d.h. solchen, wo die pfade von der wurzel zu einem blatt alle annähernd gleich lang sind) ist der suchzugriff auf kosten des einfügens im mittel efficienter.

Eine weiterentwicklung ist der **B-baum**. Jeder B-baum gehört einer **classe** $\tau(k, h)$ an, wobei $h \geq 0, k > 0$ natürliche zahlen sind. Ein solcher baum ist entweder leer (d.h. besteht aus keinen knoten und hat nicht einmal eine wurzel) oder erfüllt diese eigenschaften:

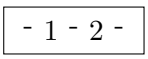
- Jeder pfad von der wurzel zu einem blatt ist $h - 1$ lang (d.h. der baum ist balanciert),
- jeder knoten außer blättern hat zwischen $k + 1$ und $2k + 1$ kinder (die wurzel darf weniger haben, hat aber als nicht-blatt mindestens 2),
- jedes blatt hat zwischen k und $2k$ einträge (d.h. sätze, die wurzel darf auch hier weniger haben, ohne minimum).

Sätze sind in allen knoten zwischen den referencen zusammen mit schlüsseln gespeichert (in blättern zeigen die referencen aber auf nirgendwo hin). Die inneren knoten sind seiten mit verweisen auf ihre kinder, die schlüssel stehen jeweils zwischen den kindern. Alle einträge im kind links vom schlüssel haben einen kleineren identificator als der schlüssel, alle rechts haben einen größeren. Es handelt sich um einen mehrwegbaum mit höhe $h_B = O(\log_{k+1} n)$ (da wir bei maximaler füllung $(k + 1)^{h_B}$ blätter haben). Die maximalen zugriffskosten h_B sind annähernd die selben wie die mittleren.

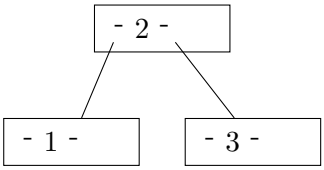


Das einfügen eines schlüssels in eine volle B-baum-seite läßt den mittleren schlüssel (median) der übervollen seite in den elternknoten wandern, wobei entsprechende referencen angelegt werden. Bei der wurzel gibt es keinen elternknoten und ein neuer muss angelegt werden, dieser wird die neue wurzel.

Beispiel:

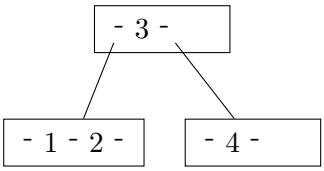


wird beim einfügen von 3 zu:

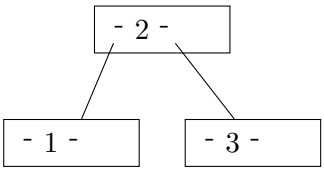


Beim löschen eines satzes aus einer seite kann es passieren, dass die seite danach zu wenig gefüllt ist. Bei einem inneren knoten müssen wir beim löschen einfach den nächstkleineren oder nächstgrößeren schlüssel aus seinen kindern in den knoten holen, eine zu leere seite tritt daher nur in den kindern auf. Bei einer blattseite ist dies nicht möglich. In diesem fall führen wir einen **ausgleich** bzw. eine **rotation** durch – hat die linke oder rechte geschwisterseite genug elemente, so schieben wir den am weitesten rechts bzw. links stehenden schlüssel der geschwisterseite in den elternknoten und tragen den alten elternknotenschlüssel in die zu leere seite ein.

Beispiel:

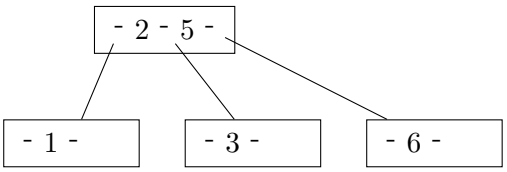


wird beim löschen von 4 zu:

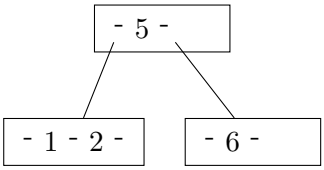


Hat die linke und rechte nachbarseite die minimale anzahl einträge, d.h. ist ein ausgleich nicht ohne weiteres möglich, so **mischen** wir, indem wir die zu leere seite mit der linken oder rechten geschwisterseite concatenatieren, wodurch wir wieder eine ausreichend große seite erhalten. Der entsprechende schlüssel im elternknoten wird dabei auch in die concatenierte seite verschoben. Ist die resultierende elternseite zu leer, so mischen wir diese mit einer nachbarseite oder gleichen für sie aus.

Beispiel:

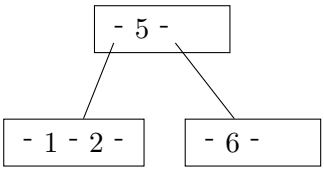


wird beim löschen der 3 zu:

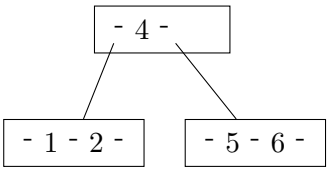


Wir können den **belegungsgrad** β des B-baums, d.h. die ausnutzung der seiten im B-baum in procent zur maximalen ausnutzung, optimieren, indem wir einen größeren **split-factor** m beim einfügen verwenden (bisher war $m = 1$). Das bedeutet, dass wir vor der nutzung des einfügeverfahrens für volle blattseiten einen ausgleich mit den rechten $m - 1$ geschwisterseiten versuchen. Erst wenn alle m seiten übervoll sind, werden die m seiten zusammengelegt und so wie beim herkömmlichen einfügen auf $m+1$ seiten verteilt, wobei die schlüssel so verteilt werden, dass, wenn keine gleiche anzahl von schlüsseln an jede neue seite gegeben werden kann, die rechten seiten weniger haben.

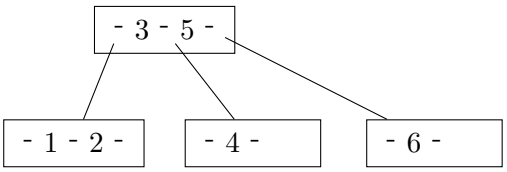
Beispiel mit $m = 2$:



wird beim einfügen der 4 zu:



und nach einfügen einer 3 zu:



Eine variante von B-bäumen sind B*-bäume. Dabei sind die schlüsseln in inneren knoten nicht mehr mit daten verbunden, es handelt sich also um reine wegweiser, die jetzt auch mehrmals im baum vorkommen können. Einzig die blattknoten haben weiterhin daten gespeichert.

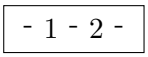
- Dadurch wird der B*-baum zu einer **index**-structur, die nur der suche dient, und die daten können zusätzlich zum suchbaum noch eine sequentielle liste (mit oder ohne cluster-bildung) bilden um so den sequentiellen zugriff schnell durchführen zu können (man kann den B*-baum als B-baum mit den elementen einer sequentiellen

liste als kinder ansehen). Wir können die sequentiellen daten sogar mit mehreren verschiedenen B-bäumen nach unterschiedlichen suchcriterien ordnen.

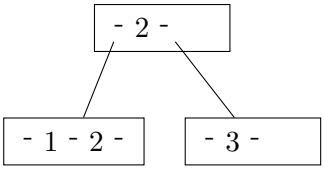
- Die schlüssel der inneren knoten müssen damit nicht mehr eindeutig sein und können die gleichen schlüssel wie die in z.b. blättern sein. Bei einem vergleich mit einem knoten zur suche im baum ist bei uns der linke teilbaum der, der bei gleichheit verfolgt wird (und nicht nur bei einem kleineren gesuchten schlüssel).
- Zudem wird ohne speicherung der daten in den inneren knoten die anzahl möglicher kindreferencen der knoten und damit der verzweigungsgrad bzw. fan-out des baumes und die efficienz der directen suche größer.
- Beim B*-baum muss die anzahl der einträge in blättern nicht mehr die gleiche sein wie die anzahl der schlüssel in indexbaumknoten, daher schreiben wir $\tau(k, k^*, h^*)$ für eine classe von B*-bäumen mit wurzelpfadlänge h^*, minimalen blatteinträgen k^* und mindestens $k + 1$ schlüsseln in inneren knoten.

Da in den inneren knoten keine daten mehr gespeichert sind, muss beim einfügen in eine volle seite der median auch im linken blattknoten erhalten bleiben.

Beispiel:



wird beim einfügen von 3 zu:



Beim löschen bleiben die schlüssel in den inneren knoten erhalten.

Der speicherbedarf von string-schlüsseln innerer knoten kann in manchen fällen auf externspeichern mit **präfix-suffix-comprimierung** optimiert werden, um mehr platz für kindreferencen und somit einen höheren verzweigungsgrad zu haben. Dazu haben wir für jeden schlüssel

- die position V, ab der er sich vom vorgänger unterscheidet (1, wenn kein vorgänger existiert),
- die position N, ab der er sich vom nachfolger unterscheidet (1, wenn es keinen nachfolger gibt),
- die anzahl $F = V - 1$ der zeichen, die mit dem vorgänger übereinstimmen und daher nicht noch einmal gespeichert werden
- und die länge $L = \max(N - F, 0)$ des comprimierten schlüssels

(nur F und L werden beim schlüssel gespeichert). Das ende eines schlüsselstrings wird bei uns im comprimierten schlüssel als punct geschrieben, ähnlich zu null-terminierten strings in z.b. der C-programmiersprache.

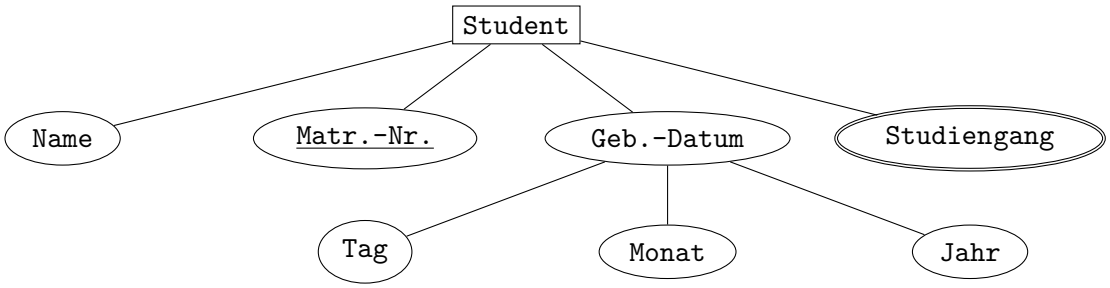
Schlüssel	V	N	F	L	Compr. Schlüssel
ILLUSION	1	6	0	6	ILLUSI
ILLUSTRATION	6	4	5	0	–
ILLYRIEN	4	3	3	0	–
ILSE	3	3	2	1	S
ILTIS	3	2	2	0	–
IM	2	3	1	2	M.
IMAGINAER	3	1	2	0	–

Der ursprüngliche schlüssel kann aus der präfix-suffix-compression nicht complett wiederhergestellt werden, für die wegweiser-schlüssel in den inneren knoten ist dies aber auch nicht nötig.

