Database Zusammenfassung:

Relational model, algebra, and calculus

Practice quantifiers
move between RA, DRC, natural language
be precise (e.g., qualified names do not exist in RA; use renaming)

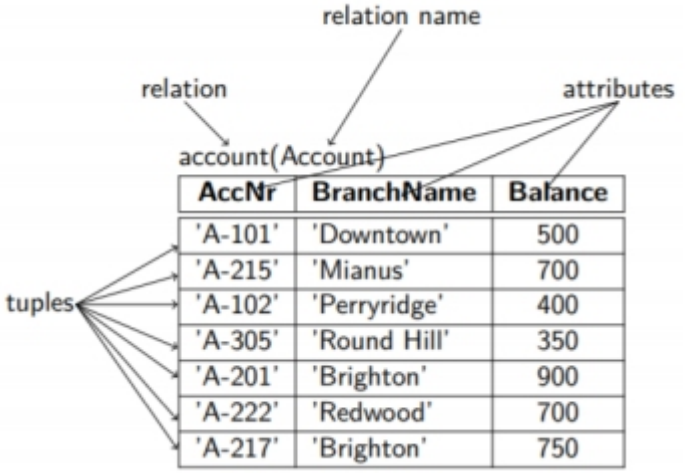
SL02: The relational model

SQL	Relational Algebra	Domain Relational Calculus
table	relation	predicate
column	attribute	argument
row	tuple	-
query	RA expression	formula

A file-based management can create conflicting data and data duplication.
A database allows you to share data easily and prevents conflicting and duplicated data.
The data association and operation methods that a database uses is called its data model. We use the relational Model. A relational DB processes data using the easy-to understand concept of a table.

1. The Relational Model
   Attribute values must be atomic (indivisible), value can be acc. Nr but not a set of acc. Nrs.
   Domain:
   - Set of allowed values for each attribute
   - Has logical definition (e.g. US_phone_nr = set of 10 digit phone nrs)
   - Has data-type or format defined (phone nr. (ddd)ddd-dddd)
   - Null is member of every domain

   Tuple: ordered set (list of values)
   Relational instance: r(R) = relation r on relation schema R ; this is a subset of the Cartesion product of the domains of its attributes
   Set: elements can be everything.
   Relation: elements are tuples, given sets D1, D2, … then relation subset of D1 x D2 x …
   unordered, order of tuples irrelevant



AccNr	BranchName	Balance
'A-101'	'Downtown'	500
'A-215'	'Mianus'	700
'A-102'	'Perryridge'	400
'A-305'	'Round Hill'	350
'A-201'	'Brighton'	900
'A-222'	'Redwood'	700
'A-217'	'Brighton'	750

   X = {{(3)}} → (3) is tuple, {(3)} is relation, {{(3)}} is database
   Database: multiple relations

A **domain** D is a set of atomic data values.
- phone numbers, names, grades, birthdates, departments
- each domain includes the special value `null`

With each domain a **data type** or format is specified.
- 5 digit integers, yyyy-mm-dd, characters

An **attribute** A_i describes the role of a domain in a relation schema.
- PhoneNr, Age, DeptName

A **relation schema** $R(A_1, ..., A_n)$ is made up of a relation name R and a list of attributes.
- $Employee(Name, Dept, Salary)$, $Department(DName, Manager, Address)$

A **tuple** t is an ordered list of values $t = (v_1, ..., v_n)$ with $v_i \in dom(A_i)$.
- $t = (\text{'Tom'}, \text{'SE'}, 23K)$

A **relation** $r \subseteq D_1 \times ... \times D_n$ over schema $R(A_1, ..., A_n)$ is a set of n-ary tuples.
- $r = \{(\text{'Tom'}, \text{'SE'}, 23K), (\text{'Lene'}, \text{'DB'}, 33K)\} \subseteq Names \times Departments \times Integer$
- $s = \{(\text{'SE'}, \text{'Tom'}, \text{'Boston'}), (\text{'DB'}, \text{'Lena'}, \text{'Tucson'})\}$

A **database** DB is a set of relations.
- $DB = \{r, s\}$

Constraints:
Conditions that must be satisfied by all valid relation instances
4 main types of constraints:

- Domain constraints (each value in a tuple must be from the domain of its attribute)
- Key constraints
  - Superkey → possible to identify a unique tuple of each possible relation. Often different set of attributes
  - Candidate key → minimal, is superkey and no subset of it is a superkey
  - Primary Key → one of the candidate keys (principal means of identifying tuples within relation)
- Entity Constraints
  - Primary key attr. of each relation no null values

ID	Name	CuStreet
1	'N. Jeff'	'Binzmühlestr'
	'T. Hurd'	'Hochstr'

ID cannot be primary key

ID	Name	CuStreet
1	'N. Jeff'	'Binzmühlestr'
2	'T. Hurd'	'Hochstr'

ID can be primary key

- Referential integrity constraints
  - Foreign key → attribute corresponding to primary key of another relation; only values occurring in primary key may occur in foreign key of referencing relation; (or formulated differently: Every value used as a foreign key NEEDS to show up as a key value in the referenced relation)

Query language

- Procedural**: how to do it**, for query optimization
  - Relational algebra
- Declarative: **what to do**, not for query optimization
  - Tuple relational calculus
  - Domain relational calculus

Now we are going to look at two basic query languages:

- Relational algebra
- Domain relational calculus

2. Relational Algebra

Is closed (= **all objects in the RA are relations**) because operators take one or two relations as inputs and produce a new relation as result.
Basic operators:

- Set operations:
  these operations work upon one or more sets of rows to produce a new set of rows.
  In short, they determine which rows from the input appear in the output.
  - **Union** r U s,
    r, s or both (everything **once**)
    Performing a union operation extracts all rows in two tables and combines them.
    both must have the same schema (equal nr. And types of attributes)
    eliminates duplicates after union (like projection)
    Example: $\pi_{CustName}(depositor) \cup \pi_{CustName}(borrower)$
    Example: $r \cup s$

r

A	B
'α'	1
'α'	2
'β'	1

s

A	B
'α'	2
'β'	3

$r \cup s$

A	B
'α'	1
'α'	2
'β'	1
'β'	3

  - **Difference** r – s
    r but not s ($t \in (r - s) \Leftrightarrow t \in r \wedge t \notin s$)
    Extracts rows from just one of the tables. (Extracts from 1 everything which is not in 2 or vice versa)
    both must have the same schema and the same arity (# of attributes n of its relational schema)
    Example: $r - s$

r

A	B
'α'	1
'α'	2
'β'	1

s

A	B
'α'	2
'β'	3

$r - s$

A	B
'α'	1
'β'	1

  - **Cartesian Product** r x s
    combine all tuples (= rows) from r with all tuples from s.
    names of r and s must be disjoint, otherwise renaming must be used (naming conflict).
    new size = |r| * s
    Example: $r \times s$

r

A	B
'α'	1
'β'	2

s

C	D	E
'α'	10	'a'
'β'	10	'a'
'β'	20	'b'
'γ'	10	'b'

$r \times s$

A	B	C	D	E
'α'	1	'α'	10	'a'
'α'	1	'β'	10	'a'
'α'	1	'β'	20	'b'
'α'	1	'γ'	10	'b'
'β'	2	'α'	10	'a'
'β'	2	'β'	10	'a'
'β'	2	'β'	20	'b'
'β'	2	'γ'	10	'b'

Ex: all customers who have loan at Perryridge: ∏CustName(∂BranchName ='Perryridge' (∂LNr =LoanNr (p Custname, LNr (borrower) x loan)

- **Intersection** r n s = r-(r-s) = r – Difference r – s
  same schema and arity required
  extracts everything 1 & 2 both have (rows where r and s are equal)

Example: $r \cap s$

r

A	B
'α'	1
'α'	2
'β'	1

s

A	B
'α'	2
'β'	3

$r \cap s$

A	B
'α'	2

- **Renaming** $\rho_{r(A1,\ldots,An)}(E)$
  Changes the relation name to r and the attribute names to $A_1,\ldots, A_k$
  can also do only r or only attributes

Example: $\rho_{s(X,Y,U,V)}(r)$

r

A	B	C	D
'α'	'α'	1	7
'β'	'β'	23	10

s

X	Y	U	V
'α'	'α'	1	7
'β'	'β'	23	10

Changed from r to s and from A,B,C,D to X,Y,U,V

- Relational operators:
  Relations behave as sets, so there are no duplicate values in them.
  - **Projection** π
    Extracts data vertically from a table. Extracts columns from a table.
    Deletes columns that are not listed.
    No duplicate rows in results since relations are sets.
    $\pi_{Column\_name\_1, [\ldots], Column\_name\_x}(Relation)$

Example: $\pi_{A,C}(r)$

r

A	B	C
'α'	10	1
'α'	20	1
'β'	30	1
'β'	40	2

$\pi_{A,C}(r)$

A	C
'α'	1
'β'	1
'β'	2

  - **Selection** σ
    Extracts data horizontally from a table. (It extracts rows)
    condition is called selection predicate and consists of terms connected by **and, or, not.**
    $\sigma_{selection\_Predicate}(Relation)$

Example: $\sigma_{A=B \wedge D>5}(r)$

r

A	B	C	D
'α'	'α'	1	7
'α'	'β'	5	7
'β'	'β'	12	3
'β'	'β'	23	10

$\sigma_{A=B \wedge D>5}(r)$

A	B	C	D
'α'	'α'	1	7
'β'	'β'	23	10

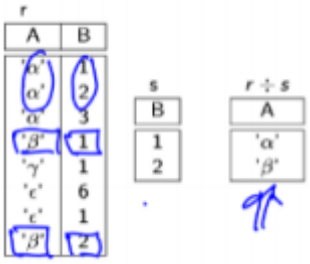
- **Division** r ÷ s

  Good for queries "for all"

  extracts all rows whose **column values** match those in the second table, BUT only returns columns that don't exist in the second table.

  So → 1. Check out which tuples are the same for all available attributes.

  2. Return from table 1 the columns that are not present in table 2 for all found matches.



Properties of the Division Operation

- Property
  - Let $q = r \div s$
  - Then q is the largest relation satisfying $q \times s \subseteq r$
- Definition in terms of the basic algebra operation

  Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

  $$r \div s = \pi_{R-S}(r) - \pi_{R-S}((\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r))$$

  To see why
  - R-S,S: all attributes of R that are not in S, followed by all attributes of S
  - Thus, $\pi_{R-S,S}(r)$ reorders attributes of r
  - $\pi_{R-S}(\pi_{R-S}(r) \times s) - \pi_{R-S,S}(r))$ gives those tuples t in $\pi_{R-S}(r)$ such that for some tuple $u \in s, t \circ u \notin r$.

T = R x S

T ÷ S = R

R:

Vater	Mutter	Kind	Alter
Hans	Helga	Harald	5
Hans	Helga	Maria	4
Hans	Ursula	Sabine	2
Martin	Melanie	Gertrud	7
Martin	Melanie	Maria	4
Martin	Melanie	Sabine	2
Peter	Christina	Robert	9

S:

Kind	Alter
Maria	4
Sabine	2

R+S

Vater	Mutter
Martin	Melanie

- **Join**
  - **Natural Join:**

    primary key of one table is foreign key of table 2.

