Basic Definitions

Superkey → possible to identify a unique tuple. Often different set of attributes
Candidate key → minimal superkey
Primary Key → one of the candidatekeys
Foreign key → attribute corresponding to primary key of another relation

Relational Algebra

select $\sigma_{BranchName='Perryridge'}(account)$, $\sigma_{A=B \wedge D>5}(r)$
project $\pi_{AccNr,Balance}(account)$
union r U s, both must have same schema, eliminate duplicates after union
difference r - s, both must have same schema
Cartesian product r x s = combine all tuples from r with all tuples from s. new size = |r|*|s|
rename $\rho_{(A1,...,An)}(E)$, changes the relation name to r and the attribute names to A1,...,Ak
intersection $r \cap s = r-(r-s)$, same schema required
theta join $r \bowtie_\theta s$: $r \bowtie_{B<X \wedge D=Y} \rho_{(X,Y,Z)}(s)$)
equi join, only equality conditions
natural join: r |x| s with R(A,B,C,D) and S(E,B,D) equivalent to: $\pi_{A,B,C,D,E}(\sigma_{B=Y \wedge D=Z}(r \times \rho_{(E,Y,Z)}(s))$
combination of all attributes that are identical in r and s.
outer join, 1. natural join, 2. adds left and/or right tuples not been combined, gaps filled with "null"
division r ÷ s, good for queries "for all"

r

A	B
'α'	1
'α'	2
'α'	3
'β'	1
'γ'	1
'ϵ'	6
'ϵ'	1
'β'	2

s

B
1
2

r ÷ s

A
'α'
'β'

aggregate functions ϑ avg, min, max, sum, count
$_{G1,G2,...,Gn}\vartheta_{F1(A1),F2(A2),...,Fn(An)}(E)$
G1,G2 ...,Gn is a list of attributes on which to group (can be empty)
Each F_i is an aggregate function
Each A_i is an attribute name
$_{BranchName}\vartheta_{sum(Balance)}(account)$
assignment temp ← $\pi_{R-S}(r)$
generalized proj.: proj. with aritmethic functions $\pi_{a,b-c}$
Deletion: acc ← acc−$\sigma_{BranchName='Perryridge'}(acc)$
Insertion: acc ←acc ∪ {('A-973','Perryridge',1200)}
Updating: acc ← $\pi_{AccNr,BranchName,Balance*1.05}(account)$

Domain Relational Calculus

Domain Independence: RC/DRC only permits domain independent expressions i.e only expressions with sensible answers and no infinite results.
Logical symbols: ∧ ∨ ¬ ⇒ ∀ ∃ …
build-in predicate symbol: = < > ≤ ≥ ≠ …
Short notation: $r(\_, \_X, \_) = \exists U,V,W(r(U,V,X,W))$
Example: first and lastname of all employees with salary above 50'000$: {*FN,LN* | ∃*Sal*(*emp*(*FN,_,LN,_,Sal*) ∧ *Sal* > 50000)}

SQL

Domain types: CHAR(length n), VARCHAR(maxLength n), INTEGER, SMALLINT, NUMERIC(precision p, digits_right_of_comma n), REAL, DOUBLE PRECISION, FLOAT(precision n)

```
CREATE TABLE branch (BranchName CHAR(15)
NOT NULL, BranchCity CHAR(30), Assets
INTEGER,
PRIMARY KEY (BranchName))

CREATE TABLE depositor (CustName
VARCHAR(15),
AccNr INTEGER,
FOREIGN KEY (AccNr) REFERENCES
account(AccNr))

Drop Table
Alter Table: alter table r add A D;
```
(A= column, D= Domain)

```
SELECT (DISTINCT)*
FROM r, s
WHERE r.A = s.A
GROUP BY A
HAVING A<10
```

Aggregate operations in select clause:
Avg, min, max, sum, count

Conditional Statement:

```
case
when cond1 then result1
when cond2 then result2
Else result
End
```

rename tables/columns with 'as' clause

select specific column: relationName.colName
Cartesian product: from borrower as B, loan as L
Join: from t1 (natural) inner join t2 (on t1.col1 = t2.col3)
(not) exists: select CustName from depositor where not exists (select…
Subqueries: select X from p where X in **(select Y from a)**
-some: 5 < some r (read: 5 < some tuple in table r)(some = in)
Query expressions: `union, intersect, except`
NULL Values: "is NULL" can be used to check for NULL Values
- Any comparison with NULL returns unknown
- "**order** by ... desc/asc" to order → asc für aufsteigend, desc für absteigend

Insertion:
Insert Into account (Kolonne1, Kolonne2, Kolonne3) Values('x','y','z'),('x2','y2','z2')[optional falls Reihenfolge geändert werden soll]
Deletion:
delete from account where Kolonne1='XY'
Update: update account set Balance=Balance*1.05 [where …]

Anomalies:
Insert: Schema zwingt den Benutzer Daten einzufügen, die nicht zwingend nötig sind.
Delete: Durch das Löschen von Teilattributen, die logisch zusammengehören, geht Information verloren, die eigentliche logisch unabhängig ist.
Update: Änderungen an einem Tupel führen zu Änderungen an einem anderen Tupel

view: create view v as <query expression>
not updatable if "distinct", "group", "having", "from" more than one table
with: with minAmount(X) as select min(balance) from account → is stored temporarily and to use with following query in the "from" statement: select AccNr from account, min_amount, where …
recursion: `WITH RECURSIVE`

Integrity constraints
not null, **primary key**(attribute),
unique (A1, A2, A3) → states that A1, A2, A3 form a candidate key,
check(Predicate)
foreign key(attribute_of_other_table) references [table]
assertion: create assertion <name> check <predicate> → always fulfilled and checked on every update

User Defined Functions (UDF):
CREATE FUNCTION accountCnt (CName VARCHAR(9)) RETURNS INTEGER AS $$ DECLARE accCnt INTEGER; BEGIN SELECT COUNT(*) INTO accCnt FROM depositor WHERE depositor.CustName = CName; RETURN accCnt; END; $$ LANGUAGE PLPGSQL;