Wir können auch nur den gleichen anfang durch präfix-comprimierung einsparen und das ende nicht weglassen, dann lassen sich die schlüssel auch aus dem comprimierten baum reconstruieren.

Capitel 3: ER-Modell

Im **entity-relationship-modell** stellen wir die typen der objecte (entities) unserer miniwelt mit den sie beschreibenden attributen und beziehungen (relationships) zwischen den objecten als graph dar. Wir modellieren dabei nur entity-mengen als typen/classen der objecte und nicht die eigentlichen objecte als instancen /ausprägungen davon. Die menge der ausprägungen eines typs E nennen wir seinen inhalt E^t. Die einzelnen entitäten ändern sich oft, wohingegen die entitätsmengen in der regel gleich bleiben. Die historie, d.h. nicht mehr bestehende, vergangene beziehungen, modellieren wir normalerweise nicht im ER-diagramm.



Attributen ist dabei immer einen wertebereich zugewiesen; die werte, die die attribute für die einzelnen entitäten einer entitätsmenge annehmen, müssen im wertebereich liegen. Wir haben auch **mehrwertige attribute** (dargestellt als doppelte ovale) – die keinmal,

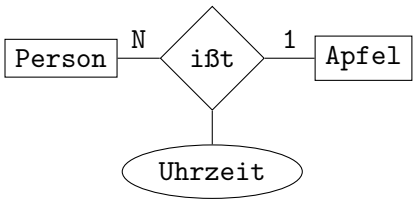
einmal oder mehrmals vorhanden sein dürfen (z.b. kann eine person mehrere vornamen haben, also wäre ein attribut `Vorname` von `Person` als mehrwertig zu modellieren) – , ebenso wie zusammengesetzte attribute, die aus anderen attributen aufgebaut sind. Ein einwertiges attribut einer entitätsmenge ist ein **schlüsselcandidat**, wenn es **eindeutig** ist (d.h. wenn verschiedene entitäten der menge nicht den gleichen wert für dieses attribut haben) und wenn es **irreducibel** ist (d.h. wenn das attribut nicht zusammengesetzt ist aus attributen, unter denen andere schlüsselcandidaten zu finden sind). Einer oder mehrere davon werden bei der specificierung der entitätsmenge als **primärschlüssel** ausgewählt (dargestellt durch unterstreichen); dabei handelt es sich um einen eindeutigen identificator für entitäten dieses typs.

Formal geben wir obiges beispiel-entity-relationship-diagramm wie folgt an: Für den entity-typ `Student` gilt `Student = ({Name, Matr.-Nr., Geb.-Datum(Tag, Monat, Jahr), {Studiengang}}, {Matr.-Nr.})`, wobei das erste tupel-element die menge der attribute von `Student` ist, während letzteres den primärschlüssel angibt. Man beachte die besonderen schreibweisen für zusammengesetzte und mehrwertige attribute.

Die wertebereiche der einwertigen attribute geben wir an als `W(Name) = char(30), W(Matr.-Nr.) = int(6), W(Tag) = W(Monat) = int(2), W(Jahr) = int(4)` (wir können bei entitäts- und beziehungstypen auch runde statt geschweiften klammern verwenden und die mengen somit als folgen schreiben). Für zusammengesetzte attribute schreiben wir `dom(Geb.-Datum) = W(Tag)` $\times$ `W(Monat)` $\times$ `W(Jahr) = int(2)` $\times$ `int(2)` $\times$ `int(4)` und für mehrwertige attribute `dom(Studiengang)` $= 2^{\texttt{W(Studiengang)}} = 2^{\texttt{char(50)}}$ oder alternativ `dom(Studiengang) = P(W(Studiengang)) = P(char(50))` (da man den wert als element der potenzmenge ansehen kann). Zusammensetzungen aus mehrwertigen attributen schreiben wir analog als `dom(A) = dom(B)` $\times$ `dom(C)`.

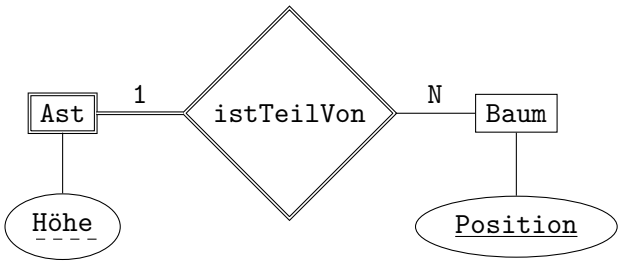
Beziehungen zwischen zwei entitäten (d.h. vom **grad** 2) stellen wir als eventuell beschriftete verbindungslinie dar. Das herstellen einer beziehung nennen wir connect. Beziehungen zwischen mehr als zwei entitäten, also von einem höheren grad, werden als verbindungslinien aller beteiligten entitätstypen mit einer raute dargestellt, welche mit der art der beziehung beschriftet ist. Die rautendarstellung brauchen wir auch, wenn wir die beziehung selbst mit attributen versehen möchten. Eine beziehung zweiten grades ist von einer teilnehmenden entität aus gesehen vom **beziehungstyp** 1:n, wenn die entität mit n anderen entitäten in dieser beziehung steht (z.b. kann eine person mehrere äpfel essen, während ein apfel nur von einer person gegessen werden kann). Analog sind n:1-, n:m- und 1:1-beziehungen definiert. Im diagramm schreiben wir allerdings stattdessen an die verbindungslinien zur raute die anzahl der beziehungen dieses typs, an der eine entität des typs teilnehmen kann, ohne die anzahl anderer teilnehmer zu beachten, z.b. kann eine person mehrmals an einer `ißt`-beziehung teilnehmen (`N`), aber ein apfel kann – unter der annahme, dass äpfel nur complett und von einer person alleine gegessen werden – nur an einer solchen beziehung teilnehmen (Achtung! Bei UML wird es genau an die andere seite geschrieben! Nicht mit UML aus z.b. SE2 verwechseln!). Dies ist eine integritätsbedingung. Wir können die verbindungslinien zu rauten auch mit den rollen der entitäten beschriften, was besonders nützlich bei der specificierung der teilnahme

mehrerer entitäten des selben typs in verschiedenen rollen an der beziehung ist (z.b. bei einer beziehung `elternteilVon` von `Person`en).



Den typ der beziehung schreiben wir formal als `ißt = ({Person, Apfel}, {Uhrzeit})`, somit gilt für eine beziehung r dieses typs $r \in \texttt{Person}^t \times \texttt{Apfel}^t \times \texttt{dom(Uhrzeit)}$. Wir schreiben $\texttt{ißt}^t$ für die menge solcher relationships. Bei verwendung von rollennamen werden diese vor den entitätsnamen geschrieben und mit einem schrägstrich von diesem getrennt.

Wir können entitätstypen **existenzabhängig** bzw. **schwach** bezüglich anderen entities machen, indem wir die teilnahme an einer beziehung zu den anderen fordern. Dazu zeichnen wir eine doppelte verbindungslinie zur raute der relationship und umranden die raute außerdem doppelt. Zur eindeutigen identification kann dann ein eigenes attribut (dies wird gestrichelt unterstrichen) in combination mit dem primärschlüssel einer entität, von der die zu identificierende identität existenzabhängig ist, an stelle eines eigenen primärschlüssels verwendet werden. Eine existenzabhängige entität muss nicht von allen entitäten abhängen (erzeugt werden), mit denen sie in einer beziehung steht (referenciert wird). Existenzabhängigkeit ist eine integritätsbedingung.



Um eine relationship von einem grad größer als 2 in einem system zu modellieren, welches nur relationships von grad 2 unterstützt, ist die modellierung der beziehung als eine existenzabhängige entität mit beziehungen zu allen beteiligten entitäten notwendig.

Erweiterungen zum ER-Modell

Wir können das ER-Modell erweitern um weitere integritätsbedingungen und übliche beziehungen der objectorientierten modellierung.

- **Cardinalitätsrestrictionen**: Statt die anzahl der an einer beziehung teilnehmenden ausprägungen nur als `1` oder `N` zu schreiben, können wir z.b. `[3,5]` für 3 bis 5 teilnehmende objecte im diagramm schreiben. Formal schreiben wir

dann `card(nimmtTeil, Person)=[3,5]`. In der fragestunde wurde uns gesagt, dass wir als erweiterte cardinalitätsrestrictionen nur `[0,1]`, `[0,n]`, `[1,1]` und `[1,n]` verwenden sollten; auf den folien gab es aber auch beispiele wie `[1,20]`. Letzteres sei aber zumindest schlechter stil.

- Wir erlauben auch die darstellung einzelner ausprägungen.
- Wir können default-werte an attributen angeben wie `Farbe = rot`,
- Wir können operationen für entitäten, d.h. was das object tun kann (was in der programmierung methoden sind), angeben.
- **Abstraction**: Wir führen übliche concepte der objectorientierung ein, die vom DBS als integritätsbedingungen geprüft werden:
  - **Classification** bedeutet, dass wir mit `instance-of-` bzw. `io`-beziehungen zeigen, dass ein object eine ausprägung eines entitätstyps ist, wodurch wir attribute, beziehungen, &c. nur einmal für den typ angeben müssen, wenn ihm mehrere entitäten angehören; `instantiation` ist das gleiche concept, aber im umgekehrten fall, dass wir schon eine entitätsmenge haben und dafür instancen angeben. Mehrfachvererbung ist möglich. Geerbtes kann überschrieben werden (overriding) – bei wertebereichen wird stattdessen die schnittmenge gebildet und attribute werden nicht überschrieben.
  - **Generalisierung** und dual dazu **specialisierung** bedeutet, dass wir `subclass-of-` bzw. `superclass-of-` bzw. `sc-` bzw. `is-a`-beziehungen darstellen, um anzuzeigen, dass ein entitätstyp von einem anderen entitätstyp attribute, beziehungen, default-werte, operationen und integritätsbedingungen erbt. Bei mehrclassenmitgliedschaft kann das system nicht mehr alle integritätsbedingungen garantieren. Eine menge von subclassen der selben classe nennen wir vollständig, wenn sie alle subclassen der classe umfaßt. Eine solche menge kann disjunct oder überlappend sein. `is-a`-beziehungen dürfen wir auch statt mit der beschriftung `is-a` mit einer besonderen art von pfeil darstellen; siehe die musterlösung zu übungsblatt 4.
  - **Element-association** ist die zusammenfassung von objecten zu mengen mit einer `element-of-` bzw. `eo`-beziehung. **Mengen-association** ist das analoge concept für mengen über eine `subset-of-` bzw. `ss`-beziehung.
  - **Element-aggregation** ist die darstellung einer `part-of-` bzw. `po`-beziehung für zusammengesetzte objecte (aggregate). Aggregation nennen wir exclusiv, wenn kein anderes object diesen teil haben kann, sonst nennen wir sie gemeinsam. Exclusive aggregation heißt in UML auch **composition**. Es gibt auch die Componentenaggregation mit `component-of-`/`co`-beziehungen bzw. `consists-of`-beziehungen – diese wird verwendet, wenn ein zusammengesetztes object auch wieder aus zusammengesetzten objecten aufgebaut ist (siehe übungsblatt 6). Componenten einer subcomponenten von einer entität

A sind auch componenten von A. Das DBS überträgt prädicate auf aggregate als **upward implied predicate** (z.b. das eingenommene volumen) bzw. auf componenten als **downward implied predicate** (z.b. der preis).