    The natural join connects tables on the same field values together bzw. 1 column stays in the joined table.

    Combination of all attributes that are identical in r and s if there are several columns which are the same.

    r |x| s with R(A,B,C,D) and S(E,B,D) equivalent to: $\pi_{A,B,C,D,E}(\sigma_{B=Y \wedge D=Z}(r \times \rho_{(E,Y,Z)}(s))$

ex: name and loan amount of customers who have loan

∏CustName,Amount(borrower|x|loan)

Example:

- $r \bowtie s$ with $R(A, B, C, D)$ and $S(B, D, E)$
- Schema of result is (A, B, C, D, E)
- Equivalent to: $\pi_{A,B,C,D,E}(\sigma_{B=Y \wedge D=Z}(r \times \rho_{(E,Y,Z)}(s)))$

r

A	B	C	D
'α'	1	'α'	'a'
'β'	2	'γ'	'a'
'γ'	4	'β'	'b'
'α'	1	'γ'	'a'
'δ'	2	'β'	'b'

s

B	D	E
1	'a'	'α'
3	'a'	'β'
1	'a'	'γ'
2	'b'	'δ'
3	'b'	'ε'

$r \bowtie s$

A	B	C	D	E
'α'	1	'α'	'a'	'α'
'α'	1	'α'	'a'	'γ'
'α'	1	'γ'	'a'	'α'
'α'	1	'γ'	'a'	'γ'
'δ'	2	'β'	'b'	'δ'

compare to outer join initial tables

loan ⋈ borrower

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'

- **Theta join:**

- **Outer join:**

Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

right outer join → preserves tuples from right and fills the rest with null

loan⟖ borrower

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-155'	*null*	*null*	'Hayes'

left outer join → preserves tuples from the left and fills the rest with null

loan⟕borrower

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-260'	'Perryridge'	1700	*null*

full outer join → preserves all tuples

loan⟗ borrower

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-260'	'Perryridge'	1700	*null*
'L-155'	*null*	*null*	'Hayes'

Aggregate Functions:
Takes a collection of values and returns a single value as a result

- Avg → average value
- Min → minimum value
- Max → maximum value
- Sum → sum of values
- Count → number of values

3. Relational Calculus

In relational calculus we specific what data we want → declarative
An expression in relational calculus creates a new relation which is specified with variables ranging over:

- Tuples of relations (TRC)
- Domains of attributes of relations (**DRC → we focus on this**)

An expression in DRC form (first order logic):

$$\{\langle x_1, x_2, \ldots, x_n \rangle | P(x_1, x_2, \ldots, x_n)\}$$

- $X_1,\ldots,x_n$ represent domain variables
- P represents a formula composed of atoms
- Form always { X_1, …, X_n | formula} where X,…., X are the only free variables in the formula

Atoms in DRC **can have one** of the following forms:

- $<x_1,\ldots ,x_n> \in r$, where r is a relation with n attributes
  - $x_1,\ldots ,x_n$ are variables which take on values from the domains of the attributes of relation r
- x θ y, where x and y are domain variables and θ is a comparison operator (<, <=, = =/=, >, >=)
  - x and y have to have domains that can be compared by θ.
- X θ c, where x is a domain variable, θ is a comparison operator and c is a constant
  - C is a value from the domain of x

Free & Bound variable

- Variable is free if it is not quantified
- Variable is boud if it is quantified

Query: "Get the names and dates of birth for all students with a date of birth after 1.1.1991"

DRC: $\{\langle n, d \rangle | \exists s(\langle s, n, d \rangle \in \text{student} \wedge d > 1991\text{-}01\text{-}01)\}$

Only values left of | are allowed to be free

FOPL (first order predicate logic) Equivalences:

FOPL Equivalences

- $\forall X(A) = \neg\exists X(\neg A)$
- $A \Rightarrow B = \neg A \vee B$
- $A \wedge \exists X(B) = \exists X(A \wedge B)$ if X is not free in A
- $A \wedge \forall X(B) = \forall X(A \wedge B)$ if X is not free in A
- $A \wedge \neg\exists X(B) = A \wedge \neg\exists X(A \wedge B)$ if X is not free in A

Set theory

- $A - B = A - (A \cap B)$

Domain independence:

- Relational calculus only permits domain independent expressions (that permit sensible answers e.g. no infinite results)
- Not decidable, syntactic criteria to ensure it

- $emp(X)$ is domain independent
- $\neg emp(X)$ is not domain independent
- $stud(X) \wedge \neg emp(X)$ is domain independent
- $X > 6$ is not domain independent

SQL

SL03: SQL

Based on multisets (bags) {{…}}, **element can occur multiple times**

DRL → retrieves data

DML → Inputs and retrieves data

DDL → Creates a table

Data Control language (**DCL**) → Manages user access (so multiple user can operate at the same time)

Use SQL functions to define, operate and control data.

Search for data using a **SELECT** statement (DRL)

Insert, update and delete data using **INSERT, UPDATE**, and **DELETE** statements (DML)

Create a Table using the **CREATE TABLE** statement (DCL)

Specify a condition using a **WHERE** phrase

DRL:

Standard format/SFW-block (DRL):

SELECT what_data_is_needed_bzw_which_Attribute (**DOES NOT REMOVE DUPLICATES!!)**

FROM table_name, another_relation_name

WHERE condition/predicate

AND condition/predicate;

Domain types:

- CHAR(length n)
- VARCHAR (maxLength n)
- INTEGER
- SMALLINT
- NUMERIC(precision p, digits_right_of_comma n)
- REAL
- DOUBLE PRECISION
- FLOAT(precision at least n)

The conditions in the WHERE phrase can be set via operators:

- Comparison operators: <, <=, =, <>, >=, >
- Logical operators: AND, OR, NOT (and BETWEEN)

Patterns:

If you know exactly what to search for you can use pattern matching via wildcard characters:

- WHERE product_name LIKE '%n';
- WHERE product_name LIKE 'n%';
- WHERE product_name LIKE Smi_t or Smit_ (for only one character)

Searches:

- BETWEEN
- IS NULL

SELECT *
FROM product
WHERE unit_price
BETWEEN 150 AND 200;

SELECT *
FROM product
WHERE unit_price **is NULL;** (Searches for a null)

Joins:

Inner - Joins:

Equi join → combining columns with the same names and rows with the same value are designated as join conditions for joining tables.

SELECT emp.first_name, emp.department_id, dept.department_name
FROM emp, dept
where emp.departement_id=dep.department_id
AND emp.department_id=90;_

SELECT Attribute_names
FROM relation_one R1, relation_two R2
WHERE prof_pers_no = pers_no; (join predicates)
(Renaming the relations for faster coverage)

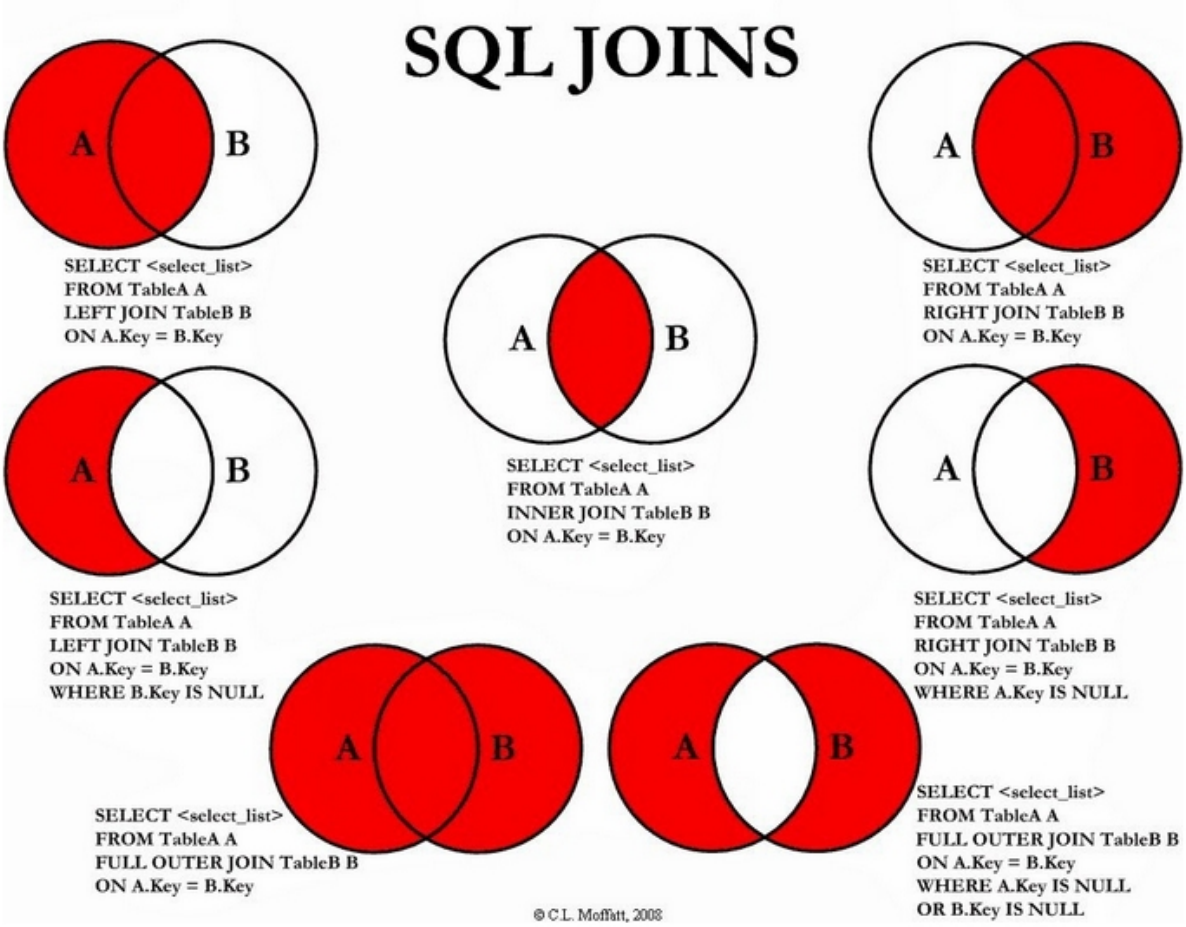
Natural join → joining columns with the same name into one is called a natural join; no need to mention the common column.

