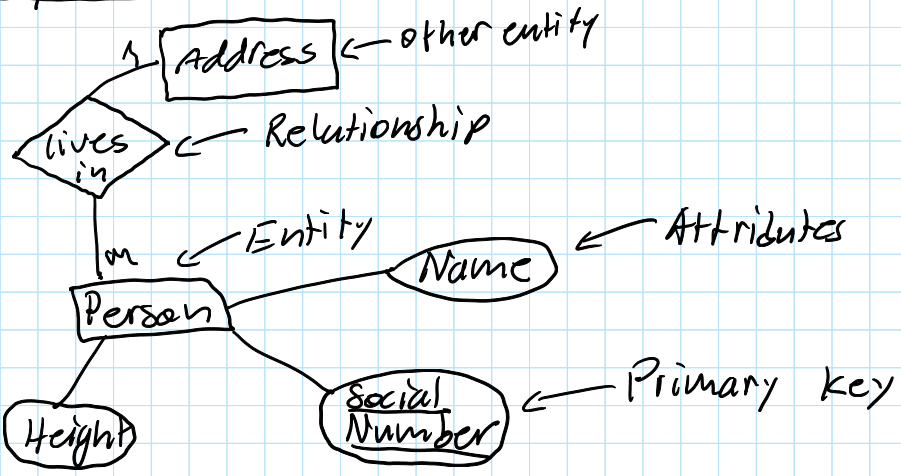
Entity Relationship Model

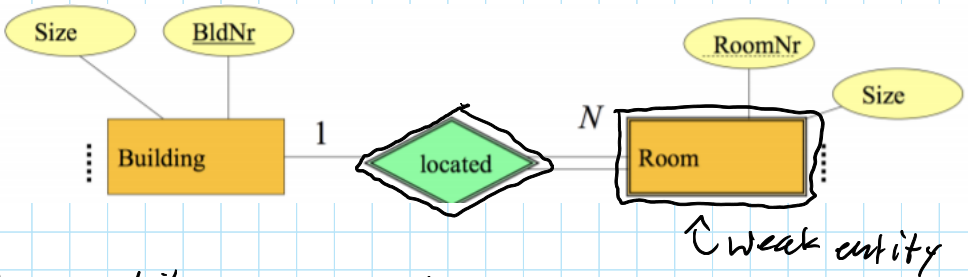


- **Candidates keys**: Set of attributes, that we can later choose one of to be the primary key

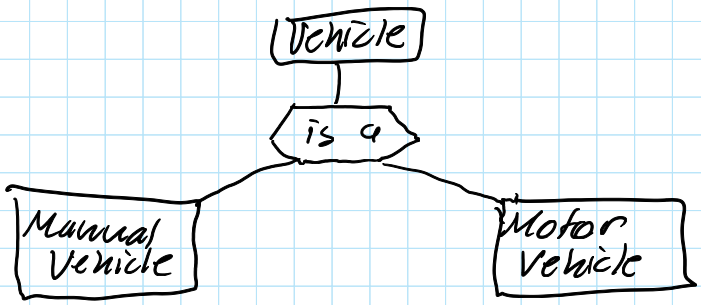
- **Cardinality**:
  - 1:1
  - 1:N
  - N:1
  - N:M

  } A relation, even of type 1:1, optionally connects entities!

- **Weak entity**:
  - Existence of room depends on the existence of the associated building
  - Primary key of a room: BldNr + RoomNr



- **Generalization**: - Like a parent



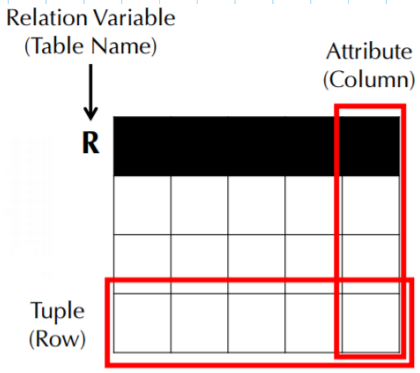
- A vehicle can be both manual-vehicle and motor-vehicle at the same time
- It does not require, that each entity of Vehicle is either a manual-vehicle or motor-vehicle