Wenn wir aus einem text ein ER-modell erstellen sollen, kann es helfen, mögliche entitäten, relationen und attribute im text verschiedenfarbig zu unterstreichen, bevor man mit dem zeichnen beginnt. Die formale notation des beispiels auf den folien (als `DECLARE RELATIONSHIP RELATION PERS.-ANGEHÖRIGE ...`) müssen wir nicht können.

Capitel 4: Relationenmodell

Beim relationenmodell stellen wir die objecte unserer miniwelt als tupel („zeilen“) in relationen („tabellen“) dar. Die anzahl der werte in einem tupel nennen wir seinen grad (nicht zu verwechseln mit z.b. dem grad einer beziehung). Die typinformationen zu den „spalten“ werden in einem relationenschema gespeichert. Integritätsbedingungen nennen wir hier relationale invarianten. Beziehungen des ER-modells entsprechen genau wie entitäten relationen im relationenmodell. Relationen können formal auch wie in der mathematik als teilmengen des cartesischen products der wertebereiche der für eine relation eindeutig benannten attribute („spalten“) geschrieben werden. Die reihenfolge der tupel und attribute wird nicht beachtet. Das concept von primärschlüsseln besteht noch immer. Verweise auf tupel eventueller anderer relationen werden über deren primärschlüssel oder andere schlüsselcandidaten als **fremdschlüssel** (foreign keys) umgesetzt (wir unterstreichen fremdschlüssel gestrichelt; sind es zudem noch primärschlüssel werden sie einmal normal und einmal gestrichelt unterstrichen). Bei einer cyclischen kette von verweisen sprechen wir von einem geschlossen referentiellen pfad. Verweise und andere werte dürfen, sofern für die entsprechende spalte nicht die integritätsbedingung `NOT NULL` für existenzabhängigkeit specificiert wurde, auch für manche oder alle tupel fehlen; an stelle des wertes steht dann der wert `NULL`.

Im gegensatz zum erweiterten ER-modell kann im relationenmodell keine semantik und keine abstraction specificiert werden; auch sind nur 1:n-beziehungen ohne attribute modellierbar, ohne sie als eigene relation darzustellen. Zum specificieren von beziehungen verwenden wir relationen, die die primärschlüssel der beteiligten entitäten selbst als primärschlüssel haben.

Partitionierung

Wir können relationen in teilrelationen zerlegen über **horizontale partitionierung**, wobei classen anhand von prädicaten zur selection aufgeteilt werden (zum beispiel kommen alle personen ab einer bestimmten körpergröße in eine menge `Riese` und die übrigen in `Zwerg`), und über **verticale partitionierung**, wobei mengen nach unterschiedlichen z.b. zugriffsberechtigungen aufgespalten werden.

Generalisierung und aggregation im relationenmodell

Zur übertragung einer generalisierung im ER-modell ins relationenmodell können wir entweder

- mit horizontaler partitionierung alle entitätstypen vollständig in je einer eigenen „hausclasse“ speichern (also ohne vererbung, aber mit bester speicherausnutzung für die daten),
- oder wir nutzen verticale partitionierung und speichern in einer relation für jede entitätsmenge eine für die allgemeinste classe eindeutige ID jedes objects der menge zusammen mit den für diesen vererbungsschritt neu hinzugekommenen attributen und beziehungen.
- Eine dritte möglichkeit wäre, in allen entitätsmengen alle attributswerte der menge für alle objecte der menge redundant zu speichern, auch wenn sie schon anderswo gespeichert sind („volle redundanz“). Dadurch wird zwar der zugriff einfacher, aber das ändern von daten wird umständlicher, es muss dafür gesorgt werden, dass die daten consistent bleiben und der speicherplatzverbrauch ist hoch.
- Alternativ kann man auch nur eine relation für die ganze hierarchie verwenden („hierarchierelation“), wo die attribute, die ein object nicht hat, einfach den wert `NULL` zugewiesen bekommen.

Aggregation können wir im relationenmodell nicht direct modellieren.

Relationenalgebra

Auf den relationen im relationenmodell kann man verschiedene operationen durchführen. Dies sind zum einen die üblichen mengenoperationen: Vereinigung (bei gleichen attributen, d.h. bei **vereinigungsverträglichkeit**), differenz (ebenso), schnitt (ebenso), die symmetrische differenz und das cartesische product (wobei wir nach berechnung des cartesischen productes die paare aus attributswertfolgen ersetzen durch concatenationen der folgen). Bei letzterem führen wir eine implicite umbenennung durch, wenn wir namensconflicte durch gleichnamige attribute bekommen, so dass z.b. ein attribut `NAME` einer relation `PERSON` zu `PERSON.NAME` wird.

Beispiel:

SCHÜLER	NAME	ALTER
	Chris	20
	Hilary	18
	Leslie	21

{(Chris, 20), (Hilary, 18)} $\cup$ {(Leslie, 21)} = {(Chris, 20), (Hilary, 18), (Leslie, 21)}
{(Chris, 20), (Hilary, 18)} $\setminus$ {(Chris, 20)} = {(Hilary, 18)}
{(Chris, 20)} $\triangle$ {(Chris, 20), (Hilary, 18)} = {(Hilary, 18)}

`{(Chris, 20), (Hilary, 18)}` $\cap$ `{(Chris, 20)} = {(Chris, 20)}`
`{(Chris, 20), (Hilary, 18)}` $\times$ `{(Leslie, 21)} = {(Chris, 20, Leslie, 21), (Hilary, 18, Leslie, 21)}`

Eine weitere operation ist die selection σ_P für ein prädicat P, wobei $\sigma_P(R)$ ein relationstupel $t \in R$ dann und nur dann enthält, wenn $P(t)$ gilt. In der logischen formel P verwenden wir als variablennamen die attributsnamen und belegen die variablen beim prüfen von $P(t)$ mit den entsprechenden werten. Zum beispiel ist $\sigma_{\texttt{NAME='Chris'} \wedge \texttt{ALTER}<21}(\texttt{SCHÜLER}) = \texttt{\{(Chris, 20)\}}$.

Mit der projection $\pi_{A_1,\ldots,A_n}(R)$ erhalten wir eine copie der reation R, bei der alle attributswerte außer $A_1, \ldots, A_n$ gestrichen sind. Es ist $\pi_{\texttt{NAME}}\texttt{(SCHÜLER)} = \texttt{\{(Chris), (Hilary), (Leslie)\}}$. Duplicate werden hierbei in der relationalen algebra eliminiert.

Mit dem **rename-operator** können wir eine implicite umbenennung von attributsnamen durchführen: $\rho_{\texttt{VORNAME}\leftarrow\texttt{NAME}}(\texttt{SCHÜLER})$. Wir können auch die namen von relationen ändern: $\rho_{\texttt{PERSON}}(\texttt{SCHÜLER})$. Beides ist besonders nützlich in verbindung mit dem cartesischen product.

Eine weitere operation ist der **verbund** bzw. **join**, der bei nicht vereinigungsverträglichen relationen nützlich ist. Ein **Θ-join** für einen arithmetischen vergleichsoperator Θ (z.b. $\neq$ oder $\leq$) von zwei relationen ist $R \underset{A_R \Theta A_S}{\bowtie} S = \sigma_{A_R \Theta A_S}(R \times S)$ für attributsnamen A_R und A_S (statt attributsnamen können auch spaltennummern angegeben werden). Ist Θ die gleichheit, so sprechen wir von einem gleichverbund bzw. equijoin. Sind im gleichverbund alle tupel von R und S in irgendeiner combination enthalten, d.h. lassen sich alle tupel von R oder von S über projection wiederherstellen, so nennen wir den join verlustfrei oder lossless. Beispiel: `{(Chris, 20), (Hilary, 18)}` $\underset{\texttt{SCHÜLER.NAME=FÄCHER.NAME}}{\bowtie}$ `{(Chris, Bio), (Chris, Sport), (Hilary, Sport), (Leslie, Bio)} = {(Chris, 20, Sport), (Chris, 20, Bio), (Hilary, 18, Sport)}`.

Der **natürliche verbund** $R \bowtie S$ von relationen R und S liefert uns all die elemente von $R \times S$, bei denen die werte der gleichnamigen attribute von R und S gleich sind. Das gleiche beispiel wie oben mit natürlichem verbund: `{(Chris, 20), (Hilary, 18)}` $\bowtie$ `{(Chris, Bio), (Chris, Sport), (Hilary, Sport), (Leslie, Bio)} = {(Chris, 20, Chris, Sport), (Chris, 20, Chris, Bio), (Hilary, 18, Hilary, Sport)}`.

Die **division** $R \div S$ ist eine operation, deren resultat die attribute von R hat, die nicht in S vorkommen, und worin alle projicierten tupel aus R enthalten sind, die combiniert mit allen tupeln aus S in R vorkommen. Beispielsweise ist `{(Chris, Sport), (Chris, Bio), (Hilary, Sport), (Leslie, Bio)}` $\div$ `{(Sport), (Bio)} = {(Chris)}`.

Der **intervallverbund** (**band join**) $R \underset{A_R[c_1,c_2]A_S}{\bowtie} S$ selectiert alle tupel des cartesischen products $R \times S$, bei denen der wert von A_S in einem intervall $[-c_1, c_2]$ um den wert von A_R liegt, d.h. nicht um mehr als c_1 kleiner und nicht um mehr als c_2 größer ist.

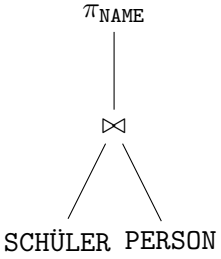
Beispiel: Sei $R :=$ `{(Chris, 20), (Hilary, 18)}` und $S :=$ `{(Leslie, 21)}`. Dann ist $R \underset{R.ALTER[1,1]S.ALTER}{\bowtie} S =$ `{(Chris, 20, Leslie, 21)}`.