SELECT * from emp
Natural join
Dept WHERE department_id=90;

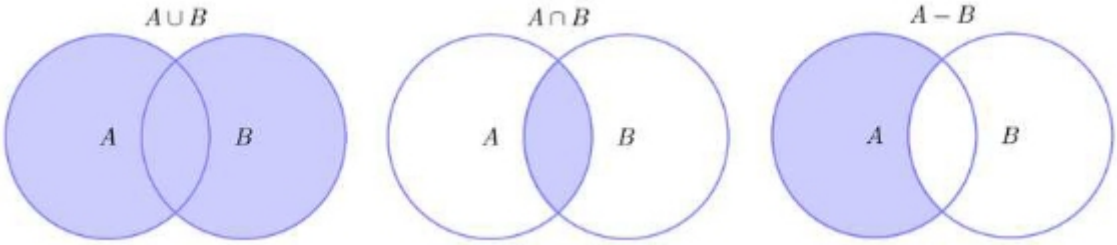
Outer – Joins:

SELECT
FROM R[natural|left outer|right outer|full outer] join S[on R.A = S.B];
(this will lead to cartesian product tho)

SELECT *
FROM student, attends, lecture
WHERE student.stud_no = attends.stud_no
AND attends.course_no = lecture.course_no;
(qualified attribute names to resolve name collisions)



Set operations:



Union:
SELECT *
FROM prof1
UNION (ALL → **UNION ALL if I want to keep the duplicates; Union will remove duplicates**)
SELECT *
FROM prof2

Intersection: (which values are found in both tables)
SELECT *
FROM prof1
INTERSECT
SELECT *
FROM prof2

Set Difference: (which values are found in prof1 but not in prof2)
SELECT *
FROM prof1
EXCEPT
SELECT*
FROM prof2

Sorting:

- Relations are not sorted by default
- Has to be done by order by clause (ascending (default) or descending)

SELECT *
FROM student
ORDER BY date_of_birth **desc**, name;

Aggregate Functions: (=set functions)
SELECT Count(), Sum(), Avg(), Max(), Min()
(to summarize Attributes or whole tuples)
SELECT count(distinct date_of_birth)
From student;

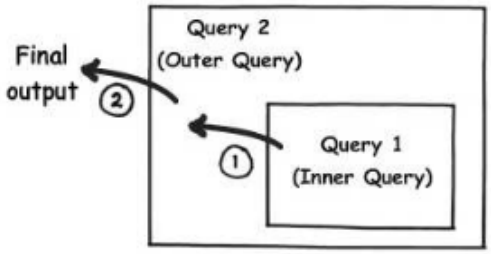
→ If we have multiple Attributes in the Select clause and don't have multiple aggregate functions we have to use the aggregate function in a subquery/nested query first to summarize and after that we can select the other attributes we want.

Nested Queries/subqueries:

- Nested sfw-blocks can appear in the where, from and even the select clause
- Basically, the "inner" query computes some kind of **intermediate result** that is used in the "outer" query

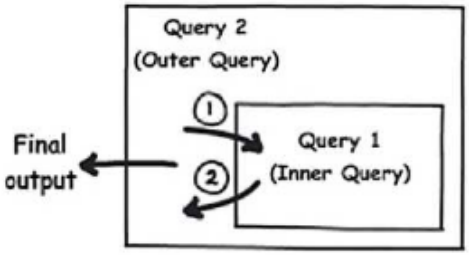
Uncorrelated subquery:

inner query does not refer to attributes of the outer one



Correlated subquery:

inner query references attributes used in the outer query



Uncorrelated Subquery:

Query: "Return the names of all students who attend the lecture with course number 5"

```
SELECT S.name
FROM student S                    } outer query
WHERE S.stud_no in
        (SELECT A.stud_no
        FROM attends A            } inner query
        WHERE A.course_no = 5);
```

→Subquery (inner query) runs only once and first; then the outer query runs

Correlated subquery:

Query: "Find all prof. with assistants who work in different areas"

```
SELECT distinct P.name
FROM professor P, assistant A     } outer query
WHERE A.mgr = P.pers_no
AND exists
        (SELECT *                 } inner query (any tuple?)
        FROM assistant B
        WHERE B.mgr = P.pers_no
        AND not (A.area = B.area));
```

→We need the attribute value of A.area and P.pers_no from the outer query

→For every tuple of the outer query, we have to run the inner query

→exists predicate is true if the subquery contains at least one tuple

First the outer tuple gets selected and passed to the inner query for verification (the inner query needs input to verify its own where clause); then the Where clause of the outer query is evaluated to check if its going to the final output table.

Grouping:

```
SELECT course_no, count(*) as no_of_students
FROM attends
GROUP BY course_no; (if an attribute is not in here, then it must appear in aggregated form in the select clause)
```

attends

stud_no	course_no
1	1
2	1
4	1
1	2
4	2
2	3
4	3

→

course_no	no_of_students
1	3
2	2
3	2

Having:

Having is used to do filtering after the grouping/aggregating (because where clause is evaluated before grouping/aggregating)

Query: “Find all the professors who give more than three lectures”

```
SELECT prof_pers_no, count(course_no) as no_of_lect
FROM lecture
GROUP BY prof_pers_no
having count(*) > 3;
```

Views:

- ➔ Used to simplify queries (some queries are too complex; so we break them down with views)
- ➔ Virtual relation so to speak
- ➔ Can also be used to restrict access(users may only access views f.e.)
- ➔ Create temporary views via the **WITH** clause

Query: “Output the names of all professors who give at least one lecture that is worth more than the average number of credits and who have more than three assistants”

```
CREATE VIEW aboveAvgCredits (as) (temporary: WITH aboveAvgCretids(cno, pno) as(…), many Assistants(mgr) as (…))
SELECT course_no, prof_pers_no
FROM lecture
WHERE credits >
        (SELECT avg(credits)
        FROM lecture);
```

→Creates the virtual relation aboveAvgCredits(course_no, prof_pers_no)

```
CREATE VIEW manyAssistants (as)
SELECT mgr
FROM assistant
GROUP BY mgr
HAVING count(*) > 3;
```

→Creates virtual relation manyAssistants(mgr)

```
SELECT name
from professor
WHERE pers_no in
        (SELECT prof_pers_no
        FROM aboveAvgCredits)
AND pers_no in
        (SELECT mgr
        FROM manyAssistants);
```

→Putting it all together

Recursive SQL:

Imagine Prerequisites for advanced courses; if you want all the prerequisites which go deeper than 1 layer you have to go recursive.
we do this via transitive closures.
A recursive query consists of three parts:

- ➔ An initial (non-recursive) subquery (executed once → initializes the content of the temporary view)
- ➔ A recursive subquery (each time it is executed, it sees only the rows added in the previous iteration; it gets evaluated until no more rows are added (leaves are reached) to the temporary view)
- ➔ A final query (on the transitive closure)

```
with recursive r(a1, a2, ..., an) as (
        ⟨ initial subquery ⟩
union
        ⟨ recursive subquery ⟩
)
select  *
from    r;
```

Requirement example:

```
with recursive req(courseno) as (
        select  prerequisite
        from    requires
        where   advanced = 5259
union
        select  r.prerequisite
        from    requires r, req q
        where   q.courseno = r.advanced
)
select  *
from    req;
```

DML:

- ➔ Insert data
- ➔ Delete data
- ➔ Change data

One must consider the constraints of the relation when inserting, deleting or changing data.

Inserting:

Insert statements adds (new) tuples to a table

INSERT INTO professor
VALUES(123456, 'Kossmann', 012)

INSERT INTO professor(pers_no, name)
VALUES(123456, 'Kossmann')

- ➔ Adds the newly created tuple (123456, 'Kossmann', NULL) to the relation professor

INSERT INTO professor(pers_no, name)
SELECT pers_no, name
FROM assistant
WHERE pers_no = 11111;
→copy paste from other relations

Deleting:

Removes tuples from a relation (which satisfies predicate in where clause)

DELETE FROM professor
WHERE pers_no = 123456;

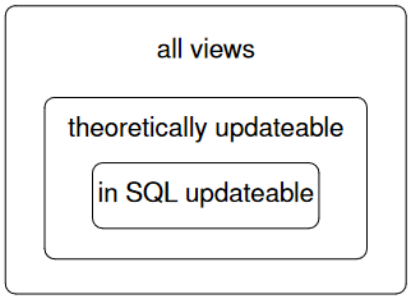
- ➔ If no where clause everything is deleted

Updating:

Where clause identifies the tuples to be changed
Set clause determines the new attribute value
UPDATE professor
SET office_no = 121
WHERE pers_no = 123456;

Views can only be updated if:

- ➔ It contains only one base relation
- ➔ The key of this relation is present
- ➔ There is no aggregation, grouping, or removal of duplicates



DDL:

Used to define the schema of a database

Constraints:

Constraints (formulated for requirements) are there to keep data consistent
The most important ones:

- ➔ PRIMARY KEY
- ➔ NOT NULL (must have a value)
- ➔ UNIQUE (may not contain duplicate values)
- ➔ CHECK CLAUSES
- ➔ REFERENTIAL INTEGRITY (Every value used as a foreign key needs to show up as a key value in the referenced relation)
- ➔ DEFAULT VALUES

Primary Key, Not Null & Unique:

CREATE TABLE relationname(
BranchName **CHAR**(15) **NOT NULL UNIQUE PRIMARY KEY**,
BranchCity **CHAR**(30) **NOT NULL**,
Assets **INTEGER UNIQUE**)

CREATE TABLE integrity constraints: NOT NULL, PRIMARY KEY → automatically ensures not null, UNIQUE

Check Clauses:

CREATE TABLE professor(
Pers_no **INTEGER NOT NULL UNIQUE PRIMARY KEY**,
Name **VARCHAR (80) NOT NULL**,
Office_no **INTEGER UNIQUE**
CHECK (office_no > 0 and office_no < 9999),
);

Range of a domain can be restricted f.e.

CREATE TABLE attends(
stud_no **INTEGER**,
course_no **INTEGER**,
CHECK (stud_no not in
SELECT e.stud_no
FROM examines e
WHERE e.course_no = attends.course_no
AND grade >=5)),
PRIMARY KEY (stud_no, course_no));

➔ Entire Queries can be checked

Referential Integrity – Foreign Key References:

CREATE TABLE professor**(**
Pers_no **INTEGER PRIMARY KEY**
…
);

CREATE TABLE lecture(
Course_no **INTEGER PRIMARY KEY**,
…,
Prof_pers_no **INTEGER NOT NULL**,
FOREIGN KEY(prof_pers_no) */the attribute naming of the foreign key in the lecture relation/*
REFERENCES professor(pers_no) */the referenced relation with the attribute name of said relation/*
);

Referential Integrity - Cascade:

If a key value is updated, the corresponding foreign key values are set to the new value (or NULL)
If a key value is deleted, the corresponding tuples with this foreign key value are deleted

```
create table lecture (
  course_no      integer primary key,
  . . .
  prof_pers_no   integer not null,
  foreign key (prof_pers_no)
  references professor(pers_no)
                 [on delete {set null | cascade}]
                 [on update {set null | cascade}]
);
```

Default Values:

If an **attribute value is not specified in an INSERTION statement**, the database uses a **default value**. **If no default value is specified, then NULL is used.**

CREATE TABLE assistant(
Pers_no **INTEGER NOT NULL PRIMARY KEY**,
Name **VARCHAR(0) NOT NULL**,
Area **VARCHAR(200) DEFAULT** 'computer science'
);

Index Structures:

An index speeds up the retrieval, but slows down update operations
Synatx:
CREATE [UNIQUE] INDEX index name
ON TABLE relation (attribute [asc | desc],
attribute [asc | desc], …) ;

Removing Objects :

Remove objects with the drop statement:

- ➔ **DROP TABLE** relation name; (delete all info about dropped table)
- ➔ **DROP VIEW** view name;
- ➔ **DROP INDEX** index name;

ALTER TABLE:

- add columns to existing table: alter table r add A D(A=column to be added, D = domain of A)
- drop columns from existing table: alter table r drop A (A = column of table r)

DCL:

Contains operations to control the flow of transactions
Transaction = sequence of interactions between an application/user and the DBMS

- ➔ some insights into SQL queries in Applications
- ➔ some stuff about embedded SQL
- ➔ Dynamic SQL

Active Databases:

With the help of constraints (PRIMARY KEY, FOREIGN KEY, CHECK CLAUSES) we increased the data semantics → anything that the database can take care of, does not have to be done separately in each application. But this can be taken one step further beyond constraints.