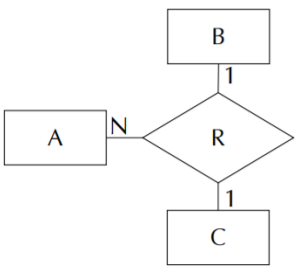
Keys

- Key: Must be unique
- Superkey: Any set of attributes, which identify an entry uniquely
- candidates key: Minimal set of superkeys
- Primary key: Just one particular candidates key
- Foreigner key: Attribute in a table, which has a relation to another table, where that attribute is the primary key
  - For every foreign key one of the two conditions must hold:
    1. Value of foreign-key is NULL
    2. Referenced tuple must exist

Relation Model

- Student: {[Legi: Integer, Name: String, Semester: Integer]}
- attends: {[Legi: Integer, Nr: Integer]}
  - Primary key müssen Beide sein, da es sich um eine m:n-Relation handelt

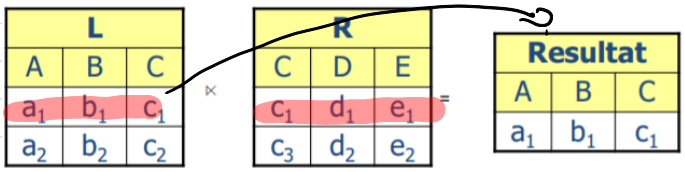




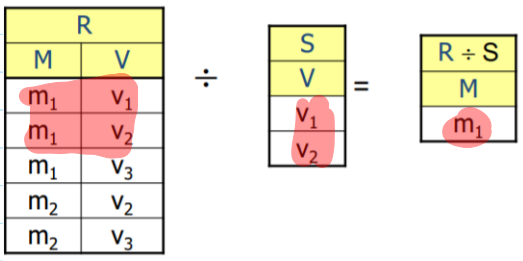
Primary key: $R(\underline{A}, \underline{B}, C)$ or $R[\underline{A}, B, \underline{C})$

Relational Algebra

- Selection: $\sigma_{semester > 10}(Student)$
- Projection: $\pi_{Level}(Professor)$: only outputs the different levels
- Cartesian product: Jede Zeile mit jeder anderen Zeile
- Natural join: $A \bowtie B$: Matches tuples (= inner join)
- Left outer join: $\bowtie$: natural join + unmatched tuples from A
- Right outer join: $\bowtie$ natural join + unmatched tuples from B
- Full outer join: $\bowtie$ natural join + unmatched tuples from A/B
- Left semi join: $\ltimes$ tuples from A matching tuples from B



- Right semi join: $\rtimes$ tuples from B matching tuples from A
- Rename: $\rho_{Requirements \leftarrow prerequisite}(requires)$
- Intersection: $A \cap B$
  - Only works, if both tables have same attributes
- Relational division:



---

n-way join

- $A \bowtie B \bowtie C \Rightarrow$ 3-way join
- Possible ways to order n-way join: $\frac{2n!}{n!}$

SQL (=structured query language)

Data types ← #chars

- **character** *(n)*, ***char*** *(n)*
- **character varying** *(n)*, **varchar** *(n)*
- **numeric** *(p,s)*, **integer**
- **blob** or **raw** for large binaries
- **clob** for large string values (character large object)
- **date**

Create Tables

- **create table** Professor
  (PersNr **integer not null,**
  Name **varchar** (30) **not null**
  Level **character** (2) **default** „AP“);

Select *
From A, B
WHERE A.id = B.id

≙

Select *
From A
Join B ON A.id = B.id

Delete a Table

- **drop table** Professor;

Select partid
FROM part

≙

select p.partid
FROM part p

Modify the structure of a Table

- **alter table** Professor **add column**(age **integer**);

Management of Indexes (Performance tuning)

- **create index** myIndex **on** Professor(name, age);
- **drop index** myIndex;

- **Delete tuples**

```
delete Student
where Semester > 13;
```

- **Update tuples**

```
update Student
  set Semester= Semester + 1;
```

```
select   PersNr, Name, Level
from     Professor
order by Level desc, Name asc;
```

```
select distinct Level
from Professor
```

} Eliminate duplicates

```
select Name
from Professor, Lecture
where PersNr = ProfNr and Title = `Operating Systems' ;
```

$$\prod_{\text{Name}}(\sigma_{\text{PersNr}=\text{ProfNr} \wedge \text{Title}='\text{OperatingSystems}'}(\text{Professor} \times \text{Lecture}))$$

