Stephan Brumme, SST, 3.FS, Matrikelnr. 70 25 44
mail@stephan-brumme.com

Aufgabe 1

DBMS: DataBase Management System
- Software, die für die technische Verwaltung der Datenbanken, die entsprechend dem Data Dictionary (DD) aufgebaut sind, zuständig ist
- Verbindungsstelle zur Anwendungssoftware, Bereitstellung einer abstrakten Schnittstelle
- Abschirmung der Datenbanken nach außen, d.h. Sicherung der Datenunversehrtheit

DD: Data Dictionary
- enthält die Datenbankschemata, die die Struktur und Konsistenzbedingungen der zugrundeliegenden Datenbanken bestimmen
- jedes DBMS enthält min. 1 Data Dictionary

DDL: Data Definition Language
- Beschreibungssprache, die den Aufbau des DD festlegt bzw. ändert
- ist Bestandteil von SQL, d.h. nur für relationale Datenbanken geeignet

ODL: Object Definition Language
- objektorientiertes Gegenstück zu DDL
- keine feste Syntax, da jede nach Programmiersprache, in der sie eingebettet ist, verschieden

SQL: Structured Query Language
- relationale Datenbanksprache (sogenannte 4^{th}GL), die Definitionssprache (DDL) und Manipulationssprache (DML) umfasst
- weitgehend standardisierte Syntax (SQL92 mit verschiedenen Levels)
- keine direkte Anbindung an Programmiersprachen
- Offenlegung aller Daten für alle Befehle, lediglich geringe Rechteverwaltung möglich
- Datenstrukturen beschränken sich auf 2-dimensionale Strukturen, d.h. Tabellen
- schnell und einfach zu erlernen, sehr starke Verbreitung

OQL: Object Query Language
- objektorientiertes Gegenstück zu SQL
- kann nur Daten abfragen, nicht verändern, oft erweitert durch vordefinierte Abfragen
- einfache Einbettung in Programmiersprachen, wie C++, Java, etc.
- weniger mächtig als SQL

Aufgabe 2

a) Die Wertidentität hängt von den Attributwerten ab. Sind zwei Tupel in einer relationalen Datenbank in all ihren Attributen und ihrem Schlüssel identisch, so sieht sie das System als identisch an.
Bei Objektidentität sind diese beiden Tupel noch unterscheidbar, da sie jeweils über einen systemweit einzigartigen Identifier verfügen. In einer objektorientierten Datenbank sind zwei Objekte nur genau dann identisch, wenn sie über den gleichen Identifier verfügen. Deshalb muss zusätzlich zur Identität der Begriff der Gleichheit eingeführt werden, die der Identität bei relationalen Datenbanken entspricht und sich allein nach den Attributen/Schlüsseln richtet.

b)

OID	Vorname	Name	(Bag) alteNamen	Geburtsdatum
2115498843218736	Eva	Sperber	Specht, Zeisig, Fink	30.06.1955

c) Die Objekt-Identität besitzt alle Eigenschaften eines Primärschlüssels. Sie kann daher den Primärschlüssel von relationalen Datenbanken ersetzen. Da sie aber systemgestützt generiert wird, kann der Benutzer eigene, zusätzliche Primärschlüssel definieren, wie z.B. Kunden-Nr. etc., die ebenfalls ein-eindeutig sind und das Tupel identifizieren können.

Aufgabe 3

kartesisches Produkt RxS:
`SELECT * FROM R,S`

A	B	C	D	E
1	2	3	3	1
1	2	3	6	2
1	2	3	1	7
4	5	6	3	1
4	5	6	6	2
4	5	6	1	7
7	8	9	3	1
7	8	9	6	2
7	8	9	1	7

`SELECT a,b,c,d FROM R,S WHERE R.c=T.c`

A	B	C	D
1	2	3	2
4	5	6	4

`SELECT * FROM R,S WHERE R.c = S.d`

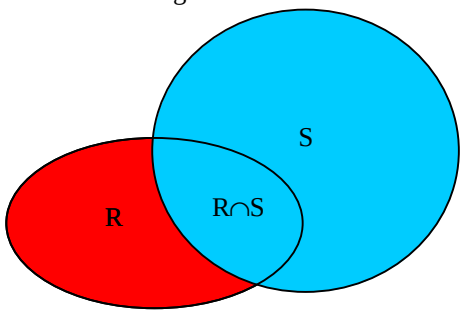
A	B	C	D	E
1	2	3	3	1
4	5	6	6	2

Ich beschränke mich aus Platzgründen auf diesen einen Θ-Verbund.