Triggers: automatically executed on defined event:
create trigger [name] before/after [insert/delete/update/...] on [table] for each row/statement execute procedure [function]
Example: create or replace function borrowerUpdate()
Returns trigger as $$
BEGIN Insert into History values (Date(Now()), new.idc);
Return new END;
$$ LANGUAGE plpgsql;
Create trigger tr_update
After update on Borrower
For each row execute procedure borrowerUpdate()

Relational Database Design

Finding a DB Design where simple semantics, minimal redundancy, update anomalies and null values, optimal join-base.
Functional dependencies: Set of attributes X determines (implies) set of attributes Y: SSN → Name. A candidate key always functionally determines all attributes of the relation.
Armstrong's inference rules:
Reflexivity: Y ⊆ X ⊨ X → Y
Augmentation: X → Y ⊨ XZ → YZ
Transitivity: X → Y , Y → Z ⊨ X → Z
Decomposition: X → YZ ⊨ X → Y , X → Z
Union: X → Y , X → Z ⊨ X → YZ
Pseudotransitivity: X → Y , WY → Z ⊨ WX → Z

closure F⁺: of a set F of FDs is the set of all FDs that can be inferred from F
closure X⁺: of a set of attributes X with respect to F is the set of all attributes that are functionally determined by X
equivalent: two sets of FDs F and G are equivalent if: every FD in F can be inferred from G and and every FD in G can be inferred from F ⇒ F and G are equivalent if $F^+ = G^+$
- F covers G if every FD in G can be inferred from F ⇔ $G^+ \subseteq F^+$
F and G *equivalent* if F covers G and G covers F

lossless join decomposition: R1 & R2 form lossless join if; (R1 ∩ R2) → (R1 - R2) is in F+ or
(R1 ∩ R2) → (R2 - R1) is in F+
Projection: Given a set of dependencies F on R, the projection of F on Ri, denoted by F|Ri where attr(Ri) is a subset of attr(R), is the set of dependencies X →Y in F+ such that the attributes in X ∪Y are all contained in attr(Ri).
Dependency Preservation: $(F|R_1 \cup ... \cup F|R_m)^+ = F^+$
Multivalued Dependency: {Brand} --» {Prod'Country} and {Brand} --» {Product}, what means that every Product is produced in every country.
Rules: X --» Y, Y --» Z ⊨ X --» (Z-Y)(transitivity); X → Y ⊨ X --» Y (replication) ; X --» Y |=X --» (R−(X ∪Y)) (complementation) ; X --» Y,W ⊇Z |=WX --»YZ (augmentation); X --»Y,∃W(W ∩Y =∅,W →Z,Y ⊇Z)|=X →Z(coalescing).
1NF: doesn't allow: composite attributes ("address" consists street, zip, country), multivalued attributes (multiple values in tuple) and nested relations (value of tuple is relation). 1NF normally is part of the definition of a relation. NOT in 1NF:

DNum	DMgrSSN	DLoc
5	334455	{Bellaire, Sugarland, Houston }

2NF: Each attribute not contained in a candidate key is dependent from ALL candidate keys (not only a part of the keys. This concludes that every relation shows exactly one issue. Relations with only one primary key attribute are automatically in 2NF. NOT in 2NF (<u>SSN</u>, <u>PNum</u>, Hours, EName, PName). SSN → EName, PNum → PName

SSN	PNum	Hours	EName	PName
1234	1	32.5	Smith	ProductX

3NF: X→A is either trivial, or X is a superkey, or A is contained in a candidate key of R. No transitive dependency of a non-candidate key to a candidate key within a relation: SSN → DName, DName → DLoc means that DLoc is transitively dependant on SSN → not allowed. Decompose the relation to get 3NF. NOT in 3NF because SSN → DNum, DNum → DName

SSN	BDate	Addr	DNum	DName
1234	1965	Houston	5	Research

BCNF: for X → Y either it is trivial or X is a superkey. NOT in BCNF because {Student, Course} → book and book → course.

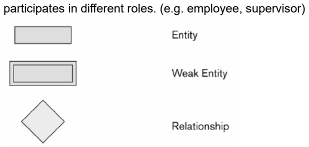
Student	Course	Textbook
Smith	Data Structures	Bertram

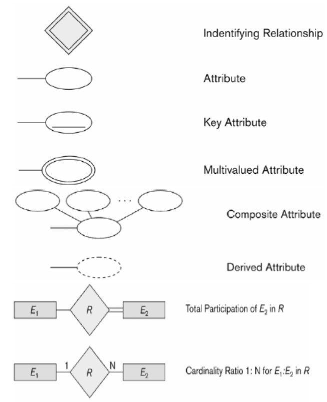
4NF: X->> Y either trivial or X is a superkey

Algorithm for BCNF Normalization
Set D := { R };
while a relation schema Q in D is not in BCNF **do**
 find a functional dependency X →Y in Q that violates BCNF;
 replace Q in D by two relation schemas (Q−Y) and (X ∪ Y);
(Assumption: No null values are allowed for the join attributes.)
Result is a lossless join decomposition of R. Not necessarily dependency preserving.

Entity-Relationship (ER) Model

Entities: objects in mini-world (employee); displayed with squares; **weak** entity types have no key attribute and must participate in an indentifying relationship type. Weak entity types are identified by a partial key of the weak entity type and the particular entity they are related to in the identifying entity type.
Attributes: properties to describe an entity (birthday). may be **simple** or **composite** (address → street, city, zip); **multivalued** {Degrees of a Person}; **derived** (calculated); displayed with ovals; derived ovals dashed, multivalues double ovals, key attributes are underlined
Relationship: diamond-shaped boxes between entities.
cardinality constraint: specifies maximum participation (1:1),(1:N),(M:N) example: employee:department = n:1 (employee works for 1 dep; dep can have n employees
participation constraint: specifies minimum participation. total participation (double line): one or more, it's mandatory to participate on this relation
single line: zero; optional participation.
In a recursive relationship type: The same entity type participates in different roles. (e.g. employee, supervisor)





subclasses: is-a relationship; Triangle shaped;

Disjointness Constraint: an entity can be a member of at most one of the subclasses of the specialization. Annotate edge in ER diagram with *disjoint*; If not disjoint, specialization is **overlapping**: that is the same entity may be a member of more than one subclass of the specialization. Annotate edge in ER diagram with overlapping (or no annotation).