Triggers:

Triggers are general-purpose actions that are automatically invoked by certain events.
They are a mechanism for formulating complex constraints in a DBMS.
Executes a series of SQL statements whenever data is

- → Inserted,
- → Deleted
- → Or updated

In a specific table.
Triggers require an **Event**, and **Activation Time**.

Events activating the trigger have to be specified:

- → **INSERT ON** table-name
- → **DELETE ON** table-name
- → **UPDATE** (**OF** column-name) **ON** table-name

Activation Time activation of the trigger:

- → BEFORE <Event>
- → AFTER <Event>
- → F.e. **BEFORE INSERT ON** R

Granularity:

- → FOR EACH ROW
- → FOR EACH STATEMENT

Transition variables:
We often need to know the state of the DB before and after the event.
Information is made available via transition variables:

- → Old tuple variable: tuple values before change
- → New tuple variable: tuple values after change
- → Old table variable: table before change
- → New table variable: table after change

Trigger Condition:
IF there is a trigger condition, then the trigger is only activated if this condition is true
Uses the keyword **WHEN** (instead of WHERE)
WHEN (newrow.salary < oldrow.salary)
WHEN (SELECT count(*) FROM oldtable) > 100

Trigger Body:
Consists of one or more SQL statements
Trigger body is executed atomically → all or nothing
Other building blocks:

- → Assignments (can be used to modify values to be inserted or updated)
- → Signals (can be used to raise errors

Before Triggers:
Act as constraints, checking conditions before allowing certain operations to go ahead

```
CREATE TRIGGER empt_start_salary
        NO CASCADE BEFORE INSERT ON emp
        REFERENCING NEW AS newrow
        FOR EACH ROW
        SET (salary, bonus) =
                (SELECT saraly, bonus
                FROM startingPay
                WHERE jobcode = newrow.jobcode);
```

Trigger limiting salary increases:

```
CREATE TRIGGER  emp_inc_salary
        NO CASCADE BEFORE UPDATE OF salary ON emp
        REFERENCING OLD AS  oldrow NEW AS newrow
        FOR EACH ROW
        WHEN (newrow.salary > 1.5 * oldrow.salary)
        SET newrow.salary = 1.5 * oldrow.salary;
```

After Triggers:
Can interact with the outside world

This trigger checks for a decreasing salary and interacts with the outside world via a log file:

```
CREATE TRIGGER emp_dec_salary
        AFTER UPDATE OF salary ON emp
        REFERENCING OLD AS oldrow NEW AS newrow
        FOR EACH ROW
        WHEN (newrow.salary < oldrow.salary)
        BEGIN ATOMIC
                VALUES(logEvent('Salary decrease',
                                CURREN TIMESTAMP,
                                Oldrow.empno));
                SIGNAL SQLSTATE '...' ('Salary decrease?);
```

- Next up is a trigger for a meteorological database keeping a table with highest temperatures up-to-date:

```
CREATE TRIGGER extreme_temp
  AFTER UPDATE ON temperatures
  REFERENCING NEW AS newrow
  FOR EACH ROW
  WHEN (newrow.temp >
           (SELECT hightemp
            FROM   extremes
            WHERE  place = newrow.place))
  UPDATE extremes
    SET hightemp = newrow.temp,
        highdate = CURRENT DATE
    WHERE place = newrow.place;
```

Recursive Triggers
The body of a trigger may apply some updates to a DB → these updates may in turn cause other tirggers to activate.
A recursive trigger is a trigger which reactivates itself due to changes that occur when the trigger activates.

```
CREATE TRIGGER add_sales_tax
  AFTER UPDATE ON invoice
  REFERENCING NEW AS newrow
  FOR EACH ROW
  UPDATE invoice
    SET newrow.total_price =
        1.08 * newrow.total_price;
```

Constraints should be preferred over triggers since constraints are more declarative.

Relational Design Theory

Relational schemas have a lot of redundant data which can express themselves in anomalies when updating, inserting or deleting.
To improve the quality of relational schemas we have to get rid of that redundant data.
We measure the quality of a relational schema via Normal Forms.
If we are not satisfied with the quality bzw. The stage of the Normal Form (NF1-4), we can use a decomposition algorithm to improve said quality.
All decomposition algorithms guarantee recoverability.
Preservation of dependencies can be guaranteed up to 3NF
Functional Dependencies → Keys → Normal Forms → Decomposition into wished Form

Anomalies of a DB:

- ➔ Update anomaly (changing a single fact requires to touch multiple tuples)
- ➔ Insertion anomaly (inserting a tupel without all the req. information results in lots of NULL)
- ➔ Deletion anomaly (Deleting a tuple removes more information than intended)

Anomalies appear when a relation has redundant data.
This happens because certain attribute values determine other attribute values → there are **functional dependencies** FD between attributes.

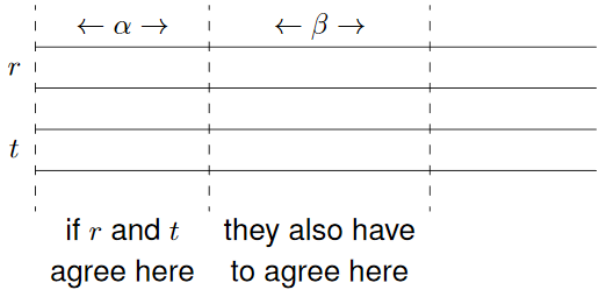
Relational Database Design Guidelines:

- ➔ Guideline 1: each tuple in relation should only rereset one entity or relationship instance
- ➔ Guideline 2: design schema that doesn't suffer from insertion, deletion and update anomalies
- ➔ Guideline 3: relations should be designed such that their tuples will have as few NULL values as possible; attribute that are null values shall be placed in separate relations
- ➔ Guideline 4: relations should be designed such that no spurious (wrong) tuples are generated if we do a natural join of the relation

Functional Dependencies:

- ➔ Used to define normal forms for relations
- ➔ Constraints
- ➔ **X → Y: X functionally determines Y, value of X determines a unique value for Y**

(X and Y are attribute sets of a relational schema R)



(Alpha = X and Beta = Y)

FD constraint must hold for all possible instances r(R)

Via FDs we can compute a key

A key K has the following properties:

- ➔ 1. K ⊆ R
- ➔ 2. K → R (the key is complete)
- ➔ 3. There is no K' ⊂ K such that K' → R (the key is minimal)

If K satisfies all the properties above It is a candidate key. (a relation may have several candidate keys)

Choose one candidate key as the primary key

If k does only fullfill properties 1 & 2 its called a superkey (**candidate key ⊆ superkey**)

If K is candidate key of R then K functionally determines all attributes in R.

Armstrong's Axioms:

Can be used to check properties 2 and 3

Let α, β and γ be subsets of R

Three Axioms:

- ➔ Reflexivity: β ⊆ α ⇒ α → β
- ➔ Augmentation: If α → β, then αγ → βγ
- ➔ Transitivity: α → β and β → γ ⇒ α → γ

Using them to find the keys is tedious; you can do iterations with the closure set of attributes

Closure of a Set of Attributes:

AC(α) is the closure of a set of attributes α. That means the set of attributes which can be FD from α.
Or said differently: **AC(α) contains all the attributes functionally determined by α**.
Figuring out AC(α) with Armstrong's Axiom is tiresome; we use a simple algorithm:

$AC := \alpha$
while (AC still changes) do
    for each FD $\beta \rightarrow \gamma$ in $\mathcal{F}_{\mathcal{R}}$ do
        if ($\beta \subseteq AC$)
        then $AC := AC \cup \gamma$

Step 1: look for attributes that **do not appear (also those that do not appear anywhere) on RHS** of any FD, these have to be part of the key
Step 2: attributes that **only appear on the RHS** of FDs, then they are dead ends (you cannot reach other attributes from these); adding them would violate minimality and hence these are never part of the keys
Step 3: (these are the ones we have to check) maybe attributes meaning not in part 1 or part2

Design theory can be used to check the quality of a schema. This is done via normal forms (NFs): 1NF, 2NF, 3NF, BCNF, 4NF
Meaning reducing data redundancy and improving data integrity basically dealing with data anomalies.

First Normal Form (1NF):

A relational schema is in 1NF, if all attribute values are atomic (no substructures allowed): (**a single cell shall not hold multiple values**)

➔ An attribute may take on only one value from its domain (f.e. char(30))

parents (in 1NF)

mother	father	child
Mary	John	Jen
Mary	John	Dave
Sue	Mike	Kate
...	...	...

No new attributes which are filled with nulls, or changes to the domain like f.e. a list…
(Superkey on the left hand side (for every single FD))

Second Normal Form (2NF):

It has to be in 1NF
Table also should not contain partial functional dependency or put different; every non key attribute has to be fully functional dependent on the key attribute in the relation. Which means, that there may not be a Subset of the key on which the non key attribute is dependent. (AB → C has to hold; if A → C is true, then C is not fully functionally dependent on AB und thus its not the second normal Form)
(Superkey on the left hand side (for every single FD))

Third Normal Form (3NF):

It has to be in 2nd NF

There should be no **transitive functional dependency for non-prime attributes (so no dependency on anything else than the actual relation key; transitive relations for key attributes don t matter)**

If C is dependent on B and B is dependent on A then C is dependent on A → this would violate the third normal form. (because in third NF if we have only one key attribute in the relation we can still have redundancy via transitive relations among the attributes in the relation.)

Split the table, so that all the non-key attributes are fully functional dependent only on the primary key.

Which means one of these holds: (Every attribute in this schema is part of some candidate key)

- α→β is trivial, bzw. β ⊆ α
- α is a superkey (for every single FD)
- Every attribute in β is part of a candidate key

Every non-key attribute has to depend on the key, the whole key, **and nothing but the key**, so help me Codd

BCNF:

It has to be in 3rd Normal Form

Every functional dependency A → B, then A has to be the Super Key of that particular table

Insert a new column to remove the non-prime attributes functional dependency.

Every attribute (including key attributes) has to depend on the key, the whole key, and nothing but the key (no transitive attributes anymore)

Thightened conditions:

- α→β is trivial, bzw. β ⊆ α
- α is a superkey (whatever is on the left hand side of the dependency is a superkey)

Multi-valued Dependencies:

When throwing together two completely independent aspects, we get rid of the FDs but there will still be redundancies.