Sei #X die Anzahl Zeilen pro Tabelle. Dann umfasst das kartesische Produkt #P=#A · #B Zeilen.
Für die Tabellen R und T entstehen 3·3=9 Zeilen (mit jeweils 5 Attributen), für die zwei großen Tabellen sind es schon 1.200·150=180.000 Zeilen. Selbst wenn jede nur einen einzelnen 32-Bit-Integer speichern soll, hat man damit fast 1 MByte Speicherverbrauch.

Aufgabe 4

Am besten wird die Gleichung in einem Venn-Diagramm deutlich:



Der Ausdruck $R \cap S$ ist genau die Fläche, die sowohl von R als auch von S überdeckt wird. Den dazu äquivalenten Ausdruck $R-(R-S)$ zerlege ich in zwei Teile: $R-S$ ist die rein rote Fläche. Wenn man diese Fläche von der gesamten roten Fläche subtrahiert, so bleibt nur der Teil übrig, der von R und S überdeckt wird (d.h. $R \cap S$).

Der `Intersect`-Befehl ist nicht im SQL89-Standard enthalten und wurde erst in SQL92 implementiert. Nicht alle Systeme unterstützen ihn, u.a. kann auch Access nicht damit umgehen.
Eine Möglichkeit, `Intersect` zu emulieren, besteht in der Verwendung der eben gezeigten Gleichung.
Schwierig wird es nur, wenn das DBMS keine Sub-Selects vernünftig unterstützt, wie es bei MySQL der Fall ist.

```
SELECT PersonenID FROM Kunde
INTERSECT
SELECT PersonenID FROM Mitarbeiter;
```

wird zu:

```
SELECT PersonenID FROM Kunde, Mitarbeiter
WHERE Kunde.PersonenID = Mitarbeiter.PersonenID;
```

Die `Union` kann noch einfacher in SQL umgesetzt werden:

```
SELECT PersonenID FROM Kunde, Mitarbeiter
```

Die Differenz oder `Minus` ist etwas aufwändiger:

```
SELECT PersonenID FROM Kunde AS k
WHERE NOT EXISTS
      (SELECT * FROM Mitarbeiter
       WHERE k.PersonenID = Mitarbeiter.PersonenID);
```

Aufgabe 5

Das Hauptproblem besteht darin, dass man Daten redundant auf mehreren Rechnern hat. Wenn ein Datum geändert werden soll, so muss dies auf allen Rechnern (und möglichst synchron) erfolgen. Hält man sich nicht daran, dann hat man inkonsistente Datensätze und kann u.U. nicht nachverfolgen, welche korrekt sind und welche aktualisiert werden müssen.

`SELECT`-Operationen sind problemlos durchführbar, da sie keine Daten ändern, sondern nur miteinander verknüpfen. Schwierig wird es bei `UPDATE`, `INSERT` usw.. Sie sollten atomar sein, um parallel laufende Abfragen nicht zu beeinflussen. Diese Atomarität kann mit Semaphoren hergestellt werden, die die Datenbank kurzzeitig verriegeln (database lock) und die alleinige Kontrolle übernehmen.
Schwieriger als die Zugriffssperre einer einzelnen Datenbank ist die globale Aktualisierung aller Rechner, die die gleiche Datenbank verteilt gespeichert haben. Zwar ist es technisch möglich, dies sofort und überall durchzuführen, allerdings leidet die Performance erheblich. Hier muss man dann auf ähnliche Verfahren wie bei Multiprozessor-Caches zurückgreifen (ähnlich wie MESI-Protokoll ?).