} From Rel. Algebra to SQL

```
select s.Name, l.Title
from Student s, attends a, Lecture l
where s.Legi = a.Legi and
      a.Nr = l.Nr;
```
} shorter names

```
select Title, PersNr as ProfNr
from Lecture;
```
} Rename Attributes

```
( select Name
  from Assistant )
union
( select Name
  from Professor);
```
} union / intersect / minus

```
select avg (Semester)
from Student;
```
} Functions: avg, max, min, count, sum

```
select Name
from Professor
where PersNr (not) in ( select PersNr
                        from Lecture);
```
('1001', '1002')

} Subqueries

where PersNr in (...)
AND PersNr not in (...)

Null-Values in SQL

- 0 + null => null
- 0 * null => null
- null = null => unknown
- null < 13 => unknown

Cases

```
select Legi, ( case when Grade >= 5.5 then 'sehr gut'
                    when Grade >= 5.0 then 'gut'
                    when Grade >= 4.5 then 'befriedigend'
                    when Grade >= 4.0 then 'ausreichend'
                    else 'nicht bestanden' end)
from tests;
```

Group by / having

```
group by origin
having count (orderid)
       < 10
```

Comparisons

- "%" represents any sequence of characters (0 to n)
- "_" represents exactly one character
- N.B.: For comparisons with = , % and _ are normal chars.

```
select *
from Student
where Name like 'Kossman%';
```

Views

```
CREATE VIEW FlightFromPEK AS
SELECT * FROM Flight WHERE orig=PEK;

SELECT *
FROM FlightFromPEK
WHERE dest = ZRH;
```

Inline
With FlightFromPek AS (--)

- Not updatable:
  - MAX/count/MIN/AVG
  - Distinct-keyword
  - Join
  - Group by or Having
  - More than one table
  - Subqueries
  - Not include the primary key in select
  - Arithmetic expressions

Recursion

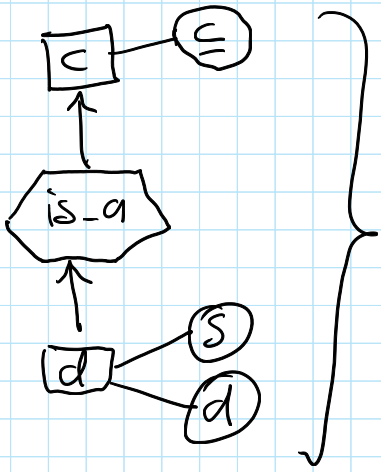
```
With
   AncesterofD as (
        select  parent
        from    parentOf
        where   D = child;

        UNION

        select  p2.parent
        FROM    AncestorofD,
                ParentOf
        WHERE   AncestorofD.ancester = parentof.child;
   )
```

Recursive call of

Converting Relationships

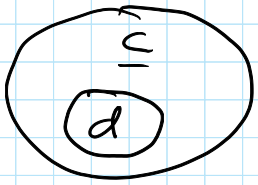
- M:N => A:(a) / B:(b) / rel:(a,b)
- 1:N => A:(a) / B:(b, a)
- 1:1 => rel:(a, b) or rel:(a, b)

Specification



d:(⊆, s, d)

↑ d is a subset, if we know c, we also know d



Constraints

- How database makes sure changes are consistent and do not cause trouble
- Problems: · Inserting tuples without a key
  · Adding references to tuples that do not exist
  · Negative age
- CREATE TABLE Student
  (sID int primary key, ← constraint
  sName text,
  GPA real not null); ← constraint

- Only one primary-key, but if some other column is as well unique, use keyword "unique"
- Primary key over two columns:

```
CREATE TABLE Student
(sID int,
sName text,
GPA real,
primary key (sID, sName));
```

Constraint key
primary key (..., ...)

Constraint test
check (...)