Completeness Constraint:
Total specifies that every entity in the superclass must be a member of some subclass in the specialization/generalization. Shown in ER diagrams by a double line.
Partial <u>allows</u> an entity not to belong to any of the subclasses. Shown in ER diagrams by a single line.

ER-to-Relational Mapping

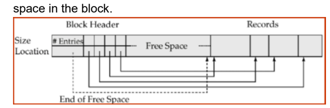
1. Mapping **regular entity** types: strong entity types; include all simple attributes, one key as primary key; A composite attribute is flattened into a set of simple attributes. If the chosen key is composite, the set of simple attributes that form it will together form the primary key.
2. **Weak entity** type W with owner E: create a relation schema R and include all simple attributes (or simple components of composite attributes) of W as attributes of R. Include as foreign key attributes of R the primary key attributes of the relations that correspond to the owner entity types. The primary key of R is the combination of the primary keys of the owners and the partial key of the weak entity type W, if any.
3. Mapping Binary **1:1 relations**: Three possible approaches: a) (if possible, take total participation-party) include as pk the foreign key of the other relation. b) when both participations are total (two lines), merge the two entities. c) create a third relation with both primary key of the connected entities.
4. Mapping Binary **1:N relations**: Include in N-side of the relation the pk of the other side. Include simple attributes of the 1:N relation type as attributes of the N-side.
5. Mapping Binary **M:N relations**: Create new relation and take the primary keys of linking entities as foreign keys. Their combination = pk. Also include attributes (simple and composite) of the relation.
6. Mapping of **multivalued attributes** (double circled): create new relation with attribute A and add primary key of relating entity as Attribute K. PK = A and K. if A is composite, we include its simple components.
7. Mapping **n-ary relations**: Create new relation, add all pk's of participating entities as foreign key. Also include simple(composite) attributes of n-ary relation itself.
8. Mapping **specialization**/generalization: 4 possibilities:
a) create normal relation for superclass with primary key k and all superclass Attributes. For each subclass, create another relation with same primary key k and all Subclass Attributes (without superclass attributes). Works for any specialization/generalization.
b) Only for total participation (double line). Create only relations for subclasses, add pk and attributes of superclass and Attributes of subclass.
c) only for disjoint subclasses. One single relation with all attributes of superclass, all subclasses and in addition one type-attribute (to show, which specialized type it is). Generates many null-values.
d) Works for overlapping and disjoint specialization. Same as option c, but for each specialization/subclass own type-attribute (mostly called "flags").

Physical Database Design

Disk contains platters. Platters have tracks which contain of sectors.
Access time of a disk: seek time (position arm over correct track) → avg seek time is ½ of the worst seek time(typically 2-10 ms); rotational delay (to find the right sector) → avg delay is ½ of the worst case. Rotational latency = time until target sector under head (typically 4-11 ms for 5400-15000 rpm).
Total Access = $T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$ Example: linear search for one type (total 20'000): avg = (20'000/2)/[tuples/block] * (seek + rot.delay + block transfer.time). binary search: lg#tuples.
data-transfer time: transfer data from/to disk.
block = continuous sequence of sectors of a track. Data between Ram and Disk are transferred in blocks; fix length (4-16kB). Separated by **interblock gaps.**

Buffer Replacement Policies
LRU strategy: Replace the block least recently used
MRU strategy: Replace the block most recently used
Pinned block: Memory block that is not allowed to be written back to disk.
Toss immediate strategy: Frees the space occupied by a block as soon as the final tuple of that block has been processed
=> MRU + pinned block is the best choice for the join(nested loop).

Fixed length records:
spanned = split→ pointers;
unspanned = records may not cross boundaries; free space in blocks if records do not fit. deletion/growing difficult; -> **Free List:** store address of first deleted record in header. Sore address of second deleted record in first record ..
variable length records:
slotted page structure: header stores number of record entries, location and size of each record) and end of free space in the block.



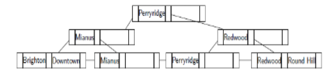
Organization of Records in Files:
Heap: place anywhere
Sequential (ordered by a defined search key), chained via pointers, difficult at insertion → locate record, if free marker, else insert in an overflow block, deletion → make deletion marker on free gaps. Periodic reorganization of files
Hash (hash function to find records).

Index Structures for Files

search key = (set of) attribute to look up records
clustering index: same order of data and index, usually sparse
non-clustering index: different order of data and index, usually dense
sparse index: index entry for some tuples only, applicable when records are sequentially ordered on the search key
dense index: index entry for each tuple
primary index: clustering index, search key is candidate key
secondary indexes: also called non-clustering indexes;
Two concepts for pointers:
a) duplicate index entries
b) buckets: index points to a bucket where all pointer in this "category" are stored.
Example: Relation with 6M tuples, primary Index, 50 tuples/block, 200indexes/block, random distributed between 0-100M. 1. Select > 75M: find first entry & scan through: total blocks for index = 30'000. Log(30'000) for first entry; 120'000/4 for block copies. → 15+30k.
SQL: Create (unique) index [name] on [table](columns) → can take long time, slow down DB. But indices are handled automatically by DBMS.

B+-Tree

Is a multilevel-index with following advantages: automatically maintains #of levels; reorganizes itself on insertion/deletions; no periodical reorganization necessary.
Properties: Ordered search keys, balanced tree.
Root: 0 to m-1 if it is a leaf, otherwise at least 2 children.
Internal nodes: between (ceiling)(m/2) and m children.
Leaf nodes: (ceiling)(m-1)/2 to m-1 search key values.
Number of levels = (ceiling)$\log_{m/2}(K)$
K = #values. One node = one disk block



Find record with search key value k (dense index):
1. Set C = root node
2. **while** C is not a leaf node **do**
 Search for the largest search key value ≤ k
  if such a value exists, assume it is K_i
  then set C = the node pointed to by P_i
  else set C = the node pointed to by P_0
3. **If** there is a key value Ki in C such that Ki = k
  then follow pointer Pi to the desired record or bucket
  else no record with search key value k exists
Insertion:

