IDB SS23 Solutions & Notes

Igor Dimitrov Jacob Rose Jonathan Barthelmes

Table of contents

IDB SS 23 Loesungen & Notizen

IDB SS 23 Loesungen & Notizen

Part I

Loesungen

1 Grundlagen der Relationalen Algebra

1.1 Grundlagen der Logik

a)

1. Wenn zwei Tiere im selben Lebensraum leben, essen sie auch das selbe.

   falsch: t_1 (Gepard) und t_9 (Uganda-Grasantilope) haben den gleichen lebensraum "Regenwald" aber andere Ernaehrungen; Karnivore und bzw. Herbivore.

   erfuellbar: Jede Datenbank, die ein einziges Tier enthaelt erfuellt diese Aussage automatisch.

2. Fuer jedes Zootier existier ein anderes Zootier, welches entweder die selbe Nahrung isst oder im selben Lebensraum lebt. ***wahr***

3. Es existieren drei Zootiere, so dass erstes und zweites ,sowie zweites und drittes den gleichen Lebensraum teilen aber erstes und drittes nicht.

   falsch und nicht erfuellbar: $l(x,y)$ ist eine Aequivalenzrelation. Somit gilt Transitivitaet: $l(x,y) \wedge l(y,z) \rightarrow l(x,z)$

4. Es gibt keine zwei unterschiedliche Tiere, die sowohl der gleichen Familie zugehoerig sind als auch den gleichen Lebensraum teilen.

   falsch: t_6 und t_{10} sind beide Sakiaffen mit dem Lebensraum Regenwald.

   erfuellbar: Jede Datenbank mit einem einzigen Element efuellt diese Aussage automatisch.

b)

1. $\forall x \in T \exists y \in T : x \neq y \wedge fam(x,y) \wedge \neg ls(x,y)$
2. $\forall x \in T \forall y \in T : fam(x,y) \wedge le(x,y) \wedge er(x,y) \rightarrow x = y$
3. $\forall x \in T \forall y \in T : fam(x,y) \rightarrow er(x,y)$

1.2 Relationale Algebra

1. Gebe die Modelle von Flugzeugen, die so heissen, wie einer aus dem Personal.

Modell
Quack

2. Gebe die crew ID der Mitarbeiter, die nicht an den afugelisteten Fluegen beteiligt sind.

cid
c090

3. Gebe die Flugnummer der Fluege, die in Deutschland starten.

Flugnr
DB2013
DB2341

4. Gebe alle Modelle aus, fuer die eine Crew-Mitglied zugelassen ist.

Zulassung
A320
B787
A380
A340
B747

5. Gebe alle Namen von Piloten aus, die fuer eine Maschine zugelassen sind mit Reichweite ≤ 10000.

Name
Pan
Schmitt

6. Gebe Start, Ziel und Modell fuer alle Modelle aus, die ungeiegnet fuer einen Flug sind, weil sie die Strecke nicht fliegen koennen.

Start	**Ziel**	**Modell**
FRA	JFK	A320
JFK	FRA	A320
CDG	LAX	A320

7. Waehle aus Fluegen die gleichen Flugnummern, die an unterschiedlichen Tagen fliegen, d.h gebe Flugnummern der Fluegen, die Rundfahrten sind.

Flugnr
DB2013

8. Gebe die Laender aus, aus denen keine Flugzeuge starten.

Land
Deutschland

1.3 Datenmanagementsysteme

a) XML und HTML basieren sich beide auf **SGML** - eine Metasprache, mit deren Hilfe man verschiedene Markup-sprachen fuer Dokumente definieren kann.

XML ist eine erweiterbare Markup-sprache, die zur Darstellung & Speicherung hierarchisch strukturierter Daten und zur Definition & Entwicklung neuer Markup-sprachen verwendet wird. XML Dokumente haben eine Baumstruktur und bestehen aus **Elemente**, die durch **Tags** Ausgezeichnet werden. XML hat keinen vordifienerten Satz von Tags, wobei die genaue Struktur eines XML-Dokuments durch den **Dokumenttypdefinition** festgelegt werden kann.

HTML beschreibt die semantische Struktur und Formattierung der Inhalte von Webseiten und war urspruenglich eine Anwendung von SGML. Im Gegensatz zu XML hat HTML einen festen Satz von Tags, die fuer die Auszeichnung der Elementen verwendet werden koennen. Streng genommen ist HTML kein XML hat aber im wesentlichen die gleiche Struktur wie ein XML-Dokument. (Hierarchische Baumstruktur, Elemente, Tags, DOM).

Fuer XML gibt es viele standarte Werkzeuge, die XML Dokumente auf Wohlgeformtheit pruefen und porgrammatsich verarbeiten koennen, z.B. wie

- XML-Prozessor/Parser,
- **XQuery**: die standarde XML Abfrage- und Transformationssprache,
- **XPath**: Untersprache von XQuery, die XQuery unterstuetzt,
- **XSLT**: Sprache die speziell dazu geiegnet ist, XML Dokumente in andere Formate umzuwandeln.

Diese Tools stehen in XML Datenbanken zur verfuegung und XML Datenbanken sind fuer die Arbeit mit XML-Dokumenten optimiert. Somit koennen HTML-Dokumente mit den etablierten zahlreichen XML Tools optimal verarbeitet werden, wenn sie in einer XML Datenbank gespeichert werden.

Ein weiterer Vorteil ist, dass eine XML-Datenbank kein oder nur ein vereinfachtes Datenschema (Beziehungsschema/Tabellen) braucht, da die Daten schon durch das Dateiformat strukturiert werden. Bei einer relationael Datenbank muss das Schema explizit definiert werden. D.h. um ein HTML-Dokument in einer RDB zu speichern, oder um ein Dokument aus einer RDB zu exportieren muss jedes mal eine Transformation zwischen der HTML-Darstellung und relationalen Darstellung des Dokuments durchgefuehrt werden. Weiterhin funktioniert die Abbildung zwischen den Dokument-orientierten und relationalen Modellen nicht immer gut und wird als **object-relational impedence mismatch** bezeichnet.

b) **Vorteile**:

- Man benoetigt kein vordefiniertes Schema
- Kommt gut mit vielen Lese- und Schreibzugriffen zurecht.

Nachteile:

- Geringe konsistenzt/Gueltigkeit der Daten.

- Weil es weniger Einschraenkungen gibt, koennen die Abfragen nicht so gut optimiert werden wie bei den relationalen DBen.

1.4 Feedback

```
Rose, Dimitrov, Barthelmes

Aufgabe 1:
    a) Ja, technisch gesehen erfüllen Datenbanken, die nur ein Tier
 ↪  enthalten, die 1 und 4, wäre halt nur besser gewesen, wenn ihr ein
 ↪  "normales" Beispiel mit mindestens 2 Tieren gewählt hättet :D
```

```
9/9 Punkte

Aufgabe 2:
    1. Es wird das MODELL gesucht, nicht der NAME :D Also A380
    5. Nicht ganz, es sind eher alle Personen, die nicht für ein
 ↪  Flugzeug mit Reichweite >10000 zugelassen sind.
    6. "Ungeeignet"? Habt ihr nie von technischen Zwischenstopps gehört?
 ↪  :p
    7. Nicht unbedingt Rundfahrten
    8. "... Länder aus, die einen Flughafen haben, aus dem..."

8/11 Punkte

Aufgabe 3:

Habt ihr ChatGPT verwendet? Das sieht sehr nach ChatGPT aus...

4/4 Punkte

Insgesamt 21/24 Punkte
```

2 Relationale Algebra und SQL

2.1 Relationale Algebra - Fortsetzung

1. $\pi_{\texttt{pid, Name}}(\sigma_{\substack{\texttt{Rolle="Pilot",} \\ \texttt{Reichweite}\geq 15000}}(\texttt{Personal} \bowtie \texttt{Zulassung} \bowtie \texttt{Modell}))$
2. $\pi_{\texttt{Name}}(\sigma_{\texttt{Land='USA'}}(\beta_{\texttt{Code}\leftarrow\texttt{Ziel}}(\texttt{Flug}) \bowtie \texttt{Flughafen} \bowtie \texttt{Flugzeug}))$
3. $\pi_{\texttt{Code, Land}}\Bigg(\sigma_{\texttt{Name='F. Kohl'}}\Bigg(\texttt{Flugzeug} \bowtie$
   $\Big(\beta_{\texttt{Code}\leftarrow\texttt{Start}}(\pi_{\texttt{Start, fid}}(\texttt{Flug})) \cup \beta_{\texttt{Code}\leftarrow\texttt{Ziel}}(\pi_{\texttt{Ziel, fid}}(\texttt{Flug}))\Big)\Bigg)\Bigg)$
4. $\pi_{\texttt{pid, Name}}(\texttt{Personal}) -$
   $\pi_{\texttt{pid, Name}}(\texttt{Personal} \bowtie \texttt{Crew} \bowtie \sigma_{\texttt{Datum}<07.04.2013}(\texttt{Flug}))$

2.2 SQL-Anfragen

1. **SQL**:

```
select distinct C from R3
```

Ergebniss:

```
{{C: 7},
 {C: 8}}
```

2. **SQL**:

```
select distinct * from R2
where B = rot
```

Ergebniss::

```
{{B: rot, C: 9}}
{{B: blau, C: 8}}
```

3. **SQL**:

   ```
   select distinct * from R2
   intersect
   select distinct * from R3;
   ```

   Ergebniss:

   ```
   {{B: blau}, {C: 7}}
   ```

4. **SQL**:

   ```
   select * from R2
   union
   select * from R3
   ```

   Ergebniss:

   ```
   {{B: blau,  C: 7},
   {B: rot,   C: 8},
   {B: rot,   C: 9},
   {B: gruen, C: 8},
   {B: gelb,  C: 7}}
   ```

5. **SQL**:

   ```
   select * from R3 except (
       select * from R2
   );
   ```

   Ergebniss:

   ```
   {{B: gruen, C: 8},
   {B: gelb,  C: 7}}
   ```

6. **SQL**:

   ```
   select distinct * from
   R1 natural jo R2
   ```

   Ergebniss:

   ```
   {{A: q, B: rot, C: 8},
   {A: q, B: rot, C: 9}}
   ```

7. **SQL**:

   ```
   select distinct * from
   R1, R2
   ```

   Ergebniss:

   ```
   {{A: q, R1.B: rot,   R2.B: blau,  C: 7 },
   {A: q, R1.B: rot,   R2.B: gruen, C: 8},
   {A: q, R1.B: rot,   R2.B: gelb,  C: 7},
   {A: r, R1.B: gruen, R2.B: gelb,  C: 7},
   {A: r, R1.B: gruen, R2.B: gruen, C: 8},
   {A: r, R1.B: gruen, R2.B: gelb,  C: 7}}
   ```

2.3 Entsprechungen in SQL und der relationalen Algebra

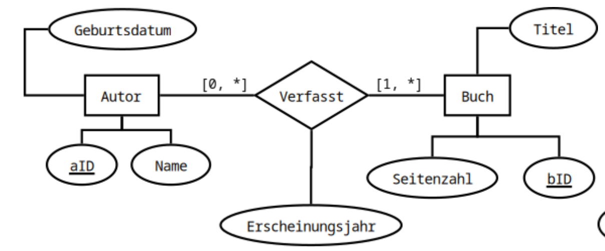
1. Die Anfragen entsprechen sich liefern jedoch nicht das gleiche Ergebniss, da der SQL-Ausdruck Duplikate zulaesst, waehrend bei der relationalen Abfrage die Duplikate entfernt werden.
2. 1. Die SQL-Anfrage liefert die **Bezeichnung** der Modelle, die nach Flughafen 'CDG' fliegen/geflogen haben.
   2. Der relationale Ausdruck liefert die Sitzplatzkapazitaeten der selben Modelle aus der SQL-Anfrage.

   Somit sind die Ausdrucke nicht Aequivalent und sie entpsrechen sich nicht.
3. 1. Die erste SQL-Anrage gibt die ID's der Co-Pilote und die Bezeichnungen der Modelle aus, dafuer sie zugelassen sind.
   2. Die zweite SQL-Anfrage gibt genau das gleiche Ergebniss wie die erste Anfrage. Man beachte, dass natural join in SQL immer von einem Kreuzprodukt und Selektionsoperationen simuliert werden kann.

   Somit sind die beiden Anfragen Aequivalent

2.4 ER-Modell

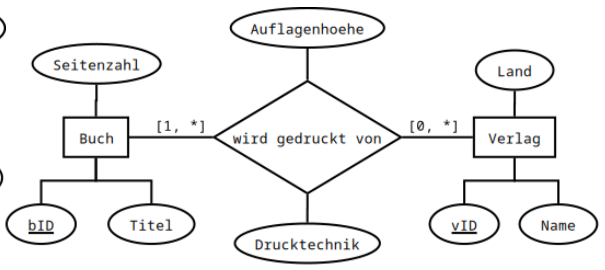
See the diagrams



Author kann beliebig viele Buecher schreiben.
(Moeglicherweise hat ein Autor kein Buch geschrieben)

Ein Buch muss mindestens von einem Author verfasst werden.
Ein buch kann von mehrerenAuthoren mitverfasst werden.

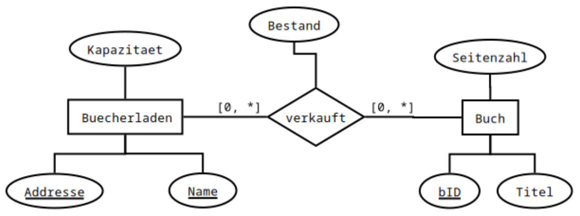
Figure 2.1: Zeile 1



Ein Verlag kann beliebig viel Buecher drucken.
Ein Verlag kann noch keine Buecher gedruckt haben - z.B ein neu gegruendeter Verlag
Ein Verlag hat einen Name und seinen Sitz in einem Land.

Ein Buch wird mindestens von einem Verlag gedruckt,
kann aber von mehreren Verlage gedruckt werden - z.B die Bibel.

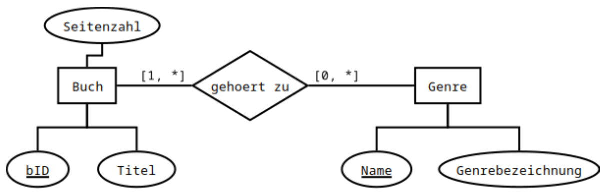
Figure 2.2: Zeile 2



Ein Buecherladen hat einen beliebig grossen Katalog von Buecher, die im Laden verkauft werden.
Der Katalog kann leer sein.
Ein Buecherladen wird Anhand einer Kombination seiner Addresse und seines Names unterschieden.

Ein Buch kann in beliebig vielen oder in keinen Laden verkauft werden.

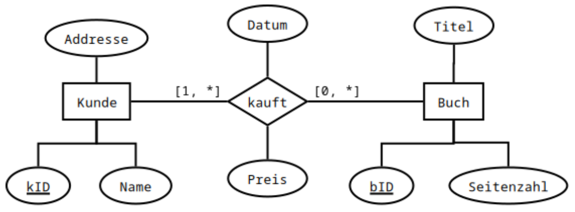
Figure 2.3: Zeile 3



Ein Buch gehoert zu mindestens einer Genre.

Es kann beliebig viele Bucher zu einer Genre geben.

Figure 2.4: Zeile 4



Ein Kunde kann beliebig viele Buecher kaufen, muss aber zumindest ein Buch gekauft haben,
um in Datenbank als 'Kunde' eingetragen zu werden.

Ein Buch kann von beliebig vielen Kunden gekauft werden.

Figure 2.5: Zeile 5

2.5 Feedback

```
Zur Aufgabe 1.

1. Richtig, wenn auch > statt >= gemeint

2. Richtig 3. Sollte passen 4. Richtig

Zur Aufgabe 2:

1. Richtig 2. Ergebnis: Wieso B: Blau wenn ihr nach rot selektiert => -
 ↪  0.25 P.

3.-7. Richtig

Zur Aufgabe 3:

1. Richtig unter der Annahme, dass Tabelle mehr als die abgedruckten
 ↪  Beispieldaten enthält (Stichwort Distinct)

2. Ebenfalls richtig

3. Dito

Zur Aufgab 4:

1. Richtig 2. Verlage sollen laut ML bitte mindestens ein Buch verlegen
 ↪  => - 0.25P. 3. Laut ML bitte [1,*] => - 0.25 P.

4. Richtig 5. Ebenfalls
```

3 ER-Modellierung

3.1 ER-Modellierung: Staedte

ER-Schema: Stadt

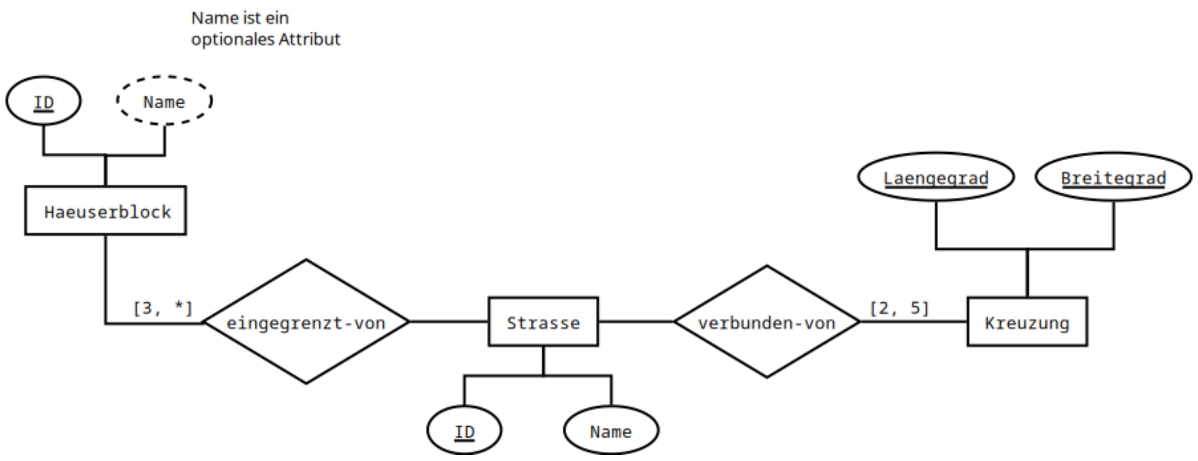


Figure 3.1: ER-Schema: Stadt

3.2 ER-Modellierung: Filmstudio-Datenbank

1. ER-Schema Figure 3.2
2. *Integriteatsbedingungen*
   1. i.A. koennen die Wertebereiche der Attribute im ER-Diagram nicht spezifiert werden, z.B. wie
      1. Erscheinungsjahr eines Films darf nicht in der Zukunft liegen oder ein sehr altes Datum wie 1776 sein.
      2. Gage eines Regissuers muss $>$ 30,000 € sein
      3. Globale Bedingungen wie z.B. **Gesamtgehalt** aus mehreren Filmen darf nie ueber 1000000 € sein koennen auch nicht spezifiziert werden.