- Primary key can't be NULL
- Unique attribute can be NULL
- Checking constraints
  - GPA real check [(GPA <= 4.0 and GPA >= 0.0), or (...)]
- Foreign key : FOREIGN KEY (personID) REFERENCES Persons(personID)
  - on delete cascade
    - ↳ If record is deleted in Persons, it is also deleted in the new table
  - on delete set null
    - ↳ If record is deleted in Persons, it foreign-key entry in the new table is set to null
  - on delete set default
  - on delete no action
  - on delete restrict

  } causes an error ↳ No modification

  ↳ Same for on modify

Redundancy

- Update anomaly: We need to update the general information in every row
- Insertion anomaly: Add a new general element row
- Deletion anomaly: If we remove all elements, the general element is also deleted

---

Functional dependencies

Professor: {[PersNr, Name, Level, Room, City, Address, Zip, AreaCode, Canton, Population, Direktion]}

- {PersNr} → {PersNr, Name, Level, Room, City, Address, Zip, AreaCode, Canton, Population, Direktion}
- {City, Canton} → {Population, AreaCode}
- {Zip} → {Canton, City, Population}
- {Canton, City, Address} → {Zip}
- {Canton} → {Direktion}
- {Room} → {PersNr}

- $\alpha \subseteq R$ is a superkey if: $\alpha \rightarrow R$

Minimal Basis

- Minimal set of functional dependencies

1. Left side reduction
   - $\hat{x}Y \rightarrow A$
   - $Y \rightarrow X$
   - ↳ $Y \rightarrow A$
2. Right side reduction
   - $CF \rightarrow BD$
   - $BC \rightarrow D$
   - ↳ $CF \rightarrow B$
3. Eliminate FDs: $X \rightarrow \emptyset$
4. Apply union rule to FDs with same left side

Decomposition of relations

- $R = R_1 \cup R_2$ ↳ Only proves, that relation is lossless and not, that it is lossy!
- Decomposition is lossless, if
  - $(R_1 \cap R_2) \rightarrow R_1$ or
  - $(R_1 \cap R_2) \rightarrow R_2$
    ↑ Intersection

↳ $S = S_1 \cup S_2$

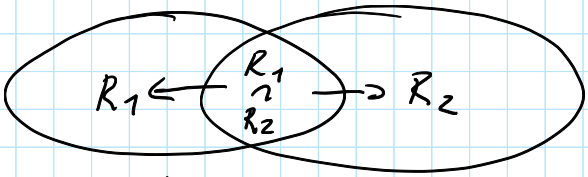
$S = S_1 \bowtie S_2$

Drinker		
Pub	*Guest*	*Beer*
Kowalski	Kemper	Pils
Kowalski	Eickler	Hefeweizen
Innsteg	Kemper	Hefeweizen

$\Pi_{Pub, Guest}$ $\Pi_{Guest, Beer}$

Visitor	
Pub	*Guest*
Kowalski	Kemper
Kowalski	Eickler
Innsteg	Kemper

Drinks	
Guest	*Beer*
Kemper	Pils
Eickler	Hefeweizen
Kemper	Hefeweizen



Lossy decomposition

- $R_1 \cap R_2 = Guest$
- Guest ↛ Beer
- Guest ↛ Pub

Normal Forms / Normalization

- Tequnique of organizing the data into multiple related tables to minimize data-redundancy

1st normal form

- Each attribute should only contain one element
- A column should contain values of same type
- Each column / attribute should have a unique name

2nd normal form (= Dependency on primary key)

- All other columns must fully depend on all primary-keys (not just one primary key)

IDSt	LastName	IDProf	Prof	Grade
1	Mueller	3	Schmid	5
2	Meier	2	Borner	4
3	Tobler	1	Bernasconi	6

⇒ IDSt | LastName

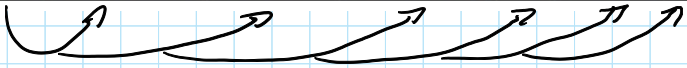
IDProf | Prof

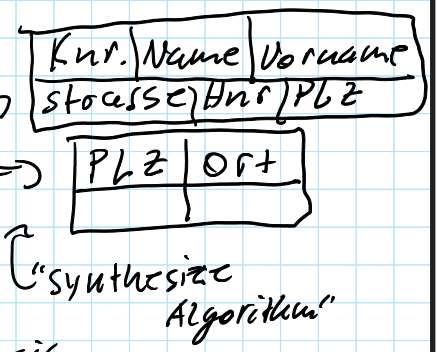
IDSt | IDProf | Grade

3th Normal Form (= No transitive Dependencies)

- If a attribute is dependant on a non-primary-key attribute

Kunde						
Knr.	Name	Vorname	Straße	Hnr.	PLZ	Ort
007	Mustermann	Max	Musterstr.	1	12345	Musterort