Algo: B+TreeInsert(L,k,p)
if *L is not yet full* **then**
 insert (k,p) into L
else
 create new node L';
 if *L is a leaf* **then**
  $L := L + (k,p); k' := L[\lceil (m+1)/2 \rceil]$;
  move entries greater or equal to k' from L to L';
 else
  $L := L + (k,p); k' := L[\lceil m/2 \rceil]$;
  move entries greater or equal to k' from L to L';
  delete entry with value k' from L;
 if *L is not the root* **then** B+TreeInsert(parent(L),k',L') ;
 else create new root with children L and L' and value k' ;

Deletion:

Algo: B+TreeDelete(L,k,p)
delete (p,k) from L;
if *L is root with one child* **then** root := child;
else if *L has too few entries* **then**
 L' is previous sibling of L [next if there is no previous] ;
 k' is value in parent that is between L and L';
 if *entries L and L' fit on one page* **then**
  if *L is leaf* **then** move entries from L to L';
  else move k' and all entries from L to L';
  B+TreeDelete(parent(L),k',L)
 else
  if *L is leaf* **then**
   move last [first] entry of L' to L;
   replace k' in parent(L) by value of first entry in L [L'];
  else
   move [first] last entry of L' to L;
   replace k' in parent(L) by value of first entry of L [L'];
   replace value of first entry in L [L'] by k';

coalesce
redistribute

Hashing

With hash function → find correct bucket within constant access time. → no index necessary. Different search keys (sk) lead sometimes to same bucket with h(sk) → seq. bucket search
An **ideal hash function** distributes **uniform** and **random**.
overflow chaining: if bucket is full, push it in the overflow bucket, linked to the normal bucked.
hash index: to find search key values → buckets with pointers to the data.
static hashing: not good because define #of buckets in beginning → many overflows in growing DB. Vice-versa, space wasted on shrinking DB.
Extendable hashing (dynamic): get bit representation of value. Take (only as much as needed) bits and compute hash value. Follow pointer to bucket.
Insertion: 1. Compute hash value and follow pointer
2. **If** free room in bucket **then** insert **else** bucket must be split and insertion re-attempted.
Split a bucket j when inserting search key value K_j:
If more than one pointer to the bucket **then** create new bucket, redirect pointer, rehash values of old bucket and new value, reinsert accordingly.
If only one pointer to the bucket **then** increment i of address table, replace each entry in the table by two entries, that point to the same bucket, recompute new bucket address table entry for K_j
Deletion: Remove it, remove bucket if empty. Coalescing of buckets can be done (can coalesce only with a buddy bucket having same value of i_j and same i_j -1 prefix if it is present).

Query Processing

Evaluation plan = execution plan = access plan. Every query is processed in 3 steps: 1. parsing & translation into RA query tree 2. optimization (chose plan with lowest cost) 3. Evaluation

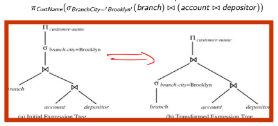
sorting: if relation fits into memory→ quicksort, else →
external sort-merge (read part of the relation into Ram, sort them, merge always two blocks;
cost [#blocks] * (2*$\log_{M-1}$(#blocks/M) + 1), M = #tuples in buffer.
Step 1: Create N sorted runs (N=#blocks/M), then sort
Step 2: Merge passes (N>=M): M-1 runs are merged.
Selection Evaluation Strategies:

A1: linear search, always works, scan & check all records → Avg cost = #blocks/2;
A2: binary search, only if ordered on attribute and contiguously stored → cost = (ceiling) $\log_2$(#blocks) + #satisfying blocks;
A3: primary index + equality on candidate key, single satisfying record → cost = heightOfTree + 1;
A4: primary index + equality on non-key → cost = heightOfTree + #blockswithsearchvalue;
A5: secondary index + equality on search key. if search-key is a candidate key → cost = heightOfTree + 1; if search-key is not cand. key → cost = heightOfTree + #BucketsWithSearchKey + #retrievedrecords;
A6: primary index on A + comparison cond. a ≥ x → find first tuple with index, scan relation sequentially. a ≤ x → scan relation sequentially until condition is false;
A7: secondary index on A + comparison cond, a ≥ x → find first index, scan index from there. a ≤ x → scan leaf pages of index until cond. is false.

Join Evaluation Strategies (r(outer)⋈s(inner)):
Nested loop join: two "for loops" checking for every tuple in r and s if they satisfy the condition.
cost = [#tuplesofR] * [#blocksS] + [#blocksR];
cost (if s fits entirely in memory) = [#blocksS] + [#blocksR]
Block nested loop join: same as before, but loops over each blockR, blockS, tuple in R, tuple in S, check if tupleR = tupleS (4 "for loops")
cost(worst) = [#blocksR] * [#blocksS] + [#blocksR];
cost(best) = [#blocksS] + [#blocksR]
cost(using M-2 disk blocks) = (ceiling)([#blocksR]/(M-2)* [#blocksS]+ [#blocksR])
Indexed nested loop join: Index lookups can replace file scans if join is an equi-join or natural join and index is available on the inner relation's join attribute (and) index can be constructed just to compute a join.
cost = [traversing index + fetching tuple] * [#tuplesR] + [#blocksR]
Merge join: each relation has a pointer, they move synchronized through the sorted relation.
cost = [#blocksR] + [#blocksS] (+sorting cost)
Hash join: same hash function for both relation
cost = 3*(blocksR + blocksS)

Query Optimization

- **Example:** Find the names of all customers who have an account at any branch located in Brooklyn.
  - $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$
    - Produces a large intermediate relation
    - Transformation into a more efficient expression $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie (account \bowtie depositor))$