We could store such cases way more efficient in two tables.

skills	
pers_no	lang
3002	English
3002	Italian
3005	German
3005	Italian

skills	
pers_no	proglang
3002	C
3002	Java
3005	C

For all tuples with the same value for α, all possible β, γ-combinations have to appear; then we have found a Multivalued Dependency in our Relation. formally:

Let $\alpha, \beta, \gamma \subseteq \mathcal{R}$ (with $\alpha \cup \beta \cup \gamma = \mathcal{R}$)

$\alpha \twoheadrightarrow \beta$ holds if for every instance R:
for every pair of tuples t_1, t_2 in R with $t_1.\alpha = t_2.\alpha$
exists a tuple $t_3 \in R$ with

- $t_3.\alpha = t_1.\alpha (= t_2.\alpha)$
- $t_3.\beta = t_1.\beta$
- $t_3.\gamma = t_2.\gamma$

MVDs are a generalization of FDs, i.e., every FD is an MVD (but not necessarily the other way around)
An important property of MVDs is: **α→→β ⇒ α→→ γ for γ = R – α – β (if γ is basically the rest)**

Fourth Normal Form (4NF):

The 4NF is a generalization of BCNF to MVDs
A relational schema is in 4NF, if for every MVD α→→β at least one of the following conditions holds:

- α→→β is trivial, i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$
- α is a superkey

Decomposition of Relations:

A relational schema can be transformed into a higher normal form by taking it apart.

- Decompose a schema R into subschemas $R_1, \ldots, R_n$ with $R_i \subset R$ for $1 <= i <= n$

A decomposition needs to have two properties to be useful:

- Recoverability of information
  - $R = R_1 \cup R_2$ and $R_1 = \pi_{R1}(R)$, $R_2 = \pi_{R2}(R)$
  - **A decomposition recovers all information if for all R instances $R = R_1 \bowtie R_2$ holds**
    - We can do this if one of the following conditions holds:
      - $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_1 \in \mathcal{F}_{\mathcal{R}}^+$
      - $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_2 \in \mathcal{F}_{\mathcal{R}}^+$
      - Or formulated differently: If we can derive either subschema R1 or R2 from the intersection of R1 and R2 then we are Fine. (so we split the table, and then The attribute on which we intersect the tables has to be able to cover the other attributes or rather via that attribute we have to be able to reconstruct the other relation; the coverage of that intersection attribute has to be one of the relations).
  - If done wrong we end up with mure tuples than before when reconstructing
- Preservation of dependencies

Decomposition Algorithms:

3NF Synthesis Algorithm:

- Prerequisites:
  - **Minimal Basis**
    - $\mathcal{F}_c \equiv \mathcal{F}$, i.e., $\mathcal{F}_c^+ = \mathcal{F}^+$
    - There are no FDs $\alpha \rightarrow \beta$ in $\mathcal{F}_c$ in which α or β contain unnecessary attributes
    - The left-hand side of every FD in $\mathcal{F}_c$ is unique
      - This can be achieved by applying the union rule: $\alpha \rightarrow \beta, \alpha \rightarrow \gamma \Rightarrow \alpha \rightarrow \beta\gamma$

- How do we check for "unnecessary" attributes?
  - $\forall A \in \alpha$:
    $(\mathcal{F}_c - \{\alpha \to \beta\} \cup \{(\alpha - A) \to \beta\}) \not\equiv \mathcal{F}_c$
  - $\forall B \in \beta$:
    $(\mathcal{F}_c - \{\alpha \to \beta\} \cup \{\alpha \to (\beta - B)\}) \not\equiv \mathcal{F}_c$
- We do this in two steps: first looking at the left-hand side (LHS) and then at the right-hand side (RHS)
  - **Minimizing LHS**
- For the LHS of every FD $\alpha \to \beta \in \mathcal{F}$ check
  - for all $A \in \alpha$ if A is unnecessary, that means:
    $\beta \subseteq AC(\mathcal{F}, \alpha - A)$
  - If yes, replace $\alpha \to \beta$ by $(\alpha - A) \to \beta$.
  - **Minimizing RHS**
- For the RHS of every (remaining) FD $\alpha \to \beta \in \mathcal{F}$ check
  - for all $B \in \beta$ if B is unnecessary, that means:
    $B \in AC(\mathcal{F} - \{\alpha \to \beta\} \cup \{\alpha \to (\beta - B)\}, \alpha)$
  - If yes, replace $\alpha \to \beta$ by $\alpha \to (\beta - B)$
  - **If present, remove all FDs of the form α → ∅**
  - **Merge all FDs that have the same LHS (union rule):**

    $\alpha \to \beta_1, \ldots, \alpha \to \beta_n$ become $\alpha \to \beta_1 \cup \cdots \cup \beta_n$

- Apply Algorithm:
  - 1. For every FD α → β in F_c create a subschema R_α = α ∪ β
    - With F_α := {α' → β' ∈ F_c | α' ∪ β' ⊆ R_α}
  - 2. Add a schema R_K that contains a candidate key
  - 3. Eliminate redundant schemas, i.e., if you find $R_i \subseteq R_j$, drop R_i

Example:

Let's apply this algorithm to `finance` with
$\mathcal{F}_{\mathcal{R}_c} = \{B \to O, S \to D, I \to B, IS \to N\}$

- $\mathcal{R}_B(B, O)$
- $\mathcal{R}_S(S, D)$
- $\mathcal{R}_I(I, B)$
- $\mathcal{R}_{IS}(I, S, N)$ (contains key)

Decomposition in BCNF:

- Start with $Z = \{\mathcal{R}\}$
- If there is still an $\mathcal{R}_i \in Z$ that is not in BCNF
  - Find an FD $(\alpha \to \beta) \in \mathcal{F}^+$ with
    - $\alpha \cup \beta \subseteq \mathcal{R}_i$
    - $\alpha \cap \beta = \emptyset$
    - $\alpha \to \mathcal{R}_i \notin \mathcal{F}^+$
  - Decompose $\mathcal{R}_i$ into $\mathcal{R}_{i_1} := \alpha \cup \beta$ and $\mathcal{R}_{i_2} := \mathcal{R}_i - \beta$
  - Remove $\mathcal{R}_i$ from Z and add $\mathcal{R}_{i_1}$ and $\mathcal{R}_{i_2}$:
    $Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i_1}\} \cup \{\mathcal{R}_{i_2}\}$

Decomposition into 4NF:

- Very similar to BCNF, just replace FD with MVD
- Start with $Z = \{\mathcal{R}\}$
- If there is still an $\mathcal{R}_i \in Z$ that is not in 4NF
  - Find an MVD $(\alpha \twoheadrightarrow \beta) \in \mathcal{F}^+$ with
    - $\alpha \cup \beta \subseteq \mathcal{R}_i$
    - $\alpha \cap \beta = \emptyset$
    - $\alpha \rightarrow \mathcal{R}_i \notin \mathcal{F}^+$
  - Decompose $\mathcal{R}_i$ into $\mathcal{R}_{i_1} := \alpha \cup \beta$ and $\mathcal{R}_{i_2} := \mathcal{R}_i - \beta$
  - Remove $\mathcal{R}_i$ from Z and add $\mathcal{R}_{i_1}$ and $\mathcal{R}_{i_2}$: $Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i_1}\} \cup \{\mathcal{R}_{i_2}\}$

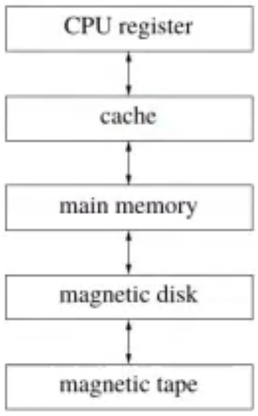
Data Storage:

Using tuple identifiers (TIDs) and slotted pages is a clever way of storing tuples on a disk.
Indexes accelerate access to data items, but makes updates more expensive.

B^+ - trees are the standard index structures used in relational systems. They are suited for point and range queries.

Memory Hierarchy:

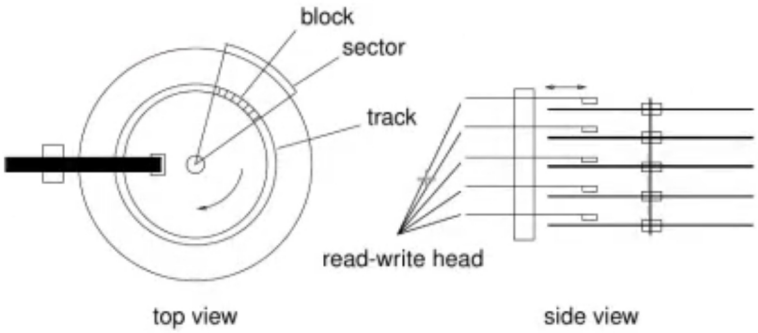
- Memory is organized hierarchically
- The higher in the hierarchy, the faster, smaller, and more expensive
- Differences in orders of magnitude
- We focus on main memory and magnetic disks



CPU register at the top → then some caches between the main memory (=RAM) and the CPU register, because it takes quite a while to transfer data from the main memory to the CPU and the CPU is way faster than the main memory. So basically, every time the main memory is sending data to the CPU the CPU is doing “nothing” for quite a while. Chaches (multiple cachelevels as well usually) are there for frequently called for data. Magnetic disks (today some use SSDs). (also caches between the main memory and magnetic disk). Accessing between main memory and magnetic disk and CPU register and caches is sequential (a lot) which is nice for the caches. Main focus on main memory and magnetic disk interaction or us.
Huge differences in terms of speed between different steps (orders of magnitude). (CPU its in my hand, Cache its in my room, Main memory its somewhere outside of my house; in a store or something, Magnetic disk its not even in my city anymore; several hours of travel away, magnetic

tape its not even in the same continent anymore; somewhere I have to take an intercontinental flight.)

Magnetic Disk



- This is a simplified, schematic view

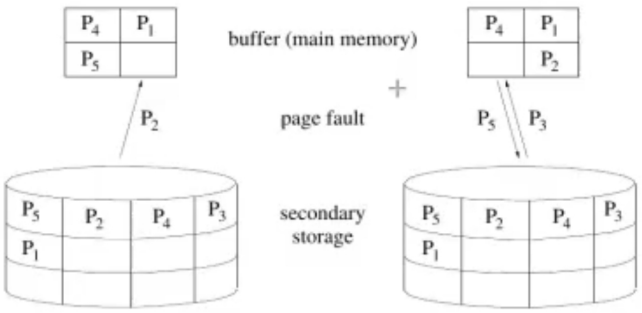
Usually 4 disks on top of each other where the read-write heads operate on 4 different surfaces (they can work in parallel).
Data storage on the blocks.

- Reading a block from disk:
  - Position the read-write head over the correct track (seek time)
  - Wait until sector/block appears under the head (rotational latency)
  - Read the content of the block (data transfer rate)

Physically positioning the read-write head takes a lot time (relative to electronic operations).
Basically, a RAM which keeps its state even when electricity is switched off.

Buffer Management

- All data processing takes place in main memory
- Data not in main memory has to be fetched from disk



All data processing takes place in main memory!
After that it has still to be sent to the CPU (but we focus on main memory and the magnetic disk)
Its alright if we want to access for example P2 –> it will just fetch it from the disk. However then the main memory is full. If we want f.e. page 3 we have to drop P5 to fetch P3 from the disk. That's basically buffer management. (Strategies like LRU, Clock etc. which handle the efficient swapping of data in main memory and disk storage.)