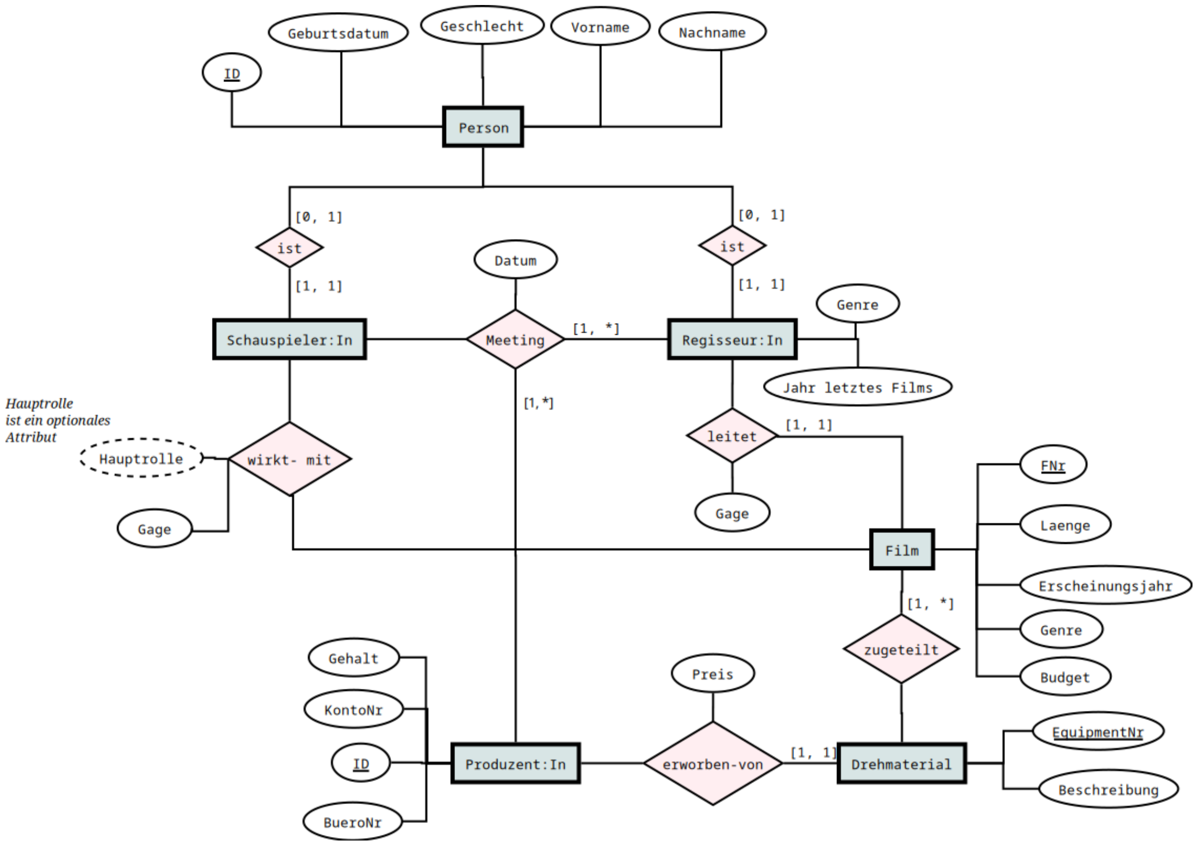
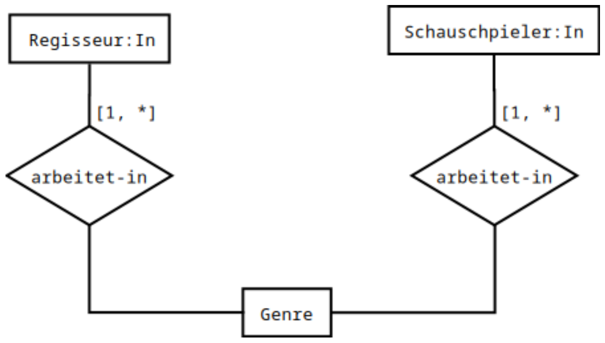


Figure 3.2: ER-Schema: NetMovie DB

2. In der Spezifikation heisst es, dass in jedem Film genau zwei Hauptrollen gibt. In unserem ER-Schema haben wir Hauptrolle als eine optionale Attribute des Beziehungstyps “wirkt-mit” modelliert. Diese Kardinalitaet kann somit nicht in unsrem ER-Schema bestimmt werden.

3. *Alternative Modellierungen*

   a) Gage als Attribute der Entitaet **Film** modellieren.
   b) Eine neue Entitaet “**Genre**” einfuehren, und “arbeitet-in” Beziehungen zwischen Regisseur-Genre, und zwischen Schauspieler-Genre modellieren:



3.3 Feedback

> `Punkte: 29.0/30`

4 Uebersetzung ER-Schema in Relationenschemata

4.1 Uebersetzung eines ER-Schemas

- Addresse(Ad_ID, PLZ, Stadt, Strasse, Hausnr)
- MusikerIn(M_ID, Name, Geb_Datum, Ad_ID → Addresse)
- Instrument(Name, Stimmung)
- spielt(M_ID→MusikerIn, (Name, Stimmung)→Instrument, bevorzugt)
- Musikstueck(MS_ID, Titel, Laenge, M_ID→MusikerIn)
- Album(A_ID, Titel, Release_Datum, Preis, Tracks, M_ID→MusikerIn)
- erscheint(M_ID→Musikstueck, A_ID→Album, TrackNr)
- spielt_mit(M_ID→MusikerIn, MS_ID→Musikstueck)

4.2 Uebersetzung eines ER-Schemas mit Hierarchien

1. *Relationales Schema*

- Personal(Pers_ID, GebDat, Name, Vorname)
- MitarbeiterIn(Pers_ID→Personal, Bonus)
- KundIn(KundenID→Personal, Branche)
- ManagerIn(Pers_ID→MitarbeiterIn, Sektion)
- ProgrammiererIn(Pers_ID→MitarbeiterIn, Abschluss)
- Programmiersprache(ProgSP)
- kann(ProgSP→Programmiersprache, Pers_ID→ProgrammiererIn, level)

2. *Weitere Methoden fuer is-a*:

i)
- **Vorteil**: Vermeidung der Redundanz und moeglichen Inkonsitenzenen, die dadurch enstehen koennen.
- **Nachteil**: Erhoehter Rechenaufwand durch Zugriff auf Attribute der Oberentitaet nur mit Join.

ii)
- **Vorteil**: Vermindertee Rechenaufwand durch direkten Zugriff auf Attribute ueber Tupel einziger Relation.

- **Nachteil**: Redundante Speicherung der gleichen Informationen.

iii)
- **Vorteil**: Vermindernde Komplexitaet des Datenbanks durch kleinere Anzahl von Relationen (eine Relation statt zwei oder drei)
- **Nachteil**: Moegliche Inkonsitenzen durch den vielen Nullwerten, die von dem Nutzer bei Insertoperationen explizit als null gesetzt werden muessen.

4.3 Feedback

```
Punkte: 26/28

Zur Aufgabe 1:

Bei Fremdschlüsselverweisen auf Relationen (z.B. in MusikerIn bitte
↪  sowas wie wohnt->Adresse) schreiben => -2P.

Rest der 1. passt.

Zur Aufgabe 2:

Zur 1:

Passt alles.

Zur 2:

Ebenfalls.
```

5 SQL-Anfragen

5.1 Grundlegende Anfragen

1.

```
select real_name, created_at
from twitter_user tu
where typ = 'lobby' and
date(created_at) <  timestamp '2009-06-30'
order by created_at
```

Table 5.1: A 1.1

real_name	created_at
Sascha Lobo	2007-05-08 21:10:26
netzpolitik	2007-10-24 14:34:50
Ulrich Müller	2009-01-07 14:50:51
Mehr Demokratie e.V.	2009-04-02 19:36:30
CCC Updates	2009-04-16 14:04:59
abgeordnetenwatch.de	2009-04-25 04:14:23
LobbyControl	2009-05-07 14:48:55

2.

```
select twitter_name, like_count
from twitter_user tu , tweet t
where tu.id = t.author_id
and t.like_count between 22000 and 25000;
```

Table 5.2: A 1.2

twitter_name	like_count
MAStrackZi	22713
n_roettgen	24329
SWagenknecht	24656
n_roettgen	22974

3.

```sql
select distinct h.txt
from
    twitter_user tu ,
    tweet t ,
    hashtag_posting hp ,
    hashtag h
where tu.id = t.author_id  and
    t.id = hp.tweet_id and
    hp.hashtag_id = h.id and
    tu.real_name = 'LobbyControl' and
    t.created_at between '2023-01-01' and '2023-01-15'
order by h.txt
```

Table 5.3: A 1.3

txt
ampel
autogipfel
autolobby
bundestag
eu
exxon
exxonknew
korruptionsskandal
lindner
lobbyregister

4.

```
select *
from hashtag h
where h.id in (
    select hp1.hashtag_id
    from hashtag_posting hp1, hashtag_posting hp2
    where hp1.tweet_id = hp2.tweet_id and
    hp1.hashtag_id = hp2.hashtag_id and not
    hp1.pos_start = hp2.pos_start
)
```

Anzahl der Ergebnisse:

Table 5.4: A 1.4 Anzahl der Ergebnisse

count
437

5.

```
select real_name, follower_count
from twitter_user
where created_at <= '2010-01-01'
and follower_count >= all (
    select follower_count
    from twitter_user
    where created_at <= '2010-01-01'
)
```

Table 5.5: A 1.5

real_name	follower_count
Sascha Lobo	761419

5.2 String-Funktionen

1.

```sql
select txt
from tweet
where txt ilike '%openai%'
and retweet_count >= 20
```

Table 5.6: A 2.1

txt
RT @_SilkeHahn: Guten Morgen: #OpenAI ist doch schon lange #ClosedAI. Seit der ersten Milliarde durch Microsoft 2019 lassen sie sich nicht mehr in die Karten schauen. @netzpolitik_org legt gekonnt den Finger in offene Wunden. Der "Technical Report" schweigt sich auf 98 https://t.co/ix1EYCAOKG… https://t.co/hAW5rbRUtH
RT @DrScheuch: Die meisten Aerosolforscher haben damals schon sehr starke Zweifel am Sinn von Ausgangssperren geäußert, (#openairstattausgangssperre) wurden aber nicht gehört. @Karl_Lauterbach, die NoCovid Modellierer und @janoschdahmen waren die lautesten Befürworter. https://t.co/bL7QMaoyKP

2.

```sql
select txt, char_length(txt) as Laenge
from named_entity
where char_length(txt) >= all (
    select char_length(txt)
    from named_entity
)
```

Table 5.7: A 2.2

txt	laenge
Arbeitsgemeinschaft Sozialdemokratischer Frauen	47

3.

```sql
select txt
from named_entity
where char_length(txt) >= 4
and txt like reverse(txt)
```

Table 5.8: A 2.3

txt
DAAD
GASAG
CIMIC
ABBA

5.3 Exists-Operatoren

1.

```
select *
from named_entity ne
where not exists (
    select *
    from named_entity_posting nep
    where nep.named_entity_id = ne.id
)
```

Anzahl der Ergebnisse:

Table 5.9: A 3.1 Anzahl der Ergebnisse

count
621

2.

```
select txt
from tweet t
where exists (
    select *
    from hashtag_posting hp, hashtag h
    where hp.hashtag_id = h.id
    and t.id = hp.tweet_id
    and h.txt = 'klima'
) and exists (
    select *
    from named_entity_posting nep, named_entity ne
    where nep.named_entity_id = ne.id
    and t.id = nep.tweet_id
    and ne.txt = 'Berlin'
)
```

Table 5.10: A 3.2

txt
Erinnerung: Bis 31.01.2023 um 23:59 (Europe/Berlin) könnt ihr noch Themenvorschläge zur #nr23 machen. Z.B. zu #Klima-,#Sozial-,#Sport-,#Medizin-, #Lokal-, #Nonprofit- #Datenjournalismus, #crossborder #Diversität #Pressefreiheit etc. Call for Papers: https://t.co/zyyNCsFC0y https://t.co/aDfQOzRG1W
RT @DieLinke_HH: Das #Klima retten, nicht den #Kapitalismus! Heute haben wir mit 12.000 anderen in #Hamburg beim #Klimastreik protestiert.

Klare Ansage an Rot/Grün in Hamburg und die Ampel in Berlin: Der Stillstand bei der Klimapolitik muss aufhören! https://t.co/jbfQrCcQuh | |So sieht die #Verkehrswende von @spdhh und @GRUENE_Hamburg aus

Frei nach Fairy Ultra: Während in Berlin schon das #9EuroTicket anläuft, werden in #Hamburg noch die Preise erhöht

Soziale #Klimapolitik geht anders!

@9euroforever #Klima @LinksfraktionHH https://t.co/Fi0eC8km1M |

3.

```
select tu.twitter_name , tu.real_name
from twitter_user tu
where exists (
    select *
    from tweet t , conversation c
    where t.author_id = tu.id and
    t.id = c.id and
    array_length(c.tweets, 1) >= 70 and
    t.created_at >= '2023-02-15'
)
```

Table 5.11: A 3.3

twitter_name	real_name
salomon_alex	Alexander Salomon
HendrikWuest	Hendrik Wüst

5.4 Aggregat-Funktionen und Gruppierung

1.

```sql
select ne.id, ne.txt, count(*) as Anzahl
from named_entity ne , named_entity_posting nep
where ne.id = nep.named_entity_id
group by ne.id
order by count(*) desc
```

Anzahl der Ergebnisse:

Table 5.12: A 4.1 Anzahl der Ergebnisse

count
15744

2.

```sql
select tu.real_name, count(*) as anzahl
from twitter_user tu , tweet t
where tu.id = t.author_id and
tu.typ = 'politician' and
t.created_at > '2022-01-01' and
tu.tweet_count > 2000
group by tu.id
order by anzahl desc
```

Anzahl der Ergebnisse:

Table 5.13: A 4.2 Anzahl der Ergebnisse

anzahl_der_ergebnisse
913

3.

```
with erg(id, anzahl) as (
    select nep.tweet_id, count(*) anzahl
    from named_entity_posting nep, tweet t
    where nep.tweet_id = t.id
    group by tweet_id
    having count(*) >= all (
        select count(*)
        from named_entity_posting nep
        group by tweet_id
    )
)
select created_at, txt
from tweet
where id in (
  select id
  from erg
)
```

Table 5.14: A 4.3

created_at	txt
2023-01-06 07:50:05	Adrian, Alexander, Ariturel, Björn, Christian, Christian, Christian, Christopher, Cornelia, Danny, Dirk, Frank, Heiko, Johannes, Kai, Katharina, Kurt, Maik, Martin, Michael, Oliver, Robbin, Roman, Sandra, Scott, Stefanie, Stefan, Stephan, Stephan, Sven.

Verdächtig?   @cduberlin |

4.

```
select date(created_at), count(*)
from tweet
group by date(created_at)
```

```
having date(created_at) > '2022-12-31'
and count(*) >= all (
    select count(*)
    from tweet
    group by date(created_at)
    having date(created_at) > '2022-12-31'
)
```

Table 5.15: A 4.4

date	count
2023-01-25	3568

5.5 Feedback

```
Punkte: 42.5/43

Zur Aufgabe 1:

1.-5. Richtig

Zur Aufgabe 2:

1.-3. Richtig

Zur Aufgabe 3:

1.-3. Richtig

Zur Aufgabe 4:

1. Richtig 2. Gemeint waren nur Tweets im Datensatz, sonst wäre das
 ↪  etwas einfach (mit HAVING anzahl >2000) =- 0.5 P.

3.-4. Richtig
```

6 Rekursion, Relationale Algebra und SQL

note: html Version der Abgabe (fuer die leichtere Kopierung der Code Blocks)

6.1 Data Definition Language (DDL) und Rekursion in SQL

1.

```sql
create table taxonomy(
    id int,
    name varchar,
    primary key(id),
    parent int,
    foreign key (parent) references taxonomy(id)
);

insert into taxonomy
values
    (0, 'animals', null),
    (2, 'chordate', 0),
    (1, 'athropod', 0),
    (6, 'mammals', 2),
    (5, 'reptiles', 2),
    (3, 'insects', 1),
    (4, 'crustacean', 1),
    (9, 'carnivora', 6),
    (8, 'scaled reptiles', 5),
    (7, 'crocodiles', 5),
    (10, 'cats', 9),
    (11, 'pan-serpentes', 8);
```

2.

```
select name
from taxonomy
where parent = 2
  union
select name
from taxonomy t1
where exists (
    select name
    from taxonomy t2
    where t1.parent = t2.id
    and t2.parent = 2
)
```

Table 6.1: A 1.2

name
carnivora
reptiles
crocodiles
scaled reptiles
mammals

3.

```
with recursive subCatOfChordate(id, name) as (
    select id, name
    from taxonomy t
    where t.parent = 2
        union
    select t.id, t.name
    from taxonomy t, subCatOfChordate s
    where t.parent = s.id
)
select id
from subcatofchordate
```

Table 6.2: A 1.2

id
6
5
9
8
7
10
11

6.2 Relationale Algebra und SQL

1.

- **rel**: $\pi_{\texttt{real\_name, tweet\_count, follower\_count}}\Big(\sigma_{\texttt{created\_at > 01.01.2019, follower\_count > 8000, tweet\_count > 1000, like\_count > 1000}}(\beta_{\texttt{author\_id}\leftarrow\texttt{id}}(\texttt{twitter\_user}) \bowtie \beta_{\texttt{ca}\leftarrow\texttt{created\_at}}(\texttt{tweet}))\Big)$
- **sql**:

```
select tu.real_name, tu.tweet_count, tu.follower_count
from twitter_user tu
where tu.created_at > '2019-01-01'
and tu.follower_count > 8000
and tu.tweet_count > 1000
and exists (
    select *
    from tweet t
    where t.author_id = tu.id
    and t.like_count > 1000
)
```

Table 6.3: A 2.1

real_name	tweet_count	follower_count
Rote Socke Türk-Nachbaur	16692	21283
Ursula von der Leyen	3675	1295550
Verteidigungsministerium	8923	120387
Carmen Wegge	1029	9355

real_name	tweet_count	follower_count

2.

- **rel**: $\pi_{\texttt{txt, author\_id, created\_at}}(\sigma_{\texttt{like\_count > 1000}}(\texttt{tweet}) -$
  $\pi_{\texttt{txt, author\_id, created\_at}}\Big($
  $\sigma_{\texttt{created\_at > ca}}(\sigma_{\texttt{like\_count > 1000}}(\texttt{tweet}) \times \beta_{\texttt{ca}\leftarrow\texttt{created\_at},\texttt{ai}\leftarrow\texttt{author\_id},\texttt{t}\leftarrow\texttt{txt}}(\sigma_{\texttt{like\_count > 1000}}(\texttt{tweet})))\Big)$
- **sql**:

```
select t.txt, t.author_id, t.created_at
from tweet t
where t.like_count >= 1000
and t.created_at <= all (
    select created_at
    from tweet
    where like_count >= 1000
)
```

Table 6.4: A 2.3

txt	author_id	created_at
Die Leute haben heute aus Trotz geböllert, oder? Das nahm ja kein Ende. Genial! Danke.	814970546366610437	2023-01-01 00:17:32

3.

- **rel**: $\pi_{\texttt{hi, hashtag\_id}}\Bigg(\sigma_{\texttt{ti < tweet\_id}}\Big($
  $\beta_{\texttt{ti}\leftarrow\texttt{tweet\_id}}(\sigma_{\texttt{hi < hashtag\_posting}}(\texttt{hashtag\_posting} \bowtie \beta_{\texttt{hi}\leftarrow\texttt{hashtag\_id}}(\texttt{hashtag\_posting})))$
  $\bowtie$
  $\sigma_{\texttt{hi < hashtag\_posting}}(\texttt{hashtag\_posting} \bowtie \beta_{\texttt{hi}\leftarrow\texttt{hashtag\_id}}(\texttt{hashtag\_posting}))\Big)\Bigg)$
- **sql**:

```
with hashtagpairs as (
    select
        hp1.hashtag_id h1_id,
        h1.txt h1_txt,
        hp2.hashtag_id  h2_id,
        h2.txt h2_txt,
        hp1.tweet_id tid
    from hashtag_posting hp1, hashtag_posting hp2, hashtag h1, hashtag
     ↪  h2
    where hp1.tweet_id  = hp2.tweet_id
    and h1.id = hp1.hashtag_id
    and h2.id = hp2.hashtag_id
    and hp1.hashtag_id  < hp2.hashtag_id
)
select hpr1.h1_txt, hpr1.h2_txt
from hashtagpairs hpr1
where exists (
    select *
    from hashtagpairs hpr2
    where hpr1.h1_id = hpr2.h1_id
    and hpr1.h2_id = hpr2.h2_id
    and hpr1.tid < hpr2.tid
)
```

Table 6.5: A 3.3 Anzahl der Ergebnisse

count
178346

6.3 Regulaere Ausdruecke in SQL

```
select tu.real_name, regexp_count(t.txt, '\m[[:upper:]]{2,}\M') as cnt,
 ↪  t.txt
from tweet t, twitter_user tu
where tu.typ = 'politician'
and t.author_id  = tu.id
and regexp_count(t.txt, '\m[[:upper:]]{2,}\M') >= all (
    select regexp_count(txt, '\m[[:upper:]]{2,}\M')
    from tweet
)
```

Table 6.6: A 3

real_name	cnt	txt
Udo Hemmelgarn, MdB	41	RT @Georg_Pazderski: BITTE BITTE BITTE BITTE

BITTE https://t.co/snBGvZGABI |

6.4 Feedback

```
Punkte: 33/33

Zur Aufgabe 1:

1.-3. Richtig

Zur Aufgabe 2:

1.-3. Richtig

Zur Aufgabe 3:

Geht einfacher, aber richtig.
```

7 SQL und Anfragesprachen

note: html Version der Abgabe fuer leichtere Kopierung der Codeblocks.