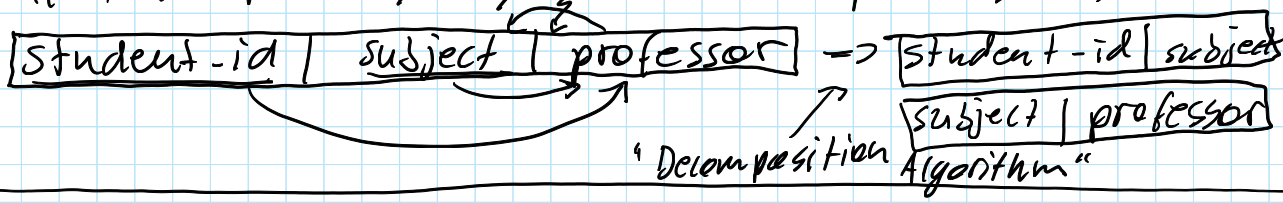




- Algorithm: 1. compute minimal Basis
  2. For each func. dependency create table
  3. If needed, att table only containing old primary keys

BCNF / 3.5 Normal form

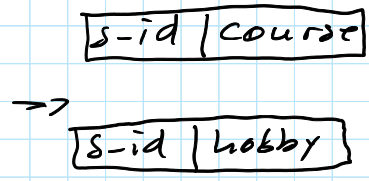
- If a non primary-key derives a primary-key



4th normal form

- No multi value dependencies (no 1:n or m:n relations)
- Multi-value-dependency
  - If for a value single value of A, more then one value of B exists
  - Have at least 3 columns

s_id	course	hobby
1	Science	Cricket
1	Maths	Hockey



- s_id ->-> course
- s_id ->-> hobby

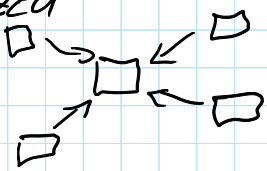
} student can have multiple courses/hobbies

OLTP

- Online transaction processing
- Workload with lots of updates
- Run over 3NF and many tables
- E.g. E-commerce

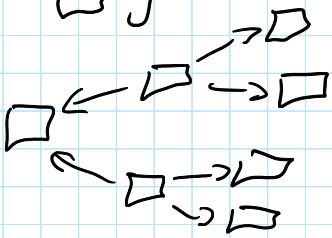
OLAP

- Online analytical processing
- Workload with heavy and complex queries
- Take hours to process
- Run denormalized
- Star schema:



Denormalized

- Snowflake schema:



Normalized form of star schema

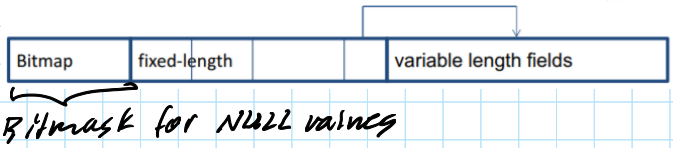
- E.g. Weather forcast

How data is stored

- Records: Tuples
- Pages: collections of records from same table
- Blocks: Parts/collections of pages

Data manager / Buffer manager

- Tablespace: space for a database on a disk
- Structure of a record

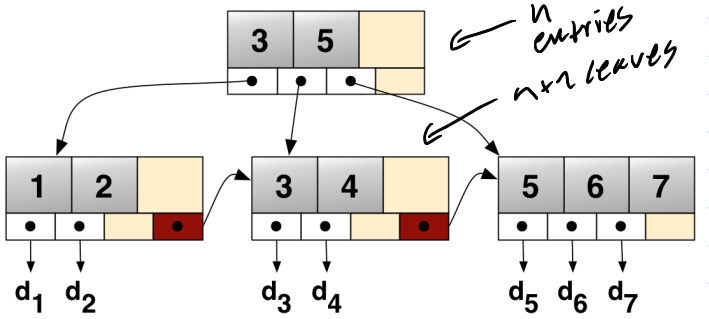


Bitmask for NULL values

- Structure of a page
  - Header at the end of the page
    - Number of records in the current table
    - Slot directory: 0: slot free
      1: slot used
  - Unpacked order: Free/used slots not ordered
  - Packed order: Free/used slots ordered