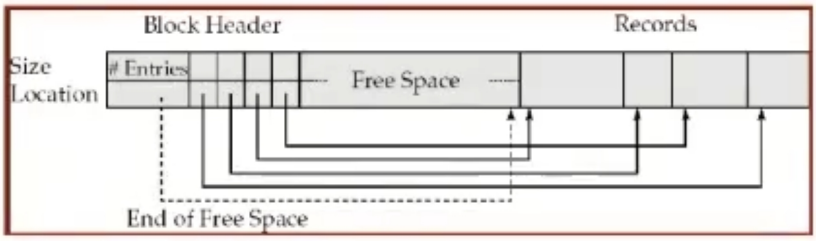
Storing Relations (How DBs store their data):

- Databases usually don't address the physical blocks of a disk directly
- They use *pages* or *logical blocks* that are mapped to physical blocks
- Tuples or records are then stored on these pages
- As records may contain variable-length fields, we need *slotted pages*

How are tuples like varchars which are variable stored on those pages?

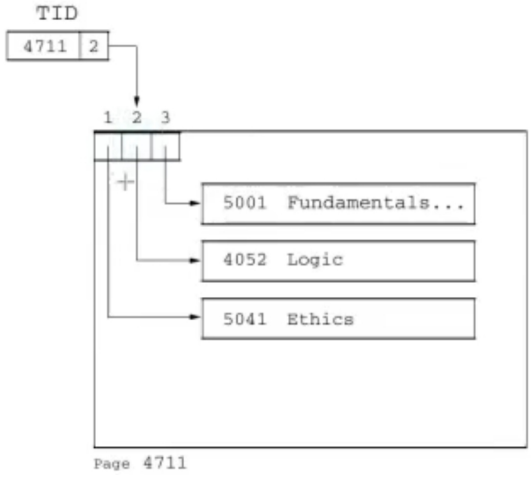
Slotted Pages:

- Every page includes a *page header*, containing information about
  - number of record entries
  - location and size of each record
  - end of free space



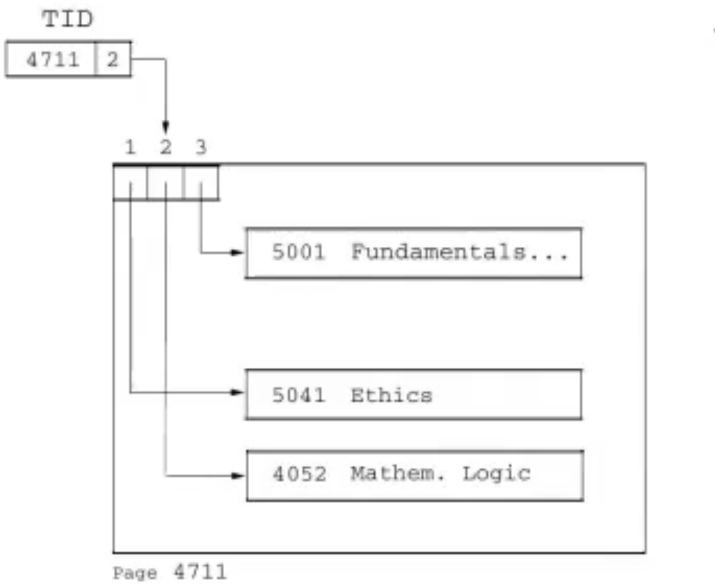
Whenever Block Header and Records meet, we ran out of space on the page.
Blockheaders tell me where on the page records are actually stored.

Tuple Identifiers (TIDs):



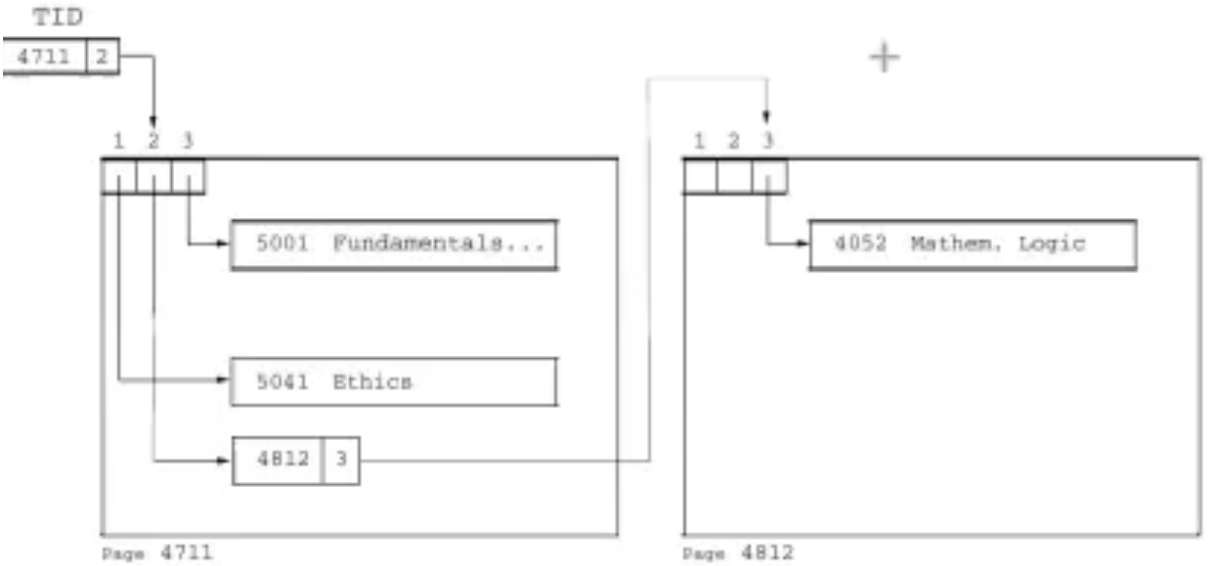
These slots in the header are called TIDs (the IDs in the Block Header are TIDs). They are unique descriptions for whatever I search on the disk.
First part of the TID tells me the page (here 4711) and the second number tells me where the Tuple is stored on the page (it's a slot number → where the pointer is stored in the page header).
Example: go to page 4711, follow the pointer which is stored in slot number 2 in the page header and look up what we want to know.
Every tuple in our DB can be uniquely identified by a TID.
When creating an index on a table, we store TIDs in this index. After the index lookup we get a set of TIDs and with that we can look up the data. That why we don t want to change the original TIDs ever.

Moving a Tuple Within a Page



Tuples might change and we have to move it around in the page. But the TID stays the same. The only thing we have to change is the pointer in the slot number 2 (because the position on the page where the pointer points to has changed).

Moving a Tuple to Another Page



WE want to keep the original TID. So instead of changing the TID we move the tuple to a new page, which basically changes the TID because the page and the slot number have changed, but we just copy that new TID to the location where the old TID has been. Inefficiency is implemented to some extent. (we basically give a pointer to a pointer).
If I move to yet another page, I just change the pointer in the first page. Basically I only want one level of indirection. I just update the pointer of the original TID to point to the now newly changed pointer again.

Data Transfer:

- Transferring all the tuples into main memory is the straightforward way to process queries
- Unfortunately, it is also the most expensive approach

- Looking at this more closely, we find out that
  - often only a tiny fraction of all tuples is needed to answer a query
  - queries often access similar parts of the data
  - hard disks allow random access

Index Structures:

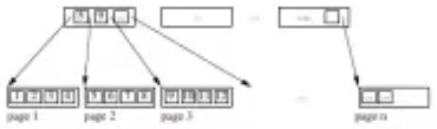
- Index structures exploit this to reduce the amount of transferred data
- Only the part that is needed to answer the query is fetched

- The two most important approaches for indexing:
  - hierarchical (trees)
  - partitioning (hashing)

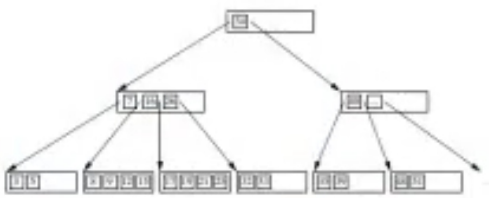
We use those properties named in Data transfer to have a more optimal approach to transferring data (only what we really need, not billions of tuples in a big DB).

Hierarchical Indexes:

- We look at two hierarchical index structures:
  - ISAM (Index-Sequential Access Method)



  - B-trees



B-trees are the standard industry structure. Every DB basically supports B-trees.

ISAM:

Main Idea is:

- Sort tuples according to indexed attributes
- Put an index on top of this

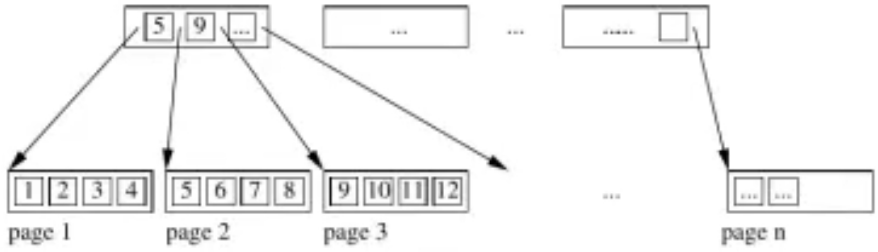
Similar to a thumb index in a book (get you to the right letter directly so you don t look through the entire alphabet first)

Example:

- We are looking for the student with student number 13542
- Assume that the tuples in the relation `student` are ordered by student number

+

index pages



data pages

Everything is sorted by student numbers. WE put some index pages on top. So the first index tells us page 1 goes up to 5 (exclusive), index 2 tells us the page goes up to 9 (exclusive), …

We basically search through the index pages and search for 13542 until we find a smaller number followed by a larger number and then we know which page to look on. This is way faster because the number of index pages is considerably smaller than the data pages.

Find the starting point of the range in the index → then go to the data pages and scan sequentially in the data pages.

Issues with ISAM

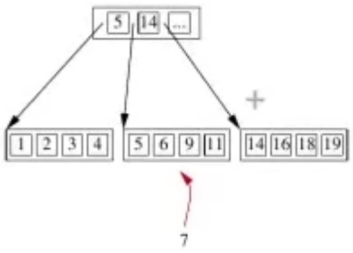
- Although searching an ISAM is straightforward and quick, maintaining the index can be expensive

+

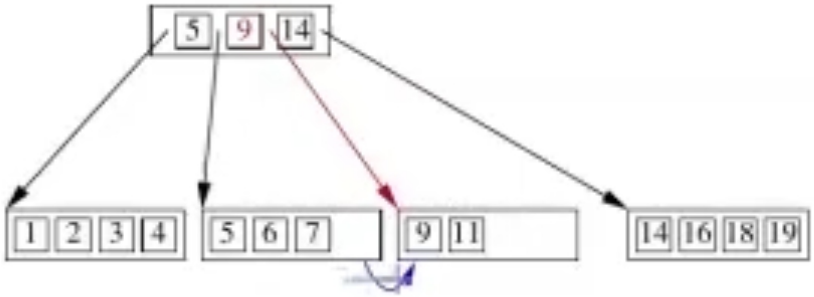
- The trouble starts when pages become full and we have to split them

Inserting a new item:

- We want to insert an item into a data page that is full:



- We have different options of handling this
- We could insert a new entry into the index, adding the value 9 and a new pointer (red option)
- However, that would mean moving index entries (possibly destroying contiguous storage)



- Or we could treat the new page as an overflow page (blue option)
- We don't have to change the index, but search takes longer with overflow pages

Careful with Red and Blue option; have to be seen separately.