7.1 Fortgeschrittene SQL-Anfragen

1.

```sql
select tu.real_name , tu.twitter_name
from twitter_user tu
where tu.typ = 'politician'
and exists (
    select *
    from twitter_user tu2
    where tu2.twitter_name <> tu.twitter_name
    and tu2.real_name = tu.real_name
)
```

Table 7.1: A1.1

real_name	twitter_name
Martin Hagen	_MartinHagen
Martin Hagen	MartinHagenHB

2.

```sql
select
    tu.real_name real_name,
    tu.twitter_name twitter_name,
    tu.follower_count follower_count,
    tu.tweet_count tweet_count,
    array_length(c.tweets, 1) conversation_length
from tweet t, conversation c, twitter_user tu
where t.id = c.id
```

```
and tu.id = t.author_id
and array_length(c.tweets, 1) >= all (
    select array_length(c2.tweets, 1)
    from conversation c2
)
```

Table 7.2: A1.2

real_name	twitter_name	follower_count	tweet_count	conversation_length
Tom Schreiber	TomSchreiberMdA	5328	48186	86
Christian Lindner	c_lindner	653690	18882	86

3.

```
select ne.txt, ne.id, count(*)
from
    tweet t,
    hashtag_posting hp,
    hashtag h,
    named_entity ne,
    named_entity_posting nep
where t.id = hp.tweet_id
and hp.hashtag_id = h.id
and h.txt ilike 'energie'
and nep.tweet_id = t.id
and nep.named_entity_id = ne.id
group by ne.txt, ne.id
having count(*) >= 4
order by count(*) desc
```

Table 7.3: A1.3

txt	id	count
Deutschland	31	18
Bayern	240	7
Thüringen	526	6
Anschluss	1741	5
Berlin	2	4
Bernhard Stengele	11253	4
CDU	65	4

txt	id	count
Europa	217	4
Bund	655	4

4.

```
select
    ne.id entity_id,
    ne.txt entity_txt,
    date(t.created_at) datum,
    count(*) anzahl
from
    tweet t ,
    named_entity_posting nep ,
    named_entity ne
where t.id = nep.tweet_id
and ne.id =nep.named_entity_id
group by ne.id, ne.txt, date(t.created_at)
order by count(*) desc
limit 5
```

Table 7.4: A1.4

entity_id	entity_txt	datum	anzahl
6	Ukraine	2023-02-24	761
2	Berlin	2023-02-12	427
28	Bundestag	2023-03-17	286
2	Berlin	2023-02-10	283
1425	CSU	2023-03-17	259

7.2 Relationale Algebra und Tupelkalkuel

1.

- **umg**: Was sind die echten Namen von allen Twitter Benutzern, die Lobbyisten sind, die einen Tweet mit ueber 2000 Likes veroeffentlicht haben, der die EU oder die USA erwaehnt?
- **tup**:

$$
\begin{aligned}
\{\langle \texttt{tu.real\_name}\rangle \mid \texttt{tu} \in \texttt{twitter\_user} \land \texttt{tu.typ='lobby'} \land \exists \texttt{t} \exists \texttt{ne} \exists \texttt{nep}(&\\
& \texttt{t} \in \texttt{tweet} \land \\
& \texttt{ne} \in \texttt{named\_entity} \land \\
& \texttt{nep} \in \texttt{named\_entity\_posting} \land \\
& \texttt{t.id = nep.tweet\_id} \land \\
& \texttt{ne.id = nep.named\_entity\_id} \land \\
& \texttt{t.like\_count > 2000} \land \\
& \texttt{t.author\_id = tu.id} \land \\
& (\texttt{ne.txt = 'EU'} \lor \texttt{ne.txt = 'USA'}))\}
\end{aligned}
$$

2.

- **umg**: Was sind die IDs aller Authoren, die zwar einen Tweet mit dem Hashtag "openai" verfasst haben aber keinen mit dem Hashtag "chatgpt".
- **tup**:

$$
\begin{aligned}
\{\langle \texttt{t.author\_id}\rangle \mid \texttt{t} \in \texttt{tweet} \land \exists \texttt{h} \exists \texttt{hp}(&\\
& \texttt{h} \in \texttt{hashtag} \land \\
& \texttt{hp} \in \texttt{hashtag\_posting} \land \\
& \texttt{h.id = hp.hashtag\_id} \land \\
& \texttt{hp.tweet\_id = t.id} \land \\
& \texttt{h.txt = 'openai'}) \land \\
& \neg \exists \texttt{h} \exists \texttt{hp}(\\
& \texttt{h} \in \texttt{hashtag} \land \\
& \texttt{hp} \in \texttt{hashtag\_posting} \land \\
& \texttt{h.id = hp.hashtag\_id} \land \\
& \texttt{hp.tweet\_id = t.id} \land \\
& \texttt{h.txt = 'chatgpt'})\}
\end{aligned}
$$

7.3 Feedback

> `Punkte: 28/28`

8 Physiche Datenorganisation und Baeume

8.1 Seiten und Saetze

1. Satzlaenge `twitter_user`

Attribute	Typ	Satzlaenge
`id`	`bigint`	8 byte
`follower_count`	`integer`	4 byte
`tweet_count`	`integer`	4 byte
`typ`	`char(11)/char(5)` “politician”/“lobby”	12 byte (1 byte Overhead)
`created_at`	`timestamp`	8 byte
`twitter_name`	`text`	12 byte
`real_name`	`text`	18 byte
—	—	—
$\sum$		**54 byte**

2. Speicherplatz der Header

   - Jede **Page** hat 24 Byte Header
   - Jedes **Tupel** hat 23 Byte Header

   D.h. jedes Tupel hat 54 Byte Nutzdaten + 23 Byte Header = 77 Byte.

3. Groesse der Bloecke im PostgreSQL:

```
select current_setting('block_size');
```

Table 8.2: block size

current_setting
8192

```
select count(*)
from twitter_user
```

Table 8.3: Anzahl der Tupel in der Relation twitter user

count
1825

Anzahl der Tupel pro Seite ca.:

```
round(8192 / 77)
```

```
[1] 106
```

Somit ist die Anzahl der Seiten ungefaehr:

```
round(1825 / 106)
```

```
[1] 17
```

4. Anzahl der Seiten der Relation ‘twitter_user’:

```
select relname, relpages
from pg_class
where relname = 'twitter_user'
```

Table 8.4: Anzahl der Seiten fuer twitter user

relname	relpages
twitter_user	22

Also in Wirklichkeit werden 22 Seiten gebraucht statt 17 Seiten. D.h. mehr Speicher. Die Gruende dieser Abweichung sind u.a. mehr Speicher fuer:

- Pageheader
- Zeiger auf die Tupel
- Special-/Free Space in Pages
- Optionalen Zusatzelementen wie Null Bitmap in den Tuples

8.2 B-Baeume

1. B-Baum 1.1 Figure 8.1

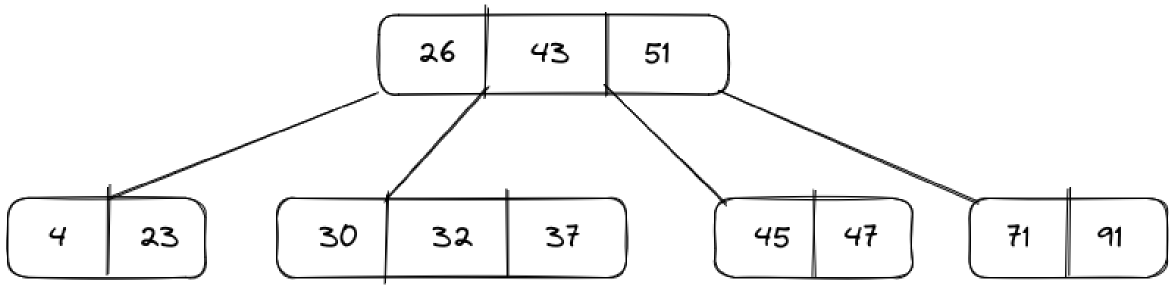


Figure 8.1: B-Baum: A2.1

2. B-Baum 1.2 Figure 8.2

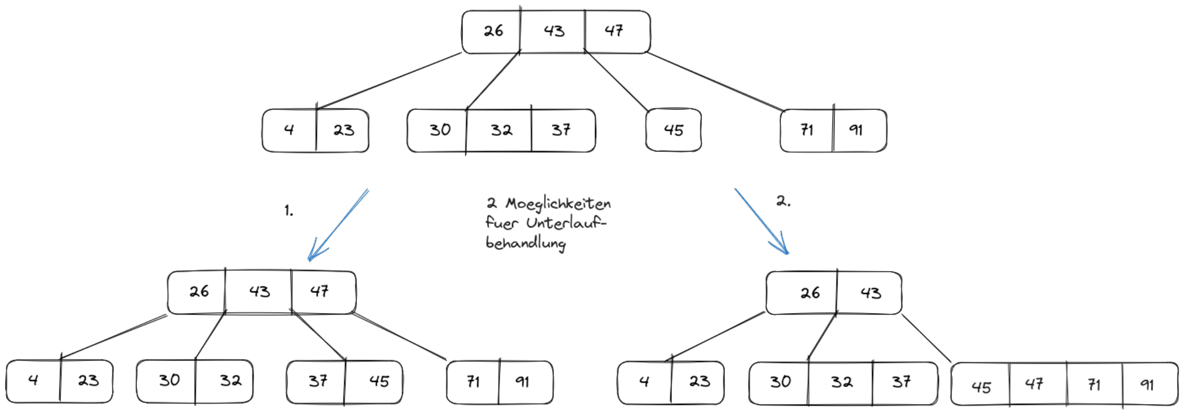


Figure 8.2: B-Baum: A2.1

8.3 B$^+$-Baeume

1. B+-Baum 3.1 Figure 8.3

2. Die Elemente in der sortierten Reihenfolge in den B+ Baum einfuegen, aber nicht durch ein normales Insert, sondern direkt an das Blatt ganz rechts einfuegen. Dadurch spart man sich die look-up Operation $\mathcal{O}(\log_m(n))$ des insert, die ein groeseres Element als die bisherigen sowieso ganz rechts in Baum ablegen wuerde.

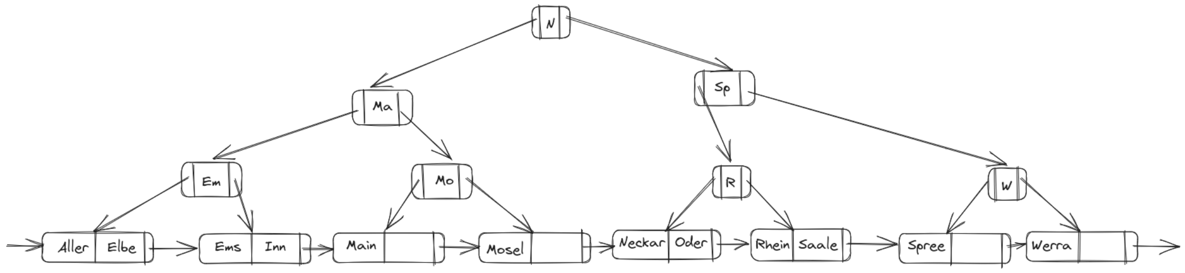


Figure 8.3: B-Baum: A2.1

8.4 Feedback

```
Punkte: 19.5/24.0
- A1: 5.0/8.0
- A2: 6.5/8.0
- A3: 8.0/8.0

Zur Aufgabe 1:

1. typ hat Länge 4, da enum => -0.5 P.

2. Tupel Header richtig, Attribut-Größe fehlt => -0.5 P.

3. So war die Aufgabe nicht gemeint, man sollte aus den Werten der 1.
 ↪ und 2. das Ausrechnen => -2 P.

4. Richtig

Zur Aufgabe 2:

1. Richtig 2. Beide Varianten nicht ganz richtig:

Grundstruktur teilweise richtig: Links sehr nah an ML man müsste nur 43
 ↪ und 37 tauschen (der angesprochene Leichtsinnsfehler), rechts bei
 ↪ Löschen eines Nicht-Blattknotens wird Element durch
 ↪ (längen-)lexikographisch nächst kleineres Element ersetzt, in diesem
 ↪ Fall die 47. Ab dann ändert sich logischerweise auch euer Baum im
 ↪ Vergleich zur ML.

 => -1.5 P.
```

```
Zur Aufgabe 3:

1. Diese Woche war die ML falsch, deswegen habe ich hier einen Fehler
 ↪  beim Korrigieren gemacht. Nur Ma zu M wäre schön.

2. Habe ich übersehen, ist richtig.
```

9 Hashing, Indexzugriffe und Sortierung

9.1 Erweitarbares Hashing

1. Einfuegen der Woerter des Limerick

   a.

h	bucket
00	Who, feared
10	Man, said
01	beard
11	Old

   b.

h	bucket
000	hen, four
001	who, feared
10*	man, said
010	beard, built
011	owls, wren
110	larks, nests
111	old, two

2. Loeschen der Woerter

h	bucket
000	hen, four
001	who, feared
10*	man, said
010	built
011	wren
110	larks

h	bucket
111	—

Der Bucket fuer 111 ist leer und 010 bis 110 sind nur halbvoll. Somit wird die Spercherzelle vom Bucket 111 unnoetig reserviert / blockiert, und die Tabelle ist unausgeglichen

Eine Loesung waere "Buckets zu verkleinern" also zwei Buckets der lokalen tiefe t zu einem bucket der tiefe $t-1$ zusammenzufassen, wenn "zu wenig" Elemente dort enthalten sind bzw. die Buckets leer sind.

Ein Vorteil hiervon waere eine hoehere Speichereffizienz. Ein Nachteil der erhoehte Aufwand durch pruefen der Fuellmenge, mehr Kopieren, etc. Ausserdem koennte es zu vielen Kopieroperationen kommen, bei unguenstigen Einfuege-/Loeschoperationen. (Wiederholtes loeschen & einfuegen auf **einen** Bucket).

9.2 Sortierung

a. 25.000 Bloecke, 48 Bloecke im HS

1. $\frac{25000}{48} \approx 521$
2. $\lceil \log_{47}(521) \rceil \approx 1,625 \approx 2$
3. $2 \cdot 25000 \cdot (1+2) = 150000$
4. passt bereits.

b. 24,000,000 Bloecke Relation, 64 Bloecke im HS

1. Anzahl der Partitionen: $\frac{24000000}{64} = 375000$
2. $\lceil \log_{63} 375000 \rceil = 4$
3.

$$\begin{aligned} 2 \cdot 24.000.000 \cdot (1 + \lceil \log_{63} 375000) &= 2 \cdot 24.000.000 \cdot (1+4) \\ &= 5 \cdot 2 \cdot 24.000.000 \\ &= 240.000.000 \end{aligned}$$

4.

$$\begin{aligned} \log_x \frac{24 \cdot 10^6}{x} = 2 \iff x^2 &= \frac{24 \cdot 10^6}{x} \\ x^3 &= 24 \cdot 10^6 && (\text{Mult } x) \\ x &= \sqrt[3]{24 \cdot 10^6} && (\sqrt[3]{\bullet}) \\ x &\approx 289 \end{aligned}$$

9.3 Zugriffmethoden

1. b) ist effizienter. Nur Tupel mit $d \in [0, 500)$ Kommen fuer die Ergebnisrelation in Frage. Dies laesst sich schnell mit B$^+$-Baum ausgeben, schneller als 7,8 mill Tupel zu scannen.
2. a) ist effektiver, denn es werden auf jeden Fall alle Elemente ausgegeben ausser $d = 1000000$. Fuer dieses muss gepreuft werden ob $y > 42 \land y < 50$. Aber weil eh so gut wie alle Elemente ausgegeben werden macht Index-zugriff keinen Sinn.
3. Gesuch sind mindestens alle Tupel t mit $d \in [7100000, 7150000)$. Dies laesst sich gut mit B$^+$-Baum abfragen. Deshalb b).
4. Hier lohng es sich weider mit dem B$^+$-Baum eine Vorauswahl an Elementen zu machen, da $d \in I_1[0, 100] \lor d \in I_2[8800, 9000) \lor d \in I_3[7000000, 7001000)$ ein wesentlich kleinerer Wertebereich ist als die komplette Relation.

9.4 Feedback

```
Punkte 24.0/28.0
- A1: 10.0/10.0
- A2: 8.0/10.0
- A3: 6.0/10.0

Zur Aufgabe 1:

1.-2. Richtig

Zur Aufgabe 2:

1. Richtig 2. Richtig 3. Richtig 4. a) Trotzdem solltet ihr ausrechnen,
 ↪  wie viele Blöcke benötigt werden b) Es wird noch ein weiterer Block
 ↪  zum Sortieren benötigt => -2 P.

Zur Aufgabe 3:

1. Falsch, da Tupel bereits sortiert sind => -2 P. 2. Richtig 3. Richtig
 ↪  4. Richtig
```

10 Anfragebearbeitung

10.1 Join-Strategien

1. cost: $b_r + \lceil \frac{b_r}{\text{mem}-1} \rceil \times b_s$
   R: 115000 Tupel, 30 pro Seite $\Rightarrow$ 3834 Seiten
   S: 22500 Tupel, 100 pro Seite $\Rightarrow$ 225 Seiten
   mem = 70

   Somit: cost $= 225 + \lceil \frac{225}{69} \rceil \times 3834 = 15561$

   Kosten sind minimal g.d.w. b_r Komplett in dem Speicher passt und noch eine Seite frei ist fuer $b_s \Rightarrow 226$:

   $$\text{cost} = 225 + \lceil \frac{225}{226-1} \rceil \cdot 3834 = 4059$$

2. cost: $b_r + p \cdot b_s$ mit $p = \lceil \frac{b_r}{\text{mem}-1} \rceil$
   $\Rightarrow$ Kosten gleich wie Block-Nested-Loop.

3. Da die Formeln zu Berechnung gleich sind, sind beide Algorithmen gleich optimal bei einer RAM-Groesse von 226 Sieten. (Kosten wie in 1.)

4. Y ist primary-key von S $\Rightarrow$ alle Werte sind Unique. D.h. ein tupel in R kann mit max **einem** Tupel aus S matchen **aber** ein Tupel aus S kann maximal mit allen Tupel aus R matchen.

   D.h. maximal: 115000 Tupel wenn alle aus R matchen.