V(A,r) = number of distinct values in r for attribute A;
SC(A,r) = average number of records satisfying equality on a.
Chose cheapest evaluation strategy using heuristic and rule-based optimization strategies.
Equivalence rules:
ER1: $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
ER2: $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
ER3: $\pi_{L_1}(\pi_{L_2}(\dots(\pi_{L_n}(E))\dots)) = \pi_{L_1}(E)$
ER4: (a) $\sigma_\theta(E1 \times E2) = E1 \bowtie_\theta E2$
(b) $\sigma_{\theta_1}(E1 \bowtie_{\theta_2} E2) = E1 \bowtie_{\theta_1 \wedge \theta_2} E2$
ER5: $E1 \bowtie_\theta E2 = E2 \bowtie_\theta E1$
ER6: (a) (E1 ⋈ E2) ⋈ E3 = E1 ⋈ (E2 ⋈ E3)
(b) $(E1 \bowtie_{\theta_1} E2) \bowtie_{\theta_2 \wedge \theta_3} E3 = E1 \bowtie_{\theta_1 \wedge \theta_3} (E2 \bowtie_{\theta_2} E3)$
ER7: (a) $\sigma_{\theta_0}(E1 \bowtie_\theta E2) = \sigma_{\theta_0}(E1) \bowtie_\theta E2$ (When all attributes in θ0 involve only the attributes of one of the expressions (E1) being joined)
(b) $\sigma_{\theta_1 \wedge \theta_2}(E1 \bowtie_\theta E2) = \sigma_{\theta_1}(E1) \bowtie_\theta \sigma_{\theta_2}(E2)$ (When θ1 involves only the attributes of E1 and θ2 involves only the attributes of E2)
ER8: Let L1 and L2 be sets of attributes from E1 and E2, respectively.
(a) if θ involves only attributes from L1∪L2:
$\pi_{L_1 \cup L_2}(E1 \bowtie_\theta E2) = \pi_{L_1}(E1) \bowtie_\theta \pi_{L_2}(E2)$
(b) Consider a join E1 ⋈θ E2. Let L3 be attributes of E1 that are involved in join condition θ, but are not in L1 ∪L2, and let L4 be attributes of E2 that are involved in join condition θ, but are not in L1 ∪L2, and
$\pi_{L_1 \cup L_2}(E1 \bowtie_\theta E2) = \pi_{L_1 \cup L_2}(\pi_{L_1 \cup L_3}(E1) \bowtie_\theta \pi_{L_2 \cup L_4}(E2))$
ER9: E1∪E2 =E2∪E1 , E1∩E2 =E2∩E1
ER10: (E1∪E2)∪E3 =E1∪(E2∪E3)
(E1∩E2)∩E3 =E1∩(E2∩E3)
ER11: $\sigma_\theta(E1-E2) = \sigma_\theta(E1)-\sigma_\theta(E2)$
$\sigma_\theta(E1 \cup E2) = \sigma_\theta(E1) \cup \sigma_\theta(E2)$
$\sigma_\theta(E1 \cap E2) = \sigma_\theta(E1) \cap \sigma_\theta(E2)$
Also $\sigma_\theta(E1-E2) = \sigma_\theta(E1)-E2$ and similarly for ∩ in place of −, but not for ∪.

ER12: $\pi_L(E1 \cup E2) = \pi_L(E1) \cup \pi_L(E2)$

Cost-based optimization: use equivalence rules, for each generated plan, use cost formulas for relation → take candidate with least cost. Cost-based optimization is expensive but on big relation worth. Example: X=attributes in r1 and r2: r1 join r2 = |r1| * |r2| / V(X,r2) not the same as r2 join r1 = |r2| join |r1| / V(X,r1)

heuristic optimization: perform selection & projection early. Perform most restrictive operations (selection, join..) before other similar operations → reduce size of relation early.

Steps in typical heuristic optimization:
1. Break up conjunctive selections into a sequence of single selection operations (rule ER1).
2. Move selection operations down the query tree for the earliest possible execution (rules ER2, ER7(a), ER7(b), ER11).
3. Execute first those selection and join operations that will produce the smallest relations (rule ER6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (rule ER4(a)).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (rules ER3, ER8(a), ER8(b), ER12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

Transaction Processing

Transaction States: Active; Partially committed (after final statement); Committed (completed, changes permanent); Failed; Aborted (state recovered as before transaction → restart or kill transaction)
ACID = Atomicity (all operations of the transaction or none); Consistency (no violation of integrity constraints); Isolation (transactions do not influence each other); Durability (persistent changes after commit).
Schedule: **Serial** = one after another;
concurrent = multiple transactions at same time;
Serializable = schedule is equivalent to a serial schedule;
Conflict when write/write or read/write or write/read
conflict equivalent schedules = schedule transformed in another schedule without conflict
conflict serializable schedules = schedule is conflict equivalent to a serial schedule; attention on **blind writes** = write operations without reading
Precedence graph: T1 → T2 if a) T1 write before T2 read; b) T1 read before T2 write c) T1 write before T2 write; If cycle → not serializable. Acyclic → order graph topological
Recoverability: If T1 → T2, T2 can't commit before T1 does, even if it's already finished. Can lead to cascading rollback(single transaction fail leads to multiple rollbacks) of all following transactions.
Cascadeless schedules: For each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj. This avoids cascading rollbacks.

Locks & Deadlock

X-lock(item) read and write possible; **S-lock**(item) only read; **U-lock**(item) for releasing the lock;
If any transaction holds an exclusive lock on a data item no other transaction may hold any lock on that item.
Dirty read = T2 sees uncommitted changes of T1;
Nonrepeatable read = read same field and get different results;
Phantom read = read multiple rows twice and get different results
Two phase locking protocol (normal): 1) growing phase (locks possible); 2) shrinking phase (unlocks possible);
Lock-point = transition point from phase 1 to phase 2; no guarantee for deadlock-freedom, but ensures serializability; lock conversions, upgrades, downgrades possible (from X-lock to S-lock, vice-versa)
Strict two-phase locking: must hold all exclusive locks until commit.
Rigorous two-phase locking: All locks hold until commit → serializable upon their commit
Schedule order: schedules, correct schedules, conflict serializable schedules, 2PL schedules, serial schedules
isolation levels:

Phenomena / Isolation level	Dirty read	Nonrepeatable read	Phantom read
read uncommitted	yes	yes	yes
read committed	no	yes	yes
repeatable read	no	no	yes
serializable	no	no	no

Examples

SQL
Find the full name and the age of persons who will have birthday within the next 60 days
SELECT fname, minit, lname, age(date_birth)
FROM Person
WHERE EXTRACT(year FROM age(date_birth)) <> EXTRACT(year FROM (age (date(now())+60, date_birth)));

FD
Consider the relation Cust(idc; fname; lname; street; city; state; zip) with the following functional dependencies:
idc → fname lname street city state zip
street city state → zip
zip → state
Cand. Key: idc
Satisfies 1NF, 2NF
Decomposition into 3NF:
Cust1(idc, fname, lname, street, city, state), cand key: idc
Cust2(street, city, state, zip), c.keys: street, city, state and street, city, zip
BCNF violations: zip → state
Decomposition into BCNF:
Cust1(idc, fname, lname, street, city, state)
Cust3(street, city, zip)
Cust4(zip, state)
Minimal cover
Find the minimal cover Fmin of
F = {A → CD;AB→D;D→E;C → D}
Solution: Fmin = {A → C,C → D,D → E}.