Query Algorithms

- Query selectivity: #Tuples in result vs #Tuples in DB
- Attribute cardinality: How many distinct values for attribute
- Scan
  - Iterate over every tuple in the DB
- Indexes
  - Avoid a full table scan ⇒ Build an index over some attributes
  - B+ tree: - Self balancing tree
    - Each node can have more than n children
    - Search, Insertion, Deletion: $O(\log(n))$



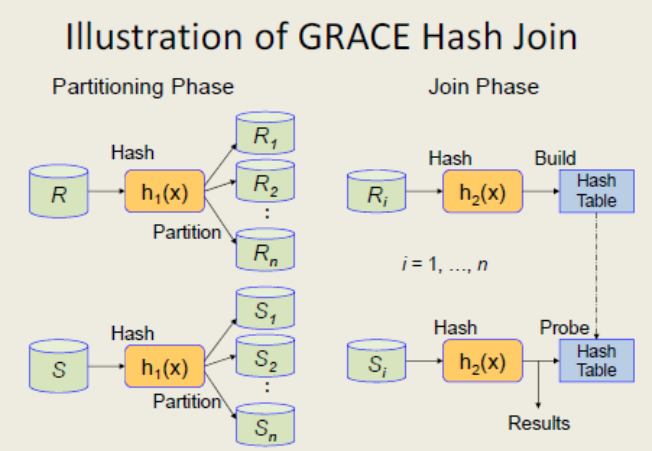
  - Two ways for storing data: - Pointer to tuple
    - Tuple inside tree
  - Hash-index
- External Sort (= One pass sort) ← every item is read/written once
  - Data does not fit in main-memory
  - N: # pages
  - M: # pages that can be stored in the buffer

1. Phase (Create Runs): Load tuples in buffer, sort, write back, do this for all tuples

2. Phase: (Merge Runs) Load from each run the X-th tuple and do sorting again

- Joins

  - Nested loop join: $O(A \cdot B)$
  - Sort-merge join: $O(A \cdot \log(A) + B \cdot \log(B))$
  - Canonical hash join: 1. Hash table A
    2. Match table B with same hash function
    - $O(A + B)$
  - Grace Hash join: - If we have too many entries for one hash function



  - $O(A + B)$

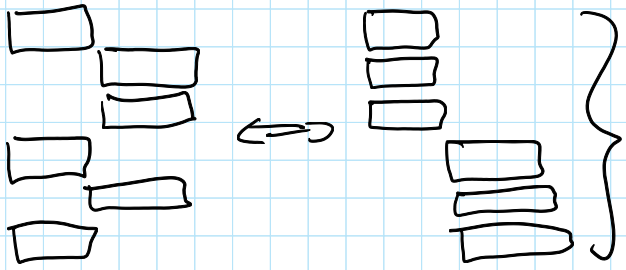
Query Execution

1. Query is translated into relational-algebra-tree
2. Tree is converted into different plans
3. Cost of each plan is approximated
4. Cheapest plan is choosen

- Make different trees
  - $\sigma_{\theta_1 \wedge \theta_2} = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
  - $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
  - $\Pi_{t_1}(\Pi_{t_2}(\Pi_{t_3}(E))) = \Pi_{t_1}(E)$
  - $E_1 \bowtie E_2 = E_2 \bowtie E_1$
  - $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
  - $\sigma_\theta(E_1 \bowtie E_2) = [\sigma_\theta(E_1)] \bowtie E_2$
  - $E_1 \cap E_2 = E_2 \cap E_1$
  - $E_1 \cup E_2 = E_2 \cup E_1$
  - $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$
- The earlier we process selections, less tuples we need to manipulate higher up in the tree

Transaction

- Series of commands, extracted from a query
- Defines a transition between two different states
- Ends in commit (=finished) or abort (=rollback)
- **ACID principle**
  - Atomicity: a transaction is executed in its entirety or not at all
  - Consistency: consistent state → transaction → consistent state
  - Isolation: a transaction executes as if it were alone in the system } 2-phase locking
  - Durability: committed changes of a transaction are never lost and can be recovered ↳ Recovery
- **Concurrency**: conflict serializable
  - We can translate two transactions into a serial schedule with a sequence of **non-conflicting** swaps
    Reordering allowed

