

# 2017-1 Data Modelling And Databases

60851 characters in 10004 words on 1686 lines

Florian Moser

February 20, 2018

## 1 INTRODUCTION

### 1.1 DataBase Management System (DBMS)

tool that helps run & develop data intensive applications  
push the complexity of dealing with data (storage, processing, persistence)  
to the database  
share the database  
database is a tool

**many form of engines**  
relational

### 1.2 usages

avoid redundancy & inconsistencies  
rich access to data  
synchronize concurrent access to data  
recovery after system failures  
security & privacy  
facilitate reuse of data  
reduce cost & pain of doing something useful

### 1.3 database abstraction layers

data independence  
logical data independence

## 2 DATA MODELING

### 2.1 data modeling

#### conceptual model

captures the world to be represented (domain), collection of entities & their relations (Entity-Relationship)  
manual modelling of ER schema, semi-automatic transformation to XML, relational, hierarchical, object-oriented, ...  
entity relationship, UML

#### logical model (schema)

mapping of the concepts to concrete logical representations  
flat file (SQLITE), relational model (SQL), network model (COBOL), hierarchical model (IBM IMS/FastPath), object-oriented model (ODMG 2.0), semi-structured model (XML), deductive model (datalog, prolog)

#### physical model

implementation in concrete physical hardware

## 3 ER MODELING

### 3.1 database design

#### 3.1.1 factors

- information requirements
- processing requirements
- DBMS
- Hardware/OS

#### 3.1.2 steps

##### 3.1.2.1 requirements engineering

###### 3.1.2.1.1 create a book of duty

###### describe information requirements

objects used, identifiers, relationships, attributes)

###### describe processes

examination, degree, ...

###### describe processing requirements

cardinalities (how many entries), distribution (how many relationships), workload (how often a process is carried out), priorities & service level agreements

#### 3.1.2.2 conceptual modeling

create a conceptual design (ER)

#### 3.1.2.3 logical modeling

create a logical design (schema)

#### 3.1.2.4 physical modeling

hardware / OS does this

### 3.2 building blocks

#### ellipsis (yellow)

properties

#### rectangles (orange)

objects

#### scewed rectangles (light green)

relations

#### underline

key

#### underline with dashes

secondary key

#### text besides relationships

roles (for example "attends")

#### double border relations & objects

weak entities, must have secondary key, relation AND object are weak

#### is-a (6 edges polygon, blue)

"inheritance", generalization, arrow points to more general object (less specific one) employee(salary) → is-a → person (id, name). likes traits!

#### part-of (looks like relation, but green)

frame - part of - bicycle (no arrows drawn)

### 3.3 creating models

avoid redundancy

KISS

#### two binary vs one ternary

different use cases

#### attribute vs entity

attribute if 1:1 relationship, else entity

#### partitioning of ER models by domains good practice

#### do not

redundancy, performance improvements

#### do

less is better, concise, correct, complete, comprehensible

### 3.4 good to know about

#### 1:1

rel(\_a\_,b), additional key \_b\_

#### n:1

A(\_a\_, b), B(a)

#### rule of thumb

put finger on all defined relations (all but one). if missing relation is a 1 → fully defined

#### min/max

(2,\*), (1,6), ...

#### weak entities

key is defined by primary key of related object & own secondary key. happens for example for building → room. double borders for relationship, entity & connecting line

#### weak relations

do not exist! Only weak entity matters; and then mark the defining relation as weak too. (order\_entry is not weak; even if order is)

### generalization

using the is-a structure, more specific points to more general. one entity can have multiple specializations

### aggregation

using the part-of structure

**Professor 1 — <gives> — N Lecture**

### min max

(1:N)  $\Rightarrow$  (0,\*) (0,1)

**to be part of a relationship is optional; but if the relationship is established; it needs to connect all entities part**

### tertiary

MyTert (\_a\_, \_b\_, c); additional key: \_a\_, \_c\_

**same underline for same key, even if it spans multiple attributes!**

**R(\_a , b\_, c)**

### ER to Schema

- 1) entities to tables
- 2) relationships to entities
- 3) combine entities which have generalization (copy all fields of more general into more specific)
- 4) combine relationships into entities where possible (same primary key)

### 3.5 difficulties

#### 3.5.1 create global schema from different views

#### want

no redundancy, no conflicts, avoid synonyms & homonyms

#### difficulties

detect generalizations, synonyms, different concepts of attribute domain

## 4 Unified Modeling Language

### 4.1 basics

#### short

UML

#### ER vs UML

attribute, generalization same

entity vs class

relationship vs association

weak entity vs compositor

#### key differences

methods are part of classes

keys are not part of UML

UML models explicitly aggregation

UML supports instance modeling

#### additional material

use cases

sequence diagram

object diagram

### 4.2 building blocks

#### 4.2.1 create small tables for classes

##### first row

name of object (professor)

##### second row

property list & their type (PersNr: String, Age: Int)

##### last row

method names (promote())

#### 4.2.2 associations

arrows, with min/max notation at the end (1, 1..\*, 0..\*, 0..1)

##### example

Professor 1 — gives  $\rightarrow$  0..\* Lecture

**other way around as compared with ER**

#### 4.2.3 aggregation

arrow (with scewed rectangle as arrow head, not filled out)

#### example

Professor —<WHITE> Group

**"belongs to"**

#### 4.2.4 composition

arrow (with scewed rectangle as arrow head, filled out)

#### example

Room —<BLACK> Building

**"is part of"**

#### 4.2.5 generalization

arrows, with triangle as head

#### example

Professor —|> Person

**"is a"**

## 5 RELATIONAL DATA MODEL

### 5.1 definitions

#### 5.1.1 relation

R untermenge  $D_1 \times D_2 \times D_3$

#### example

Addressbook untermenge string  $\times$  string  $\times$  int

#### 5.1.2 tuple

t element\_of R

#### example

("hi", "mom")

#### 5.1.3 schema

associates labels to domains

AddrBook: {[\_Id\_: int, Name: string, Number: int]}  $\rightarrow$  underline keys, done here with KEY

#### 5.1.4 instance

set of db

#### 5.1.5 key

monomial set of attributes that identify each tuple uniquely

#### 5.1.6 primary key

select one key, use it to refer to tuples

### 5.2 transform ER to RM

entities to relations  $\rightarrow$  Student: {[\_Number\_: string, Name: string]}

relationships to relations  $\rightarrow$  attends: {[\_Number\_: string, \_Lecture\_: string]}

#### naming attributes

use names of roles if provided, else use key attribute name, else invent new names

**merge relations with same key  $\rightarrow$  {[\_A\_: string, B: int]} + {[\_A\_: string, C: string]}  $\rightarrow$  {[\_A\_: string, B: int, C: string]}**

#### generalization

copy all attributes of generalization into more specific relation, or not. but keep all tables (do not remove general tables like Person!)

#### weak entities

must contain all keys (also from strong one)

### 5.3 relational algebra

#### 5.3.1 atoms

basic expressions

relation in database

constant relation

#### 5.3.2 operators

composite expression

selection, projection, cartesian product, rename, union, minus  
no recursion!

#### 5.3.3 selection s (sigma)

condition

s(semester > 10)(Student)

### 5.3.4 projection p (pi)

selection of columns  
 $p(\text{semester})(\text{Student})$

### 5.3.5 cartesian product cp (x)

all for all ( $n \times m$  set)

### 5.3.6 (natural) join j ( $|><|$ )

$S \text{ nj } R = p(A_n, R, B_n, C_n)(s(R, B_n = S, B_n)(S \text{ j } R))$

### 5.3.7 theta join tj ( $|><|$ theta)

two relations with no common attributes + binary operator theta  
 $S \text{ tj } R = s(\text{theta})(S \times R)$

### 5.3.8 rename r (p)

rename relation names to join multiple same tables  
 $s(L1.\text{age} = L2.\text{age})(r(L1)(\text{Student}) \times r(L2)(\text{Student}))$   
rename attribute names  
 $r(\text{newName} <- \text{oldname})(\text{Student})$

### 5.3.9 set minus m (-)

gives difference of attributes (not data!)