Der **äußere verbund** (**outer join**) $R \rtimes\!\!\bowtie S$ ist die vereinigung des natürlichen verbunds $R \bowtie S$ (äußerer natürlicher gleichverbund) mit allen tupeln aus R oder S, die am verbund nicht teilnehmen, ergänzt um `NULL`-werte. Für `NULL`-werte verwenden wir das symbol $\equiv$. Statt des natürlichen verbunds können wir auch einen herkömmlichen gleichverbund nutzen: $R \underset{A_R=A_S}{\rtimes\!\!\bowtie} S$. Wir können auch nur die nicht vorkommenden tupel von R um `NULL`-werte ergänzen, damit bekommen wir den **linken äußeren gleichverbund** $\rtimes\!\!\bowtie$. Analog ist der **rechte äußere gleichverbund** $\bowtie\!\!\ltimes$ definiert. Beispiel: Sei $R :=$ `{(Chris, 20), (Hilary, 18), (Jesse, 21)}` und $S :=$ `{(Leslie, 21)}`. Dann ist $R \underset{R.ALTER=S.ALTER}{\rtimes\!\!\bowtie} S = R \underset{R.ALTER=S.ALTER}{\bowtie\!\!\ltimes} S =$ `{(Chris, 20, ≡), (Hilary, 18, ≡), (Jesse, 21, Leslie)}`, aber $R \underset{R.ALTER=S.ALTER}{\bowtie\!\!\ltimes} S =$ `{(Jesse, 21, Leslie)}`.

Optimierung

Wir können anfragen in form von operationen der relationalen algebra formulieren und als operatorbaum aufschreiben.

Beispiel: $\pi_{\texttt{NAME}}(\texttt{SCHÜLER} \bowtie \texttt{PERSON})$:



Das DBS optimiert anfragen so, dass diese möglichst zeitsparend durchgeführt werden, d.h. so, dass nicht unnötig tupel und attribute durchsucht oder verarbeitet werden, obwohl schon klar ist, dass sie nicht gebraucht werden. Wir optimieren, indem wir eine anfrage so gestalten, dass die anzahl der in verarbeitungsschritten verarbeiteten tupel minimal wird – dazu können wir die verbindungslinien in operatorbäumen beschriften (bewerten) mit der zu erwartenden anzahl an tupeln und attributen davon, die für den nächsten schritt bearbeitet werden müssen. Ein factor für die bewertungen ist, wie sehr selectionen tupelelemente erwartungsgemäß herausfiltern – bei einer prüfung auf gleichheit mit einem bestimmten wert in der selection und n möglichen attributswerten wäre der **selectivitätsfactor** beispielsweise $\frac{1}{n}$. Der **join-selectivitätsfactor** (JSF) ist der selectivitätsfactor der selection beim join.

Bei der optimierung versuchen wir daher, projectionen (anders als bei normaler relationaler algebra ohne zeitaufwendige duplicateliminierung) und noch mehr selectionen

möglichst früh durchzuführen. Wir fassen unäre operationen (z.b. selection, projection) danach zusammen; dies markieren wir, indem wir ein oval um die zusammengefaßten operationen ziehen. Cartesische producte mit darauf folgenden selectionen können wir zu joins verschmelzen. Gemeinsame teilbäume erhöhen die zeitkosten nur einmal, da die ergebnisse zwischengespeichert werden können. Verbünde werden so möglichst efficient angeordnet; bei mengenoperationen werden zunächst kleinere mengen verknüpft.

Wir können ausnutzen, dass vereinigung, schnitt, verbunde und cartesische producte associativ und commutativ sind und selectionen wie projectionen commutativ sind und die für die selection gilt, dass $\sigma_{A \wedge B \wedge C}(R \times S) = \sigma_A(\sigma_B(R) \times \sigma_C(S))$ für ein B, welches nur attribute von R enthält, und ein C, welches nur attribute von S enthält.

Capitel 5: SQL

SQL (**Structured Query Language**) (früher SEQUEL – Structured English Query Language – genannt; der name mußte aus markenrechtsgründen aufgegeben werden) ist eine standardisierte sprache zur formulierung von anfragen und datenmanipulationsbefehlen (DML-befehlen; DML steht für data manipulation language) an datenbanksysteme. Die sprache ist ebenso mächtig und ähnlich aufgebaut wie das relationenmodell. Unsere relationen werden in SQL tabelle (table) genannt. Die befehle und anfragen werden entweder von einem menschlichen benutzer oder einem anwendungsprogramm gestellt. Ein DBS muss zur nutzung von SQL die ACID-eigenschaften erfüllen.

Anfragen

1. Eine anfrage (query) in SQL beginnen wir mit `SELECT`.
2. Darauf folgt ein `DISTINCT`, wenn wir gleiche tabellenzeilen im endergebnis der anfrage streichen lassen wollen, andernfalls ein `ALL` (was wir auch weglassen dürfen, da es sich um den default-fall handelt).
3. Es folgt in jedem fall entweder ein stern `*`, welcher alle attributsnamen ins ergebnis übernommen werden läßt, oder eine liste der attributsnamen, deren spalten wir in der ergebnistabelle haben wollen. Nach jedem namen können wir auch `AS` mit einem neuen namen schreiben, in den wir das attribut in der ergebnistabelle umbenennen. Statt eines attributs können wir in `SELECT`-clauseln auch einen wert (wie `'Ja'`) oder eine **aggregatfunction** auf attribute verwenden, z.b. als `AVG(ALTER)`, optional mit einem `DISTINCT` oder `ALL` in der klammer vor dem attributsnamen. Aggregatfunctionen arbeiten immer nur mit nicht-`NULL`-werten; werte werden nur ausgegeben, wenn auch tupel in der relation sind. Aggregatfunctionen sind `AVG`, `MAX`, `MIN`, `SUM` und `COUNT` (letzteres kann auch mit dem argument `*` benutzt werden und zählt dann alle zeilen unabhängig von den attributswerten).

4. Dann kommt die FROM-clausel, bestehend aus FROM und dem namen einer tabelle oder einer sicht. Aus dieser wählen wir die ergebnistupel aus, die die folgenden criterien erfüllen. Wir können auch mehrere tabellen bzw. sichten angeben (z.b. FROM SCHÜLER S, PERSON P), woraus dann das cartesische product gebildet wird; die attribute werden dann implicit nach dem angegebenen namen umbenannt wie in der relationalen algebra (symmetrische notation). Es kann auch explicit ein join an stelle des cartesischen productes angegeben werden, aus dem wir dann zeilen auswählen. Dazu schreiben wir z.b. FROM SCHÜLER S JOIN PERSON P, gefolgt von entweder ON und der join-bedingung oder von USING gefolgt von einem oder mehreren gemeinsamen attributsnamen in klammern (für einen gleichverbund). Wir können auch z.b. FROM SCHÜLER S NATURAL JOIN PERSON P schreiben für einen natürlichen verbund. Bei USING und NATURAL JOIN werden gleiche spalten nur einmal aufgeführt.

5. Es folgt optional die WHERE-clausel aus WHERE und einer bedingung als folge von durch AND und OR getrennten prädicaten wie NAME = CHRIS (analog zu selectionen), eventuell negiert mit NOT und eventuell mit einem quantificierenden ALL oder SOME nach dem vergleichsoperator. Wir können auch ein prädicat der form NAME IN ('Chris', 'Hilary') oder – als schlechtere, nicht automatisch optimierte alternative zur symmetrischen notation in der FROM-clausel – sogar NAME IN (SELECT NAME FROM ...) nutzen. EXISTS und NOT EXISTS prüft die nachfolgende menge oder mit SELECT specificierte relation auf nicht-leerheit bzw. leerheit. Mit x BETWEEN y AND z können wir sicherstellen, dass ein attributswert in einem intervall liegt. NOT können wir auch vor das IN oder BETWEEN statt vor das prädicat stellen. Ein prädicat A IS NULL oder A IS NOT NULL prüft, ob der wert von A ein bzw. kein NULL-wert ist. Mit dem LIKE-prädicat lassen sich ähnlichkeitssuchen durchführen, ähnlich wie suchen mit regulären ausdrücken: NAME LIKE '%LAR_' ist wahr für zeilen mit einem NAME-wert, der beliebig beginnt und auf LAR und genau einen weiteren buchstaben endet. Für phonetische ähnlichkeit setzen wir ein ? vor den begriff und für semantische eine $\sim$ (beides ist nicht teil des SQL-standards und wird von einem DBS wahrscheinlich nicht auf diese art unterstützt). SIMILAR TO tut das gleiche wie LIKE, aber mit richtigen regulären ausdrücken.

6. Wir können GROUP BY benutzen, um die aggregatfunctionen nur auf die gruppe von zeilen, die bei dem hier genannten attribut den gleichen wert wie in der entsprechenden ergebnistabellzeile haben, anzuwenden. Zum beispiel gibt SELECT GRUPPENNR, AVG(ALTER) FROM PERS GROUP BY GRUPPENNR zu jeder gruppennummer das durchschnittsalter aus, statt den durchschnitt unter allen gruppen auszugeben. Wenn wir eine aggregatfunction benutzen und gleichzeitig andere tupel wie GRUPPENNR auswählen, müssen wir diese in der GROUP BY-clausel angeben.

7. Optional folgt auf ein GROUP BY eine clausel HAVING mit einem prädicat, welches auch aggregatfunctionen beinhalten darf. Dadurch schränkt man die zeilen, die GROUP BY zurückgibt, ein, bevor sie von aggregatfunctionen im SELECT-teil be-

arbeitet werden.

8. Man kann dann `ORDER BY` mit einer liste von attributsnamen angeben, nach denen die zeilen in der angegebenen reihenfolge geordnet werden. Es kann nur bei der äußeren, nicht bei einer verschachtelten, `SELECT`-anfrage verwendet werden. Ohne `ORDER BY` wird die reihenfolge vom DBS gewählt, so dass kein unnötiger sortieraufwand nötig ist.

Wir können für attribute immer, auch in argumenten zu aggregatfunctionen, arithmetische operationen benutzen, z.b. als `PREIS/2`.

SQL-befehle sind, im gegensatz zu den daten, die ihnen übergeben werden, nicht casesensitive, d.h. statt `SELECT` kann man auch `select` schreiben (aber nicht `'CHRIS'` statt `'Chris'`).

`NULL`-werte liefern in logischen ausdrücken als wahrheitswert `UNKNOWN` bzw. `?`, außer bei booleschen ausdrücken wie einem wahren term gefolgt ver`AND`et mit einem `NULL`-wert, der unabhängig vom `NULL`-wert schon wahr oder falsch ist. Es gilt immer `NULL` $\neq$ `NULL`.

Wir können mengenoperationen auf die ergebnisse von `SELECT`-anfragen durchführen, indem wir zwischen mehrere `SELECT`-anfragen `UNION`, `INTERSECT` oder `EXCEPT` schreiben.