Determine Candidate Keys
1. Check if there exist attributes of relation schema R that do not occur in a functional dependency of Fr. Such attributes are part of every candidate key of R.
2. Check if there exist attributes of relation schema R that only occur on the left side of the functional dependencies of Fr. Such attributes are part of every candidate key of R.
3. Check the closure for each set of attributes with one element while considering the outcome of the two checks made before.
4. Check the closure for each set of attributes with two elements while considering the outcome of the checks made before. Consider only sets that are not a superset of candidate keys determined in the previous step.
5. …(with three/four…)

Locks
Consider transactions:
T1: r(A); r(B); if A = 0 then B:= B+1; w(B)
T2: r(B); r(A); if B = 0 then A:= A+1; w(A)
Add lock and unlock instructions to the serial schedule:
T1: sl1(A), r1(A), xl1(B), r1(B), if A = 0 then B:= B+1, w1(B), ul1(A), ul1(B)
T2: sl2(B), r2(B), xl2(A), r2(A), if B = 0 then A:= A+1, w2(A), ul2(B), ul2(A)
Show a 2PL schedule that leads to a deadlock:
sl1(A), r1(A), sl2(B), xl1(B), r1(B), xl2(A), r2(A)

DRC
The groups of islands such that the group contains exactly two islands of each type (e.g., volcanic, coral, etc.)

{Islands | island(,Islands, , , ,) ∧
∀Type(island(, , , ,Type,) ⇒
∃i1,i2(island(i1,Islands, , ,Type,) ∧
island(i2,Islands, , ,Type,) ∧ i1 != i2 ∧ !∃
i3(island(i3,Islands, , ,Type,) ∧ i1 != i3∧i2 != i3)))}

SQL Recursion
find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)
WITH RECURSIVE mgr(EmpName,MgrName) AS (SELECT EmpName, BossName FROM empl UNION SELECT empl.EmpName, mgr.MgrName FROM empl, mgr WHERE BossName = mgr.EmpName) SELECT * FROM mgr;

SL07 20, 21
Consider a disk as follows: block size B = 512 Bytes, interblock gap size G = 128 Bytes, blocks per track B/T = 20, tracks per surface T/S = 400, double-sided disks D = 15, seek time st = 30 msec, 2400 rotations per minute. Determine the following values:
1. total capacity per track = 20*(512+128) = 12.8 KB
2. useful capacity per track = 20*512 = 10.24 KB
3. number of cylinders = number of tracks = 400
4. useful capacity per cylinder = 10.24*15*2
5. transfer rate tr = (2400/(60*100))*12.8 = 512Bytes/msec
6. block transfer time btt = B/tr = 512/512 = 1 msec
7. rotational delay rd = ((60*1000)/2400)*0.5= 12.5msec(time for half a rotation)
8. bulk transfer rate btr = tr *(B/B+G) = 512 * 0.8 = 409.6 Bytes/msec
9. block read time = rd+btt+st = 30 + 12.5 + 1 = 43.5msec
10. time for 20 random reads = 20 * 43.5 = 870msec
11. time for 20 sequential reads = 20 * btt + st + rd = 62.5msec [or 20 * B/btr + st + rd = 67.5]

SL07 37,38
Assume a disk with the following characteristics: block size B = 512 Bytes, blocks per track = 20, tracks per surface = 400, number of double-sided disks = 15, rotations per minute = 2400 rpm, seek time = 30 msec. Assume a relation Emp(N 30 Bytes, SSN 9 Bytes, A 40 Bytes, P 9 Bytes) with 20'000 tuples. Determine the following values:
1. HD capacity = 20 ∗ 512 ∗ 400 ∗ 2 ∗ 15 = 122.9MB
2. size of 1 Emp tuple = 30 + 9 + 40 + 9 + 1 = 89Bytes (1 as, e.g., a deletion marker)
3. blocking factory (bfr) of Emp (= number of tuples per block) = (floor)B/89 = 5tuples/block
4. number of blocks used by Emp (unspanned organization)= 200000/5 = 4000blocks
5. number of blocks used by Emp (spanned organization) = 200000 ∗ (89/(512 − 4)) = 3505blocks (4 Bytes for pointer to next block)
6. average time for a linear search in Emp (contiguous file) = 2000 ∗ btt + st + rd = 2.04sec (half of blocks on avg, btt=1msec, st=30msec, rd=12.5msec

SL08 31, 32
Consider e 1SSN=MgrSSN d with rd =50 (number of tuples in relation d), re =5000, bd =10 (number of blocks for relation d), be =2000, nb =6 (number of available buffer blocks). Compute the number of IOs for the following evaluation strategies:
1. Block NL, e 1 d, 4 blocks for e (1 block for d, 1 block for result) = (ceiling)(b_e/4)∗ b_d + b_e = 7000
2. Block NL, e ⋈ d, 4 blocks for D = be ∗ bd + be = 22000
3. Block NL, d ⋈ e, 4 blocks for D = (ceiling)(bd/4)∗ b_e + b_d = 6010
4. Indexed NL, e ⋈ d = (we assume a B+ index on d.MgrSSN of height 4) b_e + n_e(4 + 1) = 17000
5. Indexed NL, d ⋈ e = b_d + n_d(4 + 1) = 260

SL08 38
Consider bc =400, nc =100000, bd =100, nd =50000, disk IO time = 10 msec, memory access time = 60 nsec. Compare the execution times for NL and sort merge (best case):
# of IOs = bc + bd for both cases (best case = enough memory, relations already sorted)
# of tuple comparisons in memory:
– NL = nc ∗ nd
– SM = nc + nd (unique attribute values)
exact time NL: 10 ∗ (bc + bd) + 60 ∗ 10−6(nc ∗ nd) = 5msec + 3msec = 8msec
exact time SM: 10 ∗ (bc + bd) + 60 ∗ 10−6(nc + nd) = 5msec + 0.9msec = 5.9msec