### R-S

all attributes in R which are not in S

### 5.3.10 relational division rd ( $./.$ )

$R \text{ rd } S$   
S contains at least one column of R  
S contains multiple rows  
outputs all tuples from R which have all the values supplied by S  $\rightarrow$  so if S is ( $v_1, v_2$ ) then R outputs  $m_1$  if R contains ( $[m_1, v_1], [m_1, v_2]$ ) (so both  $v_1$  &  $v_2$  connected to  $m$ )  
result set contains all columns from R which are not in S

can be rewritten as

$p(R-S)(R) - p(R-S)((p(R-S)(R) \times S) - R)$

”student who attends all lectures”  $R:\text{attends}, S:\text{lecture}$

### 5.3.11 union u (U)

combines two sets with same attributes

### 5.3.12 intersection i (turned U)

only works if both relations have same schema (same attribute names & attribute domains)  
 $i = R - (R - S)$

### 5.3.13 semi-join (left) sjl ( $|><$ )

tuples from left matching tuples from right (only left columns)

### 5.3.14 semi-join (right) sjr ( $><|$ )

tuples from right matching tuples from left (only right columns)

### 5.3.15 left outer join loj ( $_-|><|$ )

natural join + unmatched tuples from left (left+right columns)

### 5.3.16 right outer join roj ( $|><|_-$ )

natural join + unmatched tuples from right (left+right columns)

### 5.3.17 full outer join foj

natural join + unmatched tuples from left/right (left+right columns)

## 5.4 tuple relational calculus

$\{t \mid P(t)\}$

more advanced

$\{t \mid t \text{ element\_of Student} \wedge \text{there\_is } a \text{ element\_of attends}(a.\text{is\_valid\_entry} \Rightarrow a.\text{student\_id} = t.\text{student\_id} \wedge a.\text{is\_too\_late})\}$

tuple relational calculus

standard calculus; atoms, formulas, constants, ...

safety

restrict to finite answers (semantic not syntactic property!  $\{n \mid \text{not } (n \text{ element\_of Table})\}$  would not be valid)  
result must be a subset of the domain of the formula (domain: all constants, all relations used in formula)

## 5.5 domain relational calculus

similar to relational calculus, but tuples are ”written out” and specific properties are selected

example

$\{[l, n] \mid \text{there\_is } s ([l, n, s] \text{ element\_of Student} \wedge \text{there\_is } v, p, g ([l, v, p, g] \text{ element\_of tests} \wedge \text{there\_is } a, r, b ([p, a, r, b] \text{ element\_of Professor} \wedge a = \text{'Curie'}))\}$

safety

same as relational calculus

## 5.6 Codd’s theorem

relations, tuple relational (safe only), domain relational (safe only) algebra are all equivalent  
SQL is based on relational calculus  
SQL implementation is based on relational algebra

## 6 SQL

### 6.1 defintions

SQL

Structured Query Language

DDL

Data Definition Language

DML

Data Manipilation Language

Query

Query language

### 6.2 DDL

#### 6.2.1 base

CREATE TABLE person (nr INTEGER NOT NULL, street VARCHAR(50) DEFAULT ”hi mom”, CONSTRAINT my\_const CHECK nr > 0, CONSTRAINT my\_const\_2 UNIQUE(nr), CONSTRAINT my\_frkey FOREIGN KEY (other\_person\_id) REFERENCES person(id));

can add

new columns, CONSTRAINT, FOREIGN KEY, PRIMARY KEY

CREATE TABLE person (id INTEGER, email VARCHAR(50), CHECK (email IS NOT NULL));

DROP TABLE person

ALTER TABLE person ADD COLUMN (age INTEGER)

#### 6.2.2 indexes

CREATE INDEX my\_index ON person (age, nr)  
DROP INDEX my\_index

#### 6.2.3 columns

ALTER TABLE person ADD COLUMN age INTEGER NOT NULL DEFAULT 0;  
ALTER TABLE person ALTER COLUMN age TYPE varchar(50);  
ALTER TABLE person ALTER COLUMN age SET NOT NULL;  
ALTER TABLE person DROP COLUMN;

#### 6.2.4 constaints

ALTER TABLE person ALTER COLUMN age TYPE varchar(50);  
ALTER TABLE person ADD CONSTRAINT age\_bigger CHECK (age > 12);

### 6.3 DML

#### 6.3.1 inserts

INSERT INTO person (age, street) VALUES (12, ”streeT”);  
INSERT INTO copy\_table (age, street) SELECT \* FROM table;  
INSERT INTO copy\_table SELECT \* FROM table;

#### 6.3.2 sequence

CREATE SEQUENCE my\_sql INCREMENT BY 1 START WITH 1;  
my\_sql.nextval  
DROP SEQUENCE my\_sql;

#### 6.3.3 updates

snapshot semantics

mark tuples which are affected by update; then implement update on affected tuples

UPDATE person SET age = 12 WHERE age = 11;

UPDATE persone SET age = age + 1;

### 6.3.4 deletes

DELETE FROM person WHERE age = 11;

## 6.4 ETL

extract, transform, load

### 6.4.1 populating a database needs

#### extract

data from file

#### transform

into a form the database can use (type transformation, formatting)

#### load

insert into db as a bulk operation

## 6.5 Queries

### 6.5.1 select / from

SELECT \* FROM person;  
SELECT age FROM person;

### 6.5.2 where

SELECT \* FROM person WHERE age = 11;

### 6.5.3 order

SELECT \* FROM person ORDER BY age ASC, name DESC, age

### 6.5.4 distinct

SELECT DISTINCT age FROM person;

### 6.5.5 rename

SELECT \* FROM person p;  
SELECT age as person\_age FROM person;

### 6.5.6 join

SELECT \* FROM person p, hero h WHERE h.id = p.id;

### 6.5.7 set operations

(SELECT age FROM person) UNION (SELECT age FROM hero)  
//removes duplicates  
(SELECT age FROM person) UNION ALL (SELECT age FROM hero)  
(SELECT age FROM person) INTERSECT (SELECT age FROM hero)  
(SELECT age FROM person) MINUS (SELECT age FROM hero) //same  
as except  
(SELECT age FROM person) EXCEPT (SELECT age FROM hero)  
//same as minus

### 6.5.8 functions

SELECT \* FROM person WHERE SUBSTRING(name, 1, 3) = 'flo';  
SELECT \* FROM person WHERE date < NOW() AND  
date.part('year',date) = 2017 OR date = date('12.07.08');  
EXTRACT(YEAR FROM orderdate);

### 6.5.9 aggregate

SELECT AVG(age) FROM person;  
SELECT MAX(age) FROM person;  
SELECT MIN(age) FROM person;  
SELECT COUNT(age) FROM person;  
SELECT SUM(age) FROM person;

### 6.5.10 with

create views only available in the subquery → be careful with the ,  
WITH my\_table AS (SELECT \* FROM person), my\_table\_2 AS (SELECT  
\* FROM person) SELECT \* FROM my\_table, my\_table\_2

### 6.5.11 EXCEPT

distinct rows from left which are not part of right

### 6.5.12 INTERSECT

distinct rows which are both in right & left set

### 6.5.13 subqueries

SELECT age FROM person AS p1 WHERE (SELECT MIN(p2.age)  
FROM person p2 WHERE p2.name = p1.name) > 20  
WITH my\_range AS (SELECT age FROM (SELECT \* FROM person pe  
ORDER BY pe.age LIMIT 0.55 \* (SELECT COUNT(\*) FROM person))  
AS lowest55 ORDER BY age DESC LIMIT 0.1 \* (SELECT COUNT(\*)  
FROM person))  
SELECT name, (SELECT AVG(age) FROM person) FROM person,  
(SELECT \* FROM hero WHERE name = "hi mom")

### 6.5.14 grouping

SELECT AVG(age), name FROM person GROUP BY name;

### 6.5.15 having

SELECT AVG(age), name FROM person GROUP BY name HAVING  
AVG(age) > 11;

### 6.5.16 exists

SELECT name FROM person p WHERE NOT EXISTS (SELECT \*  
FROM hero h WHERE p.name = h.name)

### 6.5.17 in

SELECT name FROM person p WHERE name NOT IN (SELECT name  
FROM hero h)

### 6.5.18 all

SELECT name FROM person p WHERE age ≥ ALL(SELECT age FROM  
hero h)

### 6.5.19 for all tricks

SELECT a.Legi FROM attends a GROUP BY a.Legi HAVING  
COUNT(\*) = (SELECT COUNT(\*) FROM Lecture);

### 6.5.20 NULL

#### arithmetic results in NULL

NULL \* 2 → NULL

#### comparisation results are UNKNOWN

NULL > 2 → UNKNOWN

#### nulls are not equal null

NULL <> NULL = true

#### nulls group to one group

GROUP BY maybe.column has NULL

#### where

only true evaluations

#### group by

if at least one NULL exists there will be group for NULL

### 6.5.21 UNKNOWN

#### not

results in UNKNOWN

#### and

false and UNKNOWN → false; other and UNKNOWN → UNKNOWN

#### or

true or UNKNOWN → true; other or UNKNOWN → UNKNOWN

### 6.5.22 between

SELECT age FROM person WHERE age BETWEEN 2 AND 3;

### 6.5.23 in

SELECT age FROM person WHERE age IN (1,2,3);

### 6.5.24 case

SELECT age, (case when (age > 12) then "old" when (age < 12) then  
"young" else "12" end) FROM person;

### 6.5.25 like

%

any amount of characters

-

exactly one letter

**SELECT name FROM person WHERE name LIKE "Marga%"  
OR name LIKE "M\_rgareth"**

#### you can choose a escape caracter by using ESCAPE

so LIKE "80!%" ESCAPE "!" would look for "80%" text vals

### 6.5.26 joins

SELECT \* FROM person, hero WHERE person.id = hero.id; crossproduct  
then filter

SELECT \* FROM person JOIN hero ON person.id = hero.id; same as  
above

SELECT \* FROM person NATURAL JOIN hero; removed duplicated  
columns

SELECT \* FROM person LEFT JOIN hero ON person.id = hero.id; left  
table part fully filled out (n entries)