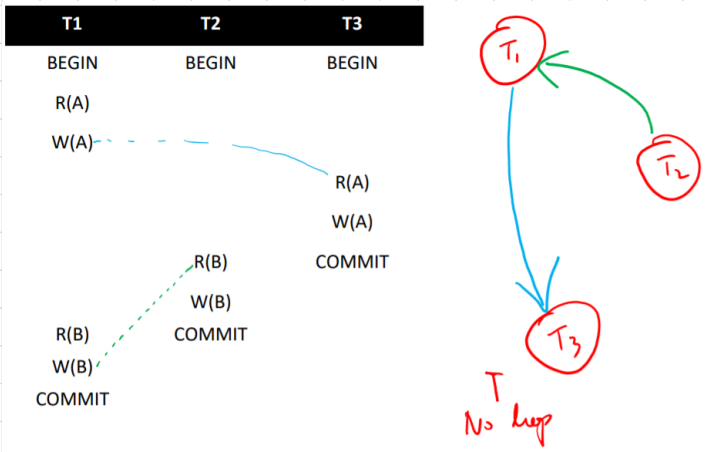


Wenn wir es so transformieren können mit non-conflict swaps, ist schedule conflict-serializable

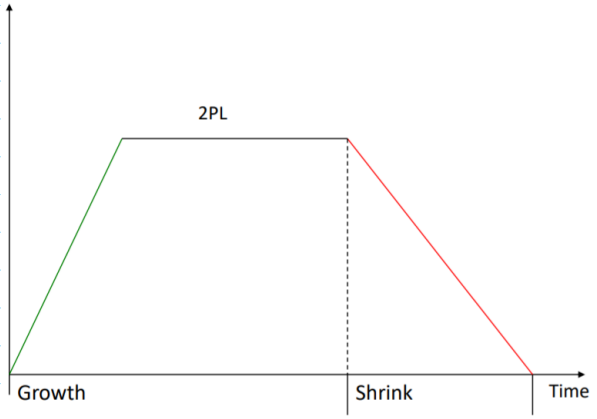
- Conflict with aborts

write(x) } conflict, because abort resets the database to the state before T_2 has written
write(x)
abort(x) }

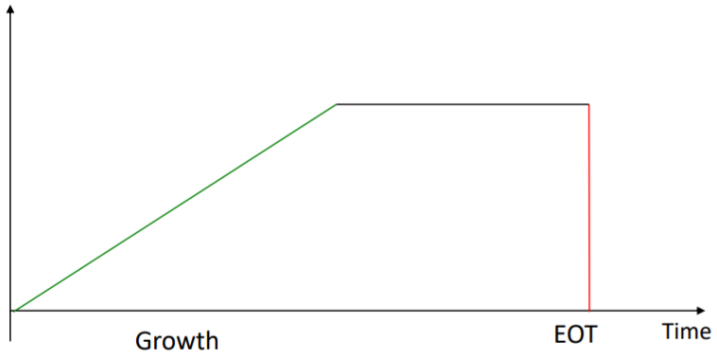
- Conflict serializable, iff dependency-graph is acyclic



- Pessimistic synchronization
  - Share lock: Only allows read operation
  - Exclusive lock: Need for write operation
- 2-phase locking



- Strict 2-phase locking



- Optimistic synchronisation
  - Snapshot isolation

Recovery

1. Application, System
   - R1 Recovery: Undo a single transaction
2. System crash: lose main memory, keep disk
   - R2 Recovery: Redo committed Transactions
   - R3 Recovery: Undo active Transactions
3. System crash: loss of disks
   - R4 recovery: Read backup of DB

- Recovable:

Recoverable [edit]

Transactions commit only after all transactions whose changes they read, commit.

$$F = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Com. & \\ & Com. \end{bmatrix} \quad F2 = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Abort & \\ & Abort \end{bmatrix}$$

If T_2 reads from T_1, T_1 must commit first

These schedules are recoverable. F is recoverable because T1 commits before T2, that makes the value read by T2 correct. Then T2 can commit itself. In F2, if T1 aborted, T2 has to abort because the value of A it read is incorrect. In both cases, the database is left in a consistent state.