Datenmanipulation

Ein befehl wie `INSERT INTO PERSON (PNR, NAME, ALTER) VALUES (2, 'Chris', 18)` fügt in die angegebene tabelle eine zeile mit zu den angegebenen attributen gehörenden angegebenen werten ein. Statt eines wertes kann auch `DEFAULT` stehen, wenn für das attribut ein default-wert definiert wurde. Nicht angegebene attribute erhalten in der zeile den wert `NULL`. Die tabelle, in die wir einfügen, muss außerdem existieren. Aus einer anderen tabelle mit compatiblen attributtypen lassen sich zeilen copieren mit befehlen wie `INSERT INTO PERSON (SELECT * FROM SCHÜLER WHERE ALTER > 17)`.

An eine `SELECT`-anfrage können wir vor der `FROM`-clausel ein `INTO` mit einem tabellennamen einfügen, wodurch das ergebnis in diese tabelle eingefügt wird.

Mit `DELETE FROM PERSON` löschen wir alle zeilen aus der angegebenen tabelle (hier `PERSON`). Mit einer optionalen `WHERE`-clausel können wir analog zu `SELECT` einschränken, was gelöscht wird.

Mit `UPDATE PERSON P SET P.ALTER = P.ALTER + 1` ändern wir die werte in zeilen einer tabelle `PERSON`. Auf `SET` folgt dabei eine commagetrennte liste von zuweisungen in form eines attributnamens, dem zeichen `=` und einem arithmetischen ausdruck. Darauf kann wieder ein `WHERE` folgen.

Bei SELECT-anfragen innerhalb von DELETE- oder UPDATE-befehlen darf die gelöschte bzw. geänderte relation nicht im FROM-teil stehen.

Datendefinition

Eine **SQL-umgebung** besteht aus einer instanz (also einer installation) eines DBMS mit einer reihe von nutzern, die in form von authorization identifiers gegeben sind, programmen (modules) zum aufruf über SQL-befehle zum stellen vorgefertigter anfragen, und **catalogen** zum speichern der metadaten. Cataloge, die selbst als datenbanken (mit vom DBS vorgegebener structur) aufgebaut sein können, umfassen mehrere **SQL-schemata**. Schemata speichern informationen zu tabellen, integritätsbedingungen, zugriffsrechten, verwendetem zeichensatz, &c. Zum zugriff auf schemata gibt es traditionell ein DBS-specifisches definitionsschema und mittlerweile auch ein vom SQL-standard genormtes informationsschema als benutzerzugängliche schnittstellen.

Wir erstellen ein schema mit CREATE SCHEMA, gefolgt vom namen des schemas (was der name der datenbank sein kann), optional gefolgt von AUTHORIZATION und dem benutzernamen des administrators des schemas, optional gefolgt von weiteren angaben wie dem zeichensatz.

Zur festlegung zulässiger werte schreiben wir CREATE DOMAIN, gefolgt vom gewünschten attributsnamen, optionalem wirkungslosem AS, dem datentyp (siehe unten), optional DEFAULT mit einem default-wert (kann auch NULL sein), und optional CHECK mit einer *geklammerten* integritätsbedingung in form eines prädicats mit VALUE als variable für einen wert des hier definierten „typs“. Vor CHECK darf noch CONSTRAINT mit einem namen stehen, den wir der bedingung geben wollen (z.b. MINDESTLOHN). Wir haben damit eine domäne definiert, die wir wie die vorgegebenen datentypen in attributdefinitionen nutzen können.

Datentypen sind

- CHARACTER (kurz CHAR) für zeichenketten mit einer theoretisch optionalen anzahl der zeichen in runden klammern,
- CHARACTER VARYING (kurz VARCHAR) für zeichenketten mit variabler länge (wieder angegeben in runden klammern; der unterschied zu **CHAR** ist, dass die zeichenketten so gespeichert werden, dass sie weniger platz brauchen, wenn sie nicht so lang sind),
- NUMERIC oder DECIMAL (kurz DEC, wir sollten DECIMAL bzw. DEC benutzen) für festcommazahlen mit optionaler klammer mit der genauigkeit (anzahl der ziffern) und eventuell noch einer weiteren angabe (scale), wie viele ziffern davon nach dem comma kommen,
- REAL für fließcommazahlen,
- INTEGER (kurz INT für ganzzahlen,

- `DATE` für datumsangaben,
- und `TIME` für zeitangaben.

Eine definition für ein attribut bzw. eine spalte besteht aus

- der angabe des namens gefolgt vom datentyp oder der domäne, aus der die werte des attributs kommen.
- Danach können wir optional mit `DEFAULT` den default-wert wie bei domänendefinitionen für das attribut überschreiben;
- auch constraints können wir hier wieder definieren, allerdings dürfen wir als prädicat hier auch `NOT NULL` oder `UNIQUE` (für schlüsselcandidaten) oder `PRIMARY KEY` (welcher implicit `UNIQUE` und `NOT NULL` ist) verwenden.
- Am ende der definition kann noch stehen, wann die integritätsbedingung geprüft werden soll – dies ist die angabe der deferrability über `INITIALLY DEFERRED` oder `INITIALLY IMMEDIATE`; letzteres stellt die consistenz für das attribut auch während transactionen sicher und ist der default-modus wenn wir die angabe weglassen.

Diese syntax der attributdefinition verwenden wir, wenn wir tabellen erstellen mit

- `CREATE TABLE`,
- dem tabellennamen,
- und in klammern einer liste von attributdefinitionen und tabellenconstraints.

Für letztere kennen wir die syntax

- `FOREIGN KEY` mit einem attributnamen in klammern zur verwendung als fremdschlüssel
- gefolgt von `REFERENCES` und dem namen der bezugstabelle (dies kann auch eine selbstreferenz sein) – optional mit angabe der in frage kommenden attribute in runden klammern –,
- gefolgt von optionalen `ON DELETE` und `ON UPDATE` mit jeweils einer referentiellen action.

`REFERENCES` kann auch bei der attributdefinition als constraint stehen, wenn der fremdschlüssel nur ein attribut umfaßt (analog zu `PRIMARY KEY`). Referentielle actionen sind `NO ACTION`, `RESTRICT`, `CASCADE`, `SET DEFAULT` und `SET NULL`; diese geben an, was getan werden soll, wenn sich das referencierte object ändert (`CASCADE` bedeutet, dass sich die änderung auch auf die referencierende zeile auswirkt (d.h. diese wird auch gelöscht oder geupdatet); `RESTRICT` und das default `NO ACTION` verhindern die löschung bzw. das update der referencierten zeile, wenn dadurch die referentielle integrität verletzt würde, wobei `RESTRICT` im gegensatz zu `NO ACTION` die integrität auch während transactionsausführungen und nicht erst am ende verlangt; `SET DEFAULT` und `SET NULL` setzen bei verletzung der integrität den fremdschlüsselwert entsprechend auf den default-wert bzw.

`NULL`). Die referentiellen beziehungen zwischen den tabellen unseres schemas nennen wir unabhängig, wenn der endzustand der tabellen nicht von der reihenfolge der durchführung referentieller actionen abhängt; unser schema heißt dann **sicheres schema**, andernfalls conflictträchtiges schema. Ein DBS sollte immer ein eindeutiges ergebnis haben – conflictträchtige schemata werden entweder verboten (durch statische analyse oder durch dynamische analyse zur laufzeit wird die sicherheit des schemas geprüft) oder man geht mit ihnen um, indem uneindeutige actionen zur laufzeit dynamisch erkannt werden und automatisch fehlschlagen (letzteres ist im actuellen SQL-standard vorgeschrieben; `IMMEDIATE` und `DEFERRED` spielen dabei eine rolle; siehe oben).

Einen aus mehreren attributen zusammengesetzten primärschlüssel können innerhalb der „`CREATE TABLE`“-anweisung hier auf einer eigenen zeile nach den attributen angeben als `PRIMARY KEY (attr_1,...,attr_n)`. (siehe folie 5-74)

Von mengen- und satzorientierter verarbeitung müssen wir die definitionen nicht können.

Die referencierten attribute können wir als kantenbeschriftungen in einem gerichteten graphen mit den relationen als knoten aufschreiben (referenzgraph); darüber drücken wir 1:n-beziehungen aus (eventuell mit existenzabhängigkeit durch `NOT NULL` oder durch definition als teil des primärschlüssels; mit `UNIQUE` stellen wir sicher, dass höchstens eine zeile mit dieser zeile in beziehung steht (1:1-beziehung); `UNIQUE NOT NULL` stellt beides sicher). 1:n-beziehungen können wir nur als eigene relationen, nicht als referenz, ausdrücken.

Auf insy.mensa-kl.de ist ein link zu SQL Fiddle, womit wir SQL üben können.

Capitel 6: Transactionen

Befehle an das SQL nutzende DBS werden im read/write-modell in operationen unterteilt, die in transactionen zusammengefaßt werden. Diese transactionen müssen die ACID-criterien erfüllen. Transactionen, die alleine, also nicht parallel mit anderen transactionen ausgeführt werden, terminieren immer mit einem consistenten DB-zustand. Ein einzelner befehl heißt dabei **atomare action**. Am ende einer transactionsverarbeitung steht ein **commit** oder ein **abort** (d.h. ein abbruch), wenn diese erfolgreich war – der effect der transaction wird beim commit der actuell gültige zustand des DBS und der erfolg wird dem nutzer bestätigt. Ein nicht erfolgreiches ende einer transaction oder ein abort zieht einen **rollback** nach sich, wobei sichergestellt werden muss, dass die transaction keine auswirkungen auf den DBS-zustand und andere transactionen hat. Einen rollback durch einen fehler im system und nicht in der transaction nennen wir erzwungen.

Bei transactionen muss garantiert sein, dass nach einem commit alle constraints erfüllt sind (transactionsconsistenz). Dafür muss DML-operationsconsistenz – das erfülltsein

aller constraints nach durchführung von einzelnen DML-operationen innerhalb einer transaction – gegeben sein. DML-operationsconsistenz wird vom datensystem garantiert. Dafür müssen die einzelnen actionen auf sätze im zugriffssystem bei der durchführung eines DML-befehls consistent sein (actionsconsistenz), ebenso wie die zugriffe auf seiten in geräten und dateien durch das speichersystem (geräte- und dateiconsistenz).

Das problem bei der transactionsconsistenz ist der mehrbenutzerbetrieb, bei dem verschiedene benutzer verschiedene transactionen parallel durchführen (nach einem verzahnten ablaufplan, d.h. teile von transactionen finden zwischen teilen anderer transactionen statt, im gegensatz zum seriellen ablaufplan, wo alle transactionen nacheinander ablaufen). Wir müssen verhindern, dass transactionen hierbei änderungen parallel ausgeführter transactionen teilweise überschreiben (haben diese dadurch keine auswirkungen, obwohl sie in serieller ausführung welche hätten, sprechen wir von einem **lost update**), oder daten verwenden, die schon ungültig sind, oder einem cursor auf daten folgen, der von einer anderen transaction aus seiner gewünschten position gebracht wurde, oder die gleichen daten mehrmals lesen sollten, wobei diese beim zweiten lesen durch eine andere transaction geändert wurden (**non-repeatable read**), &c. Das **phantom-problem** ist das problem, dass der nutzer oder das anwendungsprogramm durch eine transaction informationen aus unterschiedlichen zeitpuncten bekommt, z.b. die einzelpreise von verschiedenen articeln vor einer rabattaction und gleichzeitig schon den gesamtpreis mit rabatt, wenn dieser von der transaction später gelesen wurde. All diese probleme (**anomalien**) treten nur im zusammenhang mit datenänderungen durch DML-befehle auf.