   R: 30 Tupel Pro Seite,
   S: 100 Tupel pro Seite,
   S + R = ?

$$
\begin{aligned}
& r \in R, s \in S, p := \text{Seite} \\
& 30r = p \iff r = \frac{1}{30}p \\
& 100s = p \iff s = \frac{1}{100} \\
& x(s+r) = p \\
\Rightarrow & x(\frac{1}{30}p + \frac{1}{100}p) = p \\
\iff & x\frac{13p}{300} = p \\
\iff & x \approx 23,07 \approx 23\text{Tupel} && \text{(Annahme: nicht-Spannsaetze)}
\end{aligned}
$$

Somit: $\frac{115000 \text{ Tupel}}{23 \frac{\text{Tupel}}{\text{Seite}}} = 5000$ Seiten

10.2 Algebraische Optimierung

1.
```sql
select tu.real_name, tw.txt
from
   tweet tw,
   named_entity ne,
   named_entity_posting nep,
   twitter_user tu
where tw.author_id = tu.id
   and nep.tweet_id = tw.id
   and nep.named_entity_id = ne.id
   and tw.created_at > '2022-04-15'
   and tw.like_count > 6000
   and ne.txt like 'Berlin'
   and tu.created_at < '2015-01-01'
```

Table 10.1: A2.1

real_name	txt
Sahra Wagenknecht	Das Manifest für Frieden hat jetzt schon eine Viertelmillion Unterstützer: https://t.co/uC3H0FBZix Druck von unten kann etwas bewirken! Lasst uns den Protest gegen #Waffenlieferungen und für #Frieden und #Diplomatie nun auch die Straße tragen: Am 25. Februar um 14 Uhr in Berlin

real_name	text
Hermann Gröhe	Gemeinsam haben wir heute fraktionsübergreifend als politische Patinnen und Paten von Opfern und Verfolgten des iranischen Regimes vor der iranischen Botschaft in Berlin protestiert. Wir sehen die Verbrechen des iranischen Regime! #IranRevoIution #JinJiyanAzadi #Iran https://t.co/rDZC6KpBA2
Sahra Wagenknecht	Großartig: Eine halbe Million Unterschriften für das #ManifestfuerFrieden nach einer Woche! Herzlichen Dank & gerne weiter verbreiten: https://t.co/uC3H0FBZix Und kommt zur #Friedenskundgebung am 25.02. um 14 Uhr vor dem Brandenburger Tor in Berlin! https://t.co/2jxKTNGNpH https://t.co/WWXWoV9NlD
Sahra Wagenknecht	Ein Jahr #Krieg ist mehr als genug! Es braucht Druck von unten für #Verhandlungen. Jeder verlorene Tag kostet weitere Menschenleben und bringt uns einem 3. Weltkrieg näher. Deshalb: Kommt am 25. Februar um 14 Uhr zur großen #Friedenskundgebung nach Berlin: https://t.co/2jxKTNGNpH https://t.co/yu5vc30uv1
Sahra Wagenknecht	Nicht alle Ukrainer ziehen mit Begeisterung an die Front, und wer zu #Verhandlungen aufruft, fordert keine Kapitulation. In der Berliner Zeitung begründe ich, warum wir morgen um 14 Uhr in Berlin ein Stoppzeichen gegen weitere Kriegs-#Eskalation setzen. https://t.co/OblSye7pJ6
Sahra Wagenknecht	Auf geht's zur großen Kundgebung "Aufstand für den Frieden" am BRB Tor in Berlin - mit Reden von Alice Schwarzer, Brigade-General a.D. Erich Vad, Jeffrey Sachs, mir und Weiteren. Für jene, die nicht vor Ort sein können startet (hier) 14:00 ein Livestream: https://t.co/lNpFdi6sv7
Sevim Dağdelen, MdB	50.000 Menschen auf der #Friedenskundgebung #AufstandFuerFrieden in Berlin am Brandenburger Tor! #b2502 https://t.co/lB5NZYruWX
Sahra Wagenknecht	Heute kamen mehr als 50000 Menschen zu unserer Friedensdemo nach Berlin. Wir sind viele, das hat die Veranstaltung gezeigt. Ein herzliches Dankeschön an alle, die dabei waren!
Sahra Wagenknecht	Schätzungsweise 50.000 Menschen nahmen gestern an unserer Friedenskundgebung am Brandenburger Tor in Berlin teil. Dazu haben Alice Schwarzer & ich eine Erklärung veröffentlicht: https://t.co/gBbEHxgS8Q

2.

$$\pi_{\texttt{tu.real\_name, tw.txt}}\Bigg(\sigma_{\substack{\texttt{nep.tweet\_id = tw.id} \wedge \texttt{tw.author\_id = tu.id} \wedge \\ \texttt{nep.named\_entity\_id = ne.id} \wedge \texttt{tw.created\_at > '15.04.2022'} \wedge \\ \texttt{tw.like\_count} > 6000 \wedge \texttt{ne.txt like 'Berlin'} \\ \texttt{tu.created\_at < '01.01.2015'}}}\Big(\beta_{\texttt{tw}}(\texttt{tweet}) \times \beta_{\texttt{ne}}(\texttt{named\_entity}) \times \beta_{\texttt{nep}}(\texttt{named\_entity\_posting} \times \beta_{\texttt{tu}}(\texttt{twitter\_user})\Big)\Bigg)$$

3. Siehe Figure 10.1
4. Siehe Figure 10.2

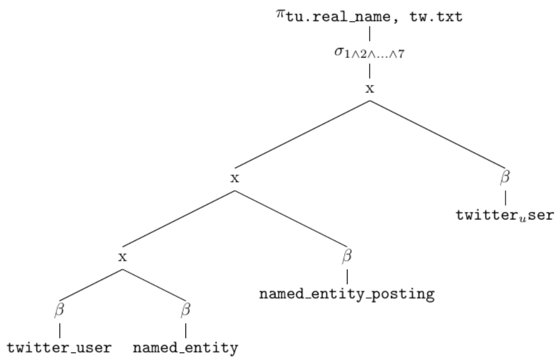


Figure 10.1: A2.3: nicht optimiert

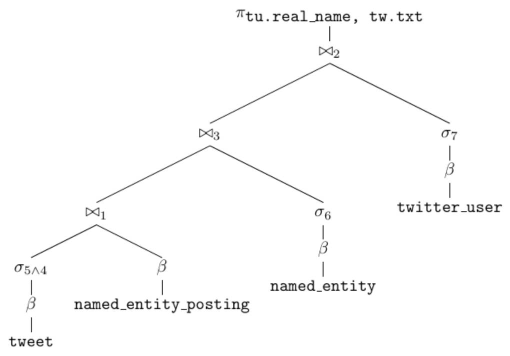
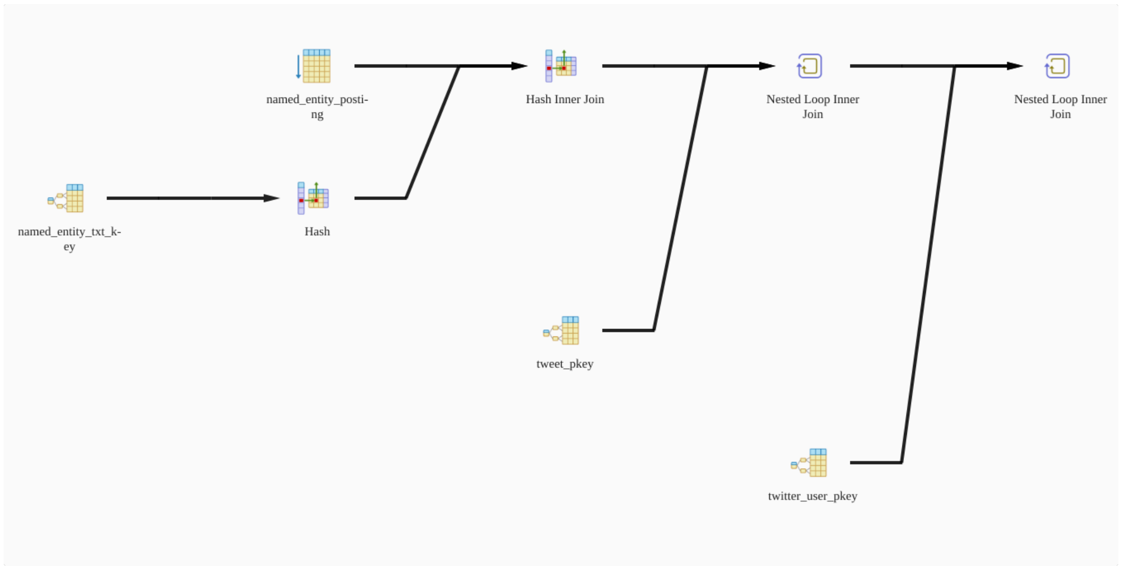


Figure 10.2: A2.4: optimiert

5. Operatorbaum mit pgadmin explain **?@fig-a2-5**



Der Operatorbaum i.A. aehnlich zu dem Operatorbaum aus 4), in dem Sinne das 1. Die Selektionen ebenfalls nach unten geschoben wurden und immer **vor** den join-Operationen stehen. Denn wir haben hier ausserdem keine Krezuprodukte, sondern 2. alle Kreuzprodukte wurden durch joins ersetzt.

Spannend it ausserdem, dass hier ein richtiger Anfrageplan vorliegt, da joins/scans genau definiert wurden (index-scan, hash inner join, …).

Darueber hinaus ist die join Reihenfolge anders: Da die Seletion ueber der Relation `named_entity` nur ein Ergebnis liefert, steht diese ganz unten und haelt alle Folgeergebnisse schlank. (Alle joins machen nur einen loop. Steht wenn man auf die joins klickt)

10.3 Feedback

```
Punkte: 18.0/23.0
- A1: 7.0/11.0
- A2: 11.0/12.0

Rose, Dimitrov, Barthelmes

Aufgabe 1:

3. Naja, es wird ja nicht nach dem optimalen Algorithmus *aus 1 oder 2*
↪  gefragt, sondern nach dem optimalen Algorithmus überhaupt
4. Falsche Formeln, falsches Ergebnis
```

```
7/11 Punkte

Aufgabe 2:

wo textdatei
3. Es fehlen die Projektionen nach den notwendigen Attributen

10/12 Punkte

Insgesamt 17/23 Punkte
```

Part II

Notes

11 Intro

DBMS:

11.1 Purposes of DB

Two main uses of databases:

- **online transaction processing**: large numer of users use the database, each user retrieving small amounts of data and performing small updates.
- **data analytics**: processing of data to draw conclusions, infer rules or decision procedures.

11.2 SQL DDL

```sql
create table department
  (dept_name  char(20),
   building   char(15),
   budget     numeric(12, 2));
```

SQL DDL supports integrity constraints

11.3 SQL DML

sql is declarative. Input several tables, output always a single table. Example:

single table input:

```sql
select instructor.name
from instructor
where instroctur.dept_name = 'History'
```

two tables as input:

```
select instructor.ID, department.dept_name
from instructor, department
where
  instructor.dept_name = department.dept_name and
  department.budget > 95000;
```

11.4 Steps to DB Design

Computer representation of some real world or imaginary complex system involves mainly two tasks:

- modelling the data aspect of the system (data requirements)
- modelling the behavior aspect of the system (functional requirements)

Modelling the data aspect falls under the domain of DBs design. Primary objects and their interrelationships must be determined according to the needs of the users of the system. Modelling behavior is a software design task. These tasks are interrelated and can reinforce/influence each other.

Steps of DB design:

1. **requirements analysis**: precise specification of client requirements (what data needs to be modelled) by consulting with domain experts and prospective clients.

2. **conceptual design**: a data model is chosen and the requirements are translated to the concepts of the data model (usually ER).

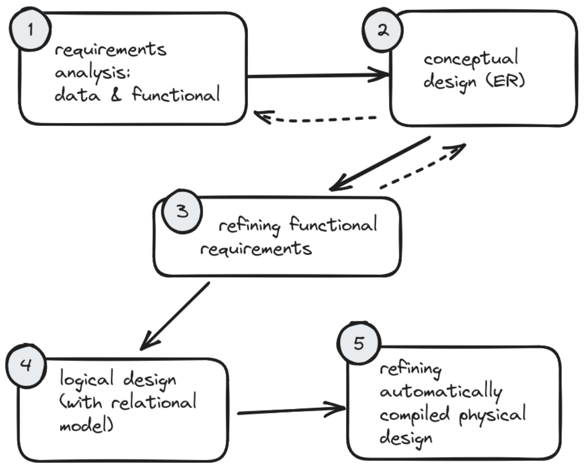
   the resulting schema is reviewed to confirm that all data requirements are satisfied and that there are no conflicts. Redundant features can be removed too.

   basically all necessary attributes to be captured are determined in this step. alternative to ER an automatic generation of tables can be achieved via a method called **normalization**.

3. **specification of functional requirements**: A precise conceptual schema facilitates determining the behavioral requirements. These in turn can reinforce and alter the data schema.

4. **logical-design phase**. Higher-level conceptual schema is mapped to the implementation data model of the database system (usually relational).

5. **physical-design phase**. The logical schema is automatically compiled to a physical implementation by the DBMS but the designer can manually refine and fine-tune the physical design.


1
requirements analysis: data & functional
2
conceptual design (ER)
3
refining functional requirements
4
logical design (with relational model)
5
refining automatically compiled physical design

11.5 DBMS architecture

The general architecture of a DBMS can be broadly regarded as consisting of **three** main components:

- **querry processor**
- **storage manager**
- **disk storage**

11.5.1 Querry processor

The querry processor realizes an easiy access to data via a querry language: users don't have to specify *how* to access the data but only *what* data to access. The particular algorithm/plan needed to access the data specified by the querry is automatically compiled by the querry processor. Querry processors try to optimize these algorithms as much as possible.

1. **DDL interpreter**
2. **DML compiler**: (query optimization)
3. **querry evaluation engine**: executes low-level instrocution generated by the DML compiler.

11.5.2 Storage Manager

DB are very large and cannot be stored entirely in the main memory. Also since persistent storage of data is always wanted, DB are stored on secomdary memory, usually disks.

Movement of data from secondary memory is much slower than movement from main memory to CPU registers. Therefore the DB tries to structure the data in a way that movement from secondary memory to main memory is minimized. This can be fine-tuned by the DB designer.

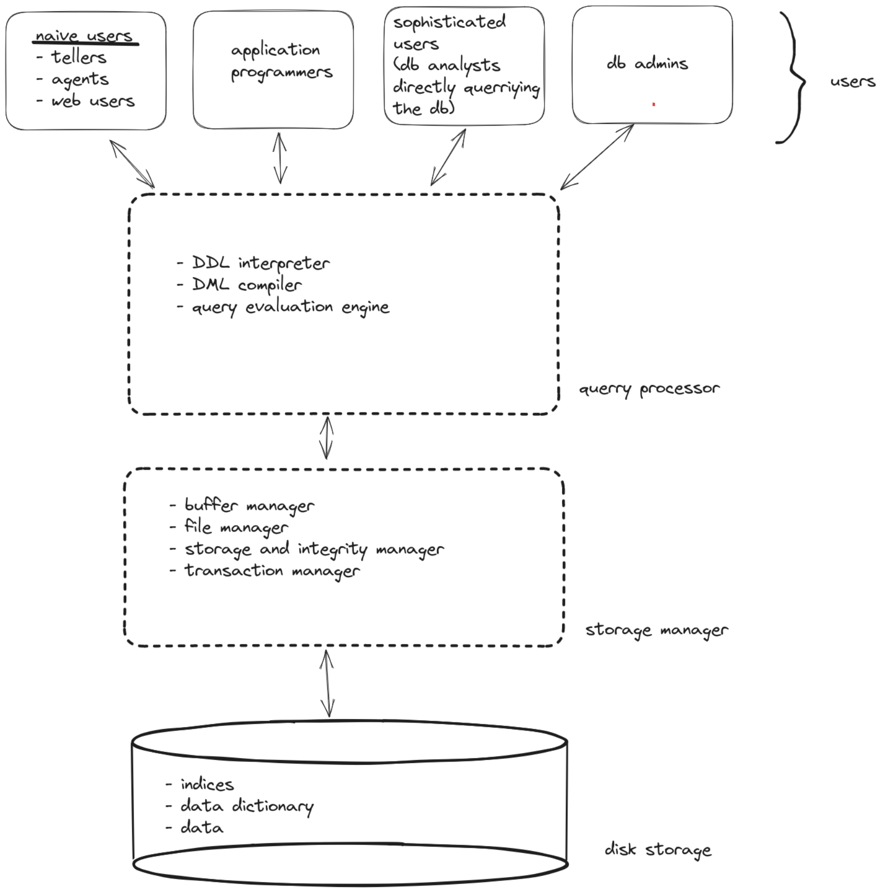
Storage Manager is responsible with the interaction with the file manager of the underlying OS. The raw data is ultimately stored on the disk using the file system of the OS. The storage manager translates the DML statements into low-level file-system commands.

Components of the storage manager:

1. **authorization and integrity manager**
2. **transaction manager**: Realizes concurrent and parallel access to data as well as the grouping sequences of db access querries as single atomic units (transactions). Ensures that DB state is consistent after such operations but also in case of failures. The transaction manager thus realizes
   1. **atomicity**:
   2. **consistency**
   3. **durability**
3. **file manager**: allocation of space on disk and data structures used to represent information stored on disk
4. **buffer manager**: critical part of the DBS, since it is responsible for fetching data from disk storage into main memory and what data to cache in main memory. Enables the db to handle data sized much larger than the size of main memory.

Storage manager also implements several data structures as part of the physical system implementation:

- **data files**: store the DB itself.
- **data dictionary**: metadata about the structure of the db, i.e. the schema. Used by many other components like the querry processor
- **indices**: Just like an index of a book can provide fast access to data by storing actualy physical address of data items having some particular value on some attribute or multiple attributes.


naive users
- tellers
- agents
- web users
application programmers
sophisticated users (db analysts directly querriying the db)
db admins
users
- DDL interpreter
- DML compiler
- query evaluation engine
querry processor
- buffer manager
- file manager
- storage and integrity manager
- transaction manager
storage manager
- indices
- data dictionary
- data
disk storage

12 Introduction to Relational Model and Relational Algebra

12.1 Relational Model Uni DB

- $\texttt{intstructor}(\underline{\texttt{ID}}, \texttt{name}, \texttt{dept\_name} \rightarrow \texttt{department}, \texttt{salary})$
- $\textbf{\texttt{course}}(\underline{\texttt{id}},\ \texttt{title},\ \texttt{dept\_name} \rightarrow\ \texttt{department},\ \texttt{credits})$
- $\textbf{\texttt{prereq}}(\underline{\texttt{course\_id} \rightarrow\ \texttt{course},\ \texttt{prereq\_id} \rightarrow\ \texttt{course}})$
- $\textbf{\texttt{department}}(\underline{\texttt{name}},\ \texttt{building},\ \ \texttt{budget})$
- $\textbf{\texttt{section}}(\underline{\texttt{course\_id}, \texttt{id},\ \texttt{semester},\ \texttt{year},\ }(\texttt{building},\ \texttt{room\_number}) \rightarrow \texttt{classroom},\ \texttt{time\_slot\_id})$
- $\textbf{\texttt{teaches}}(\underline{\texttt{instructor\_ID} \rightarrow\ \texttt{instructor}, (\ \texttt{course\_id},\ \texttt{sec\_id},\ \texttt{semester},\ \texttt{year}) \rightarrow\ \texttt{section}})$
- $\textbf{\texttt{student}}(\underline{\texttt{ID}},\ \texttt{name},\ \texttt{dept\_name} \rightarrow\ \texttt{department},\ \texttt{total\_credit})$
- $\textbf{\texttt{takes}}(\underline{\texttt{student\_ID} \rightarrow \texttt{student}, (\texttt{course\_id},\ \texttt{section\_id},\ \texttt{semester},\ \texttt{year}) \rightarrow \texttt{section}}, \texttt{grade})$
- $\textbf{\texttt{advisor}}(\underline{\texttt{student\_id} \rightarrow\ \texttt{student},\ \texttt{instructor\_id} \rightarrow\ \texttt{instructor}})$
- $\textbf{\texttt{classroom}}(\underline{\texttt{building},\ \texttt{room\_number}},\ \texttt{capacity})$
- $\textbf{\texttt{time\_slot}}(\underline{\texttt{id},\ \texttt{day},\ \texttt{start\_time}},\ \texttt{end\_time})$

12.2 Relational Algebra

12.2.1 Select Operation

- Information of all instructors from the physics department:

$$\sigma_{\texttt{dept\_name="Physics"}}(\texttt{instructor})$$

```
select *
from instructor
where dept_name = 'Physics'
```