Further Issues

- Although the number of index pages is significantly smaller than the number of data pages...
  - ...the traversal can still take some time
- Idea: why not have index pages for index pages?
- That is basically the principle of a B-tree!
  - Additionally, we can also store data in index pages
  - A B-tree can also be seen as a generalization of a binary tree

B-Trees:

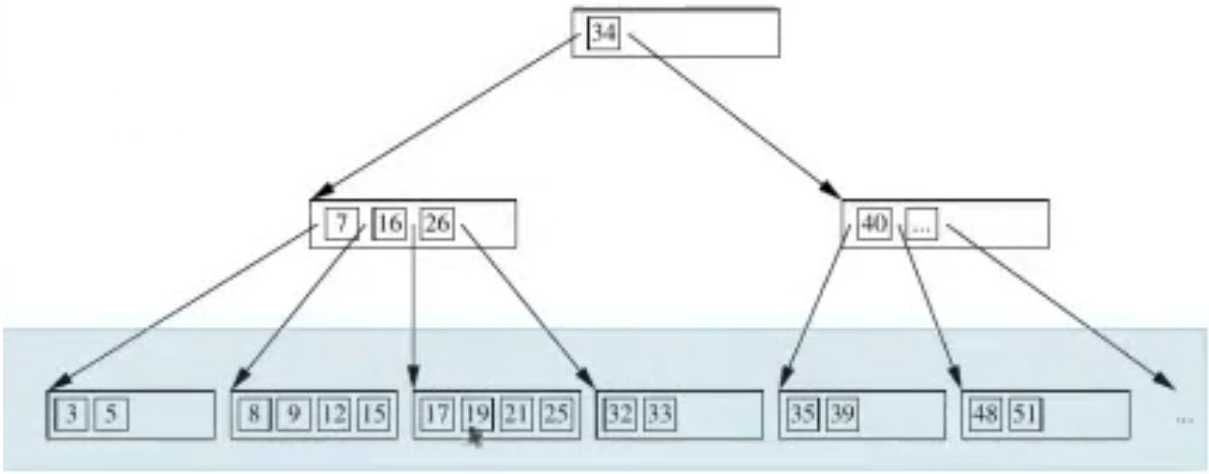
Basic problem: we don t want to scan through the index pages sequentially due to the reasons listed above. The idea → why not have index pages for index pages? We compress even more and become even faster.

Additionally, we can also store data in index pages

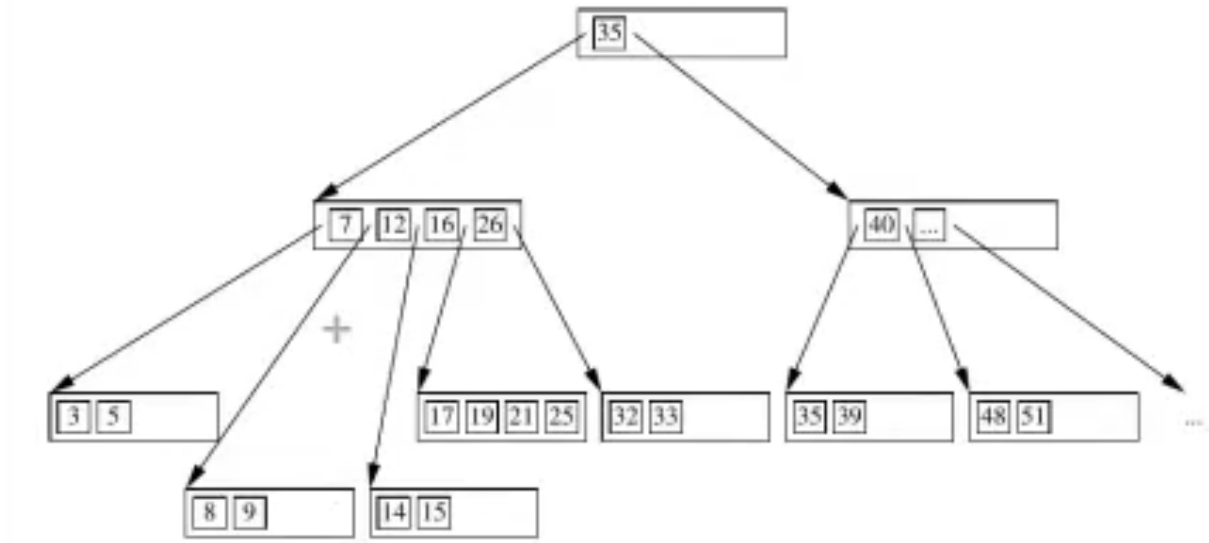
A B-tree can also be ssen as a generalization of a binary tree.

B-Tree Example:

- Let's have a look at an example:

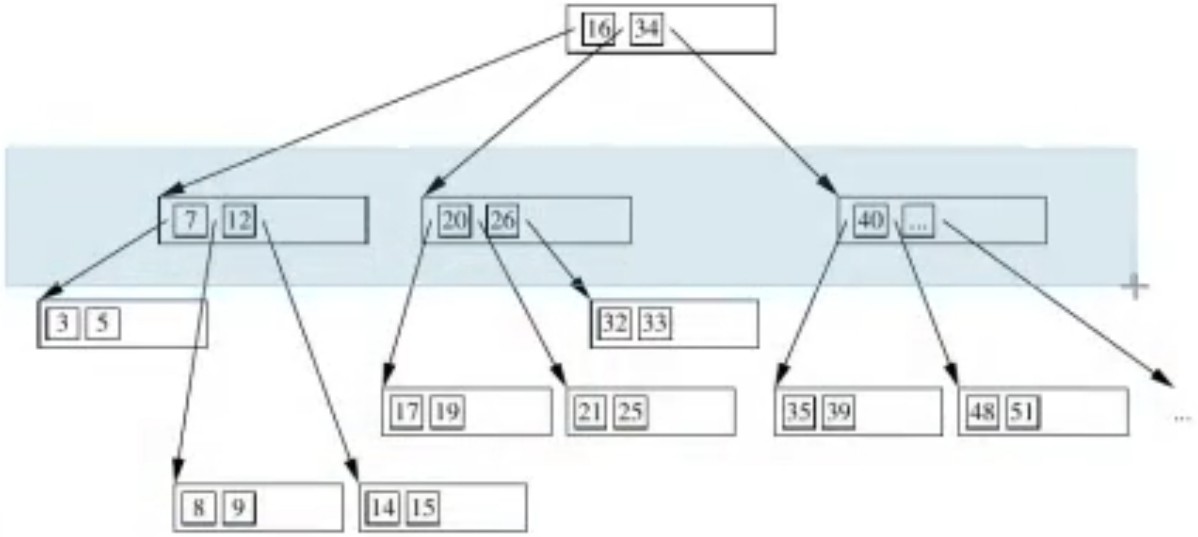


- Let's insert the data item 14 into the tree
- This leads to a page overflow: median value on the page is pushed upwards and page is split:



- Let's insert another data item: 20

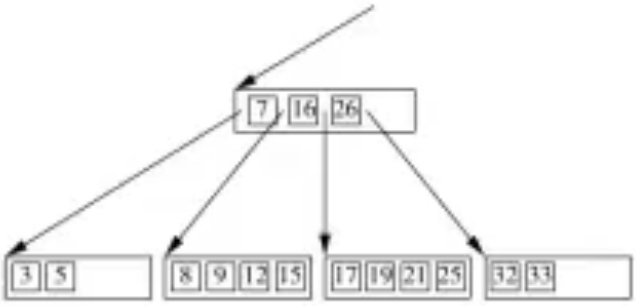
- This time, we have a page overflow on an index page
  - Treated in the same way: push median up, split page



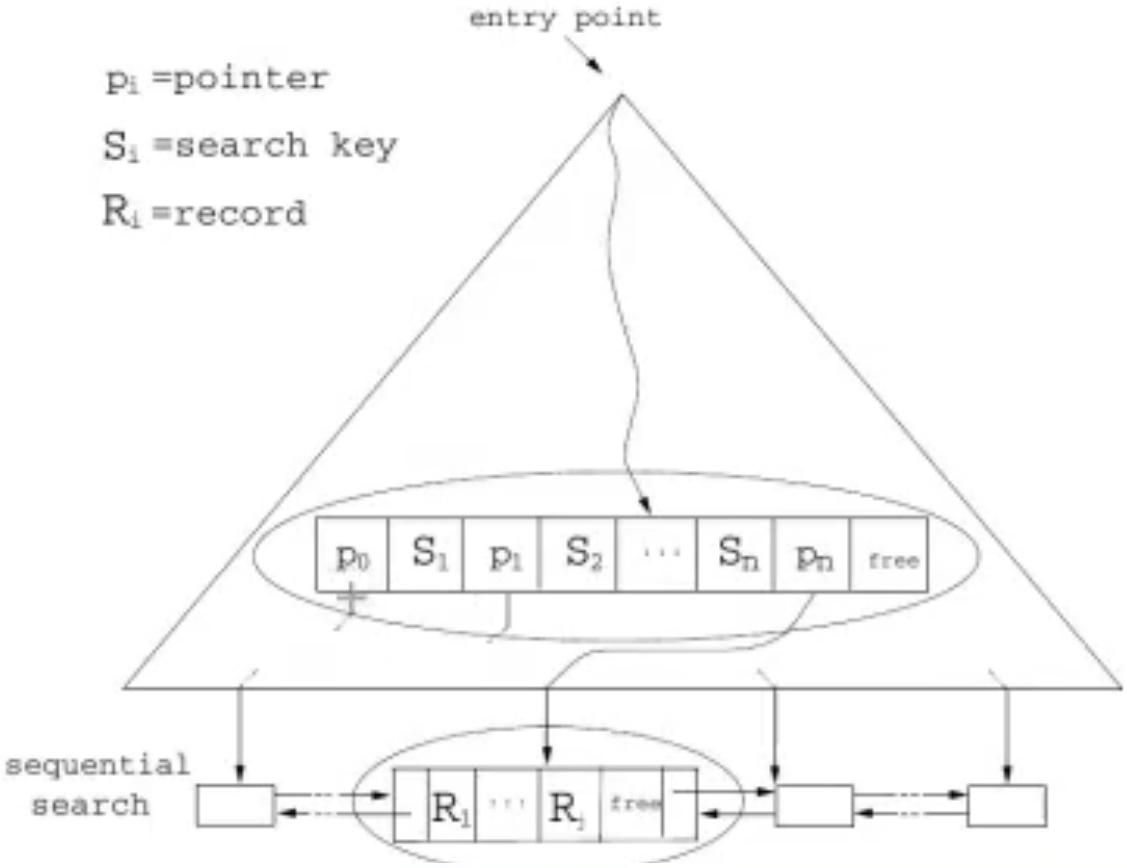
Its still three levels; it will get to 4 if the root node needs get to split up. **The root has 3 pointers!** Every number represents a tuple.