Daten, die eine transaction geändert hat, die aber noch nicht zur nutzung von anderen transactionen freigegeben sind, nennen wir schmutzig bzw. **dirty**. Operationen nennen wir **critisch**, wenn ihre durchführungsreihenfolge etwas an der ausgabe oder dem DB-endzustand ändert. In critischen operationen anderer transactionen dürfen keine daten benutzt werden, die dirty sind. Operationen mit schmutzigen daten nennen wir auch dirty (z.b. dirty read, dirty overwrite); diese sind auch anomalien im mehrbenutzerbetrieb.

Wir sagen, eine transaction liest von einer anderen, wenn sie ein datum liest, das die andere geschrieben hat, und die andere nicht vorher zurückgesetzt wurde.

`Serialisierbarkeit`

Zur nachträglichen prüfung einer von einem „**geschichtsschreiber** mitgezeichneten operationsanordnung bereits durchgeführter DML-operationen (**historie**; in form einer vereinigung mehrerer transactionen mit einer totalen oder partiellen ordnung für je ein- und mehrprocessorsysteme) auf consistenz testen wir, ob sie sich in eine serielle operationsfolge mit gleicher ausgabe und gleichem endzustand der DB umwandeln läßt - wenn ja, dann ist sie **conflictserialisierbar** (oder kurz einfach nur **serialisierbar**) und damit correct. Jede historie mit einer partiellen ordnung verfügt über mehrere topologische ordnungen, die mit der partiellen ordnung übereinstimmen. In echten DBS muss dies allerdings zur laufzeit bei noch nicht durchgeführten operationen von einem

scheduler bzw. **sperrverwalter** oder **lock manager** geprüft werden, dessen aufgabe es ist, einen serialisierbaren ablaufplan (wir sagen dazu **schedule**, allerdings werden die begriffe schedule und historie auch von manchen synonym benutzt) zu erstellen, indem nicht conflictfreie operationen (**conflictoperationen**) eine pausierung oder ein rollback ihrer transaction durch den scheduler verursachen. Schedules sind präfixe von historien. Wir nennen historien äquivalent, wenn sie conflictoperationen in der gleichen reihenfolge ausführen; conflictfreie operationen interessieren dabei nicht und müssen nicht in der ordnung specificiert werden. Geschichtsschreiber und scheduler kommen auf allen systemschichten (daten-, zugriffs- und speichersystem) zum einsatz.

Sei T_i eine transaction und A ein datum. Dann schreiben wir $r_i[A]$ für eine leseoperation in T_i auf A und analog $w_i[A]$ für eine schreiboperation, c_i für ein commit und a_i für einen abort.

Ein beispiel für eine historie ist:

$$H = \begin{array}{ccccccc} r_1[A] & \longrightarrow & r_1[B] & \longrightarrow & w_1[A] & \longrightarrow & c_1 \\ \downarrow & & & & \uparrow & & \\ w_2[A] & \longrightarrow & r_2[B] & \longrightarrow & r_2[A] & \longrightarrow & c_2 \end{array}$$

Der **conflictgraph** dazu ist: $T_1 \rightleftarrows T_2$

Topologische ordnungen dazu sind:
$H_S^1 = r_1[A] \to r_1[B] \to w_2[A] \to r_2[B] \to r_2[A] \to w_1[A] \to c_2 \to c_1$
$H_S^2 = r_1[A] \to w_2[A] \to r_1[B] \to r_2[B] \to r_2[A] \to w_1[A] \to c_2 \to c_1$
$H_S^3 = r_1[A] \to w_2[A] \to r_2[B] \to r_1[B] \to r_2[A] \to w_1[A] \to c_2 \to c_1$
$H_S^4 = r_1[A] \to w_2[A] \to r_2[B] \to r_2[A] \to r_1[B] \to w_1[A] \to c_2 \to c_1$
$H_S^5 = r_1[A] \to r_1[B] \to w_2[A] \to r_2[B] \to r_2[A] \to c_2 \to w_1[A] \to c_1$
$H_S^6 = r_1[A] \to w_2[A] \to r_1[B] \to r_2[B] \to r_2[A] \to c_2 \to w_1[A] \to c_1$
$H_S^7 = r_1[A] \to w_2[A] \to r_2[B] \to r_1[B] \to r_2[A] \to c_2 \to w_1[A] \to c_1$
$H_S^8 = r_1[A] \to w_2[A] \to r_2[B] \to r_2[A] \to r_1[B] \to c_2 \to w_1[A] \to c_1$
$H_S^9 = r_1[A] \to w_2[A] \to r_2[B] \to r_2[A] \to c_2 \to r_1[B] \to w_1[A] \to c_1$
(und weitere mit früherem c_1)

Eine historie ist genau dann serialisierbar, wenn ihr conflictgraph acyclisch ist. Obige historie H ist also nicht serialisierbar. Die classe aller conflictserialisierbaren historien nennen wir CSR oder SR. Für serialisierbare historien schreiben wir auch einfach z.b. $H_S^1 = T_1 \mid T_3 \mid T_2$ als topologische ordnung, wenn die zugehörige totale ordnung das gleiche ergebnis und die gleiche ausgabe hat wie ein serielles ausführen von T_1, T_3 und dann T_2 in dieser reihenfolge.

Wenn nach einer historie eine operation o_i vor einer anderen o_j ausgeführt werden muss, schreiben wir $o_i <_H o_j$. $o_i </_H o_j$ bedeutet $o_i \not<_H o_j$. $o_i <_i o_j$ schreiben wir für die ordnung innerhalb der i-ten transaction.

Probleme treten auf, wenn wir eine transaction mit einem abort abbrechen oder wenn

wir sie aufgrund eines fehlers abbrechen müssen.

- Zur behebung dieser probleme ist eine weitere anforderung an historien **rücksetzbarkeit**, d.h. bis zu einem commit müssen alle operationen einer transaction rückgängig gemacht werden können (da noch ein abort kommen könnte). Rücksetzbare historien sind solche, bei der eine transaction mit einer schreibe-operation, die mit einer leseoperation einer anderen transaction im conflict steht, immer vor der anderen transaction commitet. Sie bilden die classe RC (für „recoverable“).
- **Cascadierendes rücksetzen vermeidende historien**, d.h. solche, bei denen nach ausführung einer conflict-schreiboperation einer transaction diese transaction erst mit einem commit oder abort enden muss, bevor mit der operation im conflict stehende leseoperationen ausgeführt werden, bilden ACA („avoiding cascading aborts“).
- **Stricte historien**, d.h. solche, bei denen nach ausführung einer conflict-schreiboperation einer transaction diese transaction erst mit einem commit oder abort enden muss, bevor mit der operation im conflict stehende lese- *oder* schreiboperationen ausgeführt werden, bilden ST.
- Serielle historien sind eine teilmenge der stricten, serialisierbaren historien; sie bilden die menge S.

RX-Sperrverfahren

Ein lösungsansatz ist es, synchronisation paralleler transactionen durchzuführen, indem wir ressourcen sperren – d.h. andere sie benutzende transactionen pausieren –, solange sie benutzt werden. Wir weisen ressourcen einen sperrmodus zu: `NL` (no lock), `R` (read; für leseoperationen) oder `X` (exclusive; für schreiboperationen). Ein `R`-gesperrtes object kann nicht `X`-gesperrt werden; ein `X`-gesperrtes kann gar nicht gesperrt werden. Eine sperre schreiben wir als `lock(a,R)` bzw. `lock(a,X)` für eine ressource `a` und einen angeforderten sperrmodus `R` bzw. `X`. Das beenden der sperre schreiben wir als `unlock(A)`. Die actuellen sperren sind als **waits-for-graph** (WfG) darstellbar, z.b. $T_2 \xrightarrow{a} T_1$ für ein warten von T_2 auf T_1 wegen ressource a.

Trotz nutzung von sperren kann es aber immernoch zu problemen kommen, wenn manche änderungen an manchen datensätzen vor einer anderen transaction und andere nach der anderen durchgeführt werden. Wir müssen die sperren nach einer geeigneten methode einsetzen. Dazu gibt es **zwei-phasen-sperrprotocolle** bzw. **2PL-protocolle** (von **two-phase locking**). Vor jedem objectzugriff muss dabei eine sperre darauf angefordert worden sein, für ein object darf aber nicht mehrmals von der gleichen transaction eine sperranforderung gestellt werden. Spätestens beim commit müssen alle sperren wieder freigegeben worden sein. Die zweiphasigkeit, die 2PL-protocolle ausmacht, bedeutet, dass wir in einer wachstumsphase alle sperranforderungen stellen und in einer schrumpfungsphase alle sperren aufheben; zwischen den anforderungen und freigaben dürfen

auch operationen stehen.

- Bei **S2PL** (strict 2PL) werden alle `X`-sperren erst beim commit freigegeben.
- Beim **SS2PL** (strong S2PL) geben wir auch `R`-sperren erst beim commit frei, d.h. ohne operationen dazwischen werden am ende alle freigegeben.
- Beim **preclaiming** sperren wir am anfang außerdem noch alles, was wir benutzen, d.h. operationen werden erst nach der wachstums- und nur vor der schrumpfphase durchgeführt.

Bei diesem sperrverfahren kann es zu **verklemmungen** (**deadlocks**) kommen, wenn wir sperren in verschiedenen transactionen in unterschiedlicher reihenfolge anfordern.

Recovery

Nach einem crash muss zur erfüllung der ACID-criterien der alte DB-zustand nach allen schon abgeschlossenen transactionen wiederhergestellt werden. Dazu können wir transactionen wiederholen (**redo**), deren änderungen nur im hauptspeicher standen (also **transient** und noch nicht **permanent** waren), während wir schon permanent gespeicherte änderungen von transactionen, die durch den crash an ihrer verarbeitung gehindert wurden und nicht mehr ausgeführt werden können, zurücksetzen müssen (**undo**).