Table 12.1: 2 records

id	name	dept_name	salary
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

- Information of all instructors with salaries greater than 90,000 $:

$$\sigma_{\texttt{salary > 90000}}(\texttt{instructor})$$

```
select *
from instructor
where salary > 90000
```

Table 12.2: 2 records

id	name	dept_name	salary
22222	Einstein	Physics	95000
83821	Brandt	Comp. Sci.	92000

- Information about all instructors from the physics department with salaries greater than 90000:

$$\sigma_{\texttt{dept\_name = 'Physics'} \wedge \texttt{salary > 90000}}(\texttt{instructor})$$

```
select *
from instructor
where salary > 90000 and dept_name = 'Physics'
```

Table 12.3: 1 records

id	name	dept_name	salary
22222	Einstein	Physics	95000

- comparison of two different attributes of the **same** relation is possible, e.g. all departments whose name is the same as their building name:

$$\sigma_{\texttt{dept\_name = building}}(\texttt{department})$$

```
select *
from department
where name = building
```

Table 12.4: 0 records

name	building	budget

12.2.2 Project Operation

- list ID, name and salary information of all instructors:

$$\Pi_{\texttt{ID, name, salary}}(\texttt{instructor})$$

```
select i.id, i.name, i.salary
from instructor i
```

Table 12.5: Displaying records 1 - 10

id	name	salary
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000

- expressions of attributes are allowed, e.g. montly salaries:

$$\Pi_{\texttt{ID, name, salary/12}}(\texttt{instructor})$$

```sql
select id, name, salary / 12 as month_salary
from instructor
```

Table 12.6: Displaying records 1 - 10

id	name	month_salary
10101	Srinivasan	5416.667
12121	Wu	7500.000
15151	Mozart	3333.333
22222	Einstein	7916.667
32343	El Said	5000.000
33456	Gold	7250.000
45565	Katz	6250.000
58583	Califieri	5166.667
76543	Singh	6666.667
76766	Crick	6000.000

12.2.3 Composition of Relational Operations

- find the names of all instructors in the Physics department

$$\Pi_{\texttt{name}}(\sigma_{\texttt{dept\_name = 'Physics'}}(\texttt{instructor}))$$

```sql
select name
from instructor
where dept_name = 'Physics'
```

Table 12.7: 2 records

name
Einstein
Gold

12.2.4 Cartesian (Cross) Product

let $r[R]$ and $s[S]$. If $R \cap S = \emptyset$, then $r \times s$ is simply:

$$(r \times s)[R \cup S] := \{t[R \cup S] \mid t[R] \in r \wedge t[S] \in s\}$$

If $R \cap S \neq \emptyset$, equally named attributes must be distinguished. Let

$$R \tilde{+} S := R \oplus S \bigcup_{x \in R \cap S} \{R.x, S.x\}$$

Then,

$$(r \times s)[R \tilde{+} S] := \{t[R \tilde{+} S] \mid t[(R \setminus S) \cup \bigcup_{x \in R \cap S} \{R.x\}] \in t[R] \wedge t[(S \setminus R) \cup \bigcup_{x \in R \cap S} \{S.x\}] \in t[S]\}$$

Problem when $r \times r$. We must use rename.

12.2.5 Rename Operation

A whole relation can be renamed:

$$\beta_s(r)$$

Attributes of a relation can be renamed:

$$\beta_{b_1 \leftarrow a_1, b_2 \leftarrow a_2}(r)$$

Above the attributes a_1 and a_2 of r are renamed to b_1 and b_2.

Using rename we can perform cross product of a relation with itself:

$$\beta_s(r) \times r$$

sql version:

```sql
select *
from r, r as s
```

Example illustrating rename:

- Find the ID and name of all instructors who earn more than the instructor whose ID is 12121:

$$\Pi_{\texttt{i.ID, i.name}}\Big(\sigma_{\texttt{i.salary > wu.salary}}(\beta_{\texttt{i}}(\texttt{instructor} \times \beta_{\texttt{wu}}(\sigma_{\texttt{id = 12121}}(\texttt{instructor}))))\Big)$$

12.2.6 Join Operation

12.2.6.1 Natural Join

for $r[R]$ and $s[S]$ natural join is defined as:

$$r \bowtie s := \{t[R \cup S] \mid t[R] \in r \wedge t[S] \in s\}$$

e.g. `instructor` $\bowtie$ `teaches` gives all information about instructors and courses they teach:

12.2.6.2 θ-Join

General θ-join for a predicate θ is defined as:

$$r \bowtie_\theta s := \sigma_\theta(r \times s)$$

join can be expressed in terms θ-join with appropriate rename and projection operations.

12.2.7 Set Operations

for relations r and s with compatible schemes R and S (compatible means same arities and corresponding domains) simply

- $\texttt{r} \cup \texttt{s}$
- $\texttt{r} \cap \texttt{s}$
- $\texttt{r} \setminus \texttt{s}$

examples:

- courses offered in 2017 fall semester **or** in 2018 spring semester:

$$\Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Fall'} \wedge \texttt{year = 2017}}(\texttt{section})) \cup$$
$$\Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Spring'} \wedge \texttt{year = 2018}}(\texttt{section}))$$

- courses offered in 2017 fall semester **and** in 2018 spring semester:

$$\Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Fall'}\wedge\texttt{year = 2017}}(\texttt{section}))\cap$$
$$\Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Spring'}\wedge\texttt{year = 2018}}(\texttt{section}))$$

- courses offered in 2017 fall semester **but not** in 2018 spring semester:

$$\Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Fall'}\wedge\texttt{year = 2017}}(\texttt{section}))\setminus$$
$$\Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Spring'}\wedge\texttt{year = 2018}}(\texttt{section}))$$

12.2.8 Asssignment

For convenience we can name intermediate results of relational algebraic operations, by assigning them variable names:

$$\texttt{r} := \Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Fall'}\wedge\texttt{year = 2017}}(\texttt{section}))$$
$$\texttt{s} := \Pi_{\texttt{course\_id}}(\sigma_{\texttt{semester = 'Spring'}\wedge\texttt{year = 2018}}(\texttt{section}))$$
$$\texttt{r} \cup \texttt{s}$$

13 Introduction to SQL

13.1 Overview

- **Data-definition language (DDL)**:
  - defining and modifying relation schemas
  - integrity constrains
  - view definition
  - authorization: access rights to relations and views.
- **Data-manipulation language (DML)**: querry information from and modify tuples in relations
- **Transaction Control**: Specifying beginning and end points of transactions.
- **Embedded/dynamic SQL**: how SQL is embedded in general-purpose programmig languages.

13.2 SQL DDL

following can be specified:

- schema for each relation
- set of indices to be maintained for each relation
- security and authorization information for each relation
- physical storage structure of each relation on disk

13.2.1 Basic Types

- **char(n)**: fixed n-long string. (full form: **character(n)**)
- **varchar(n)**: variable-length character string with max length n. (full form **character varying(n)**)
- **int** (full form: **integer**)
- **smallint**
- **numeric(p, d)**: p digits in total, d of the digits after decimal point.
- **real**, **double precision**: floating-point / double precision floating point.
- **float(n)**: Floating-point with precision of at least n digits.

13.2.2 Basic Schema Definition

```
create table department (
    dept_name   varchar(20),
    building    varchar(15),
    budget      numeric(12,2),
    primary key (dept_name) --primary key integrity constraint
);
```

general form:

```
create table r(
    a1 domain1,
    a2 domain2,
    ...
    a_n domain_n
    [integrity constraint 1],
    ...
    [integrity constraing 2] -- these are optional
)
```

13.2.3 Basic Constraints:

- **primary key**: a1, ..., a_n together form the primary key of the relation:

  ```
  primary key(a1, a2, ..., a_n)
  ```

- **foreign key**: a1, ..., a_n together form a foreign key over a relation s, i.e. a1, ..., a_n must be a primary key of some tuple in s (existence: referential integrity)

  ```
  foreign key (a1, a2, ..., a_n) references s
  ```

- **not null**:

  ```
  name varchar(20) not null, --name can not be null
  ```

concrete example:

```
create table instructor(
  ID          varchar(5),
  name        varchar(20) not null, --name can not be null
  dept_name   varchar(20),
  salary      numeric(8,2),
  primary key (ID),
  foreign key (dept_name) references department
)
```

Note that explicitly specifying the primary key of the referenced relation

```
foreign key (dept_name) references department(dept_name)
```

is also possible but not required.

13.2.4 Altering the Schema

- remove a relation completely from the database schema:

  ```
  drop table r; --not to be confused with 'delete from'
  ```

- modify a relation schema:

  ```
  alter table r add a1 a2; -- add the attributes a1, a2 to the
   ↪  relation r
  ```

  ```
  alter table r drop a; -- drop/remove attribute a from the relation r
  ```

  some dbs' don't support dropping single attributes but only whole tables.

13.3 Basic SQL Querries

A typical sql querry has the form

```
select a1, ..., a_n
from r1, ..., r_m
where P -- P is a predicate/condition
```

```
select name
from instructor
```

Table 13.1: Displaying records 1 - 10

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick

```
select dept_name
from instructor
```

Table 13.2: Displaying records 1 - 10

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology

```
select distinct dept_name --removes duplicates
from instructor
```

Table 13.3: 7 records

dept_name
Physics
Biology
Elec. Eng.
Finance
Comp. Sci.
History
Music

Select allows arbitrary expressions of attributes. we can output 10% salary raise for instructors

```
select id, name, dept_name, salary * 1.1
from instructor
```

Table 13.4: Displaying records 1 - 10

id	name	dept_name	?column?
10101	Srinivasan	Comp. Sci.	71500
12121	Wu	Finance	99000
15151	Mozart	Music	44000
22222	Einstein	Physics	104500
32343	El Said	History	66000
33456	Gold	Physics	95700
45565	Katz	Comp. Sci.	82500
58583	Califieri	History	68200
76543	Singh	Finance	88000
76766	Crick	Biology	79200

13.3.1 Where clause.

- Names of all instructors in the CS department who have salary greater than $70,000:

```
select name
from instructor
where dept_name = 'Comp. Sci.'
and salary > 70000
```

Table 13.5: 2 records

name
Katz
Brandt

13.3.2 Joining Tables

- names of instructors, names of their departments and the names of the buildings where departments are located:

```
select i.name, d.name as dept_name, building
from instructor i, department d
where i.dept_name = d.name
```

Table 13.6: Displaying records 1 - 10

name	dept_name	building
Nishimoto	Biology	Watson
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Katz	Comp. Sci.	Taylor
Srinivasan	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor
Singh	Finance	Painter
Wu	Finance	Painter
Roznicki	History	Painter
Califieri	History	Painter

- names of instructors and identifiers of courses they have tought:

```
select i.name, t.course_id
from instructor i, teaches t
where i.id = t.instructor_id
```

Table 13.7: Displaying records 1 - 10

name	course_id
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101

- names of instructors from the CS department and identifiers of courses that they have tought:

```
select i.name, t.course_id
from instructor i, teaches t
where i.id = t.instructor_id and i.dept_name = 'Comp. Sci.'
```

Table 13.8: 8 records

name	course_id
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Katz	CS-101
Katz	CS-319
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319

13.3.3 Renaming

```
select i.name as instructor_name, t.course_id -- as can be omitted
from instructor as i, teaches as t -- as can be ommited
where i.id = t.instructor_id
```

Table 13.9: Displaying records 1 - 10

instructor_name	course_id
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101

Renaming useful when comparing tuples in the same relation:

- names of instructors whose salary is greater than at least one instructor in the biology department ≈ names of all instructors who earn more than the lowest paid instructor in the Biology department:

```
select distinct i.name
from instructor i , instructor i2
where i.salary > i2.salary and i2.dept_name = 'Biology'
```

Table 13.10: 8 records

name
Brandt
Gold
Wu
Kim
Nishimoto
Katz
Einstein
Singh

- **correlation name = table alias = tuple variable**

13.3.4 String Operations

- concatenation with `||`

```
values
('hey' || ' there!')
```

Table 13.11: 1 records

column1
hey there!

- `upper()` and `lower()`

```
values
(upper('hey there')),
(lower('HEY THERE'))
```

Table 13.12: 2 records

column1
HEY THERE
hey there

- removing spaces with `trim()`

```
values
(trim('hey    ' ) || ' there!')
```

Table 13.13: 1 records

column1
hey there!

13.3.5 Pattern Matching

- `%`: matches any string
- `_`: matches any character

- examples:
  - `intro%`: any string beginning with 'Intro'
  - `%Comp%`: any string containing 'Comp' as a substring
  - `___`: any string of exactly three characters
  - `___%`: any string of at least three characters
- concrete example; information of courses that have 'comp' as a substring in their title:

```
select *
from course c
where c.title ilike '%comp%' --ilike is case insensitive like
```

Table 13.14: 2 records

id	title	dept_name	credits
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4

*Escaping special characters like "%" with "":

```
select *
from (
    values
    ('%15')
) r(a)
where a like '\%%'
```

Table 13.15: 1 records

a
%15

- Defining custom escape characters other than "":

```
select *
from (
    values
    ('%15')
) r(a)
```

```
where a like '^%%' escape '^' -- '^' is defined as the escape character
```

Table 13.16: 1 records

a
%15

13.3.6 Ordering Display of Tuples

- ordering

```
select name
from instructor
where dept_name  = 'Physics'
order by name;
```

Table 13.17: 2 records

name
Einstein
Gold

- ordering order, multiple attributes

```
select *
from instructor
order by salary desc, name asc
```

Table 13.18: Displaying records 1 - 10

id	name	dept_name	salary
30765	Green	Music	NA
22222	Einstein	Physics	95000
83821	Brandt	Comp. Sci.	92000
12121	Wu	Finance	90000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

id	name	dept_name	salary
76543	Singh	Finance	80000
45565	Katz	Comp. Sci.	75000
10032	Nishimoto	Biology	73120
76766	Crick	Biology	72000

13.3.7 Between

```
SELECT name
from instructor i
where i.salary BETWEEN 90000 and 100000
```

Table 13.19: 3 records

name
Wu
Einstein
Brandt

13.3.8 Tuples in Where Predicates

```
SELECT i.name, t.course_id
from instructor i, teaches t
where i.id = t.instructor_id and i.dept_name = 'Biology'
```

is equivalent to

```
SELECT i.name, t.course_id
from instructor i, teaches t
where (i.id, i.dept_name) = (t.instructor_id, 'Biology')
```

Table 13.20: 2 records

name	course_id
Crick	BIO-101
Crick	BIO-301

13.4 Set Operations

Set operations eliminate duplicates by default. Duplicates retained by `all` keyword.

13.4.1 Union

corresponds to $\cup$.

- courses offered in 2017 Fall **or** 2018 Spring:

```
(
    select  course_id
    from section
    where semester = 'Fall' and year = 2017
)
UNION
(
    select course_id
    from section
    where semester = 'Spring' and year = 2018
)
```

Table 13.21: 8 records

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

13.4.2 Intersect

Corresponds to $\cap$.

- Courses offered in **both** Fall 2017 and Spring 2018:

```
(
    select  course_id
    from section
    where semester = 'Fall' and year = 2017
)
intersect
(
    select course_id
    from section
    where semester = 'Spring' and year = 2018
)
```

Table 13.22: 1 records

course_id
CS-101

13.4.3 Except

Corresponds to \.

- courses offered in the Fall 2017 but not in Spring 2018:

```
(
    select  course_id
    from section
    where semester = 'Fall' and year = 2017
)
except
(
    select course_id
    from section
    where semester = 'Spring' and year = 2018
)
```

Table 13.23: 2 records

course_id
PHY-101

course_id
CS-347

- Illustrating how `all` works:

`union all`:

```
with r(a) as (
    values
    (1),
    (1)
), s(a) as (
    values
    (1)
)
(select *
from r )
union  all
(select *
from s)
```

Table 13.24: 3 records

a
1
1
1

`intersect all`:

```
with r(a) as (
    values
    (1),
    (1),
    (1)
), s(a) as (
    values
    (1),
    (1)
)
```

```
(select *
from r )
INTERSECT all
(select *
from s)
```

Table 13.25: 2 records

a
1
1

`except all`:

```
with r(a) as (
    values
    (1),
    (1)
), s(a) as (
    values
    (1)
)
(select *
from r )
except  all
(select *
from s)
```

Table 13.26: 1 records

a
1

```
with r(a) as (
    values
    (1),
    (1)
), s(a) as (
    values
    (1)
)
(select *
from s )
except  all
(select *
from r)
```

Table 13.27: 0 records

a

13.5 Null

- `null` represents values that are not known. (can be anything).
- arithmetic expressions involving `null` produce `null`:

```
values
(1 + null)
```

Table 13.28: 1 records

column1
NA

- boolean expressions and predicates involving `null` (other than `is [not] null`) have a special truth value `unknown`.
  - e.g. `(1 < unknown) = uknown`
- truth tables for `unknown`:

p	q	p and q
`false`	`unknown`	`false`
`true`	`unknown`	`unknown`

p	q	p or q
`false`	`unknown`	`unknown`
`true`	`unknown`	`true`

`not unknown = uknown`

- tuples that evaluate to `uknown` in the `where` clause are not included in the result (just like the ones that evaluate to `false`)
- we can test if a value is or isn't null:

```
select a
from (
    values
    (1, 3),
    (2, null)
) r(a, b)
where b is null
```

Table 13.31: 1 records

a
2

```
select a
from (
    values
    (1, 3),
    (2, null)
) r(a, b)
where b is not null
```

Table 13.32: 1 records

a
1

- we can test whether the result of a predicate is or isn't `uknown`

```
select a
from (
    values
    (1, 3),
    (2, null)
) r(a, b)
where b > 2 is unknown
```

Table 13.33: 1 records

a
2

```
select a
from (
    values
    (1, 3),
    (2, null)
) r(a, b)
where b > 2 is not unknown
```

Table 13.34: 1 records

a
1

13.6 Aggregate Functions

- Take collection (i.e. aggregate, multiset) of values as input and return a **single** value.
- Built in aggregate functions in sql:
    - `avg`: input must be numeric
    - `min`
    - `max`
    - `sum`: computes the total sum of the values in the aggregate, input must be numeric
    - `count`

13.6.1 Basic Aggregation

- Average salary of instructors in the CS department:

```
select avg(salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.'
```

Table 13.35: 1 records

avg_salary
77333.33

Duplicates are retained. Retention of duplicates is obviously important for calculating averages. But sometimes we may want to eliminate duplicates:

- Find the total number of instructors that teach a course in the Spring 2018 semester:

```
select count(distinct t.instructor_id)
from teaches t
where semester = 'Spring' and year = 2018
```

Table 13.36: 1 records

count
6

as opposed to

```
select count(t.instructor_id)
from teaches t
where semester = 'Spring' and year = 2018
```

Table 13.37: 1 records

count
7

- counting all attributes with `count(*)`:

```
select count(*)
from course
```

Table 13.38: 1 records

count
13

`count(*)` retains `null` values. It is the only aggregate function that does so. All other aggregate functions ignore `null` values, including count functions applied on a single attribute. That is `count(*)` and `count(attr)` are different:

```
select count(*) as count_star, count(a) as count_attribute
from (
    values
    (1),
    (null)
) r(a)
```

Table 13.39: 1 records

count_star	count_attribute
2	1

13.6.2 Aggregation with Grouping

Instead of applying the aggregate function to a single aggregate (collection), we can apply it to multiple aggregates/collections, that consist of tuples grouped together w.r.t certain grouping attributes:

- find the average salary in each department:

```
select dept_name, avg(salary) as avg_dept_salary
from instructor
group by dept_name
order by avg(salary) desc
```

Table 13.40: 7 records

dept_name	avg_dept_salary
Physics	91000.00
Finance	85000.00
Elec. Eng.	80000.00
Comp. Sci.	77333.33
Biology	72560.00
History	59666.67
Music	40000.00

- find number of instructors working in each department:

```
select dept_name, count(*) cnt
from instructor
group by dept_name
order by dept_name
```

Table 13.41: 7 records

dept_name	cnt
Biology	2
Comp. Sci.	3
Elec. Eng.	1
Finance	2
History	3
Music	2
Physics	2

- find the number of instructors in each department who teach a course in the Spring 2018 semester:

```
select i.dept_name, count(distinct i.id) as instr_count
from instructor i , teaches t
where i.id = t.instructor_id
and t.semester = 'Spring' and t.year = 2018
group by i.dept_name
```

Table 13.42: 4 records

dept_name	instr_count
Comp. Sci.	3
Finance	1
History	1
Music	1

Warning

Note that only attributes allowed to appear in the select clause (other than the attribute being aggregated) are the attributes used in the `group by` clause. Thus

```
select a, X, avg(b) -- X doesn't appear in the group by clause below
from r -- some relation r
where P -- some predicate
group by a
```

it not legal.