Die Abwägung zwischen Performance und Sicherheit ist so schwierig, dass es keine perfekte Lösung gibt. Dies kann man immer wieder an den Problemen großer Internet-Firmen, meist sogenannte Free-Mailer, sehen.

Aufgabe 6

Die Reihenfolge der Abfragekriterien spielt bei der Datenbankperformance eine entscheidende Rolle. Das Grundprinzip lautet, dass man alle temporär erzeugten Tabellen bzw. Views möglichst klein halten sollte. Grundsätzlich vergrößern `Join`s die Tabellen, während `Select`s sie verkleinern. Als Schlussfolgerung sollte jedes `Select` möglichst früh erfolgen, jeder `Join` dagegen zum spätest möglichen Zeitpunkt eingesetzt werden. Diese allgemeine Aussage muss für den speziellen Einzelfall angepasst werden, da auch die Reihenfolge der `Join`s bzw. `Select`s untereinander wichtig ist. Hier ist die Grundregel, dass stets die `Join`-Operation zuerst erfolgen sollte, die die Ergebnismenge am stärksten einschränkt, genauso sollte die `Select`-Operation zuerst erfolgen, die die Ergebnismenge möglichst wenig aufbläht.
Diese Regeln basieren im wesentlichen auf der Anzahl der notwendigen Vergleiche für `Select`s und dem nur beschränkt verfügbaren Arbeitsspeicher sowie der Effizienz der angeschlossenen Cache-Systeme.

Projektionen wirken sich nicht auf die später erforderliche Anzahl von Vergleichen nachfolgender Befehle aus, sie belegen lediglich Speicherplatz und vermindern die Effizienz der Cache-Algorithmen. Daraus ergibt sich die Regel, dass man Projektionen auch möglichst früh durchführen sollte (wie Joins).

Aufgabe 7

Die SQL-Strings sind ebenfalls in der Seminarverwaltungs-Datenbank enthalten. Ich werde deshalb die SQL-Strings so verwenden, wie sie auch in Access tatsächlich funktionieren. Die in Access nicht implementierten Features (`Minus` etc.), die den Befehle teilweise erheblich verkürzen würden, benutze ich nicht.

a) Als Datum habe ich den heutigen Tag gewählt (16. Dezember 2000), dieser Wert muss natürlich später entsprechend angepasst werden. Die Datenbasis bildet die Tabelle `Person`, die sowohl alle `Kunden`, als auch alle `Dozenten` enthält. Nicht berücksichtigt wird dabei, dass eventuell Einträge in `Person` existieren, die weder einem `Kunden` noch einem `Dozenten` zugeordnet werden können. Diese Einträge wären aber überflüssig und würden in `Person` eine gewisse Redundanz erzeugen:

```
SELECT * FROM Person WHERE (((Person.Geburtsdatum)<#16/12/1950#));
```

Die nächste Aufgabenstellung ist schon etwas schwieriger. Das Problem liegt im wesentlichen darin, dass die Vergleichskriterien (jeweils die `Personen ID`) in den Tabellen `Dozent` und `Kunde` zu finden sind, es jedoch die dazugehörigen Persondaten zurückgeliefert werden sollen. Diese wiederum muss man aus der Tabelle `Person` extrahieren. Ich erzeuge deshalb erst eine temporäre View, die alle `Personen ID` von Kunden, die gleichzeitig auch Dozenten sind, findet. Dann durchsucht eine zweite Abfrage die Tabelle `Person` und übernimmt alle Einträge, für die in der View die entsprechende `Personen ID` existiert, in die Zieltabelle, die dem Nutzer dann angezeigt wird.
Um redundante Einträge im Ergebnis zu eliminieren ist das Schlüsselwort `DISTINCTROW` wichtig, das für eine Eindeutigkeit hinsichtlich der Wertidentität sorgt, d.h. wertgleiche Tupel aussortiert und nur einfach in der Ergebnisliste auftauchen lässt.

```
SELECT DISTINCTROW * FROM Person WHERE (((Person.[Personen ID]) In
   (SELECT DISTINCTROW Dozent.[Personen ID] FROM Dozent, Kunde
    WHERE (Dozent.[Personen ID]=Kunde.[Personen ID])));
```