Review 8.8 difficult
Consider a DB with the following characteristics: O … |r1|·|r2|
- $|r1(A,B,C)| = 1000, V(C,r1) = 900$
- $|r2(C,D,E)| = 1500, V(C,r2) = 1100, V(E,r2) = 50$
- $|r3(E,F)| = 750, V(E,r3) = 100$

Estimate the size of $r1 \bowtie r2 \bowtie r3$ and determine an efficient evaluation strategy.
✓ $cost(r_1 \bowtie r_2) = |r_1| \cdot \frac{|r_2|}{V(C,r_2)} = 1364$
$cost(r_3 \bowtie r_2) = |r_3| \cdot \frac{|r_2|}{V(C,r_1)} = 22'500$
$cost(r_1 \bowtie r_3) = 750'000$ (cr)
$cost(r_2 \bowtie r_1) = 1667$
✓ $cost(r_2 \bowtie r_3) = |r_2| \cdot \frac{|r_3|}{V(E,r_3)} = 11'250$
$cost((r_1 \bowtie r_2) \bowtie r_3) = cost(r_1 \bowtie r_2) \cdot \frac{|r_3|}{V(E,r_3)} = 10'230$
DBS 2017, SL08 73/82 M. Böhlen, IfI@UZH

Cost-Based Optimization Example/3 (# of IOs)
- Emp $\bowtie_{DNo=DNum}$ Dept
- Statistics:
  - |Emp| = 10'000 tuples ; 5 Emp tuples per block
  - |Dept| = 125; 10 Dept tuples per block
  - Hash index on Emp(DNo)
  - 4 EmpDept result tuples per block
- Plan p1: Block nested loop with Emp as outer loop
  - cost(p1) = (10'000/5) + (10'000/5) * (125/10) + (10'000/4) = 30'500 IOs
  - (10.000/4 is cost of writing final output)
- Plan p2: Indexed nested loop with Dept as outer loop and hashed lookup in Emp
  - cost(p2) = (125/10) + 125 * (10'000/125/5) + (10'000/4) = 4'513 IOs
  - 10'000/125/5 is the average number of blocks/department

bE, # of result blocks, bE, bD, # of Emp blocks, bD, 10/2, nD (Anzahl Tupel in Dept)
DBS 2017, SL08 76/82 M. Böhlen, IfI@UZH

Review 9.5/3 = ⟨T1; T2; T1'; T2'⟩ or ⟨T1; T2; T2'; T1'⟩
Assume relation r(a) = {(1), (2), (3)} and 2PL with isolation level set to serializability. What is the behavior of the following schedule:

```
T1: SELECT SUM(a) FROM p;             6
T1: INSERT INTO p VALUES (6);         done
T2: SELECT SUM(a) FROM p;             blocked
T2: INSERT INTO p VALUES (6);         blocked
T1: SELECT SUM(a) FROM p;             12
T2: SELECT SUM(a) FROM p;             blocked
T2: COMMIT;                           blocked
T1: SELECT SUM(a) FROM p;             12
T1: COMMIT;  T1 done / T2: 12; done (inserted); 18; committed
T1': SELECT SUM(a) FROM p;            18
T2': SELECT SUM(a) FROM p;            18
```

The "SOLA-Stafette" is an annual relay race that takes place in and around Zurich. Table sola stores for each SOLA edition (identified by the year) the SSN of each participating runner, the name of the team he or she belongs to, the track number assigned to this runner and the time (in seconds) he or she needed to complete the track. The attributes that compose the primary key are underlined. A sample database instance is illustrated below:

sola

Year	TeamName	TrackNr	RunnerSSN	Time
2017	'IFI Runners'	11	1234	3612
2017	'IFI Runners'	13	4567	2344
2017	'Leadville 100 Heroes'	13	6688	2019
2016	'Leadville 100 Heroes'	11	3334	2892

Tasks:
(a) Write a SQL query that returns the names of all teams that participated in year 2017 and whose runners in 2017 all had a smaller run time than the average run time for his or her track in 2017. Return every team name only once:

```
SELECT DISTINCT TeamName
FROM sola s1
WHERE s1.YEAR = 2017 AND NOT EXISTS (
    SELECT RunnerSSN
    FROM sola s2
    WHERE s2.YEAR = 2017 AND s2.TeamName = s1.TeamName
        AND s2.TIME >= (
            SELECT AVG(TIME)
            FROM sola s3
            WHERE s3.TrackNr = s2.TrackNr AND s3.YEAR = 2017 ) );
```

(b) Write a domain relational calculus (DRC) expression that returns the SSN of all runners who ran track 11 in year 2017 and finished at most 600 seconds behind the fastest runner for track 11 in year 2017.
{RS | ∃T(sola(2017, ,11,RS,T) ∧ ∃T1(sola(2017, ,11, ,T1) ∧ !∃ T2 (sola(2017, ,11, ,T2) ∧ T2 < T1) ∧ T ≤ T1 + 600))}

{RS | ∃T(sola(2017, ,11,RS,T) ∧ ∃T1(sola(2017, ,11, ,T1) ∧ ∀T2(sola(2017, ,11, ,T2) ⇒ T2 ≥ T1)∧T ≤ T1 + 600))}
(c) Write a relational algebra (RA) expression that returns the SSN of all runners who ran every track at least once. You can assume that every track number is present in attribute TrackNr of table sola:
$\pi_{RunnerSSN}(sola) -$
$\pi_{RunnerSSN}(\pi_{TrackNr}(sola) \times \pi_{RunnerSSN}(sola)) - \pi_{TrackNr,RunnerSSN}(sola))$

Equivalent solution using aggregation:
$x \leftarrow \rho_{NrT}(\vartheta_{COUNT(TrackNr)}(\pi_{TrackNr}(sola)))$
$y \leftarrow \rho_{RunnerSSN,NrT}(_{RunnerSSN}\vartheta_{COUNT(TrackNr)}(\pi_{RunnerSSN,TrackNr}(sola)))$
$\pi_{RunnerSSN}(x \bowtie y)$
1.2 Let r and s be two relations with schemas R(A,B) and S(A,C), respectively. Determine for each of the following queries if they are equivalent to r⋈s. You can assume that the relations do not contain duplicates.