2PC

Es gibt auch DBS, die verteilt auf mehreren rechnern laufen. Das problem dabei ist, dass diese jeweils nur über einen teil des zur verarbeitung von befehlen nötigen wissens verfügen. Beim **zweiphasen-commit-protocoll** (**2PC-protocoll**) haben wir einen rechner zur centralen steuerung (den coordinator) und viele agenten, die jeweils teiltransactionen durchführen. Der commit geschieht in zwei phasen. In der ersten phase (dem voting) schickt der coordinator den agenten eine prepare-nachricht, woraufhin diese mit ready antworten, wenn sie ihre arbeit erfolgreich durchgeführt haben, während sie bei einem fehler mit einer failed-nachricht /no-vote-nachricht antworten. Der eigentliche commit (oder, bei einem fehler, abort) geschieht in der zweiten phase, der decision-phase, wo vom coordinator eine commit- bzw. abort-nachricht an alle agenten gesandt wird, auf die diese mit einer ACK-nachricht (acknowledge) antworten sollten. Das protocoll blockiert, wenn ein agent nicht antwortet, erst nach einer weile schickt der coordinator an diesen agenten wieder einen aufruf zum commit bzw. abort; außerdem ist ein ausfall des coordinators fatal. Logging geschieht bei allen rechnern synchron.

Capitel 7: XML

XML ist ein datenformat zur zur maschinellen verarbeitung geeigneten speicherung structurierter und semi-structurierter daten in form eines baumes. Die XML verarbeitenden programme (z.b. XML-parser) nennen wir **XML-processoren**. XML kann auch in einem DBS als ausgabeformat genutzt werden und es gibt eine eigene anfragesprache für XML-basierte DBS. Zu einem XML-document lassen sich metadaten structuriert speichern. Deren specification ist im gegensatz zu SQL-datenbanken nicht notwendig und dient nur zur angabe von integritätsbedingungen. Zur ausgabe und für anfragen nötige daten wie attributsnamen und der aufbau des documents werden als teil der daten specificiert und gespeichert. XML ist somit flexibler, da die schemadefinition auch nachträglich geschehen kann. Auch ist XML ein offener standard, der von vielen anwendungen auch tatsächlich so eingehalten wird; daten im XML-format sind somit nicht an die nutzung mit programmen bestimmter hersteller gebunden. Datenbanksysteme können XML auch intern (nativ) benutzen (XDBS = XML DBS) und es nicht nur als ausgabeformat verwenden, dies ist aber weniger efficient. Es kann ein einzelnes XML-document für die ganze datenbank verwendet werden oder mehrere kleine teildocumente.

XML basiert wie HTML auf der sprache **SGML** (Standard Generalized Markup Language) und hat zusätzliche, strengere regeln für die structur der daten. Dadurch sind XML-documente immer SGML-documente, aber nicht andersherum.

Syntax

Ein beispiel für ein XML-document:

```
<?xml version="1.0">
<countries>
 <country>
  <name>Japan</name>
  <population year="2012">126,659,683</population>
  <GDP year="2012" currency="yen">474,558,600,000,000</GDP>
 </country>
 <country>
  <name>Germany</name>
  <GDP currency="dollar">3.401 trillion</GDP>
  <timezone daylightsavings="yes"
   specified-for="winter">
    UTC+1
  </timezone>
 </country>
```

```
</countries>
```

Ein XML-document nennen wir **well-formed**, wenn es der XML-syntax genügt.

Bei XML beginnt jeder knoten (**element**) mit einem **start-tag** in <spitzen klammern> wie `<Person>` und endet mit einem passenden **ende-tag** wie `</Person>`, welcher sich nur im schrägstrich zu beginn vom start-tag unterscheidet. Dazwischen ist sein **content**, d.h. seine daten (als text) oder kindknoten oder beides (mixed content) gespeichert. Ein element ohne content, d.h. ein **leeres element**, können wir statt <name></name> auch äquivalent als <name/> schreiben.

Will man in seinem XML-document das zeichen `<`, `>` oder `&` benutzen ohne dass es als anfang eines tags verstanden wird, muss man es als referenz `&lt;`, `&gt;` bzw. `&amp;` schreiben. Alternativ kann man text auch „wörtlich“ als seine zeichen interpretieren lassen, indem man ihn in `<![CDATA[` und `]]>` einschließt (natürlich darf er kein `]]>` als teil des textes beinhalten).

Im start-tag, aber nicht im ende-tag, können auch attribute gespeichert werden. Dazu schreiben wir vor der schließenden spitzen klammer den attributnamen, ein `=`-zeichen, und den gewünschten wert in anführungszeichen. Der name muss für das element eindeutig sein. Attribute sollten nur als attribute und nicht als kindelemente gespeichert werden, wenn sie „metainformationen“ über die in elementen gespeicherten daten enthalten, z.b. wann eine information über die bevölkerungsdichte eines landes erfaßt wurde. Da attribute im ER- oder relationenmodell in XML oft als elemente dargestellt werden, lassen sich deren attribute nicht einfach im ER- und relationenmodell abbilden, weil diese keine attribute von attributen (aspecte) unterstützen. Das abbilden aller elemente des XML-documents als eigene tabelle im relationenmodell nennt sich **shredding** und ist umständlich.

Schemata

Für ein XML-document kann optional ein schema angegeben werden, wie es structuriert ist und sein muss. Das document nennen wir nur dann **gültig** bzw. **valid**, wenn es dem schema entspricht, d.h. nur elemente und attribute des schemas in der vorgegebenen form beinhaltet.

Eine möglichkeit, ein schema zu definieren, ist eine **document type definition** (**DTD**). Diese steht entweder am anfang des documents oder in einer gesonderten datei mit einer referenz darauf am anfang des documents.

Ein beispiel einer DTD:

```
<!DOCTYPE country_list [
 <!ELEMENT countries (country*)>
 <!ELEMENT country (name, GDP, population?, timezone?)>
```

```
 <!ELEMENT name (#PCDATA | short-name | official-name)*>
 <!ELEMENT short-name (#PCDATA)>
 <!ELEMENT official-name (#PCDATA)>
 <!ELEMENT GDP (ANY)>
 <!ATTLIST GDP currency(yen | euro | dollar) #REQUIRED>
 <!ELEMENT population (ANY)>
 <!ELEMENT timezone (#PCDATA)>
 <!ATTLIST timezone
  daylightsavings(yes|no) no
  specified-for(summer|winter) #FIXED winter
 >
]>
```

Diese folgt direct auf die angabe der XML-version.

- Wir beginnen mit `<!DOCTYPE` , einem namen für unser schema, und `[`.
- Dann folgen definitionen, was elemente mit einem bestimmten namen beinhalten müssen und dürfen. Diese bestehen jeweils aus `<!ELEMENT`, einem elementnamen, und in runden klammern einer liste der unterelemente, die das element genau einmal beinhalten muss – wir können die liste auch mit `|` zu einer auswahl machen, aus der genau ein element enthalten sein muss, und wir können listen in klammern zusammenfassen –, am ende steht eine schließende `>`. Ein `?` nach einem unterelementnamen bedeutet, dass es ein- oder keinmal vorkommen muss (statt genau einmal). Ein `+` bedeutet mindestens einmal, ein `*` heißt beliebig oft. Wird anstelle der liste `EMPTY` specificiert, darf das entsprechende element weder textdaten noch kinder haben. Bei `ANY` ist der inhalt beliebig. Anstelle einer unterelementliste in runden klammern kann auch `(#PCDATA)` angegeben werden, was bedeutet, dass das element textdaten beinhalten darf. Wir können auch mit `(#PCDATA |` elementname `| ...)*` eine liste von möglichen elementen specificieren, dabei dürfen aber nur reine elementnamen in einer einzelnen mit `|` getrennten auswahl stehen, ohne quantoren wie `?` oder `*` und ohne teillisten in klammern. Im beispiel auf den folien wurde `#PCDATA` anders benutzt; dies entspricht aber nicht dem XML-standard und nicht den übungen.
- Attribute specificieren wir in der gleichen liste wie elemente; sie beginnen allerdings mit `<!ATTLIST`, gefolgt vom namen des elements, und einer mit whitespace getrennten liste von attributsdefinitionen und einem abschließenden `>`. Attributsdefinitionen bestehen aus dem namen des attributs, dem attributstyp, und einer attributsdeclaration. Ein attributstyp ist bei uns entweder `CDATA` für beliebige strings, `ID` für den einen documentweit eindeutigen identificator des elements, `IDREF` für einen wert, der anderswo im document als `ID` vorkommt, oder eine in runden klammern stehende liste von möglichen string-werten. Eine attributsdeclaration ist entweder `#REQUIRED` für attribute, die das element haben muss, `#IMPLIED` für attribute ohne default-wert, oder einen default-wert mit optionalem

`#FIXED` davor (`#FIXED` heißt, dass das attribut den default-wert haben muss).

Eine DTD muss deterministisch sein, d.h. es muss beim schrittweisen durchgehen der elementdefinitionen eindeutig sein, was für ein teil der definition gerade durchlaufen wird. Anders ausgedrückt muss sich die DTD direct in einen deterministischen endlichen automaten umwandeln lassen, statt einem nichtdeterministischen endlichen automaten zu entsprechen, der erst noch umgebaut werden muss. (Siehe zum beispiel die lösung der aufgabe 3b auf blatt 11.

XML Schema

Eine complexere alternative zu DTDs ist die beschreibung eines XML-schemas mit dem passend benannten schema-definitionsstandard **XML Schema**. Dort können attributwerte auch mit datentypen und constraints angegeben werden. Wir können elementdefinitionen nur für einen namensraum gültig machen, um zum beispiel zwischen einem `preis` als kosten eines zum verkauf stehenden objects und einem `preis` als auszeichnung zu unterscheiden. Wir können eigene typen innerhalb von namensräumen definieren. Ein XML Schema ist selbst ein XML-document.

XPath

Anfragen in einer XML-basierten datenbank können in der sprache **XPath** gestellt werden. Dabei wird die anfrage mit einer sequenz aus einträgen (**items**) in XML-syntax beantwortet; items sind entweder **knoten** (**nodes**) im baum aus den XML-elementen, -attributen, -commentaren und weiterem, oder sie sind **atomare werte** (**atomic values**). Im document werden die elemente als in der reihenfolge ihrer position im document geordnet angesehen; attribute des selben elements sind untereinander ungeordnet.

Mit einem pfadausdruck wird die anfrage gestellt:

- Dieser kann mit einem einfachen `/` beginnen, der für die wurzel des documents steht. Ein mit diesem beginnender pfad wird relativ zur wurzel ausgewertet; ohne `/` beziehen wir uns stattdessen auf einen bestimmten contextknoten. Dies ist analog zu dateipfaden in Unix-systemen.
- Der rest des pfades ist eine durch `/` getrennte folge von achsenschritten und wird sequentiell ausgewertet. In einem achsenschritt geben wir zunächst eine achse an, dann einen knotentest und optional ein oder mehrere prädicate, jeweils in eckigen klammern, die alle die knoten filtern (es müssen also alle prädicate gelten, als wären sie mit einem logischen und verknüpft).
  - Die achse gibt an, welche knoten wir weiterhin in unserer anfrage auswerten und eventuell ausgeben wollen. Achsen, die wir kennen sollten, sind `child::` – für die knoten, die kinder des actuell ausgewerteten bezugsknoten sind –,

`descendant::` – für die knoten im teilbaum mit dem actuellen knoten als wurzel, die nicht die wurzel sind –, `descendant-or-self::` – für alle solchen knoten und den actuellen knoten –, `self::` – für den actuellen knoten –, `parent::` – für die elternknoten des actuellen knotens –, `ancestor::` – für alle knoten außer dem actuellen, in deren teilbaum der actuelle knoten vorkommt –, `ancestor-or-self::` – für alle solchen knoten und den actuellen knoten –, `preceding::` – für alle knoten, die in der ordnung des documents vor dem actuellen kommen *und* keine vorfahren sind –, `following::` – für alle, die danach kommen *und* keine nachkommen sind –, und `preceding-sibling::` und `following-sibling::` – für solche knoten, die auch noch jeweils geschwisterknoten des actuellen knotens sind. Siehe folie 39 für eine diagrammdarstellung. Außerdem gibt es die achse `attribute::` für die attribut-kindknoten des actuellen knotens. Es gibt noch weitere achsen wie `namespace::` für namensräume.