13.6.3 Having Clause

Specifies a condition that applies to **groups** rather than to tuples.

- departments where average salayr is morethan $42,000:

```
select i.dept_name, avg(i.salary)
from instructor i
group by dept_name
having avg(i.salary) > 42000
order by avg(i.salary) desc
```

Table 13.43: 6 records

dept_name	avg
Physics	91000.00
Finance	85000.00
Elec. Eng.	80000.00
Comp. Sci.	77333.33

dept_name	avg
Biology	72560.00
History	59666.67

Like with `select` the attributes that are allowed in the `having` clause are either present in the `group by` clause, it is the attribute that is being aggregated.

13.6.4 Semantics of Group by and Having

Can be understood roughly as:

1. `from` is evaluated to get a relation
2. `where` predicate is applied on each tupel to get a new relation
3. tupels that agree on values of those attributes listed in the `group by` clause are placed into groups.
4. `having` clause applied to each group, the ones that satisfy it are retianed to obtain a new relation
5. `select` clause is applied to the relation to obtain the resulting relation.

A query with both `where` and `having`:

- for each course section offered in 2017, find the average total credits (`tot_cred`) of all students enrolled in the section, if the section has at least 2 students:

```
select t.course_id, t.sec_id, t.semester, avg(s.tot_cred)
from takes t, student s
where t.student_id = s.id
and t."year" = 2017
group by course_id, sec_id, semester
having count(s.id) > 1;
```

Table 13.44: 3 records

course_id	sec_id	semester	avg
CS-101	1	Fall	65
CS-190	2	Spring	43
CS-347	1	Fall	67

Aggregation with null

All aggregate functions except of `count(*)` ignore `null` values

Aggregation of Boolean Values

- the aggregate function `some()` can be applied to an aggregate consisting of boolean values to compute the disjunction of these values
- the aggregate function `every()` can be applied to an aggregate consisting of boolean values to compute the conjunction of the values:

```
select every(a)
from (
    values
    (true),
    (true)
) r(a)
```

Table 13.45: 1 records

every
TRUE

13.7 Nested Subqueries

Nested subqueries are used for:

- test for set membership with `[not] in`
- make set comparisons
- determine set cardinality by nesting queries in the `where` clause

Queries can be nested in the

- `where` clause
- `from` clause

Simple example for a subquery:

```
-- names of departments and the average pay
-- where average pay in that department is above the overall average pay
select dept_name, avg(i.salary) as avg_dep_pay
from instructor i
group by i.dept_name
having avg(i.salary) > (
    select avg(i2.salary)
    from instructor i2
)
order by avg(i.salary) desc
```

Table 13.46: 4 records

dept_name	avg_dep_pay
Physics	91000.00
Finance	85000.00
Elec. Eng.	80000.00
Comp. Sci.	77333.33

13.7.1 Testing for Set Membership

Reconsider the queries

- Find all the courses taught in **both** Fall 2017 **and** Spring 2018 semesters
- Find all the courses taught in Fall 2017 **but not** in Spring 2018

Previously we used set operations. Now we can use `in`:

```
select distinct s.course_id
from "section" s
where s.semester = 'Fall' and s."year" = 2017
and s.course_id in (
    select s.course_id
    from "section" s
    where s.semester = 'Spring' and s."year" = 2018
)
```

Table 13.47: 1 records

course_id
CS-101

```
select distinct s.course_id
from "section" s
where s.semester = 'Fall' and s."year" = 2017
and s.course_id not in (
    select s.course_id
    from "section" s
    where s.semester = 'Spring' and s."year" = 2018
)
```

Table 13.48: 2 records

course_id
CS-347
PHY-101

where

```
select course_id
from section
where semester = 'Spring' and year = 2018
```

is a nested subquery.

`distinct` is used since set operations remove duplicates by default.

13.7.2 Enumerated Sets

`[not] in` can be used on enumerated sets:

- Names of al instructors other than Mozart and Einstein:

```
select distinct name
from instructor i
where name not in ('Mozart', 'Einstein')
```

Table 13.49: Displaying records 1 - 10

name
Singh
Srinivasan
Crick
Green
Brandt
Gold
Roznicki
Califieri
Wu
Kim

Warning

enumerated sets shouldn't be confused with tupels

13.7.3 Set Comparison

Reconsider the query:

- Names of all instructors whose salary is grater than at least one instructor in the Biology department:

Previously we used the somewhat awkward solution:

```
select i.name
from instructor i , instructor i2
where i2.dept_name = 'Biology'
and i.salary > i2.salary
```

Now we can express this much more similar to natural language:

```
select i.name
from instructor i
where i.salary > some(
    select salary
    from instructor
    where dept_name = 'Biology'
)
```

Table 13.50: 8 records

name
Wu
Einstein
Gold
Katz
Singh
Brandt
Kim
Nishimoto

Contrast:

with cross product	with set comparison
`select i.name`	`select i.name`
`from instructor i , instructor i2`	`from instructor i`
`where i2.dept_name = 'Biology'`	`where i.salary > some(`
`and i.salary > i2.salary`	`    select salary`
	`    from instructor`
	`    where dept_name =`
	`     ↪  'Biology'`
	`)`

Consider another query:

- Names of instructors that have a salary greater than that of any/each instructor in the Biology department:

```
select i."name"
from instructor i
where i.salary > all (
    select salary
    from instructor
    where dept_name = 'Biology'
)
```

Table 13.52: 7 records

name
Wu
Einstein
Gold
Katz
Singh
Brandt
Kim

Warning

- `= some (...)` is identical to `in (...)`
- `<> some (...)` is **not** identical to `not in (...)`
- `<> all (...)` is identical to `not in (...)`
- `= all(...)` is **not** identical to `in (...)`

- name of the department with the highest average salary:

```
select dept_name
from instructor i
group by i.dept_name
having avg(i.salary) >= all (
    select avg(salary)
    from instructor
    group by dept_name
)
```

Table 13.53: 1 records

dept_name
Physics

13.7.4 Testing for Empty Relations

We can test whether a query has any tuples in its result (whether if it's non-empty) with `exists`. Reconsider the query

- All courses taught both in the Fall 2017 and Spring 2018:

```
select course_id
from "section" s
where s.semester = 'Fall' and s."year" = 2017
and exists (
    select *
    from "section" s2
    where s2.semester = 'Spring' and s2."year" = 2018
    and s.course_id = s2.course_id
)
```

Table 13.54: 1 records

course_id
CS-101

Above a **correlation name** from the outer querry has been used in the inner querry. This can equivalently be achieved with a usual join operation:

```
select s.course_id
from "section" s , "section" s2
where s.semester = 'Fall' and s."year" = 2017
and s2.semester = 'Spring' and s2."year" = 2018
and s.course_id = s2.course_id
```

Non-existence can be queried with `not exists`. For example we can use it to simulate **set containment**.

$$B \subseteq A \equiv \texttt{not exists (B except A)}$$

- All students who have taken all courses offered in the Biology department:

```
select s.id, s."name"
from student s
```

```
where not exists (
    (select c.id
    from course c
    where c.dept_name = 'Biology')
        except
    (select t.course_id
    from takes t
    where s.id = t.course_id)
)
```

Table 13.55: 0 records

id	name

The subquerry

```
select c.id
from course c
where c.dept_name = 'Biology'
```

finds all courses offered in the Biology department. The subquerry:

```
select t.course_id
from takes t
where s.id = t.course_id
```

finds all courses that the student 's' has taken.

Consider the query:

- How many students have taken a course by instructor with the ID '10101'

We can construct this query in three different ways:

- With the usual join operation:

```
select count(distinct t.student_id)
from takes t, teaches t2
where t.course_id = t2.course_id
    and t.sec_id = t2.sec_id
    and t.semester = t2.semester
    and t."year" = t2."year"
    and t2.instructor_id = '10101'
```

- With set membership test using `in` with a tuple instead of a single attribute

```
select count(distinct t.student_id)
from takes t
where (t.course_id, t.sec_id, t.semester, t."year") in (
    select t2.course_id , t2.sec_id , t2.semester , t2."year"
    from teaches t2
    where t2.instructor_id = '10101'
)
```

- with the `exists` construct

```
select count(distinct t.student_id)
from takes t
where exists (
    select t2.course_id , t2.sec_id , t2.semester , t2."year"
    from teaches t2
    where t.course_id = t2.course_id
    and t.sec_id = t2.sec_id
    and t.semester = t2.semester
    and t."year" = t2."year"
    and t2.instructor_id = '10101'
)
```

Table 13.56: 1 records

count
6

13.7.5 Test for Absence of Duplicates / Test for Uniqueness

Testing if subquery has duplicate tuples with `unique`:

- courses that were offered at most once in 2017:

```
select c.id
from course c
where unique (
    select
    from "section" s
    where c.id = s.course_id
    and s."year" = 2017
)
```

Since `unique` not implemented in Postgresql, we can simulate it as follows:

```
select c.id
from course c
where 1 >= (
    select count(s.course_id)
    from "section" s
    where s."year" = 2017
    and c.id = s.course_id
)
```

Table 13.57: Displaying records 1 - 10

id
BIO-101
BIO-301
BIO-399
CS-101
CS-315
CS-319
CS-347
EE-181
FIN-201
HIS-351

(For some reason this delivers a different result than two solutions below. Why?)

Another solution using aggregate functions:

```
select c.id
from course c , "section" s
```

```
where c.id = s.course_id
and s."year" = 2017
group by c.id
having count(*) <= 1
```

Table 13.58: 5 records

id
BIO-101
CS-101
CS-347
EE-181
PHY-101

Yet another far less elegant solution:

- First we find courses offered more than once in 2017:

```
select course_id
from "section" s
where s."year" = 2017
    except all
select distinct course_id
from "section" s
where s."year" = 2017
```

Table 13.59: 1 records

course_id
CS-190

- Than courses offered at most once in 2017:

```
select distinct course_id
from "section" s
where s."year" = 2017
and course_id not in (
    select course_id
from "section" s
where s."year" = 2017
    except all
select distinct course_id
from "section" s
where s."year" = 2017
)
```

Table 13.60: 5 records

course_id
BIO-101
CS-101
CS-347
EE-181
PHY-101

13.7.6 Subqueries in the From Clause

Since relations appear in the `from` clause, there is nothing preventing them being subqueries. Reconsider the query

- Average instructor salaries per department, where the average salary in that department is greater than 42000:

Previously we solved using `group by` and `having` - which is the natural way:

```
select i.dept_name, avg(i.salary) avg_salary
from instructor i
group by i.dept_name
having avg(i.salary) > 42000
```

We can re-write it without `having` using subquery in the `from` clause:

```
select dept_name, avg_salary
from (
    select i.dept_name, avg(i.salary)
    from instructor i
    group by i.dept_name
) dep_salaries(dept_name, avg_salary) --table alias
where avg_salary > 42000
```

Table 13.61: 6 records

dept_name	avg_salary
Physics	91000.00
Biology	72560.00
Elec. Eng.	80000.00
Finance	85000.00
Comp. Sci.	77333.33
History	59666.67

Contrast:

with `having`

```sql
select i.dept_name, avg(i.salary) avg_salary
from instructor i
group by i.dept_name
having avg(i.salary) > 42000
```

with nested query in `from` clause

```sql
select dept_name, avg_salary
from (
    select i.dept_name, avg(i.salary)
    from instructor i
    group by i.dept_name
) dep_salaries(dept_name, avg_salary) --table alias
where avg_salary > 42000
```

Another example:

- max total salary in a department across all departments:

```sql
select max(tot_sal) as max_tot_sal
from (
    select sum(i.salary)
    from instructor i
    group by i.dept_name
) r(tot_sal) -- table alias
```

Table 13.63: 1 records

max_tot_sal
232000

Note

Correlation variables are allowed in a `from` subquery using the `lateral` keyword.

- names of instructors, their salaries, alongside with the average salary of their department:

```
select i.name, i.salary , i.dept_name, avg_dep_salary
from instructor i , lateral (
    select avg(i2.salary)
    from instructor i2
    where i.dept_name = i2.dept_name
) r(avg_dep_salary)
```

Table 13.64: Displaying records 1 - 10

name	salary	dept_name	avg_dep_salary
Srinivasan	65000	Comp. Sci.	77333.33
Wu	90000	Finance	85000.00
Mozart	40000	Music	40000.00
Einstein	95000	Physics	91000.00
El Said	60000	History	59666.67
Gold	87000	Physics	91000.00
Katz	75000	Comp. Sci.	77333.33
Califieri	62000	History	59666.67
Singh	80000	Finance	85000.00
Crick	72000	Biology	72560.00

13.7.7 With Clause

Defines temporary relations whose definition is available only in the query in which `with` clause occurs.

- Departments with maximum budget

```
with max_budget(value) as (
    select max(d.budget)
    from department d
)
select d."name" , d.budget as budget
from department d, max_budget mb
where d.budget = mb.value
```

Table 13.65: 1 records

name	budget
Finance	120000

Alternatively we can use nested subquerries in the `with` clause or in the `from` clause:

- as a simple subquery in `where` clause:

```
select d."name" , d.budget
from department d
where d.budget = (
    select max(d2.budget)
    from department d2
)
```

- using `all` with a `where`-clause subquery:

```
select d.name, d.budget
from department d
where d.budget >= all (
    select budget
    from department d2
)
```

- as `from`-clause subquery:

```
select d.name, d.budget
from department d, (
    select max(d2.budget)
    from department d2
) bd(val)
where d.budget = bd.val
```

Using with improves readability. Consider another example:

- departments where total salary greater than the average of the total salary of all departments:

```
with dep_tot_salary(d_name, t_salary) as (
    select i.dept_name, sum(i.salary)
    from instructor i
    group by i.dept_name
)
select *
from dep_tot_salary
where t_salary > (
    select avg(t_salary)
    from dep_tot_salary
)
```

Table 13.66: 4 records

d_name	t_salary
Physics	182000
Finance	170000
Comp. Sci.	232000
History	179000

13.7.8 Scalar Subqueries

Queries returning one single tuple with one attribute are called **scalar**. Such queries can be used in place of values as **scalar subquerries**, even in `select`-clause:

- all departments along with number of instructors in each department

```
select d."name" ,
       ( -- scalare unteranfragen duerfen innerhalb select eingesetzt
 ↪   werden
        select count(*)
        from instructor i
        where d."name" = i.dept_name --d is a correlatin variable
       ) cnt
from department d
order by cnt desc
```

Table 13.67: 7 records

name	cnt
History	3
Comp. Sci.	3
Finance	2
Biology	2
Physics	2
Music	2
Elec. Eng.	1

Of course this can simply be achieved with grouping:

```
select i.dept_name , count(*)
from instructor i
group by i.dept_name
order by count(*) desc
```

Table 13.68: 7 records

dept_name	count
History	3
Comp. Sci.	3
Physics	2
Finance	2
Biology	2
Music	2
Elec. Eng.	1

Alternatively as subquery of `from` with `lateral`:

```
select d."name", res.val as cnt
from department d , lateral (
    select count(*)
    from instructor i
    where i.dept_name = d."name"
) res(val)
order by cnt desc
```

Scalar subqueries can occur in `select`, `where` and `having` clauses.

13.7.9 Queries Without From

- with `values`

```
values
(1, 'a'),
(2, 'b')
```

Table 13.69: 2 records

column1	column2
1	a
2	b

- with `select`

```
select
(1), ('a')
  union all
select
(2), ('b')
```

Table 13.70: 2 records

?column?	?column?..2
1	a
2	b

13.8 Modifying the Database

- adding - `insert into R`
- removing - `delete from R`
- changing - `update R set`

infromation from the database as opposed to querying.

13.8.1 Deletion

Expressed just like a query,

- delete tuples from r satisfying condition P:

```
delete from r -- relation
where P -- predicate
```

- delete all tuples from instructor:

```
delete from instructor;
```

or equivalently

```
delete from instructor
where true
```

The predicate can be arbitrarily complex,

- Delete all instructors that work in the Watson building:

```
delete from instructor
where dept_name in (
  select dept_name
  from department
  where building = 'Watson'
)
```

`delete` deletes tupels from a single relation, but we can still reference any number of relations nested in `select`, `from` and `where`, including the one that we are deleting from. Consider:

- delete all instructors that earn less than the overall average instructor salary

```
delete from instructor
where salary < (
  select avg(salary)
  from instructor
)
```

- delete(which is anoter way of saying 'fire') all instructors that haven't taught in the year 2018

```
delete from instructor i
where i.id not in (
  select t.instructor_id
  from teaches t
  where t."year" = 2018
)
```

above a relation other than the one being deleted was referenced in the nested subquery in the `where` clause, demonstrating we can reference arbitrary relations.

> Note
>
> deletions are performed **after** the tupels are filtered (the ones that pass the test). Otherwise the result could be influenced.

13.8.2 Insertion

Any query result delivering a collection of tuples can be inserted into a relation, as long as they agree with the cardinality and domains of the attributes of the relation:

- providing tuples explicitetly using `values`:

```
insert into course
values
('CS-437', 'Database Systems', 'Comp. Sci.', 4),
('PHY-201', 'Intro. Theo. Phys.', 'Physics', 6)
```

- or using `select`:

```
insert into course
select
```

```
('CS-437'), ('Database Systems'), ('Comp. Sci.'), (4)
  union
select
('PHY-201'), ('Intro. Theo. Phys.'), ('Physics'), (6)
```

- we can explicitely specify the order of the attributes:

```
insert into course(title, id, credits, dept_name)
  values ('Database Systems', 'CS-437', 4, 'Comp. Sci.')
```

- even omit some of the attributes from the specification (ommited attributes are set automatically as `null`):

```
insert into course (title, id)
  values ('Baking Cakes', 'BK-101')
```

Above `dept_name` and `credit` are set automatically as `null`.

More generally, results of an arbitary query can be inserted as well. Consider:

- make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

```
insert into instructor
  select s.id, s."name", s.dept_name, 18000
  from student s
  where s.dept_name = 'Music' and tot_cred > 144;
```

- insert a copy of a relation to itself:

```
insert into s
  select *
  from s
```

Note

Importantly in the above example:

```
insert into s
  select *
  from s
```

the query is evaluated before the insertion is performed. Otherwise we could face infinite loops. Also note that this operation is only possible if the relation s has no primary key defined. Otherwise duplicates are not allowed.

13.8.3 Updates

Changing some values of tuples is possible with `update`. Consider:

- Increase the salaries of all instructors by 5%

```
update instructor
set salary = salary * 1.05
```

- Raise the salary only of those instructors with pay less than $70,000

```
update instructor
set salary = salary * 1.05
where salary < 70000
```

Nested subqueries are allowed in the `where` clause, referencing the relation being updated or other arbitrary relations. Consider:

- give a 5% salary raise to instructors whose salary is less than overall average salary:

```
update instructor
set salary = salary * 1.05
where salary < (
  select avg(salary)
  from instructor
)
```

- give a 5% salary raise to instructors that have taugh more than 1 course in 2018:

```
update instructor i
set i.salary = i.salary * 1.05
```

```
where 1 < (
    select count(*)
    from teaches t
    where t.instructor_id = i.id
    and  t."year" = 2018
)
```

- give 3% raise to instructors with salary over $100,000, 5% to others.