Der jetzt folgende SQL-Befehl ist der komplexeste. Der Grund liegt einfach darin, dass kein `Minus` in Access existiert. Mein Lösungsansatz beruht darin, dass ich die Menge bestimme, die durch ein Minus aus den Zwischentabellen herausfallen würde, und dann den Befehl `Not In` anwende:

```
SELECT DISTINCTROW * FROM Person WHERE (((Person.[Personen ID]) Not In
   (SELECT DISTINCTROW Kunde.[Personen ID] FROM Dozent, Kunde
    WHERE (Dozent.[Personen ID]=Kunde.[Personen ID])) And
          (Person.[Personen ID]) In
         (SELECT DISTINCTROW Kunde.[Personen ID] FROM Person, Kunde
          WHERE (Person.[Personen ID]=Kunde.[Personen ID])));
```

Die letzte Abfrage funktioniert fast genauso wie die vorherige, ich kommentiere sie daher nicht weiter:

```
SELECT DISTINCTROW * FROM Person WHERE (((Person.[Personen ID]) Not In
   (SELECT Kunde.[Personen ID] FROM Kunde, bucht
    WHERE (Kunde.[Personen ID] = bucht.[Personen ID])) And
          (Person.[Personen ID]) In
         (SELECT DISTINCTROW Kunde.[Personen ID] FROM Person, Kunde
          WHERE (Person.[Personen ID]=Kunde.[Personen ID])));
```

b) Als aktuellen Datum habe jeweils den heutigen Tag, es ist der 16. Dezember 2000, genommen.
Die Erkennung von Minimal-/Maximalteilnehmerzahl-Vertauschungen ist relativ einfach:

```
SELECT Count(*) FROM Seminarveranstaltung
  WHERE (((Seminarveranstaltung.MinTeilnehmer)>[MaxTeilnehmer]) AND
         ((Seminarveranstaltung.bis)>#16/12/2000#));
```

Ähnlich verhält es sich mit Überbuchungen, hier sind lediglich ein paar Relationszeichen bzw. Vergleichsattribute anders:

```
SELECT Count(*) FROM Seminarveranstaltung
  WHERE (((Seminarveranstaltung.AktuellTeilnehmer)>[MaxTeilnehmer]) AND
         ((Seminarveranstaltung.bis)>#16/12/2000#));
```

Die noch nicht voll belegten Seminare folgen dem gleichen Schema:

```
SELECT Count(*) FROM Seminarveranstaltung
  WHERE (((Seminarveranstaltung.AktuellTeilnehmer)<[MaxTeilnehmer]) AND
         ((Seminarveranstaltung.bis)>#16/12/2000#));
```

Ich habe auch versucht, die letzte Teilaufgabe zu lösen, scheiterte jedoch frustriert. Die dazu notwendige SQL-Abfrage schließt zu viele Tabellen ein, so dass sie mir zu komplex und unübersichtlich wurde.

c) Rein technisch ist ein Union auf `Kunde` und `Dozent` möglich. Es ergibt sich aber eine inhaltliche Inkonsistenz, da der Umsatz als Kunde nicht unbedingt mit dem des Dozenten übereinstimmt. Selbst wenn dies anfänglich noch in Griff zu bekommen wäre, so besteht immer die Gefahr, das die Integrität im Laufe der Zeit verletzt wird. Eine Umgestaltung der Datenbank derart, dass sie vollkommen der Dritten Normalform genügt, schafft dieses Problem aus der Welt.