SELECT \* FROM person RIGHT JOIN hero ON person.id = hero.id;

right table part fully filled out (m entries)  
SELECT \* FROM person FULL OUTER JOIN hero ON person.id = hero.id; both tables in there, missing column values are null (m+n entries)

### 6.5.27 recursion

WITH RECURSIVE temp (n, fact) AS (SELECT 0, 1 UNION ALL  
SELECT n+1, (n+1)\*fact FROM temp WHERE n < 9) SELECT \*  
FROM temp;  
WITH lectures (first, next) AS (SELECT prerequisite, follow-up FROM  
requires UNION ALL SELECT t.first, r.follow-up FROM lecture t,  
requires r WHERE t.Next= r.prerequisite)  
WITH RECURSIVE R(r) AS (SELECT 1 UNION SELECT r+1 FROM  
R) SELECT r FROM R LIMIT 10;

### 6.5.28 views

CREATE VIEW person\_view AS (SELECT age, name FROM person)

#### updateable iff view involves

only one base relation; the key of that relationship; does not contain  
aggregates, group-by or duplicate-elimination

→ **view columns do not change (not even if SELECT \*) if the  
columns of base table change**

## 7 INTEGRITY CONSTRAINTS

### 7.1 ways to constraint

schema (defines domain of the data & the captured concepts)  
types (defines format & space reserved for values)  
constraints (add additional constraints to attributes & relations)

### 7.2 constraints

pre- & postconditions  
way to make sure changes are consistent and do not cause trouble later on  
control content of data and its consistency as part of the schema

#### 7.2.1 avoiding problems

inserting data without a key  
adding references to non-existing tuples  
nonsensical values for attributes  
conflicting tuples

#### 7.2.2 example constraints

keys  
multiplicity of relationships  
attribute domains  
subset relationship of generalization  
referential integrity (foreign keys do indeed reference an existing key)

#### 7.2.3 static constraints

constraints any instance of DB must meet

#### 7.2.4 dynamic constraints

constraints on a state transition of the DB

#### 7.2.5 why in database

good way to annotate schema  
db is a central points; so has to be done once & for all cases

#### sofety net

in case feature missing in app

#### useful for DB optimization

### 7.3 constraint examples

#### 7.3.1 UNIQUE

key, null each count as unique  
CREATE TABLE person (id INTEGER PRIMARY KEY, email TEXT  
UNIQUE)

#### 7.3.2 PRIMARY KEY

entity integrity  
CREATE TABLE person (id INTEGER PRIMARY KEY)  
CREATE TABLE person (id INTEGER, PRIMARY KEY (id))  
CREATE TABLE person (id INTEGER, age INTEGER, PRIMARY KEY  
(id, age))

#### 7.3.3 FOREIGN KEY

##### referential integrity

for every foreign key the value must be either NULL or the one of an

existing referenced tuple

**CREATE TABLE person (id INTEGER PRIMARY KEY,  
other\_person\_id INTEGER REFERENCES person)**

**CREATE TABLE person (id INTEGER PRIMARY KEY,  
other\_person\_id INTEGER, FOREIGN KEY (other\_person\_id)  
REFERENCES person ON DELETE SET NULL)**

**CREATE TABLE person (id INTEGER PRIMARY KEY,  
other\_person\_id INTEGER, FOREIGN KEY (other\_person\_id)  
REFERENCES person(id) ON DELETE SET NULL)**

#### 7.3.4 maintain integrity actions

CREATE TABLE person (id INTEGER PRIMARY KEY, other\_person\_id  
INTEGER, FOREIGN KEY (other\_person\_id) REFERENCES person(id)  
ON DELETE SET NULL ON UPDATE SET NULL);

→ set foreign key to null if changed/removed

CREATE TABLE person (id INTEGER PRIMARY KEY, other\_person\_id  
INTEGER, FOREIGN KEY (other\_person\_id) REFERENCES person(id)  
ON DELETE CASCADE ON UPDATE SET NULL);

→ remove if foreign element is removed, set null if foreign element is  
updated

CREATE TABLE person (id INTEGER PRIMARY KEY, other\_person\_id  
INTEGER, FOREIGN KEY (other\_person\_id) REFERENCES person(id)  
ON DELETE NO ACTION UPDATE CASCADE);

##### cascade

propagate update & delete

##### set default, set null

set references to null or default value

##### restrict

prevents deletion of primary key if it still is referenced somewhere (checked  
immediately)

##### no-action

same as restrict, but checked at the end

#### 7.3.5 implementation with triggers

ECA rule

Event → check Condition → execute Action

#### 7.3.6 checks

CREATE TABLE person (age INTEGER, CHECK (age BETWEEN 1  
and 5));  
CREATE TABLE person (gender varchar(1), CHECK (gender IN  
('m', 'f')));  
CREATE TABLE person (born INTEGER, died INTEGER, CHECK  
(born < died));

#### 7.3.7 triggers

CREATE TRIGGER enforce\_feminism BEFORE UPDATE ON Professor  
FOR EACH ROW WHEN (old.age != 1) BEGIN .... new & old, use if ..  
and .. then .. end if; END

## 8 NORMAL FORMS

### 8.1 redundancy

#### problems

waste of storage space  
need to keep duplicates up to date

#### advantages

improve locality  
space is not so much of a problem anymore, time is  
fault tolerance, availability

### 8.2 multi-version database

storage is cheap → never throw anything away

#### consequence 1

no delete (simply mark as deleted)

#### consequence 2

no update in place (create new version of tuple)

#### NoSQL Movement

denormalized data

### 8.3 functional dependency

if prop A equal then prop B equal of two tuples

$\{A\} \rightarrow \{B\}$ ,  $A \rightarrow B$  as convention

### 8.3.1 armstrong axioms

#### reflexivity

$b \text{ subset\_of } a \Rightarrow a \rightarrow b$

#### augmentation

$a \rightarrow b \Rightarrow ay \rightarrow by$  (dont forget:  $a \rightarrow y = a \rightarrow ay$ )

#### transitivity

$a \rightarrow b \wedge b \rightarrow y \Rightarrow a \rightarrow y$

complete, all other rules can be inferred from those three

#### union

$a \rightarrow b \wedge a \rightarrow y \Rightarrow a \rightarrow by$

#### decomposition

$a \rightarrow by \Rightarrow a \rightarrow b \wedge a \rightarrow y$

#### pseudo-transitivity

$a \rightarrow b \wedge yb \rightarrow g \Rightarrow ay \rightarrow g$

## 8.4 keys

### 8.4.1 superkey

(a determines whole relation)

$a \rightarrow R$

either superset of minimal key or candidate key

#### 8.4.1.1 example

##### trivial

all attributes!

##### non-trivial

take candaidate key and add unnecessary attribute

### 8.4.2 minimal key

#### for\_all A element\_of a

not  $((a - A) \rightarrow b)$  (no entry from key can be removed)

$a \rightarrow. b$

#### example

the keys which identify the row; as the id

### 8.4.3 (candidate) key

(a is minimal and determines whole relation)

$a \rightarrow. R$

#### example

the attributes which closure allows to determine all attributes

## 8.5 cardinalities

cardinalities define functional dependencies

functional dependencies determine keys

but not all functional dependencies are derived from cardinality information

## 8.6 closure of attributes

### 8.6.1 input

F (set of functional dependencies ( $A \rightarrow B$ )), a\_set (set of attributes)

### 8.6.2 formula

result = a\_set;

while (resultChanges) {

8.6.2.1 foreach (a\_set as  $a \rightarrow b$ ) {

if (a element\_of result) {

result += b;

}

}

8.6.2.2 }

8.6.2.3 //closure is deterministic & terminates

8.6.2.4 //correctly done with Mengen; so use submenge for element\_of and union for +=

## 8.7 minimal basis

8.7.1 Fc is a minimal basis for F iff

$Fc == F$  (closure of attribute sets in Fc is same as F)

8.7.1.1 all functional dependencies in Fc are minimal

for\_all A element\_of a

$(Fc - (a-b)) \text{ union } ((a-A) \rightarrow b) != Fc$

for\_all B element\_of b

$(Fc - (a-b)) \text{ union } ((a) \rightarrow (b-B)) != Fc$

8.7.1.2 in Fc, there are no two functional dependencies with the same left side

### 8.7.2 formula

#### 8.7.2.1 given

$a \rightarrow b$ , A element\_of A, B element\_of B, set of functional dependencies F

#### 8.7.2.2 reduction of left sides

if (B element\_of Closure(F, a-A)) then replace  $a \rightarrow b$  with  $(a-A) \rightarrow b$

#### 8.7.2.3 reduction of right sides

if (B element\_of Closure(F-( $a \rightarrow b$ ) union  $a \rightarrow (b-B)$ , a)) then replace  $a \rightarrow b$  with  $a \rightarrow (b-B)$

#### example

B element\_of Closure(F-( $a \rightarrow bc$ ) union  $a \rightarrow b$ , a)