– Der knotentest filtert die resultierenden knoten. Geben wir `*` an, werden alle elemente (oder für die `attribute::`-achse stattdessen alle attribute; analoges gilt für andere achsen bestimmter knotentypen) der achse weiter verarbeitet. Geben wir statt `*` einen namen an, werden nur knoten weiter bearbeitet, die von `*` ausgewählt würden und den angegebenen namen haben. Geben wir `comment()`, `text()`, `processing-instruction()`, `node()` oder in XPath 2.0 auch `element()` an, wählen wir alle knoten des angegebenen typs aus (bei `node()` sind das alle knoten, also nicht nur die element- bzw. attributsknoten). `element(name)` steht für alle elementknoten des angegebenen namens.

– Ein optionales, in eckigen klammern stehendes prädicat ist wie bei selectionen in der relationalen algebra eine bedingung, die knoten außerdem erfüllen müssen. Dabei können wir die werte von knoten aus pfadausdrücken – bezogen auf den actuellen bezugsknoten oder mit `/` auf die wurzel – und atomare werte in vergleichen benutzen, z.b. vergleicht `[child::Preis < 100]` den im element `Preis` als text (`#PCDATA`) angegebenen wert mit dem atomaren wert 100. Es ist also möglich, XML-werte als zahlenwerte zu interpretieren. Wir können bei XPath auch `EQ` als vergleichsoperator benutzen; im gegensatz zu `=` vergleicht `EQ` werte und nicht den gesamten content. Als prädicat können wir alternativ auch nur eine zahl n angeben, um den n-ten knoten auszuwählen, z.b. steht der schritt `child::Gerät[2]` für das zweite kindelement mit namen `Gerät`. Bei einer rückwärtsachse ist dabei zu beachten, dass die knoten in der sequenz zwar nicht in umgekehrter documentenreihenfolge aufgeführt, aber in umgekehrter reihenfolge nummeriert werden. Wir können auch nur einen pfadausdruck als prädicat angeben; dieser ist wahr, wenn er am anfang der sequenz, die er auswählt, einen knoten hat, und falsch, wenn es eine leere sequenz ist. Wir führen damit also existenztests durch, z.b. wählt der schritt `child::*[child::GPU]` nur die kindelementknoten aus, die selbst ein kindelement namens `GPU` haben.

- An stelle eines achsenschritts kann immer auch ein functionsaufruf stehen. Wir kennen
  - `fn:string(`pfadausdruck`)`, was den wert des angegebenen pfadausdrucks als string zurückgibt, wenn der pfadausdruck nur einen einzigen knoten zurückliefert. Liefert er keinen knoten zurück wird der leere string zurückgegeben. Wird kein pfadausdruck angegeben (also nur `fn:string()`), so wird der wert des actuellen knotens ausgegeben.
  - `fn:id(`attribut`)` functioniert analog und gibt das durch das angegebene attribut identificierte element zurück (wie bei fremdschlüsseln in SQL).
  - `fn:doc("`dateiname`")` gibt die wurzel des XML-documents mit dem angegebenen dateinamen zurück.
  - `fn:distinct-values(`anfrage`)` gibt die von einander verschiedenen werte des resultats der anfrage zurück.
  - `fn:avg(`werte`)` ist der durchschnitt der werte.
  - `fn:not(`item`)` negiert die existenz von item.

  Das präfix `fn:` ist default und kann weggelassen werden.
- Wir haben auch einige abkürzungen:
  - `child::` wird implicit angenommen und kann weggelassen werden (als default).
  - Ein punct `.` als achsenschritt steht für den actuellen bezugsknoten.
  - Zwei puncte `..` stehen für `parent::node()`.
  - `@` steht für `attribute::`.
  - Zwei schrägstriche `//` statt einem stehen für `/descendant-or-self::node()/`.

Die prüfung einer anfrage auf correctheit nennen wir statische analyse.

XQuery

Anfragen kann man auch in der mehr functionen bietenden sprache **XQuery** stellen. XPath ist in der actuellen version eine teilmenge von XQuery, d.h. XPath-ausdrücke sind auch XQuery-ausdrücke. Hybride XML- und SQL-datenbanken können SQL-anfragen mit XQuery combinieren.

Eine XQuery-anfrage schreiben wir wie ein XML-document, nur dass wir darin auch XQuery-ausdrücke in geschweiften klammern verwenden können. Die anfrage wird dann

mit dem XML-document beantwortet, dass man aus dem angegebenen XML-code erhält, wenn man alle geschweift geklammerten XQuery-ausdrücke darin durch ihren wert ersetzt.

Eine art solcher XQuery-ausdrücke sind **FLWOR-ausdrücke** (gesprochen wie englisch „flower“). Die bezeichnung FLWOR steht für die bestandteile eines solchen ausdrucks:

- Ein FLWOR-ausdruck beginnt mit einer oder mehreren `for`- oder `let`-clauseln, welche darauf folgenden variablennamen innerhalb des FLWOR-ausdrucks gültige werte zuweisen. Variablennamen beginnen stets mit einem `$`-zeichen.
  - Bei einer `for`-clausel kommt nach dem `for` eine commagetrennte liste von variablennamen mit je einem darauf folgenden `in` mit einer XQuery-anfrage – die auch ein pfadausdruck sein kann –, für deren zurückgegebene sequenz die variablen einmal jede combination aus werten annehmen und der restliche FLWOR-ausdruck für jede solchen variablenbelegung einmal ausgewertet wird und selbst etwas zurückgibt. Dies kann ähnlich zu den joins der relationalen algebra benutzt werden.
  - Bei `let` haben wir eine ähnliche syntax, nur dass statt `in` die zeichen `:=` benutzt werden und die variable nur einmal an genau die rückgabe des teilausdrucks gebunden wird.
- Mit einer optionalen `where`-clausel selectieren wir daraus nur die variablenzuweisungen, wo das auf ein `where` folgende prädicat wahr ist. Im prädicat können natürlich alle variablen benutzt werden, die mit den vorherigen `for`s und `let`s definiert wurden oder aus höher geschachtelten FLWOR-ausdrücken kommen.
- Ein optionales `order by` kann benutzt werden, um die zurückzugebenden werte umzuordnen. Bei uns folgt dabei auf ein `order by` immer ein variablenname.
- Die `return`-clausel am ende gibt an, was für jede variablenbelegung zurückgegeben wird. Mit verschachtelten FLWOR-ausdrücken in der `return`-clausel lassen sich z.b. left outer joins simulieren.

Capitel 8: Informationssuche im Web

Eine suchmaschine, die das Web nach für nutzer interessanten informationen durchsucht, muss eine riesige und rasant wachsende menge an wissen verwalten. Zudem sind unsere daten im Web meist unstructuriert, d.h. maschinenunverständlich, und sie liegen oft nicht einmal als textdateien sondern z.b. als PDF- oder office-document vor. Die hier verwendeten informationssysteme nennen wir **IRS** (**information retrieval systems**); diese speichern ganze dateien als paar aus einem documenttyp und dem wert, d.h. dem dateiinhalt.

Ein perfectes IRS würde bei suchanfragen alle relevanten und nur relevante dateien finden (retrieven). Dazu muss eine **unscharfe** suche durchgeführt werden, d.h. eine suche, bei der ergebnisse nicht eindeutig relevant sind oder nicht, sondern eine möglichst gute bewertung (**ranking**) bezüglich der suchanfrage zugewiesen bekommen. Mehrdeutige wörter müssen berücksichtigt werden; **ontologien** sind hierzu benutzte sammlungen von informationen zur bedeutung von wörtern. Für synonyme wird ein **thesaurus** benutzt, der diesen jeweils ein canonisches synonym zuweist. Der thesaurus beinhaltet in einem gut gepflegten IRS unter umständen sogar alle in den daten verwendeten möglicherweise relevanten wörter. Documente werden bei ihrer speicherung im IRS vorverarbeitet und in eine documentrepräsentation gebracht, in der z.b. für das ranking irrelevante wörter wie „ist“ entfernt sind und alle wörter in ihre grundform gebracht wurden. Im wortschatz des IRS enthaltene und zur suche verwendete schlüsselwörter nennen wir **descriptoren**.

Die qualität von ergebnissen einer IRS-suche können wir mit folgenden begriffen messen: **precision** ist der anteil $P = \frac{|r \cap g|}{|g|}$ relevanter gefundener ergebnisse unter allen gefundenen documenten, während **recall** der anteil $R = \frac{|r \cap g|}{|r|}$ gefundener relevanter documente unter allen relevanten ist. Wir sollten möglichst einen hohen wert für eines davon oder sogar beide haben.

Bei web-suchmaschinen reichen precision und recall als qualitätsmaße nicht aus; wir müssen auch die qualität der seite bezüglich efficienzcriterien wie der antwortzeit berücksichtigen (zugreifbarkeit), ebenso wie die interpretierbarkeit (d.h. verständlichkeit) durch menschliche benutzer, nützlichkeit (was ungefähr der precision entspricht) und glaubwürdigkeit. Eine liste von von menschen ausgewählten nützlichen links zu einem thema kommt suchmaschinen dabei gelesen; so eine liste nennen wir **catalog**. Nicht mehr verfügbare links sollten dabei nicht gefunden werden.

Vom Google-mitbegründer Larry Page wurde z.b. das page-rank-verfahren entwickelt, welches die relevanz von Web-seiten anhand der anzahl der auf sie verweisenden links und der auf ihnen befindlichen links bestimmter „qualität“ beurteilt. PageRank basiert auf dem HITS-algorithmus, welcher wiederholt seiten bewertet, bis die qualität einer seite relativ constant bleibt.

Eine metasuchmaschine ist eine suchmaschine, die suchanfragen an mehrere andere suchmaschinen weitergibt und deren ergebnisse zusammenfaßt und dem nutzer präsentiert.

Es gibt auch SQL-ähnliche anfragesprachen für die suche im internet.