Aufgabe 8 (Extra 1)

a) Die erste Normalform erfordert, dass jedes Attribut atomar ist. Da aber nicht festgelegt wird, wieviele Betreuer einer Seminarveranstaltung zugeordnet werden, ist dies nicht gegeben.

b) Zur Auflösung dieser Problematik ist eine neue Tabelle erforderlich, die ich „Betreut“ nenne, „Bucht“ hat damit ein Attribut weniger:

Bucht	S#	Termin	K#	Preis	Hotel	Raum	max. Teilnehmerzahl

Betreut	S#	Termin	Betreuer

c) Die zweite Normalform ist genau dann erfüllt, wenn jedes Nicht-Schlüsselattribut *voll funktional* vom Primärschlüssel abhängt. Zusätzlich muss sich die Tabelle bereits in der ersten Normalform befinden. In der Beispielaufgabe hängen das Hotel, die Raumnummer und die max. Teilnehmerzahl nur von der Seminarnummer S# und dem Termin, nicht aber von der Kundennummer K# ab. Somit ist die voll funktionale Abhängigkeit nicht gegeben, es muss erneut eine weitere Tabelle erstellt werden, die „FindetStatt“ genannt wird, „Bucht“ verkleinert sich enorm:

Bucht	S#	Termin	K#	Preis

Betreut	S#	Termin	Betreuer

Findet statt	S#	Termin	Hotel	Raum	max. Teilnehmerzahl

d) Die dritte Normalform verlangt, dass die Datenbank in zweiter Normalform vorliegt und jedes Nicht-Schlüsselattribut nicht transitiv, d.h. direkt vom Primärschlüssel abhängt.
Die max. Teilnehmerzahl wird bereits durch das Hotel und den Raum bestimmt. Dies verletzt aber die Bedingungen der dritten Normalform. Abhilfe schafft eine neue Tabelle namens „Kapazität“:

Bucht	S#	Termin	K#	Preis

Betreut	S#	Termin	Betreuer

Findet statt	S#	Termin	Hotel	Raum

Kapazität	Hotel	Raum	max. Teilnehmerzahl

e) Zu jedem Schlüsselpaar (S#,Termin) aus „Bucht“ und „Betreut“ muss ein entsprechender Schlüssel in „FindetStatt“ existieren.
Jedem Paar (Hotel, Raum) aus „FindetStatt“ ist eine maximale Teilnehmerzahl in „Kapazität“ zugeordnet.

(Bucht.S#, Bucht.Termin) $\subseteq$ (FindetStatt.S#, FindetStatt.Termin)
(Betreut.S#, Betreut.Termin) $\subseteq$ (FindetStatt.S#, FindetStatt.Termin)
(FindetStatt.Hotel, FindetStatt.Raum) $\subseteq$ (Kapazität.Hotel, Kapazität.Raum)

Aufgabe 9 (Extra 3)

Material	Mat.-Nr.	Mat.Bezeichnung	Firmenname	Anschrift	Lief. -Nr.	Bestell-Nr.	Verp.-Nr.	Preis
	12	HD-Diskette	Infotrade	...	8	5235	10	12,50
	12	HD-Diskette	Softdisk	...	10	55366	10	12,00
	10	DD-Diskette	Infotrade	...	8	8755	10	10,00
	...	...	...	...	...	...	...	...

a) Die Tabelle liegt in der ersten Normalform vor. Alle Attribute sind atomar, allerdings hängen einige Nicht-Schlüsselattribute nur von einem Teil des Primärschlüssels ab, z.B. die Materialbezeichnung nur von der Materialnummer, nicht aber von der Lieferantennummer.

b) Die Abhängigkeiten veranschauliche ich in der Form {A}→ {B} dar, was bedeutet, dass B funktional von A abhängt. Der Primärschlüssel ist grau markiert, dabei weist hellgrau auf partielle Attribute davon hin, dunkelgrau ist der vollständige:

{Materialnummer} → {Materialbezeichnung}
{Lieferantennummer} → {Firmenname}
{Lieferantennummer} → {Anschrift}
{Materialnummer, Lieferantennummer} → {Bestellnummer}
{Bestellnummer} → {Verpackungsnummer}
{Bestellnummer} → {Preis}

c) Zuerst stelle ich die zweite Normalform her, indem ich die in Aufgabe b) hellgrau schattierten Beziehungen in neue Tabellen ausgliedere:

Material	Mat. -Nr.	Lief. -Nr.	Bestell-Nr.	Verp.-Nr.	Preis
	12	8	5235	10	12,50
	12	10	55366	10	12,00
	10	8	8755	10	10,00
	...	...	...	...	...

Materialdetails	Materialnummer	Materialbezeichnung
	10	DD-Diskette
	12	HD-Diskette
	...	...