#### 8.7.2.4 remove Fd where a == empty

#### 8.7.2.5 apply union rule to Fd where a1 == a2

## 8.8 decomposition of relations

bad relation capture multiple concepts, so decompose them

#### lossless

$S = S1 \text{ union } S2$

$S = S1 \text{ natural\_join } S2$

#### prove that it is lossless (direct proof)

show that it holds for  $\text{column1} = R1 \text{ union } R2$  that  $\text{column1} \rightarrow R1$  or  $\text{column1} \rightarrow R2$

#### prove that is lossy (prove by counterexample)

let R have two entries

construct R1 and R2 from it

cosntruct R1 natural\_join R2

show that the constructed table is not identical to R

#### prove that $R = R1 \text{ natural\_join } R2$

assume R1 is of the form (a, b) and R2 is of the form (a, c)

prove R untermenge R1 natural\_join R2 (if (a,b,c) element\_of R, then ...)

prove R1 natural\_join R2 untermenge R (if (a,b,c) element\_of R1

natural\_join R2, then show that its in R1 and R2)

#### preservation of dedependencies

$FD(R)^+ == (FD(R1) \text{ union } FD(R2))^+$

$\rightarrow$  possible for lossless decomposition & but not preservation of dependencies if dependencies span the two tables

## 8.9 remember stuff

the key (1NF), the whole key (2NF) and nothing but the key (3NF), so help me codd

## 8.10 normal forms

to investige the normal form of a relation first find the candidate key (the minimal key) of the functional dependencies

for all FD example assume R(ABCD) with key AB

### 8.11 minimal basis

iff functional dependencies cannot further be reduced

#### 8.11.1 algorythm

##### left reduction

remove unnecessary part of keys from left

##### right reduction

remove unnecessary results from the right

##### remove empty FD

remove all FD which point to nothing

## 8.12 first normal form

### 8.12.1 why

only atomic domains (no JSON as value)

### 8.12.2 short

no json

### 8.12.3 formal

only atomic domains as values (no JSON, array or similar)  
"no repeating groups"

### 8.12.4 disprove 1NF

find JSON  
"find repeating groups"

### 8.12.5 prove 1NF

only atomic domains  
"columns are all different (no columns which should be rows)"

### 8.12.6 example in 1NF

[12, "hallo", "hi"]

### 8.12.7 example not in 1NF

[12, {'de':  
'hallo', 'en': 'hi'}]

(name, bestellung1, bestellung2 bestellung3)

## 8.13 second normal form

### 8.13.1 why

eliminate update anomalies

### 8.13.2 short

must depend on whole key

### 8.13.3 formal

no functional dependency exists which depends only on part of the  
candidate key  
"each column must depend on the entire primary key"

### 8.13.4 prove 2NF

check that all FD either depend on the full key or no key at all

### 8.13.5 example

valid  
 $AB \rightarrow C, C \rightarrow D, C \rightarrow A$

invalid  
 $A \rightarrow C$

## 8.14 third normal form

### 8.14.1 why

Eliminating functional dependencies on non-key fields

### 8.14.2 short

a is candidate key or B is part of candidate key

### 8.14.3 formal

B is part of the key  
B element\_of a ( $a \rightarrow B$ , trivial:  $AB \rightarrow B$ )  
a is superkey of R

### 8.14.4 prove 3NF

for all a check that its the key  
the rest; check that the right side is part of the key

### 8.14.5 example

valid  
 $AB \rightarrow C; C \rightarrow A$

invalid  
 $C \rightarrow D;$

### 8.14.6 synthesis algorithm

compute minimal basis Fc ( $A \rightarrow B, C$ ), (D,A are keys)

#### create relations for FD

for all  $a \rightarrow b$  create  $R_a := a$  union  $b$  ( $[A, B, C]$ )

#### create relations for keys (if needed)

if exists k untermenge R, and k is key of R create  $R_k := k$  ( $[A, B, C], [A, D]$ )

#### eliminate redundant (submenge relations)

eliminate  $R_a$  if exists  $R_a$  untermenge  $R_a'$  (eliminate unnecessary tables)

## 8.15 boyce codd normal form

### 8.15.1 why

eliminate FD which have no key

### 8.15.2 short

a is candidate key

### 8.15.3 formal

B element\_of a ( $a \rightarrow B$ , trivial)  
a is superkey of R

### 8.15.4 prove BCNF

for all a check that its the key

### 8.15.5 example

valid  
 $AB \rightarrow C$

invalid  
 $C \rightarrow A$

### 8.15.6 decomposition algorithm

any schemas can be converted to BCNM, but dependencies are not  
guaranteed to persist

#### 8.15.6.1 decompose cyclic

create new table for cyclic FD, and remove new key from the old relation

#### 8.15.6.2 result = a\_set

8.15.6.3 foreach (a\_set as a) {  
if (a is not in BCNF because  $y \rightarrow z$  is evil) {  
a1 = (y, z);  
a2 = (a - z);  
result = result - a + a1 + a2;  
}  
}

#### 8.15.6.4 }

## 8.16 fourth normal form

### 8.16.1 why

elimintate MVD anomalies

### 8.16.2 short

only one concept per table

### 8.16.3 formal

for every MVD the left side must be superkey

### 8.16.4 prove 4NF

for all  $a \twoheadrightarrow B$  check that a is superkey

### 8.16.5 example

valid  
 $AB \rightarrow C$

invalid  
 $C \rightarrow A$

### 8.16.6 decomposition algorithm

result = a\_set

8.16.6.1 foreach (a\_set as a) {  
if (a is not in 4NF because  $y \rightarrow z$  is evil) {  
a1 = (y, z);  
a2 = (a - z);  
result = result - a + a1 + a2;  
}  
}

#### 8.16.6.2 }

### 8.16.7 not dependecy preserving!

## 8.17 MVD (Multi Value Dependency)

written as  $a \twoheadrightarrow b$ , person\_id  $\twoheadrightarrow$  phone  
if table can be restructured; so for example (person\_id, language, skill)  
if 1:n beziehung in same table; so (person\_id, email, children\_name)

### 8.17.1 formal

for\_all t1,t2 element\_of R and  $t1.a = t2.a \Rightarrow t3, t4$  element\_of R  
create table with four rows (t1,t2,t3), three columns (A,B,C).

**left column**

all a

**middle column**

b1,b2,b1,b3

**right column**

c1,c2,c2,c1

**a  $\rightarrow \rightarrow$  b && a  $\rightarrow \rightarrow$  c**

**now write down all the conclusions for t3,t4  $\rightarrow$  proove stuff with that**

### 8.17.2 laws

**generalization, promotion**

**a  $\rightarrow$  b  $\Rightarrow$  a  $\rightarrow \rightarrow$  b**

**reflexivity, augementation, transitivity**

**multi value augmentation, transitivity (in result is g sub\_element h, so  $ag \rightarrow ah$ )**

**complement**

**a  $\rightarrow \rightarrow$  b  $\Rightarrow$  a  $\rightarrow \rightarrow$  R-b-a**

**coalesce**

**a  $\rightarrow \rightarrow$  b  $\wedge$  (y sub\_group b)  $\wedge$  (z intersect b = empty)  $\wedge$  z  $\rightarrow$  y = a  $\rightarrow$  y**

**multi-value union**

**a  $\rightarrow \rightarrow$  b  $\wedge$  b  $\rightarrow \rightarrow$  c  $\Rightarrow$  a  $\rightarrow \rightarrow$  bc**

**intersection**

**a  $\rightarrow \rightarrow$  b  $\wedge$  a  $\rightarrow \rightarrow$  c  $\Rightarrow$  a  $\rightarrow \rightarrow$  (b intersect c)**

**minus**

**a  $\rightarrow \rightarrow$  b  $\wedge$  a  $\rightarrow \rightarrow$  c  $\Rightarrow$  a  $\rightarrow \rightarrow$  (b - c)  $\wedge$  a  $\rightarrow \rightarrow$  (c - b)**

**WRONG**

**a  $\rightarrow \rightarrow$  by  $\Rightarrow$  a  $\rightarrow \rightarrow$  b  $\wedge$  a  $\rightarrow \rightarrow$  y (example: a: person\_id, b: language, c: skill)**

### 8.17.3 trivial MVD

**b sub\_group a OR b = R-a**

### 8.17.4 lossless decomposition

**(R1 union R2)  $\rightarrow \rightarrow$  R1**

## 8.18 database schema

**captures**

concepts represented

attributes

constraints & dependencies over all attributes

**good schemas**

ER as the basis

functional dependencies for constraining

normal forms for removing redundancies & anomalies

algorithms for decomposing & deriving normalized tables

## 8.19 OLTP

on-line transaction processing

workload with updates, online (time-critical), high volume of small transactions

integrity important (3NF+)

banking, shops

**TPC-C**

wholesome supplier, stocks of products in stores, districts, warehouses

## 8.20 OLAP

on-line analytical processing

workload with complex queries, data updates in batches

de-normalized schemas, with approaches as star/snowflake

marketing analysis

**TPC-H**

pricing, analysis of sales

**star schemas (not normalized tables)**

fact + domension tables (sales table with customer\_id, part\_id as fact table, customer + part tables as dimension tables)

**why denormalized**

not updated by hand, during data loading process constaints are controlled (application now responsible!)