Query	Equivalent to r ⋈ s	Not equivalent to r ⋈ s
$(r ⟕ s) \cap (r ⟖ s)$	☒	☐
$\pi_{A,B}(r \bowtie s) \bowtie \pi_{A,C}(r \bowtie s)$	☒	☐
$(r \times \rho_{(A',C)}(s)) - \sigma_{A \neq A'}(r \times \rho_{(A',C)}(s))$	☐	☒

1.3 Consider relation instance r with schema R(A,B,C,D) shown in the table below.

r

A	B	C	D
0	2	3	4
NULL	2	NULL	1
NULL	NULL	6	NULL
0	2	NULL	NULL
0	NULL	2	1
NULL	2	NULL	0

(a) Evaluate the following query on relation instance r and write down the result in the table next to the given query.

```
SELECT r1.A AS A,
       r1.B AS B,
       r1.C AS C
FROM r r1, r r2
WHERE r1.D = r2.D;
```

A	B	C
0	2	3
NULL	2	NULL
NULL	2	NULL
0	NULL	2
0	NULL	2
NULL	2	NULL

(b) Evaluate the following query on relation instance r and write down the result in the table next to the given query.

```
SELECT A,
       B,
       COUNT(D) AS Nr
FROM r
GROUP BY A, B;
```

A	B	Nr
0	2	1
NULL	2	2
NULL	NULL	0
0	NULL	1

(a) X -->> Y |= X →Y
The statement is not correct. Consider the relation schema Coach(Name,SportClass,Salary) with the following instance (N-->>S vs N->S):

Coach

	Name	SportClass	Salary
t_1	'Ana'	'Golf'	1200
t_2	'Ana'	'Tennis'	1200

$X \twoheadrightarrow YZ \models X \twoheadrightarrow Y, X \twoheadrightarrow Z$ (2.5 points)
The statement is not correct.

CoachInfo

	Name	MobileOperator	PhoneNumber	PetName	PetType
t_1	'Ana'	'Salt'	'555-55-55'	'Fluffy'	'dog'
t_2	'Ana'	'Salt'	'555-55-55'	'Coco'	'cat'
t_3	'Ana'	'Sunrise'	'111-11-11'	'Fluffy'	'dog'
t_4	'Ana'	'Sunrise'	'111-11-11'	'Coco'	'cat'

We assume, that a coach may have several phone numbers from different mobile operators (but not more than one number per operator) and may have several pets. Then the multivalued dependency Name -->> MobileOperator,PhoneNumber holds. However, the instance of CoachInfo shows the counterexample for Name -->> MobileOperator.
2.3 Given relation schema Storage(Company,Warehouse,SpaceLimit) with the following set FStorage of functional dependencies.

$F_{storage}$={Company->Warehouse, Warehouse->Spacelimit}
The rel. schem. is decomposed into S_1(C, W), S_2(W,SL).
(a) state anomaly that is eliminated by decomposition:
An update anomaly
(b) Give an instance of Storage, where the anomaly can occur, and the example of this anomaly with the explanation why it may occur in Storage.

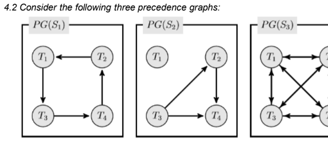
Storage

	Company	Warehouse	SpaceLimit
g_1	'Migros'	'Spacy'	1200
g_2	'Coop'	'Spacy'	1200

If we update the information about space limit for the company 'Migros' and warehouse 'Spacy' (tuple g1) to 2400, then the information, that is stored in Storage, will become inconsistent because tuple g2 includes the contradictory fact about space limit of 'Spacy'.
(c) Give the corresponding instances of Storage1 and Storage2 based on instance of Storage. Explain why the anomaly can no longer occur:
The update anomaly can not occur after the decomposition. Storage1 and Storage2 have exactly one entry per company and warehouse, respectively

4.1 Consider the following two transactions T1 and T2:
T1 = r1(A), r1(B), w1(A), w1(C), r1(A), c1
T2 = w2(A), w2(B), r2(C), w2(C), r2(A), c2
(a) Construct a schedule S1 with transactions T1 and T2 that is not recoverable and is conflict-serializable:
S1 = r1(A), r1(B), w1(A), w1(C), r1(A), w2(A), w2(B), r2(C), w2(C), r2(A), c2, c1
(b) Construct a schedule S2 with transactions T1 and T2 that is cascadeless and is not conflict-serializable. (5 points, you only receive partial points if you explain your approach)
S2 = r1(A), r1(B), w2(A), w2(B), r2(C), w2(C), r2(A), c2, w1(A), w1(C), r1(A), c1

4.2 Consider the following three precedence graphs:



Construct for each of the three precedence graphs a schedule that results in this precedence graph. The schedules have must obey the constraints on the number of allowed operations. An operation can be a read or a write operation.
(a) Schedule S1 consists of a total of 6 (six) operations:
S1 = r2(A), w1(A), r3(A), r3(B), w4(B), r2(B), c1, c2, c3, c4
(b) Schedule S2 consists of a total of 4 (four) operations:
S2 = r1(A), w3(B), w2(B), w4(B), c1, c2, c3
(c) Schedule S3 consists of a total of 7 (seven) operations:
S3 = w1(A), w2(A), w3(A), w4(A), w3(A), w2(A), w1(A), c1, c2, c3, c4

3.1 Consider the following DDL statements to create four tables. Construct an ER-diagram for this database. If relationship cardinalities are unclear, pick the ones that do not violate the DDL statements. Each table must map to one entity (either weak or strong) in the ER-diagram

CREATE TABLE a(
A INTEGER,
B INTEGER,
C INTEGER,
PRIMARY KEY (A));

CREATE TABLE b(
D INTEGER,
E INTEGER,
A INTEGER,
PRIMARY KEY (A,D),
FOREIGN KEY (A)
REFERENCES a(A));

CREATE TABLE c(
F INTEGER,
G INTEGER,
H INTEGER,
A INTEGER,
PRIMARY KEY (F,G),
FOREIGN KEY (A)
REFERENCES a(A));

CREATE TABLE d(
I INTEGER,
J INTEGER,
F INTEGER,
G INTEGER,
PRIMARY KEY (I),
FOREIGN KEY (F,G)
REFERENCES c(F,G));