Avoids cascading aborts / rollbacks (ACA) [edit]

Also named cascadeless. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

The following examples are the same as the ones in the discussion on recoverable:

$$F = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Com. & \\ & Com. \end{bmatrix} \quad F2 = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Abort & \\ & Abort \end{bmatrix}$$

In this example, although F2 is recoverable, it does not avoid cascading aborts. It can be seen that if T1 aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

The following is a recoverable schedule which avoids cascading abort. Note, however, that the update of A by T1 is always lost (since T1 is aborted).

$$F3 = \begin{bmatrix} T1 & T2 \\ & R(A) \\ R(A) & \\ W(A) & \\ & W(A) \\ Abort & \\ & Commit \end{bmatrix}$$

Note that this Schedule would not be serializable if T1 would be committed. Cascading aborts avoidance is sufficient but not necessary for a schedule to be recoverable.

- Strict history: A schedule in which a transaction can neither read or overwrite an item X until the last transaction that wrote X has committed

> On a cascadeless schedule a transaction T_2 cannot read a value a if a transaction T_1 wrote a before that and didn't commit. On a strict schedule T_2 also wouldn't be able to write a after T_1 wrote it (even if it read a before T_1 wrote it).
>
> If you read carefully, the definition of strict says "not read **or overwritten**". That's the difference.

- Undo-Logging: (= Rückgängig machen)
  - Some transactions are in the middle, how to undo them?
  - Start T
  - Update <T, X, v> (v: old value)
  - Commit T / Abort T
  - If commit is in the log, it does not mean it is on the disk
  - If T modifies database element X, write <T, X, v> to disk, before change of X is written to disk (v: old value)
  - If a transaction commits, commit must be written to disk only after all other changes are written to disk
- Redo logging (= Wiederholen)
  - Update <T, X, v> (v: new value)
- Undo/Redo Logging
  - Update <T, X, v, w> (= Store both, old and new value)
  - Flush log, whenever Commit appear in log

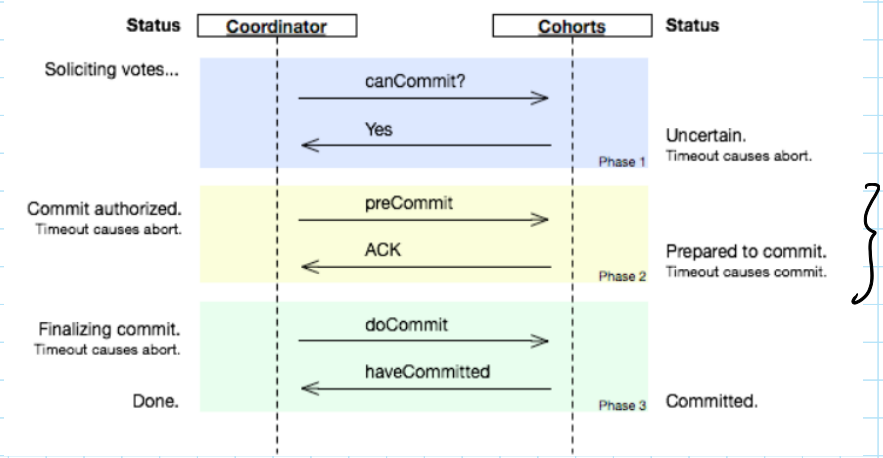
Consensus (= Verteiltes Commit Protokoll für DB-Cluster)

- Problem of reaching an agreement among all working processes on the value of a variable
- 2-phase commit

1. Coordinator sends Vote-Request to all participants (← in most cases the agent who will alter data)
2. All participants sends back Vote
3. Coordinator collects all votes
   - 3.1. All yes: Commit and send commit to all others
   - 3.2. Some No: Abort and send Abort to all which voted yes

- If time-out for waiting for response → Abort
- If time-out for waiting for Vote-Req → Abort local
- If time-out waiting for a decision → uncertainty period
- Problem that leads to inconsistency:
  1. Coordinator sends Commit to first Agent
  2. Agent commits
  3. Coordinator and first Agent fails

- 3-phase commit



} introduced by 3-phase commit

Data Replication

- When are updates propagated?
  - Synchronous (No inconsistencies / worse response time)
  - Asynchronous (Good response time / Data inconsistencies)
- Where the update can be made?
  - Primary copy => master (Load at primary-copy is large)
  - Update everywhere => group
    ↳ (copies need to be synchronized)

Key-Value-Store

- Map with keys and associating values
- Clients can put and request values per key