**snowflake schema (some dimension tables are normalized)**

so dimensiontables can itself have dimensiontables

**TPC-DS**

sell analytics over multiple channels

seven fact tables, 17 dimension, 38 columns. large!

## 8.21 modern trends

working set in main memory

column stores at physical level

OLTP & OLAP in same system

specialitation for use case (denormalization etc)

## 8.22 data cubes

analysis & reporting

preaggregated data over several dimensions

### 8.22.1 example cube dimension

year, product category, store

### 8.22.2 support

**slicing**

fixing one dimension (select specific year) (removed on dimension of the cube)

**dicing**

select some values over all dimension of cube (sales of product by product\_nr, time, region) (generalization of slicing; select multiple times & places)

**roll up & drill down**

group-by at different granuarities (sales by year or month, products by region or shop) (less or more cubes)

## 9 DMBS

### 9.1 basis

**input**

SQL

**output**

tuples

**process**

application  $\rightarrow$  DML-Compiler  $\rightarrow$  query optimizer, schema  $\rightarrow$  runtime  $\rightarrow$  query processor, data manager (Indexes, Records), Storage Manager (pages)

### 9.2 performance

now assuming all data in main memory, before assumed data on disk

**random vs sequential access**

expensive (pollutes caches, generates faults, not predictable) vs fast (hardware pefecheing works well)

### 9.3 storage system

#### 9.3.1 storage systems basic

**organized in a hierarchy**

combine different types to get desired result

**can be distributed**

organized in arrays, use cheap hardware, paralell processing, replication

**non-uniform**

varying distance to memory, sequential vs random access

#### 9.3.2 storage manager

**controls access to disks**

implements storage hierarchy (ssd  $\rightarrow$  tapes  $\rightarrow$  disks), caching, optimizes storage

**management of files & blocks**

keeps them in pages (granularity of access), keeps track of pages from DBMS (catalog)

**buffer management**

segementation of buffer pool, clever replacement policies, pins pages (no replacement)



### 9.3.3 bottleneck

cache hiarchies requires careful placement of data

### NUMA has big effect

memory access time not uniform (distance), memory on other cores may not be the same

### 9.3.4 store data

records as tuples  
collection of tuples same table as pages  
parts/collection of pages as blocks  
tablespaces is the place on disk for the db to use

### 9.3.5 data manager

maps tuples to pages

### 9.3.6 buffer manager

maps pages in memory to ones on disk

### 9.3.7 catalog

knows where which files are

### 9.3.8 oracle logical data organization

#### extend

blocks of data (~pages) for the same purpose

#### segement

collection of extends stored in the same tablespace

## 9.4 persist stuff

### 9.4.1 record structure

#### fixed length fields

direct access  
stored in system catalogs  
no need to scan to find i-th entry

#### variable length fields

access in two steps (retrieve info about variable (length, pointer), retrieve value)  
field count + delimiter or array of field offsets at beginning of space

#### NULL

bit set to indicate value is null

### 9.4.2 page structure

#### 9.4.2.1 fixed size

##### header

slot directory (1 if valid, 0 if invalid), number of records in page, other stuff

##### entry

consists of rid (record identifier), payload

##### position of record in page

slotno \* #record

→records can move inside page (on CRUD), do not need to regenerate indexes!

#### 9.4.2.2 packed vs unpacked

no space between records (only at the end of page) vs does not care (may be better for access)

#### 9.4.2.3 variable size

##### header

slot directory (length of entries), number of slots in page, pointer to start of free space

record identifier as <page\_id, record\_nr>

#### 9.4.2.4 full page but record grows

##### row chaining

placeholder PID (which points to another page); flexible, no need to update other references, but expensive

#### 9.4.2.5 fixed-sized sequence of blocks

#### 9.4.2.6 header page

references data pages (full and empty) & next header page

### 9.4.3 file

variable-sized sequence of blocks

### 9.4.4 tablespace

one or more datafiles

each datafile is only associated to one tablespace  
an object in the tablespace may spans multiple datafiles

## 9.5 buffer management

### 9.5.1 target

keep pages in memory as long as possible

### 9.5.2 crittical

replacement policy (LRU, 2Q (active, inactive page, replace inactive, accessed pages moved to active queue)), when does writeback occur of updated pages?

### 9.5.3 hides the fact that not all pages are in main memory which are operated on (which is assumed by query processor)

### 9.5.4 Buffer management of DBMS vs OS

DBMS has more insights as OS (access patterns)

#### problems

double page fault (as DBMS is on top of OS, replacement of page in Buffer may results in another replacement in OS)

### 9.5.5 access patterns

#### sequential

1 - 1000

#### hierachical

index navigation

#### random

index lookup

#### cyclic

nested loops (repeated access pattern)

### 9.5.6 replace policies

#### LRU

Least Recently Used, replace oldest used page in cache → problem  
sequential flooding (#buffer pages < #files in page)

#### MRU

Most Recently Used, replace newest used page in cache (example: buffer size 4, access: 1234512345)

### 9.5.7 DBMin

#### observations

many concurrent queries, queries are made from operators

**buffer segmentation; each operation has its own buffer size, replacement policy**

#### exaples

scan 4pages MRU, indexScan 100pages LRU

## 9.6 meta-data management

stored in tables, accessed internally with SQL

#### contains

schema (to compile queries)  
tablespaces (files)  
histograms (statistics for query optimizations)  
parameters (cost of IO, CPU speed, for optimizations)  
compiled queries (used for prepared statements)  
configuration (isolation level, AppHeap size)  
Users (login, passwd)  
Workload statistics (index advisors)

## 10 QUERY PROCESSING

### 10.1 architecture

#### compiler

sql input  
parser produces query graph model  
rewrite produces QGM  
optimizer produces QGM++  
CodeGen produces the Execution Plan

#### runtime system

Interpreter fetches the tuples

## 10.2 runtime system

### compile query into machine code

better performance, today's approach

### compile query into relational algebra & interpret

easier debugging, portability

hybrid; e.g. compile predicates

## 10.3 relational algebra algorithms

### 10.3.1 query selectivity

#tuples in result vs #tuples in table

### 10.3.2 attribute cardinality

#distinctive values for attribute

### 10.3.3 skew

probability distribution of the possible values of an attribute

### 10.3.4 selectivity of index (over an attribute)

#diff values / #total rows → high for unique keys; low for boolean

### 10.3.5 indexes

B-Tree of values;

good on primary/foreign keys, attributes with high cardinality, attributes

used in joins, conditions

needs space, maintenance, not useful if low index selectivity

### clustered index

leaves of B-Tree hold data (and not just pointer). Only once per table

## 10.4 partitioning

divide a table into smaller chunks (by hash, range) for parallel access,

cache fit, increase concurrency, distribute hot-spots

needs good heuristics to get right

## 10.5 table access

iterate over every tuple in table; match tuple against predicate

### 10.5.1 not so expensive because

selectivity, IO, block/page sizes

### 10.5.2 predictable runtime

### 10.5.3 scan

no indexes relevant; load each page

### min / max pages

1 / # of pages which can be served in one IO request

### 10.5.4 index scan

relevant index found; need to scan whole index to find corresponding results

### min / max pages

2 / entire B-Tree + all pages → use LRU

### 10.5.5 index seek

index relevant found; but don't need to scan whole index only parts of it

(BETWEEN query)

## 10.6 sorting

### 10.6.1 sort vs hash

both  $O(n \log n)$ ; hash done before for partitioning, lower constants for CPU

vs sorting more robust

### 10.6.2 needed for results; intermediate step for other operations

### 10.6.3 expensive in CPU, space

### 10.6.4 two-phase external locking

data does not fit in memory / memory already used by other queries

### 10.6.5 N size input pages, M size of buffer

### 10.6.6 One-Pass Merge

#### phase I (create runs)

load buffer space with tuples, sort tuples in buffer, write to disk, redo until all are done

#### phase II (merge runs)

use priority heap to merge tuples from runs

### special

$M \geq N$ : no merge needed,  $M < \sqrt{N}$ : multiple merges needed

### merge runs

keep pointer in each run (start at 0);

load items from pointer into memory

choose lowest item in memory

advance the pointer of the corresponding run by one and take the new item into memory → back to choosing lowest

### 10.6.7 Multi-Way Merge

generalization of one-pass merge

input file → 1-page run (pass 0) → produce 1-page sized runs

each consecutive run (called "pass") produces double sized runs (with merges)

#### 10.6.7.1 analysis IO

$O(n)$  if  $m \geq \sqrt{n}$

$2n$  if  $m > n$

$4n$  if  $n > m \geq \sqrt{n}$

$O(n \log m \ n)$  if  $m < \sqrt{n}$

#### 10.6.7.2 analysis CPU

if  $(m > \sqrt{n})$

create  $N/M$  runs of size  $M$  ( $O(N \cdot \log_2 m)$ ), merge  $n$  tuples ( $O(N \cdot \log_2 N/M)$ )