Lieferant	Lieferantennummer	Firmenname	Anschrift
	8	Infotrade	...
	10	Softdisk	...
	...	...	...

Die dritte Normalform bedingt, dass die Verpackungsnummer und der Preis, die beide funktional von der Bestellnummer abhängen, in eine separate Tabelle verschoben werden, die beiden Tabellen *Materialdetails* und *Lieferant* bleiben unverändert:

Material	Materialnummer	Lieferantennummer	Bestellnummer
	12	8	5235
	12	10	55366
	10	8	8755
	...	...	...

Bestellung	Bestellnummer	Verpackungsnr.	Preis
	5235	10	12,50
	55366	10	12,00
	8755	10	10,00
	...	...	...

Aufgabe 10 (Extra 4)

Veranstaltung	V#	Titel	Datum
	1	OO-Testen	11.11.01
	2	Einführung	02.12.02

Teilnehmer	T#	Anrede	Titel	Vorname	Nachname
	13	Herr	Dr.	Heinz	Müller
	10	Frau		Sabine	Zöllner

NimmtTeil	T#	V#
	13	1
	10	2
	13	2

a) Es muss lediglich jeder Eintrag der Tabelle *Teilnehmer* auf das Attribut Titel untersucht werden:

```
SELECT * FROM Teilnehmer WHERE Titel = „Dr.“;
```

b) Da eine Veranstaltung mehrmals stattfinden kann, taucht derselbe Titel u.U. mehrfach auf. Dies umgeht man durch das Schlüsselwort `DISTINCT` (manchmal auch `DISTINCTROW` genannt):

```
SELECT DISTINCT Titel FROM Veranstaltung;
```

c) Das `SELECT` verknüpft zwei Tabellen, ansonsten sind keine Besonderheiten zu beachten. Es ist eventuell die Reihenfolge der `WHERE`-Bedingungen zu überlegen, da der Grundsatz gilt, dass man möglich zuerst die Bedingung anwendet, die die Ergebnistabelle am stärksten einschränkt. Die gegebenen Daten würden zwar eine umgekehrte Reihenfolge suggerieren, ich bin aber der Ansicht, dass in einer großen Datenbank es mehr Teilnehmer für eine Veranstaltung gibt, als ein einzelner Teilnehmer Veranstaltungen besucht:

```
SELECT Teilnehmer.* FROM Teilnehmer, NimmtTeil
   WHERE Teilnehmer.T# = NimmtTeil.T#
     AND NimmtTeil.V#  = 1;
```

d) Das `SELECT` umfasst jetzt alle drei Tabellen.

```
SELECT Teilnehmer.* FROM Veranstaltung, Teilnehmer, NimmtTeil
   WHERE Veranstaltung.Titel = “OO-Testen“
     AND Veranstaltung.V#    = NimmtTeil.V#
     AND Teilnehmer.T#       = NimmtTeil.T#;
```

Es gibt verschiedene Wege, wie man dieses SQL-Statement optimieren könnte. Zuerst ergibt eine Betrachtung der Größe der temporär als kartesisches Produkt erzeugten Tabelle, dass sie $2 \cdot 2 \cdot 3 = 12$ Einträge mit jeweils $3 + 5 + 2 = 10$ Attributen umfasst.
Eine Variante wäre, das kartesische Produkt *Veranstaltung*×*NimmtTeil* zu erzeugen, daraus alle Teilnehmernummern von „OO-Testen“ zu extrahieren und diese gewonnene Information auf die Tabelle *Teilnehmer* anzuwenden. Das DBMS müsste jetzt zwei temporäre Tabellen erzeugen, eine mit $2 \cdot 3 = 8$ Einträgen und $2 + 3 = 5$ Attributen, die andere mit $1 \cdot 2 = 2$ Einträgen und $2 + 5 = 7$ Attributen.
Ein anderer Ansatz beruht darauf, dass man die Reihenfolge der Bedingungen optimiert. Der angegebene SQL-String scheint mir aber in dieser Hinsicht keinerlei ungenutzte Reserven zu bieten.