One solution is to write to update statements:

```
update instructor
set salary = salary * 1.03
where salary > 100000
```

```
update instructor
set salary = salary * 1.05
where salary <= 100000
```

Warning

in the above solution the order of statements is important, otherwise we could end up giving a 8% raise to instructors whose salary is 100000 or just below it.

Alternatively use `case` statement. Then the order won't be important.

```
update instructor
set salary =
  case
    when salary <= 100000 then salary * 1.05
    else salary * 1.03
  end
```

Note

General syntax of case statement:

```
case
  when P1 then res1 -- P1 is a predicate
  when P2 then res2
  ...
  when Pn then res_n
  else res0
end
```

since case statement is an expression that is evaluated to a value, it can be used any place where a value is used.

Complex subqueries can follow after `set` clause:

- set total credit of each student to the sum of the credits of courses successfully completed by the student. (Grade is not an 'F' or `null`)

```
update student
set tot_cred = (
    select sum(c.credits)
    from takes t, course c
    where t.course_id = c.id
        and t.student_id = student.id
        and t.grade is not null
        and t.grade not like 'F'
)
```

For student that haven't successfully completed a course, total credit will be set to null. Instead we may set it to 0 using `case`:

```
update student
set tot_cred = (
  case
    when select sum(c.credits) is not null then sum(credits)
    else 0
    from takes t, course c
    where t.course_id = c.id
        and t.student_id = student.id
        and t.grade is not null
        and t.grade not like 'F'
)
```

Equivalently, this can be ahieved by `coalesce` provided by some DBs.

```
update student
set tot_cred = (
    select coalesce(sum(c.credits), 0)
    from takes t, course c
    where t.course_id = c.id
        and t.student_id = student.id
        and t.grade is not null
        and t.grade not like 'F'
)
```

where `coalesce(x, y)` evaluates to x, if x is not `null`, and to y otherwise.

14 Intermediate SQL

- `join` Expressions
- view definition
- integrity constraints
- more details regarding data definition
- authorization

14.1 Join Expressions

14.1.1 Natural join

corresponds to $\bowtie$. Consider:

```
select *
from (
    values
    (1, 2),
    (3, 2)
) r(a, b) natural join
(
    values
    (2, 3),
    (2, 4),
    (1, 10),
    (5, 8)
) s(b, c)
```

Table 14.1: 4 records

b	a	c
2	3	3
2	1	3
2	3	4

b	a	c
2	1	4

14.1.2 Join Using

Possible to explicitely list attributes on which it is to be joined:

```
select *
from (
    values
    (1, 2, 5),
    (3, 2, 7)
) r(a, b, c) join
(
    values
    (2, 3),
    (2, 4),
    (1, 10),
    (5, 8)
) s(b, c) using (b)
```

Table 14.2: 4 records

b	a	c	c..4
2	3	7	3
2	1	5	3
2	3	7	4
2	1	5	4

same as

```
select *
from r, s
where r.b = s.b
```

14.1.3 Join on

```
select *
from student join takes on student.id = takes.student_id
```

Table 14.3: Displaying records 1 - 10

id	name	dept_name	tot_cred	student_id	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	00128	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	00128	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	12345	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	12345	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	12345	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	12345	CS-347	1	Fall	2017	A
19991	Brandt	History	80	19991	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	23121	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	44553	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	45678	CS-101	1	Fall	2017	F

Exactly the same as

```
select *
from student, takes
where student.id = takes.student_id
```

Advantages:

- readability in separating join conditions from other `where`-clause conditions
- **necessary** for outer joins.

14.1.4 Outer Joins

According to the definition of usual join operation

```
select *
from student join takes
on student.id = takes.student_id
```

non-matched tuples don't appear in the resulting relation. But sometimes we might nevertheless wich to include such tuples in the result.

This can be achieved by

- `left outer join`
- `right outer join`
- `full outer join`

First of all, notice that there is one student who hasn't taken any courses:

```
select *
from student s
where s.id not in (
    select t.student_id
    from takes t
)
```

Table 14.4: 2 records

id	name	dept_name	tot_cred
70557	Snow	Physics	0
12789	Newman	Comp. Sci.	NA

An he doesn't appear in the previous join operation. We can include this student in the final result using `left outer join`:

```
select *
from student s left outer join takes t
on s.id = t.student_id
order by s.id desc
```

Table 14.5: Displaying records 1 - 10

id	name	dept_name	tot_cred	student_id	course_id	sec_id	semester	year	grade
98988	Tanaka	Biology	120	98988	BIO-301	1	Summer	2018	NA

id	name	dept_name	tot_cred	student_id	course_id	sec_id	semester	year	grade
98988	Tanaka	Biology	120	98988	BIO-101	1	Summer	2017	A
98765	Bourikas	Elec. Eng.	98	98765	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	98765	CS-315	1	Spring	2018	B
76653	Aoi	Elec. Eng.	60	76653	EE-181	1	Spring	2017	C
76543	Brown	Comp. Sci.	58	76543	CS-319	2	Spring	2018	A
76543	Brown	Comp. Sci.	58	76543	CS-101	1	Fall	2017	A
70557	Snow	Physics	0	NA	NA	NA	NA	NA	NA
55739	Sanchez	Music	38	55739	MU-199	1	Spring	2018	A-
54321	Williams	Comp. Sci.	54	54321	CS-101	1	Fall	2017	A-

Notice how the tuple corresponding to student Snow who hasn't taken any courses has `null` values for all attributes that come from the `takes` relation.

Consider a simpler exampe. Consider tables:

```
with r(a, b) as (
    values
    (1, 2),
    (2, 2),
    (5, 3)
), s(b, c) as (
    values
    (2, 4),
    (2, 5)
)
select *
from r natural join s
```

Table 14.6: 4 records

b	a	c
2	1	5
2	1	4

b	a	c
2	2	5
2	2	4

Above `natural join` didn't include the tuple `(5, 3)` from `r`, since it didn't match with anything from s. But

```
with r(a, b) as (
    values
    (1, 2),
    (2, 2),
    (5, 3)
), s(b, c) as (
    values
    (2, 4),
    (2, 5)
)
select r.a a, r.b b, s.c c
from r natural left outer join s
```

Table 14.7: 5 records

a	b	c
1	2	5
1	2	4
2	2	5
2	2	4
5	3	NA

contains tuple `(5, 3, null)`.

`inner join` is just another name for all other default joins that don't include nonmatched tuples.

Alternative way to find students that haven't taken a course using `outer join`:

```
select *
from student s left outer join takes t
on s.id = t.student_id
where t.course_id is null
```

Table 14.8: 2 records

id	name	dept_name	tot_cred	student_id	course_id	sec_id	semester	year	grade
70557	Snow	Physics	0	NA	NA	NA	NA	NA	NA
12789	Newman	Comp. Sci.	NA	NA	NA	NA	NA	NA	NA

`right outer join` is symmetric to `left outer join`. Adding `(4, 7)` to s in the previous simple example:

```
with r(a, b) as (
    values
    (1, 2),
    (2, 2),
    (5, 3)
), s(b, c) as (
    values
    (2, 4),
    (2, 5),
    (4, 7)
)
select *
from r natural right outer join s
```

Table 14.9: 5 records

b	a	c
2	2	4
2	1	4
2	2	5
2	1	5
4	NA	7

`(4, 7)` didn't match with any tuple from `r` but was still included in the result with `a` set `null`.

`full outer join` is a combination of left and right outer joins. Again the previous example:

```
with r(a, b) as (
    values
    (1, 2),
    (2, 2),
    (5, 3)
), s(b, c) as (
    values
    (2, 4),
    (2, 5),
    (4, 7)
)
select *
from r natural full outer join s
```

Table 14.10: 6 records

b	a	c
2	1	5
2	1	4
2	2	5
2	2	4
3	5	NA
4	NA	7

Anotheer `full outer join` exmaple:

- all students from the Comp. Sci. department, along with course sections, if any that they have taken in Spring 2017; all course sections from Spring 2017 must be displayed even if no student from the Comp. Sci department has taken the course:

```
select s.id, t.course_id
from ( -- subquery computing relation containing students from CS
 ↪  department
    select s.id, s.name
    from student s
    where s.dept_name = 'Comp. Sci.'
) s(id, name) right outer join
( -- subquery computing relation of section taken in Spring 2017
    select t.student_id, t.course_id, t.sec_id, t.semester, t."year"
    from takes t
    where t.semester = 'Spring'
        and t.year = 2017
) t(id, course_id, sec_id, semester, year) on s.id = t.id
```

Table 14.11: 3 records

id	course_id
12345	CS-190
54321	CS-190
NA	EE-181

or an equivalent alternative formulation using `with` to factor the subqueries:

```
with s(id) as (
    select s.id
    from student s
    where s.dept_name = 'Comp. Sci.'
), t(id, sec_id, course_id, semester, year) as (
    select t.student_id, t.sec_id, t.course_id , t.semester, t."year"
    from takes t
    where t.semester = 'Spring' and t."year" = 2017
)
select s.id, t.course_id
from s right outer join t on s.id = t.id
```

Warning

Note that

```
select s.id, t.course_id
from student s right outer join takes t
on s.id = t.student_id
where s.dept_name = 'Comp. Sci.'
    and t.semester = 'Spring'
    and t."year" = 2017
```

Table 14.12: 2 records

id	course_id
12345	CS-190
54321	CS-190

would be a **`wrong`** solution, since `right outer join` is performed before the `where` clause on the full student relation and not on the student relation restricted to the CS department.

To verify previous solution compute sections offered in Spring 2017:

```
select *
from section
where year = 2017
    and semester = 'Spring'
```

Table 14.13: 3 records

course_id	id	semester	year	building	room_number	time_slot_id
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C

Indeed, course 'EE-181' hasn't been taken by any student from the CS department.

14.2 Views

Two reasons:

- Security; restricting what users can see
- Virtual tables that better correspond to the intuition of the user

`view` can be thought of extending `with` beyond use in a single query. It is possible to define arbitrarily many views on top of existing relations.

14.2.1 View Definition and Usage

Syntax:

```
create view v as <query expression>
```

Examples:

- restricted access to instuctor relation without salary information:

```
create view faculty as
    select i.id, i."name" , i.dept_name
    from instructor i
```

Then, we can access `faculty` as if it were a regular relation:

```
select *
from faculty f
```

Table 14.14: Displaying records 1 - 10

id	name	dept_name
10101	Srinivasan	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music
22222	Einstein	Physics
32343	El Said	History
33456	Gold	Physics
45565	Katz	Comp. Sci.
58583	Califieri	History
76543	Singh	Finance
76766	Crick	Biology

In **authorization** section we can see how users can be given access to views instead of or in addition to relations.

> Note
>
> view relations are not pre-computed and stored; they are computed dynamically.

- view of all course section offered by the physics department in fall 2017:

```
create view physics_fall_2017 as
    select c.id as course_id, s.id as section_id, s.building ,
     ↪  s.room_number
    from course c , "section" s
    where c.id = s.course_id
        and c.dept_name = 'Physics'
        and s."year" = 2017
        and s.semester = 'Fall'
```

Views remain available until explicitly dropped.

Examples for usage of views:

- id of physics courses offered in Fall 2017 semester, Watson building:

```
select course_id
from physics_fall_2017
where building = 'Watson'
```

Table 14.15: 1 records

course_id
PHY-101

Attribute names can be specified explicitly:

```
create view departments_total_salary(dept_name, total_salary) as
    select i.dept_name, sum(i.salary)
    from instructor i
    group by i.dept_name
```

Then we can use it:

- departments with corresponding total salaries where total salary is less than the average total salary across departments:

```
select *
from departments_total_salary dts
where dts.total_salary < (
    select avg(total_salary)
    from departments_total_salary
)
```

Table 14.16: 3 records

dept_name	total_salary
Biology	145120
Elec. Eng.	80000
Music	40000

One view may be used in defining another view.

- view listing id and room number of all physics courses offered in fall 2018 in the Watson building, using the previous view `physics_fall_2017`

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building = 'Watson'
```

14.2.2 Materialized Views

We learned that results of views are not pre-computed and stored, i.e. they are simply virtual relations that are computed on demand. But some DBs allow storing pre-computed views, that are automatically updated when the relations used in the definition of the view change.

Such views are called **materialized views**.

Some DB's only periodically update materialized views, some perform updates when they are accessed. Some DBMS allow specifying which method is to be used.

Advantages of materialized views is avoiding recomputing the relation defined by the view each time it is accessed. It can be beneficial if the relations used in view definition are very large.

14.2.3 Update of a View

Modifications are generally not allowed on views, since views usually represent partial information and inserting tuples into views would require inserting `null` values into original relations. Which is not guaranteed to work when the view is defined by joining multiple relations and the joined attribute is omitted form the final view definition.

Following works in postgresql:

```
insert into faculty
    values ('30765', 'Green', 'Music')
```

Then we see that a new tuple has been inserted into `instructor` with salary set to `null`:

```
select *
from instructor i
where dept_name = 'Music'
```

Table 14.17: 2 records

id	name	dept_name	salary
15151	Mozart	Music	40000
30765	Green	Music	NA

But in case we define a view listing ID, name and building name of instructors:

```
create view instructor_info as
select i.id, i."name" , d.building
from instructor i , department d
where i.dept_name = d.name
```

This view is created by joining instuctor and department over the attributes `instructor.dept_name` and `d.name`. Trying to insert into this view raises and error along the lines of:

```
SQL Error [55000]: ERROR: cannot insert into view "instructor_info"
Detail: Views that do not select from a single table or view are not
 ↪  automatically updatable.
```

A view is in general **updatable**:

- `from` clause has one DB relation.
- `select` clause contains only attribute names of the relation and does not have
  - expressions
  - aggregates
  - `distinct`
- Attributes not listed in the `select` clause are not `not null` or `primary key`

- query does not have a `group by` or `having` clause.

So, the view:

```
create view history_instructors as
select *
from instructor i
where dept_name = 'History'
```

would be updatable:

```
insert into history_instructors
values
('14532', 'Roznicki', 'History', 57000)
```

Note

We can still insert a non-history tuple to the `history_instructors` view:

```
insert into history_instructors
values
('10032', 'Nishimoto', 'Biology', 73120)
```

This tuple will be simply inserted into `instructor` relation and won't appear in `history_instructor`:

```
select *
from history_instructors
```

Table 14.18: 3 records

id	name	dept_name	salary
32343	El Said	History	60000
58583	Califieri	History	62000
14532	Roznicki	History	57000

```
select *
from instructor
where id = '10032'
```

Table 14.19: 1 records

id	name	dept_name	salary
10032	Nishimoto	Biology	73120

However views can be defined with a `with check option` clause at the end of the definition:

```
create view biology_instructors as
select *
from instructor i
where i.dept_name = 'Biology'
with check option
```

Then

```
insert into biology_instructors
values
('10311', 'Schmidt', 'Physics', 105000)
```

Won't be possible.

Preferable altarnative to modifying views with default insert, update and delete is the `instead of` feature found in **trigger** declarations, that allow actions designed specifically for each case.

14.3 Transactions

Transaction is a sequence of query and/or update statements. A transaction begins implicitly when an sql statement is executed. One of follwoing statements must end a transaction:

- `commit [work]`: The updates are made permanent. Afterwars transaction is automatically started.
- `rollback [work]`: undoes all the updates performed by during the transaction. DB is restored to the state before transaction started.

`commit` and `rollback` allow transactions to be **atomic**.

14.4 Integrity Constraints

Examples:

- Instructor name cannot be `null`
- Different instructors cannot have the same ID (primary key)
- Every department name in `course` relation must have matching name in the `department` relation (referential integrity)
- The budget of a department must be greater than $0,0

In general arbitrary predicates (that can be realistically tested by the DBMS).

Usually part of the `create table` command but can also be added to an existing relation with `alter table R add <constraint>`

14.4.1 Constraints on a Single Relation

`create table` may include integrity-constraint statements in addition to `primary key` and `foreign key`:

- `not null`
- `unique`
- `check (<predicate>)`

14.4.2 Not Null

Remember that null value is a member of all domains, therefore it is a legal value for every attribute in SQL by default, but it may me inapropriate for some attributes s.a.:

- student name
- department budget

declared as follows:

```
name varchar(20) not null;
budget numeric(12, 2) not null
```

This prohibits insertion of a null value and is an example of a **domain constraint**. Primary keys are implicitly not null.

14.4.3 Unique

sql supports the integrity constraint:

```
unique(a1, ..., a_n)
```

which specifies that attributs `a1, ..., a_n` form a superkey. However they are allowed to be null unless explicitly declared not null.

14.4.4 Check Clause

In a relation declaration `check (<Predicate>)` specified that `<Predicate>` must be satisfied by every tuple in the relation, which creates a powerful type system.

Exmaples:

- `check(budget > 0)` in the declaration of `department`:
- values `semester` attribute can take in the declaration of `section`:

```
create table section(
  course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6),
  year numeric(4, 0),
  building varchar(15),
  room_number varchar(7),
  time_slot_id varchar(4),
  primary key (course_id, sec_id, semester, year),
  check( semester in ('Fall', 'Winter', 'Spring', 'Summer'))
)
```

Note

In the above declaration if semester value is `null` it still does not violate the check condition, eventhough null is not one of specified values, because a check condition is violated only if it explicitly evaluates to `false`. `unknown` does not violate the check condition (comparisons with `null`). In order the avoid nulls `not null` must be explicitly specified.

`check()` can be placed anywhere in the declaration. Often it is placed right after the attribute , if it effects a single attribute. More complex `check()` clauses are listed at the end of the declaration.

According to the SQL standard arbitrary predicates and subqueries are allowed in check. But none of the current DBMS support subqueries.

Further `check()` examples:

- `course`:

```
create table course (
    ...
    credits numeric(2, 0) check (credits > 0) -- credit is numeric with
↪   two digits, must
                                              -- be greater than 0
)
```

- `instructor`:

```
create table instructor (
    ...
    salary numeric(8, 2) check (salary > 29000) -- salary less than $1
↪   mil, greater than $29k
)
```

14.4.5 Referential Integrity

Referential Integrity: Often it is needed that a value that appears in a *referencing relation* for a given set of attributes also appears in a *referenced relation*.

Foreign keys is an example of a referential integrity constraint, reminder:

```
foreign key (dept_name) references department --part of create table
↪   course (...)
```

For each tuple in `course`, value of `dept_name` must appear in `name` in `department` relation.

By default foreing-key references the primary-key attributes of the referenced relation but a list of attributes of the referenced relation can be specified explicitly. This list of attributes must either be `primary key` or `unique`:

```
foreign key (dept_name) references department(name)
```

Note

More general referential integrity, where referenced attributes do not form a candiate key is not supported by SQL. But there are alternative construct in SQL that can achieve this, eventhough none of them are supported by current SQL DBMS implementations.

Foreign key must reference a compatible set of attributes. (Cardinality and data type/domain)

14.4.5.1 Cascade

Default behaviour is to reject a transaction that violates a referential integriy constraint. But this can changed with `cascade`.

With `cascade` instead of rejecting a **delete** or **update** that violates the constraint, the tuple in the **referencing** relation is changed. (updated or deleted)

For example in `course` relation

```
create table course(
    ...
    foreign key (dept_name) references departments
        on delete cascade
        on update cascade,
    ...
)
```

- `on delete cascade`: If a tuple in `department` is deleted, all tuples in the `course` that reference that department are deleteded
- `on update cascade`: same as above

further behaviour is allowed in SQL other than `on delete cascade`

- `on delete set null`
- `on delete set default`: set to the default value of the domain.

Foreign keys are allowed to be `null`, unless explicitly specified `not null`. By default foreign key values that contain a `null` are automatically accepted to satisfy the foreign-key contraint by default.

14.4.6 Naming Constraints

Integrity constraints can be named explicitly with the keyword `constraint`:

```
salary numeric(8, 2), constraint minsalary check(salary > 29000)
```

This allows dropping constraints by name:

```
alter table instructor drop constraint minsalary
```

14.4.7 Complex Checks and Assertions

Complex conditions in checks are not implemented in practical DBMS's, but are part of SQL:

```
check (time_slot_id in (select id from time_slot)) -- in the definition
 ↪  of section
```

Current DBMS' do not provide `create assertion` or complex `check` constructs. Nevertheless, equivalent functionality can be achieved using **triggers**, including non-foreigh-key referential integrity constraints.

Assertion is a predicate that a DB should always satsify. Consider:

- for each student tuple in the `student` relation, `tot_cred` must be equal to the sum of successfully completed courses in the relation `takes`

```
create assertion credits_earned_constraint check
(not exists (
    select id
    from student s
    where tot_cred <> (
        select coalesce(sum(credits), 0)
        from takes t, course c
        where t.course_id = c.id
            and s.id = takes.student_id
            and t.grade is not null and t.grade <> 'F'
    )
))
```

General form of an SQL assertion:

```
create assertion <assertion-name> check <predicate>
```

Note

SQL does not provide

$$\forall x P(x)$$

Instead we use the equivalent

$$\neg \exists x \neg P(x)$$

which in turn can be expressed as

```
...
where not exists (
    ... -- SFW construct simulating tuples satifsying not P
)
...
```

14.5 SQL Data Types and Schemas

We covered basic DT, s.a. :

- `int`
- `varchar(<N>)`
- `numeric(<N>, <M>)`
- `float`

There are additional DTs, as well as possiblity to define custom DTs.

14.5.1 Date and Time

SQL standard supports several DTs relating to the dates and times:

- `date`: A calendar date containing a four-digit year, month and a day of the month
- `time`: The time of day in hours, minutes and seconds.
- `time(<P>)`: Same as time, where `<P>` can be used to specify the number of fractional digits for seconds.
- `time with timezone`: Same as time, with the additional information for the time zone.

- `timestamp`: A combination of `time` and `date`
- `timestamp(<P>)`: Same as `timestamp`, where `<P>` specifies the number of fractional digits for seconds. (default is 6)
- `timestamp with timezone`: self-explanatory

Exmaples:

- `date '2023-04-25'`, format: `yyyy-mm-dd`
- `time '09:30:15'`, format: `hh:mm:ss[.ff]`
- `timestampt`, format: `date time`

Individual fields can be extracted from `date` or `time` values using `extract()` function:

```
values
(extract(year from date('1999-12-12'))),
(extract (second from time '10:15:30.14'))
```

Table 14.20: 2 records

column1
1999.00
30.14

We can get current date, current time (with time zone), local time (without timezone), current time stamp (with time zone), local time stamp (without time zone):

```
values
(current_date)
```

Table 14.21: 1 records

column1
2023-10-29

```
select *
from (
    values
    (current_time(2), localtime(2))
) times(with_time_zone, without_time_zone)
```

Table 14.22: 1 records

with_time_zone	without_time_zone
21:11:34.74	21:11:34.74

```
select *
from (
    values
    (current_timestamp, localtimestamp)
) time_date(with_time_zone, without_time_zone)
```

Table 14.23: 1 records

with_time_zone	without_time_zone
2023-10-29 21:11:34	2023-10-29 21:11:34

SQL allows comparison of date and time types.

There is `interval` data type that corresponds to interval compoutations of `time` types.

```
values
('10:30:15'::time - '09:15:30'::time)
```

Table 14.24: 1 records

column1
01:14:45

```
values
(age('1999-10-13'::date, '1983-03-15'::date))
```

Table 14.25: 1 records

column1
16 years 6 mons 29 days

Arithmetic operations with `interval` type are possible:

```
values
('1 years 5 mons 20 days'::interval + current_date)
```

Table 14.26: 1 records

column1
2025-04-18

14.5.2 Type Conversion and Formatting Functions

14.5.2.1 Casting

Casting DT with `cast(<D1> as <D2>)` or with `::`:

```
values
(cast(10.2 as int)),
(10.2::int),
('1111'::int) --casting string as int
```

Table 14.27: 3 records

column1
10
10
1111

14.5.2.2 Formatting

Changing displayed format instead of the DT with `to_char`, `to_number`, `to_date`

```
values
(cast(10.2 as int)),
(10.2::int),
('1111'::int) --casting string as int;
```

Table 14.28: 3 records

column1
10
10
1111

```
values
(to_char(10, '999D99')),
(to_char(124.43::real, '999D9'));
```

Table 14.29: 2 records

column1
10,00
124,4

```
values
(to_char(localtimestamp, 'HH12:MI:SS')),
(to_char (interval '15h 2m 12s', 'HH24:MI:SS'));
```

Table 14.30: 2 records

column1
09:11:34
15:02:12

```
values
(to_date('05 Dec 2000', 'DD Mon YYYY'));
```

Table 14.31: 1 records

column1
2000-12-05

14.5.2.3 Handling Null Values

We can specify how null values should be displayed with `coalesce()`:

```
select a, coalesce(b, 0) as b
from (
    values
    (1, null),
    (2, 3)
) r(a, b)
```

Table 14.32: 2 records

a	b
1	0
2	3

14.5.3 Default Values

A Default value can be specified for an attribute in the `create table` statement:

```
create table student (
    id varchar(5),
    name varchar(20) not null,
    dept_name varchar(20),
    tot_cred numeric(3, 0) default 0
    primary key (id)
)
```

When a tuple is inserted into `student`, if no value provided for `tot_cred` its value is set to 0 by default:

```
insert into student(id, name, dept_name)
values ('12789', 'Newman', 'Comp. Sci.')
```

14.5.4 User-Defined Types

Two forms are supported:

- **distinct types**
- **structured data types**: complex data types with nested record strutures, arrays and multisets

14.5.4.1 Distinct Types

Even though `student.name` and `department.name` are both strings, they should be distinct on the conceptual level.

On a programming level assigning a human name to a department name is probably a programming error. Similarly comparing a monetary value in dollars to a monetary value in pounds is also probably a programming error. A good type system should detect such errors.

`create domain` can be used to create new types:

```
create domain dollars as numeric(12, 2) check (value >= 0)
```

New types can be used in `create table` declarations:

```
create table department(
    dept_name varchar(20),
    building varchar(15),
    budget Dollars
)
```

14.5.5 Generating Unique Key Values

DBMS offer automatic management of unique-key value generation. In the `instructor` instead of

```
id varchar(5)
```

we can write:

```
id number(5) generated always as identiy
```

Any `insert` statement must avoid specifying a value for the automatically generated key:

```
insert into instructor(name, dept_name, salary)
    values ('Newprof', 'Comp. Sci.', 100000)
```

if we replace `always` with `by default` we can specify own keys.

14.5.6 Create Table Extensions

Creating a table with the same schema as an existing table:

```
create table temp_instructor (like instructor including all)
```

A new table can be created and populated with data using a query:

```
create table t1 as (
    select i.name as i_name, i.salary as i_salary
    from instructor i
    where dept_name = 'Music'
)
with data; --optional in postgres
```

> **ℹ Note**
>
> **difference to views**:
>
> - views reflect the actual contents dynamically.
> - tables created with the above method are initiated with set values.

14.6 Index Definition in SQL

Many queries reference only a small portion of the records in the file:

- Find all instructors in the Physics department
- Find the salary of the instructor with the ID 22201

reference only a fraction of the records in the instructor relation. It is inefficient to check every record if `building` field is ‘Physics’ or if `id` field is ‘22201’.

An **index** on an attribute of a relation is a data structure that allows the DBS to find those tuples in the relation that have a specified value for that attribute efficiently (in logarithmic time), without linearly scanning through all tuples of the relation.

For example, if we create an index on the `dept_value` attribute of the relation `instructor`, DBS can find records that have any specified value for `dept_value` s.a. “Physics”, or “Music” directly, without reading all the tuples linearly.

An index can also be created on a list of attributes instead of a single attribute, e.g. on `name` **and** `dept_name` of `instructor`.

Indexes can be created automatically by the DBMS, but it is not easy to decide, therefore SQL DDL provides syntax for creating indexes manually with the `create index` command:

```
create index <index-name> on <relation-name> (<attribute-list>);
```

Example:

```
create index dept_index on instructor (dept_name);
```

Now, when a query uses `dept_name` from `instructor` it will benefit from the index and it will execute faster:

```
select *
from instructor
where dept_name = 'Music'
```

Table 14.33: 2 records

id	name	dept_name	salary
15151	Mozart	Music	40000
30765	Green	Music	NA

Named indexes can be dropped:

```
drop index dept_index;
```

14.7 Authorization

A user ma be assigned several types of authorizations on parts of the DB:

- to read data
- to insert new data
- to update data
- to delete data

Each of the above is called a **privilege**. A user may be authorized on combinations of those on parts of the DB such relations or views.

User may also be authorized on the DB Schema, s.a. create, modify or drop relations.

A user may be authorized to

- pass his authorization (**grant**)
- withdraw an authorization that was granted (**revoke**)

14.7.1 Granting and Revoking Privileges

SQL includes following privileges:

- `select` (reading data)
- `insert` (insert new data)
- `update` (updating data)
- `delete` (deleting data)

`grant` statement is used to confer authorization:

```
grant <privilege list>
on <relation/view name>
to <user/role list>
```

Grant read authorization `department` relation to users Amit and Satoshi

```
grant select on deparment to Amit, Satoshi
```

Update authorization can be granted on the whole tuple or only on a list of attributes. List of attributes on which update authorization is granted appears after `update` listed inside parantheses

Grant update authorization on `budget` attribute of the `department` relation to users Amit and Satoshi:

```
grant update on department(update) to Amit, Satoshi
```

`insert` and `delete` authorizations are analogous.

Note

The user name `public` refers to all current and future users of the system. Thus privileges granted to `public` are implicitly granted to all current and future users.

To revoke authorizations `revoke` statement is used:

```
revoke select on department from Amit, Satoshi;
revoke update (budget) on department from Amit, Satoshi;
```

14.7.2 Roles

Naturally each instructor must have the same authorizations on same relations/views. When new insturctor is appointed they must automatically receive this roles.

In UniDB example roles could be:

- `instructor`
- `teaching_assistant`
- `student`
- `dean`
- `department_chair`

Roles can be created as:

```
create role instructor
```

Roles can be granted priviliges just like users:

```
grant select on takes
to instructor;
```

Roles can be granted to users and to other roles:

```
create role dean;
grant instructor to dean --grant role instructor to role dean
grant dean to Satoshi; -- grant role dean to user Satoshi
```

When a user logs in to the DBS, actions executed by the user have all the privileges granted directly to the user, as well ones granted to roles granted to the user.

14.7.3 Authorization on Views

Consider a staff member who needs to know that salaries of all faculty in the Geology department, but is not authorized to see any information regarding other departments. Thus he must be denied direct access to the `instructor` relation, but still be able to see information from the Geology department. This can be achieved with granted authorization to a view:

```
create view geo_instructor as (
    select *
    from instructor
    where dept_name = 'Geology'
)
```

Then we can create role `geo_admin` and grant it certain privileges:

```
create role geo_admin;
grant select on geo_instructor to geo_admin;
grant update (salary) on geo_instructor to geo_admin;
```

The user who **creates** this view must have `select` authorization on `instructor` relation.

14.7.4 Transfer of Privileges

A user/role is by default not allowd to pass on their authorization to other users/roles. This can be changed with `with grant option` clause.

Example:

- We wish to allow Amit `select` privilege on `department` and allow Amit to grant it to others:

```
grant select on department to Amit with grant option;
```

Consider the granting of update authorization on the `teaches` relation. Assume that, initially DB admin grants update authorization to U_1, U_2, and U_3, with `with grant option`. These users may grant this authorization to other users, which can be represented by an **authorization graph**.

A user has an authorization *iff* there is path from root to the user.

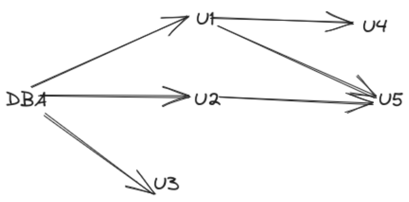


Figure 14.1: authorization-graph

14.7.5 Revoking of Privileges

Suppose DB admin decides to revoke the authorization of U_1 show in Figure 14.1. Since U_4 has received authorization from U_1, his authorization should be revoked as well. However U_5 was granted authorization by both U_1 and U_2. Since U_2s authorization was not revoked U_5 retains the authorization.

This default behavior is **cascading revokation**. It can be prevented by the keyword `restrict`:

```
revoke select on department from Amit, Satoshi restrict
```

This revocation fails, if there are any cascading effects.

Only the privilege of granting an authorization can be revoked specifically:

```
revoke grant option for select on department from Amit;
```

Now Amit can no more grant `select` privilege to other users, but still has the privilege itself.

The default cascading revokation is not appropriate in many cases. Suppose Sutoshi has the role `dean`, grants `instructor` to Amit, and later `dean` is revoked from Sutoshi, perhaps because he leaves the University. Since Amit continues to be employed he should retain the `instructor` role.

To deal with this, SQL permits privilege to be granted by a role rather than by a user. SQL has a notion of the current role associated with a session. By default it is null. It can be set with

```
set role role_name --role associated with the session
```

To grant a privilege with the grantor as the current session role, we add the clause

```
granted by current_role
```

to the grant statement, provided current session role is not null.

15 Uni DB

15.1 ER Model

15.2 Relational Model

- $\texttt{intstructor}(\underline{\texttt{ID}}, \texttt{name}, \texttt{dept\_name} \rightarrow \texttt{department}, \texttt{salary})$
- $\textbf{\texttt{course}}(\underline{\texttt{id}}, \texttt{title}, \texttt{dept\_name} \rightarrow \texttt{department}, \texttt{credits})$
- $\textbf{\texttt{prereq}}(\underline{\texttt{course\_id} \rightarrow \texttt{course}, \texttt{prereq\_id} \rightarrow \texttt{course}})$
- $\textbf{\texttt{department}}(\underline{\texttt{name}}, \texttt{building}, \texttt{budget})$
- $\textbf{\texttt{section}}(\underline{\texttt{course\_id}, \texttt{id}, \texttt{semester}, \texttt{year}}, (\texttt{building}, \texttt{room\_number}) \rightarrow \texttt{classroom}, \texttt{time\_slot\_id})$
- $\textbf{\texttt{teaches}}(\underline{\texttt{instructor\_ID} \rightarrow \texttt{instructor}, (\texttt{course\_id}, \texttt{sec\_id}, \texttt{semester}, \texttt{year}) \rightarrow \texttt{section}})$
- $\textbf{\texttt{student}}(\underline{\texttt{ID}}, \texttt{name}, \texttt{dept\_name} \rightarrow \texttt{department}, \texttt{total\_credit})$
- $\textbf{\texttt{takes}}(\underline{\texttt{student\_ID} \rightarrow \texttt{student}, (\texttt{course\_id}, \texttt{section\_id}, \texttt{semester}, \texttt{year}) \rightarrow \texttt{section}}, \texttt{grade})$
- $\textbf{\texttt{advisor}}(\underline{\texttt{student\_id} \rightarrow \texttt{student}, \texttt{instructor\_id} \rightarrow \texttt{instructor}})$
- $\textbf{\texttt{classroom}}(\underline{\texttt{building}, \texttt{room\_number}}, \texttt{capacity})$
- $\textbf{\texttt{time\_slot}}(\underline{\texttt{id}, \texttt{day}, \texttt{start\_time}}, \texttt{end\_time})$

15.3 SQL

15.3.1 DDL

- **Definitions**:

```
drop table prereq;
drop table time_slot;
drop table advisor;
drop table takes;
drop table student;
drop table teaches;
drop table section;
drop table instructor;
drop table course;
drop table department;
drop table classroom;


create table classroom
    (building       varchar(15),
     room_number        varchar(7),
     capacity       numeric(4,0),
     primary key (building, room_number)
    );

create table department
    (name       varchar(20),
     building       varchar(15),
     budget             numeric(12,2) check (budget > 0),
     primary key (name)
    );

create table course
    (id     varchar(8),
     title          varchar(50),
     dept_name      varchar(20),
     credits        numeric(2,0) check (credits > 0),
     primary key (id),
     foreign key (dept_name) references department (name)
        on delete set null
    );

create table instructor
    (ID         varchar(5),
     name           varchar(20) not null,
     dept_name      varchar(20),
     salary         numeric(8,2) check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department (name)
        on delete set null
    );

create table section
    (course_id      varchar(8),
         id         varchar(8),
     semester       varchar(6)
        check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
     year           numeric(4,0) check (year > 1701 and year < 2100),
     building       varchar(15),
     room_number        varchar(7),
     time_slot_id       varchar(4),
     primary key (course_id, id, semester, year),
     foreign key (course_id) references course (id)
        on delete cascade,
     foreign key (building, room_number) references classroom (building,
      ↪  room_number)
        on delete set null
    );

create table teaches
    (instructor_ID          varchar(5),
     course_id      varchar(8),
     sec_id         varchar(8),
     semester       varchar(6),
     year           numeric(4,0),
     primary key (instructor_ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section
      ↪  (course_id, id, semester, year)
        on delete cascade,
     foreign key (instructor_ID) references instructor (ID)
        on delete cascade
    );

create table student
    (ID         varchar(5),
     name           varchar(20) not null,
     dept_name      varchar(20),
     tot_cred       numeric(3,0) check (tot_cred >= 0),
     primary key (ID),
     foreign key (dept_name) references department (name)
        on delete set null
    );

create table takes
    (student_ID         varchar(5),
     course_id      varchar(8),
     sec_id         varchar(8),
     semester       varchar(6),
     year           numeric(4,0),
     grade              varchar(2),
     primary key (student_ID, course_id, sec_id, semester, year),
     foreign key (course_id, sec_id, semester, year) references section
      ↪  (course_id, id, semester, year)
        on delete cascade,
     foreign key (student_ID) references student (ID)
        on delete cascade
    );

create table advisor
    (student_ID         varchar(5),
     instructor_ID          varchar(5),
     primary key (student_ID),
     foreign key (instructor_id) references instructor (ID)
        on delete set null,
     foreign key (student_ID) references student (ID)
        on delete cascade
    );

create table time_slot
    (id     varchar(4),
     day            varchar(1),
     start_hr       numeric(2) check (start_hr >= 0 and start_hr < 24),
     start_min      numeric(2) check (start_min >= 0 and start_min <
 ↪  60),
     end_hr         numeric(2) check (end_hr >= 0 and end_hr < 24),
     end_min        numeric(2) check (end_min >= 0 and end_min < 60),
     primary key (id, day, start_hr, start_min)
    );

create table prereq
    (course_id      varchar(8),
     prereq_id      varchar(8),
     primary key (course_id, prereq_id),
     foreign key (course_id) references course (id)
        on delete cascade,
     foreign key (prereq_id) references course (id)
    );
```

- **Data**:

```
delete from prereq;
delete from time_slot;
delete from advisor;
delete from takes;
delete from student;
delete from teaches;
delete from section;
delete from instructor;
delete from course;
delete from department;
delete from classroom;
insert into classroom values ('Packard', '101', '500');
...
insert into department values ('Biology', 'Watson', '90000');
...
insert into course values ('BIO-101', 'Intro. to Biology', 'Biology',
 ↪  '4');
...
insert into instructor values ('10101', 'Srinivasan', 'Comp. Sci.',
 ↪  '65000');
...
insert into section values ('BIO-101', '1', 'Summer', '2017', 'Painter',
 ↪  '514', 'B');
...
insert into teaches values ('10101', 'CS-101', '1', 'Fall', '2017');
...
insert into student values ('00128', 'Zhang', 'Comp. Sci.', '102');
...
insert into takes values ('00128', 'CS-101', '1', 'Fall', '2017', 'A');
...
insert into advisor values ('00128', '45565');
...
insert into time_slot values ('A', 'M', '8', '0', '8', '50');
...
insert into prereq values ('BIO-301', 'BIO-101');
...
```

15.3.2 Example Querries