#### 10.6.7.3 complexity

$N \cdot \log(N)$  is correct, but db cares more about CPU/IO cost, constants, buffer allocation

#### 10.6.7.4 two-way sort

parallelize sorting, concurrent sorts, despite main memory large

## 10.7 joins

very common, expensive

### 10.7.1 performance factors

join algorithm, relative table sizes, number of tables to join, order of joins, selectivity, predicates of query, memory hierarchy, indexes

### 10.7.2 R table 1, S table 2

### 10.7.3 nested-loops NLJ

#### two for loops

while more get tuple from R, scan S, output if match found

#### optimization with blocks

get blocks from R, hash & compare to S (needs less S scans)

#### comparisons

$R \cdot S$

#### IO cost

$p(R) + p(S) \cdot p(R)$

#### min / max pages

2 pages / all pages of inner, one of outer → use MRU

#### good for

two small tables

### 10.7.4 canonical hash join

#### build phase

build a hash table from all tuples in S

#### probe phase

scan S with the hash keys

#### easy parallelizable

#### comparisons

S

#### IO cost

$3 \cdot p(R) + S$  (build hash table from R, compare with S)

### 10.7.5 grace hash join GHJ

build partitions of the same hash value;

scan R and S with the hash function and put tuple in correct partition

foreach partition do the canonical hash join

#### comparisons

$\max(R, S)$

#### IO cost

$3(p(S) + p(R))$  (read/write while hashing; afterwards read to compare)

**min / max pages**  
 $\sqrt{p(R)} + 1$  page of bigger relation /  $p(R) + p(S)$

**good for**  
big tables

### 10.7.6 partitioned hash join PHJ

create  $p$  partitions; then build & probe each partition

**if  $p$  too big**  
TLB-misses or cache trashing

### 10.7.7 sort merge join SMJ

sort both tables  
merge them

**comparisons**  
 $r \log(r) + s \log(s) + r + s$

**IO**  
 $5(p(R) + p(S))$  (read/write for sort, read/write for merge, read for match phase)

**min / max pages**  
2 pages for external sort & multiple merge steps or  $\sqrt{p(R)}$  for one merge step /  $p(R) + p(S)$

**good for**  
small + big table

### 10.7.8 multi-pass radix partitioning

recursively apply partitioned hash join;  
partition key is determined by  $\log_2 p$  bits of hash value

### 10.7.9 parallel radix join

**1st scan**  
local histograms

**2nd scan**  
global histogram & prefix sum  $\rightarrow$  each thread knows when to output its tuples

### 10.8 group-by

hash on group-by attribute; aggregate on hash collision  
sort on group-by attribute; aggregate sorted ranges  
 $\rightarrow$  choose what's best for the situation

## 11 OPTIMIZER

### 11.1 optimizer

plan is a tree of operators; implementation can be chosen

### 11.2 step 1 (execution model)

#### 11.2.1 pipeline execution of the tree (dataflow graph with records as unit of exchange)

define each operator independently

#### generic interface for each operator

open (cascades down the tree), next (produce next answer), close (cascades down the tree)

each operator implemented by iterator

#### 11.2.2 iterator model

open goes down the tree, as soon as everything is open  
next goes down the tree, data is passed from the very bottom to all nodes (which each call next)  
from left bottom to right bottom, up & down in between  
data flow bottom up; control top down  
advantages; generic interface, supports buffer management, no overhead in main memory, pipelining & parallelism

**disadvantages**  
poor cache locality, high overhead of method calls

**alternatives**  
vectorized; use blocks instead of tuples, fast in column stores

**improvements**  
adaptive execution, non-blocking operators, pull/push, query compilation techniques, streaming/partial results

### 11.2.3 dataflow operators

**union**  
read both sides, issue all tuples

**union without duplicated**  
read both sides; only issue tuple if it has not been issued before

**select**  
read tuple; while doesn't match predicate continue to read, else output

**projection**  
read tuple; while more output specified attributes

### 11.3 step 2 (choosing operators)

#### 11.3.1 variables

$b$ : number of blocks in relation,  $c$ : #of blocks in result set,  $h$ : cost of index lookup

#### 11.3.2 selection (file scan)

**A1 (linear search)**  
cost  $b$ ; cost  $b/2$  if on key attribute,  
always applicable, equality

**A2 (binary search)**  
cost  $\log_2(b)$  (location first block for condition) +  $c$   
ordered file, continuous blocks, equality

#### 11.3.3 selection (using indexes)

**A3 (primary index on candidate key, equality)**  
cost  $h + 1$   
equality (only one record retrieved)

**A4 (primary index on nonkey, equality)**  
cost  $h + c$   
equality (multiple records retrieved)

**A5 (equality of search-key of secondary index)**  
cost  $h+1$  bzw  $h+c$   
potentially very expensive because each record may be on another block!

#### 11.3.4 bc

bigger than condition,  $sc$ : smaller than condition

#### 11.3.5 selection with comparison

##### 11.3.5.1 A6 (primary index, comparison)

**bc**  
find first entry which is bigger with index; scan relation from there

**sc**  
start scanning from the beginning till first tuple breaks condition

##### 11.3.5.2 A7 (secondary index, comparison)

**bc**  
find first index of tuple which is bigger; continue to scan index from there and investigate pointers

**sc**  
scan leaf pages of index till first does not satisfy conditions; follow pointers to data

$\rightarrow$  **need IO for each result; maybe sequential scan is faster!**

#### 11.3.6 c1, c2, c3

conditions

#### 11.3.7 complex selections

**A8 (conjunctive selection using one index)**  
select a combinations of  $cn$  which is cheapest to do with A1-A7, apply rest after reading out

**A9 (conjunctive selection using multiple-key index)**  
use multi-key index if available

**A10 (conjunctive selection by intersection of identifier)**  
requires indices with record pointers  
do each condition with its own index  
do a conjunction of the retrieved record pointers of all indexes  
read out records; apply the rest of the tests in memory

**A11 (disjunctive selection by union of identifiers)**  
applicable if all conditions have index, otherwise do linear scan  
take union of all identifiers

**negation (not c1)**

use linear scan  
if very few records in c, and index for c1 is available use this

### 11.4 step 3 (generating equivalent plans)

#### 11.4.1 equivalence rules

conjunctive selection can be deconstructed into sequence of simple selections  
selection operators are commutative (reordering)  
only the last projection rule is needed, others can be omitted  
selections can be combined with cartesian products & theta joins (cross join + selection → theta join)  
theta / natural joins are commutative (can switch tables left & right)  
theta / natural joins are associative (can change order, care with theta; conditions must only contain values of tables left&right)  
selection / theta join can be distributed if only attributes of one of the tables matter (selection can be applied before join)  
projection / theta join can be distributed when you split values to each source table (you can do the projection beforehand + don't forget to select all attributes used in condition)  
union / intersection are commutative, associative  
selection distributes over union, intersection, minus  
projection distributes over union

#### 11.4.2 query rewrite techniques

unnesting of views (remove subqueries which are stupid)  
predicate augmentation ( $A.x = B.x$  and  $B.x = C.x \rightarrow$  augment  $C.x = A.x$ )  
can use any semantically correct rewrite

#### try to

move constraints between blocks, unnest blocks

### 11.5 histograms

#### equi-width

range same steps; age 10-20, 20-30

#### equi-depth

same number of records for range; age 10-22, 22-28, 28-40

#### multidimensional

multiple properties combined in histogram

### 11.6 enumerate alternative plans

#### task

create query execution plan

#### main principle

search through set of plans, choose paths with least estimated cost

#### query optimization is NP hard

#### algorithms

greedy heuristics (highest selectivity join first), dynamic programming, randomized, other heuristics (tips by programmer), smaller search space

### 11.7 join plans for R join S

consider all combinations of access plans  
consider all join algorithms (NL: nested loop, IdxNL: index nested loop, SMJ: sort-merge join, GHJ: grace-hash join)  
consider all orders

#### → for three way

consider prev. plans too

### 11.8 join plans for multiple

use System-R, so consider only left-deep join trees (generate fully pipelined plans)

#### 11.8.1 enumerate Left-Deep plans

find best 1-relation pass for each relation  
find best way to join result of n-1 to n relation  
for each step retain only plan overall, and cheapest plan for interesting order of tuples

#### 11.8.2 order-by, group-by used as last step, using special operator or "interesting order" plan

#### 11.8.3 avoid cartesian products

#### 11.8.4 → approach is exponential in # of tables!

#### 11.8.5 interesting sort order

if r1 combined r2 have ordered attributes that also happen to be in r4 (which is to be joined later); merge-join may be useful later

#### 11.8.6 each subset needs to have

best join order, overall and for all interesting sort orders → does not increase complexity that much

#### 11.8.7 heuristic optimization

cost-based optimization is very expensive, use fixed set of rules that typically make things cheaper  
selection as early as possible (less tuples)  
projects as early as possible (less attributes)  
more restrictive selection / joins before less restrictive ones

##### 11.8.7.1 typical steps

deconstruct conjunctive selections into sequence  
move selection down the tree  
execute joins / selections first which produce the smallest relations  
replace cartesian followed by conditions to join with select  
deconstruct & move down projections as far as possible, possible selection new things (for relations above)  
identify subtrees which can be pipelined and do it

#### more

degree of multiprogramming, reads vs writes, query result caching, stored procedures, prepared statements, materialized views (cache result of view), indexes

## 12 TRANSACTION PROCESSING

### 12.1 notion of time

concurrency control does not imply time ordering (implementation dependant)  
find a way to execute operations in parallel while ensuring correct result

### 12.2 operations

#### BOT

Begin Of Transaction

#### Commit

all changes have been made persistent

#### Abort

transaction is cancelled, all changes rollback

#### Read

read a data item

#### write

write a data item

#### a < b

a happens before b (partial order)

### 12.3 properties

transactions start with BOT and end with abort or commit (but not both)  
no more operations from transaction are possible after commit/abort  
operations within a transaction are totally ordered  
transactions can only read & write  
transactions enter the database in consistent state and exit it in one too

### 12.4 ACID

#### Atomicity

executed in its entirety or not at all  
atomic commitment protocol (like 2PC)

#### Consistency

executed over a consistent DB in its whole leaves the DB in a consistent state

#### Isolation

each transaction behaves as it would have the whole database for itself  
guaranteed by 2PL

#### Durability

no commits are lost

## 12.5 management

### concurrency control

enforcing isolation among multiple running transactions

### recovery

ensuring atomicity & durability

## 12.6 histories

### conflicting operation

two operations over the same item; one of them being a write

### history

partially ordered sequence of operations from a set of transactions

if two operations are ordered in a transaction, they are in equal order in

the history

if two operations conflict, then they are ordered with respect to each other

## 12.7 conflicts

### a1

abort, c1: commit, w1(x): write to x, r1(x): read to x

### r1(x) and a2(x)

conflict if 2 updated x

### w1(x) and a2(x)

conflict if 2 updated x

## 12.8 concurrency control theory

lost update phenomenon (T1 running, in between T2 executes & commits, T1 continues and reverts changes of T2)

phantom items (T1 running (aggregates something), T2 inserts new item, T1 continues (aggregates again, but gets different results))

### 12.8.1 serial history

correct by definition

if for all two transactions from the history all operations from one appear to be before or after the other all operations from the other transaction called serializable

### serializability graph

edges are arrows, nodes are transactions, if acyclic then alles paletti

### 12.8.2 equal histories

same transactions with same operations

conflicting operations are ordered in the same way

## 12.9 recovery theory

### R1 (abort of single TA)

undo changes of single TA

### R2 (system crash, but disks are kept)

redo committed TA's

### R3 (system crash, but disks are kept)

undo active TA's

### R4 (crash with loss of disks)

read backup of DB from tape

**undo-redo possible because databases keep logs of all transactions with their changes**

## 12.10 history types

partial order of all operations

total order of conflicting operations

### 12.10.1 conflict

write followed by read!

### 12.10.2 RC (recoverable)

commit of conflict must be before own conflict ("write" commit before "read" commit)

if T1 reads x from T2 and commits, then  $c2 < c1$

no need to undo a committed transaction

→ results of queries are guaranteed to be correct

→ if no read occurs we are fine!

### 12.10.3 ACA (avoid cascading aborts)

commit of conflict must be before read ("write" commit before read occurs)

if T1 reads x from T2, then  $c2 < r1[x]$

aborting a transaction does not cause aborting others

→ no thrashing behaviour because cascading aborts all the time

→ if no read occurs we are fine!

### 12.10.4 ST (strict)

commit / abort of conflict must be before write/read

if T1 reads or writes value written by T2 then  $c2 < r1[x] / w1[x] \ \&\& \ a2 < r1[x] / w1[x]$

undoing a transaction does not undo the changes of other transactions

→ recovery after a failure is possible after a failure

### 12.10.5 solve exercise

#### 12.10.5.1 check for write followed by read

**if none, check for write on write**

if none → strict

if one, check if commit is between → strict else ACA

**if one, check if commit of write is between**

if completely separated → strict

if multiple writes between commits → ACA

**if not, check if commits are in the correct order → recoverable**

**if not → not recoverable**

## 12.11 serializability

only committed transactions are of importance when evaluating if its serial or not

convert concrete schedule (in which transactions can overlap, the way it happened in real life) to sequential history (where transactions are absolutely ordered)

draw a graph of the concrete schedule for the conflicting operations; if there are cycles it is not serializable

### conflict equivalent

refers to the alternative sequential histories created; if the order of non-aborted conflicting operations were maintained those are conflict equivalent

### conflict serializability

if the concrete schedule is equivalent to a sequential history

### view equivalent

initial reads must be the same in the other histories, read/write conflict exists in the same order, final write must be done by the same transaction

### view serializable

if the concrete schedule is view equivalent to a sequential history

**view equivalent implies conflict equivalent**

## 12.12 general

concurrency control & recovery very powerful, but slow

### 12.12.1 tweak correctness

by clever combination; add, delete, multiply do not conflict  
view serializability enforcing

### restricting structure of transaction

no blind writes (must always read before write), only write one item, no reads after a write

### postponing correctness

eventual consistency, reconciliation, abandoning recoverability

## 12.13 database scheduler

### 12.13.1 architecture

transaction manager TM  
scheduler

### data manager

recovery manager → buffer manager

### storage system

### 12.13.2 pessimistic synchronization

#### S

shared lock, needed for reads

#### X

exclusive, needed for writes

→ **better than OS** cause knows semantics of operations, two locks

**allows concurrent reads**

### 12.13.3 2PL

serializability guaranteed, but may non-recoverable history because of access to uncommitted data

TA acquires lock before accessing object → locked if it can't get it

#### **Growth**

acquire locks, never releases

#### **Shrink**

releases locks, never acquires

**at EOT (abort or commit) all locks must be released**

**non-recoverable history possible!**

→ **if violated; non-serial history possible,**

→ **but still cascading aborts possible if transaction aborts after starting to release locks (because of uncommitted reads)**

→ **does not fulfill ACID because of uncommitted reads transaction may need to be reversed after commit!**

### 12.13.4 strict 2PL

can only release lock on commit / abort

→ avoids cascading aborts

→ visibility at commit (only ever see changed data after commit) in contrary to ACID enforced recoverability (which just says that it can't be reversed)

### 12.13.5 more on 2PL

deadlock problem; needs to be detected with wait-for graph, abort conflicting transactions

#### **OS vs DB**

both use locking, DB focus on data (collection of resources), OS on critical paths / individual resource

OS assigns to process, DB to transaction

OS keeps lock as long as used, DB till end of transaction

**no reordering of operations**

**does only accept serializable histories**

### 12.14 snapshot isolation

TA receives timestamp on start

all reads are carried out with data from before the timestamp, writes changes in buffer

on commit, DB checks for conflict (any data which newer timestamp, because other TA's accessed the same object)

#### 12.14.1 discussion

phantoms disappear (because no updates are seen while trans is running)  
no read/write ever blocked

#### **overhead**

need to keep write set of TA (efficient to abort), no deadlocks but unnecessary rollbacks

#### **write skew**

might be that two TA's modify db in a correct manner, but combined it does not fulfil requirements anymore

#### 12.14.2 reordering of R/W conflict operations

#### 12.14.3 aborts to deal with WW

#### 12.14.4 allows non-serializable histories! b1 b2 w2 r1 c1 w2 c2

### 12.15 asynchronous protocol

no possible, must contains synchronous part or be able to tolerate failure to some degree

communication fail

party may not be able to commits its own decision

### 12.16 isolation in SQL

#### **read uncommitted**

allows the reading of uncommitted values (of transactions which may be aborted later)

#### **read committed**

reads only values which are committed (but one TA may reads different values, if another TA writes in between)

#### **repeatable read**

prevent reading of different values, but phantoms may occur (can see new rows! but already read out ones will be the same; and not removed)

#### **serializable**

full isolation, no funny business, no phantoms etc

### 12.17 distributed systems

#### **eventual atomicity**

at some point; or never (failures in one will result in compensation in already committed ones)

### 12.18 the consensus problem

#### **distributed consensus**

reaching an agreement among all working processes on the value of a variable

**not difficult if system is reliable**

#### **asynchronous**

not able to tell difference between slow connection and failed process

**if failures do occur all entirely async protocols block → no protocol that guarantees consensus can be created!**

**no commit protocol can guarantee independent recovers → it will have to contact others about the decision which had been made**

### 12.19 atomic commitment

#### **AC1**

all processors reach same decision (agreement)

#### **AC2**

a processor cannot reverse its decision (consistency)

#### **AC3**

commit can only be decided if all vote yes (no imposed decisions)

#### **AC4**

if not failures occurred and all votes yes → commit (non triviality)

#### **AC5**

if failures have been repaired eventually a decision will be reached (liveness)

### 12.20 2PC protocol

coordinator sends VOTE-REQ to all, writes START-2PC to log  
all participants reply with YES or NO and abort. before sending answer persist in log

#### 12.20.1 coordinator decides

if all YES then send COMMIT to all, if some NO send ABORT to all who votes YES. writes log before sending final answer

#### 12.20.2 all participants do what operator tells them, and logs it

#### 12.20.3 failures

**if coordinator received timeout in waiting for VOTE-REQ response**

can decide to abort

**if participant times out waiting for VOTE-REQ**

can decide to abort (as no one will have commit yet for sure)

**if participant times out waiting for decision**

it must ask around with Coordinative Termination Protocol CTP; if no one is certain this is called the uncertainty period

#### 12.20.4 recovery & persistence

each state change in protocol is written to disk

#### 12.20.5 linear 2PC

exploits linear network structure

→ if vote with NO, then it can be directly send back (does not need to reach the coordinator)

#### 12.20.6 deadlock

in case operator stops responding before sending first COMMIT message (which means all participants voted YES, but no-one can continue)

if a network split happens, this will block!

#### 12.20.7 #number of messages

3(n-1)

**12.20.8 not live (cause of blocking), but safe**

### 12.21 3PC protocol

coordinator sends VOTE-REQ to all, writes START-3PC to log  
all participants reply with YES or NO and abort. before sending YES  
persist in log, after sending NO persist in log

#### 12.21.1 coordinator decides

if all YES then send PRE-COMMIT to all, if some NO send ABORT to all  
who votes YES. writes log

#### 12.21.2 coordinator waits for ACK's, ignore those who time out

#### 12.21.3 coordinator sends COMMIT

#### 12.21.4 all participants do what operator tells them, and logs it

#### 12.21.5 less timeout window than 2PC

#### 12.21.6 too expensive for practice

#### 12.21.7 in 2PC

if everybody is uncertain → would be able to commit because everybody  
voted YES!

→ but there might be processes which decided to abort and then crashed  
→ need a way to make sure all processes are uncertain (so we can commit)

#### 12.21.8 NB rule

no operational process can commit if there are any operational processes  
which are uncertain → the termination protocol takes care of this  
coordinator must make sure everybody is out of the uncertainty period  
(with PRE-COMMIT) before it can issue a COMMIT message  
when running CTP and everybody is uncertain → abort

##### 12.21.8.1 implications

**coordinator times out waiting for VOTE\_REQ**  
abort

**participant times out waiting for VOTE\_REQ**  
abort

**coordinator times out waiting for ACK**  
simply ignore those

**participant times out waiting for PRE-COMMIT**  
ask around

**participant times out waiting for COMMIT**  
ask around, but certain!

#### 12.21.9 another round of messages

after receiving all YES coordinator sends PRE-COMMIT to all  
participants and waits for all ACKs. then it sends the COMMIT message

#### 12.21.10 if by asking around any participant has a PRE-COMMIT message all can commit, else abort → problem with network split!

**12.21.11 #number of message**  
 $5(n-1)$

#### 12.21.12 live & safe if network is reliable, else may not be safe (network splitting)

### 12.22 Termination Protocol

elect new operator  
coordinator sends STATE-REQ to all  
if aborted/committed received, do this  
if some committable → send PRE-COMMIT to all  
if all uncertain → abort  
→ failures of the participants on the termination protocol can now be  
ignored

→ **only problem left**

if operator fails before sending PRE-COMMIT we have to start all over  
again

### 12.23 No communication failures

we do not allow communication failures (lost packets or similar)

#### problems

lost NO votes  
split networks (partitions) → use quorums (blocking; no decision can be  
made)  
after total failures; we need to find the last active member (because he  
alone knows the read decision made)

## 13 DATABASE REPLICATION

### 13.1 replication strategies

#### when

synchronous / asynchronous

#### where

update everywhere / primary copy

### 13.2 why replication

#### performance

local access is fast, remote slow

#### fault tolerance

different sites allow running system if one fails

#### application type

OLTP (transaction, fast updates) vs OLAP (analytical, lot of processing)

### 13.3 why not

replication needs time

higher space consumption

either lock or inconsistent state is archived by distributed write

### 13.4 concepts

#### 13.4.1 synchronous replication

full ACID

propagate changes to all instances of db immediately

#### advantage

no inconsistencies, all changes atomic, local copy always latest data

#### disadvantage

transaction has to update all sites (longer response time)

#### 13.4.2 asynchronous replication

first executes update local, then propagate changes (copies are inconsistent  
while update is going on)

#### advantage

transaction is always local (good responsetime)

#### disadvantage

data inconsistencies, local read may be old data, changes to all copies are  
not instant, replication not transparent

#### 13.4.3 update everywhere

changes can be initiated by all copies

#### advantages

any site can run an update, load is distributed

#### disadvantages

copies need to be synchronized

#### 13.4.4 primary copy

only one can update (master), all others are updated reflection the changes  
to master

#### advantage

no inter-site synchronization is needed, one site has always all updates

#### disadvantage

load at primary copy can be quite large, reading local copy may contain  
old value

### 13.5 strategies

#### 13.5.1 sync & primary copy

#### advantage

updates do not need to be coordinated, no inconsistencies

#### disadvantage

longest response time, only useful with a few updates (bottleneck primary  
copy), fault tolerant, local copies are read-only and almost useless, not  
scalable (probability of deadlocks too high)

#### conclusion

globally correct, remote writes

#### practical

too expensive (not so useful), not used

### 13.5.2 sync & update everywhere

#### advantage

consistent data, symmetrical, very fault tolerant

#### disadvantage

long response time, updates need to be coordinated

#### conclusion

globally correct, local writes

#### practical

too expensive (does not scale)

### 13.5.3 async & primary copy

#### advantage

no coordination necessary, short response time, nearly same performance as with no replication

#### disadvantage

local copies not up to date, inconsistencies

#### conclusion

inconsistencies

#### practical

feasible

### 13.5.4 async & update everywhere

#### advantage

no centralized coordination, shortest response time

#### disadvantage

inconsistencies, updates may be lost (reconciliation)

#### conclusion

inconsistencies & reconciliation

#### practical

feasible for some applications

## 13.6 ACID & replication

use 2PC for Atomicity

### 13.6.1 how to make sure serialization order is the same at all sites

use replication protocol

### 13.6.2 quorum protocol

quorum of sites can agree to perform an operation (if majority)

”Quorums are sets of sites formed in such a way so as to be able to determine that it will have a non-empty intersection with other quorums”

#### 13.6.2.1 quorum

##### equal effort

all quorums should have about the same size

##### equal responsibility

all sites participate in same number of quorums

##### dynamic reconfiguration

establishing a quorum needs to be dynamic to account for failures

##### low communication overhead

minimize messages being sent

##### graceful degradation

effort proportional to amount of failures

#### 13.6.2.2 quorum consensus protocol (weighted quorums)

each site has weight  $w_i$ , total weight is  $N$ , let  $RT$  &  $WT$  read/write thresholds, such that  $2 \cdot WT > N$ ,  $RT + WT > N$

each copy has version number

##### 13.6.2.2.1 read quorum

if total weight is greater than  $RT$

#### READ

contact sites with version number, until read quorum reached, then read the one with highest version number

##### 13.6.2.2.2 write quorum

if total weight is greater than  $WT$

#### WRITE

contact sites till write quorum is reached, then write to all a new copy with version number  $k+1$

### 13.6.3 sync & update everywhere

#### read-one, write-all

use 2PL, read locally, write with distributed locking protocol (execution is serializable) → you improve this by using quorums

#### write all available copies

read (read copy, if timeout try another),

write (send write to all locations; if reject → abort; if missing → missing write),

validation (check that all missing writes are still down and others all still up; yes → commit, no → abort)

### 13.6.4 sync & primary copy

not used in practice

### 13.6.5 async & primary copy

updates are executed at primary copy (uses 2PL) → propagates

transactions to other pages after termination

reads are executed locally

### 13.6.6 async & update everywhere

transactions are executed locally (2PL), afterwards distributed to other sites  
updates need to be coordinated!

#### use reconciliation

use patterns as latest wins, site priority, largest value, or ad-hoc (error handler decide on the spot)

## 14 KEY VALUE STORES

### 14.1 architecture

#### schema

key + BLOB, index on key (hashing)

#### deployment

horizontal partitioning, replication of partitions

#### replication strategy

quorums, async replication (eventual consistency)

#### good

very fast lookup, easy to scale, useful for lookup

#### bad

no easy support for queries, inconsistent data

#### ugly

pushed complexity to the app, some operations are costly, works well as cache but not as full application

#### got rid of

schema, sql, ACID

### 14.2 specialized engines

#### 14.2.1 amadeus workload

one table, denormalized

#### 14.2.2 cescando

made for amadeus workload

remove unpredictability, make scalable

#### predictable performance

parameters are size of scan, number of queries per scan

optimize whole workload instead of single queries → scales nice with large number of clients & complex queries

split input into parallel streams of data which can be computed on different cores, and then results are merged again

### 14.3 questions

min pages for grace hash join / compare with other joins