Deletion Algorithm

- Deletion of a value in a leaf is simple: just delete the value
- An inner node needs to stay properly connected to its children
- For example, we want to delete the value 16 from an inner node (distinguishes values in leaf nodes):



Schema of a B^+-Tree



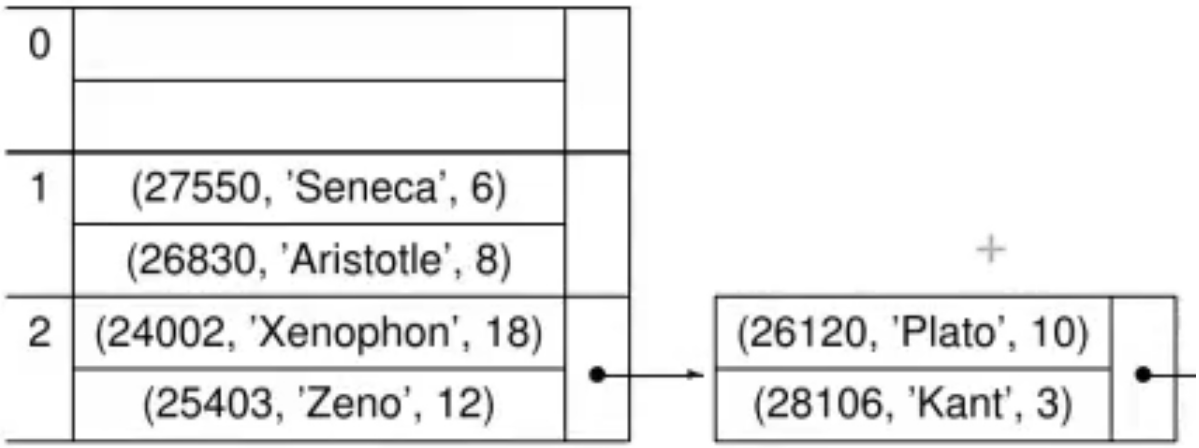
Hashing

- The value to be indexed is run through a hash function
- For instance, take $h(x) = x \bmod 3$
- Then insert value into a table according to hash value

0	
1	(27550, 'Seneca', 6)
2	(24002, 'Xenophon', 18)
	(25403, 'Zeno', 12)

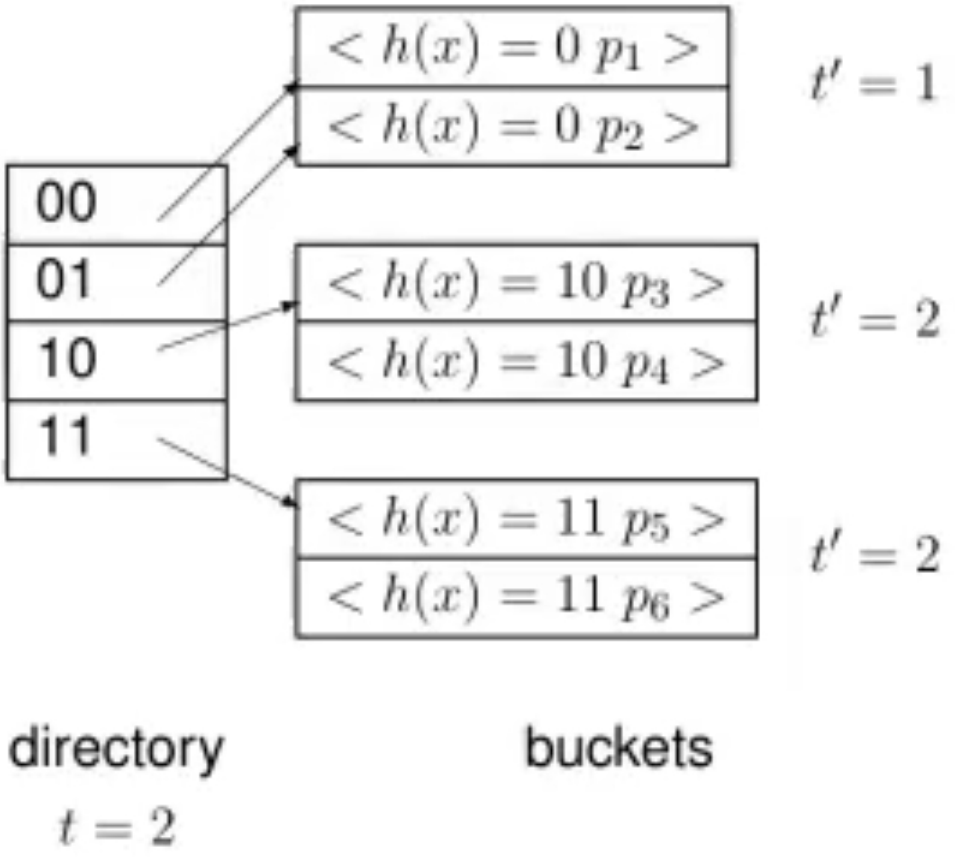
Hashing(2)

- What happens if we run out of space?
- We have to introduce overflow pages



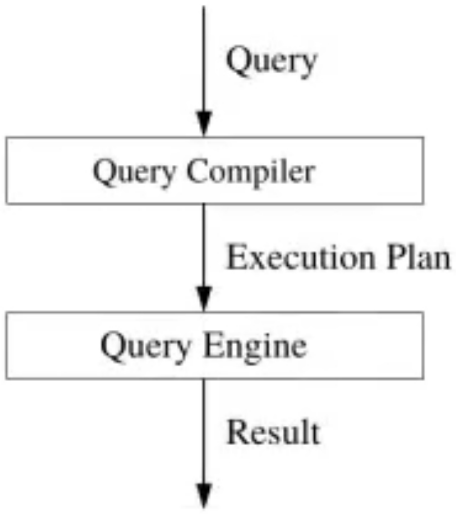
- Becomes inefficient if table becomes too small
- If we could make this adaptable...

Extendable Hashing



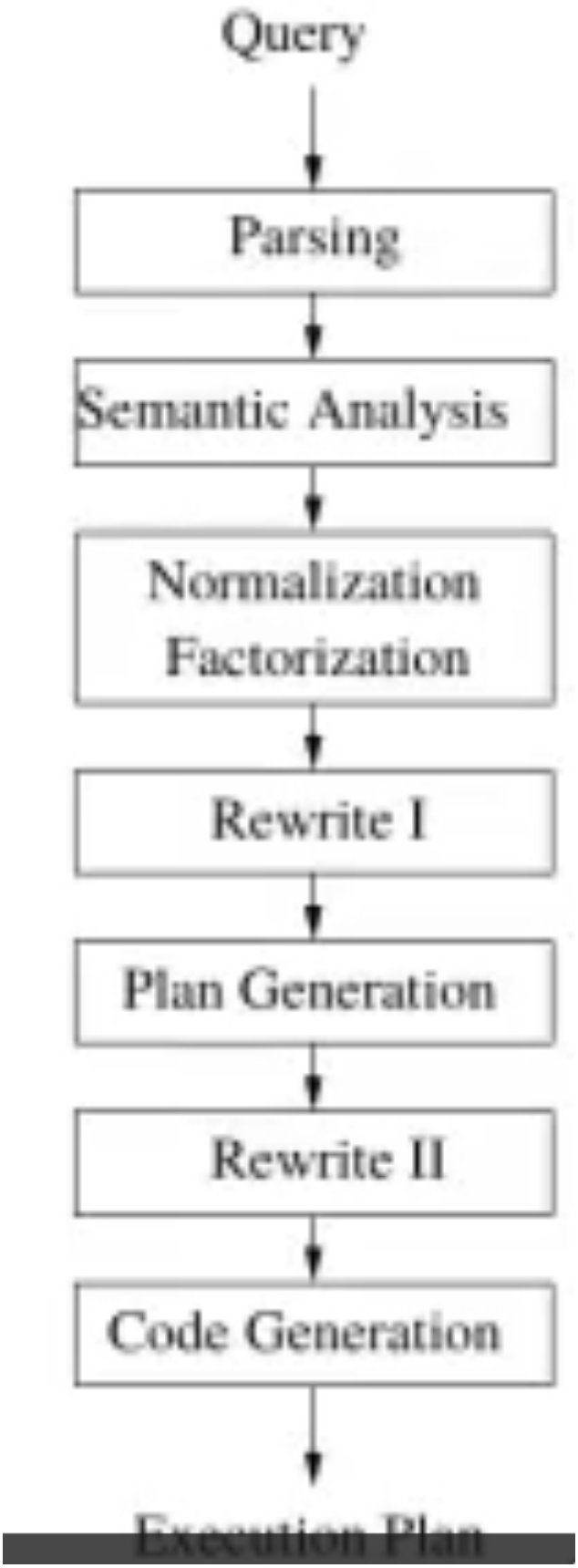
Query Processing:

High level overview:



Compilation

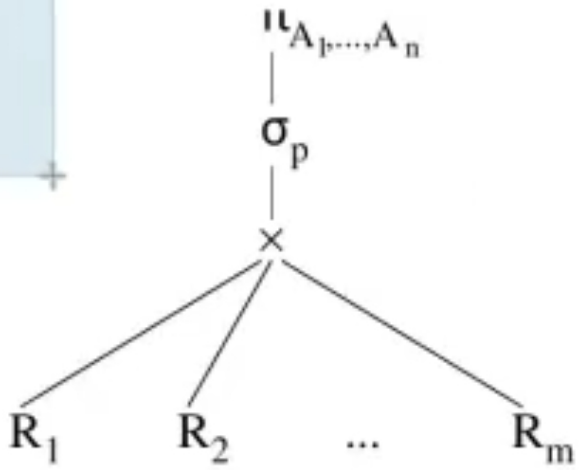
- SQL is declarative
  - Query has to be translated into a procedural program that can be run on the query engine
- DBMSs translate SQL into another format
  - A widely-used approach is the translation into relational algebra


Query
Parsing
Semantic Analysis
Normalization
Factorization
Rewrite I
Plan Generation
Rewrite II
Code Generation
Execution Plan

Canonical Translation

- There is a standard way of translating SQL into relational algebra
- Expressions in relational algebra are often represented graphically

select $A_1, \ldots, A_n$
from $R_1, \ldots, R_m$
where p



Optimization

- Canonical plan is not very efficient (e.g. it contains Cartesian products)
- Most DBMSs have query optimizers rewriting the plan to make it more efficient
- Query rewriting and optimization is a complex problem

Optimization (2)

- Why bother about query optimization as an ordinary user?
- Sometimes the query optimizer produces bad plans
- Nevertheless, most DBMSs allow a user to take a look at the generated execution plans
- Plans can be analyzed and query can be changed to make it more efficient

SUBOPTIMAL

Query Optimization 101

- A DBMS can estimate the execution costs of a plan with the help of
  - cost models
  - statistics
- Investigating all possible plans is usually too expensive
- An optimizer uses heuristics to optimize queries
- Optimization can take place on different levels:
  - Logical level
  - Physical level

Logical Level

- Starting point is the canonical relational algebra expression
- What can we do?
  - Transformation of algebraic expression into equivalent ones (ideally into faster ones)
  - A rule of thumb is to minimize the input/output of the individual operators

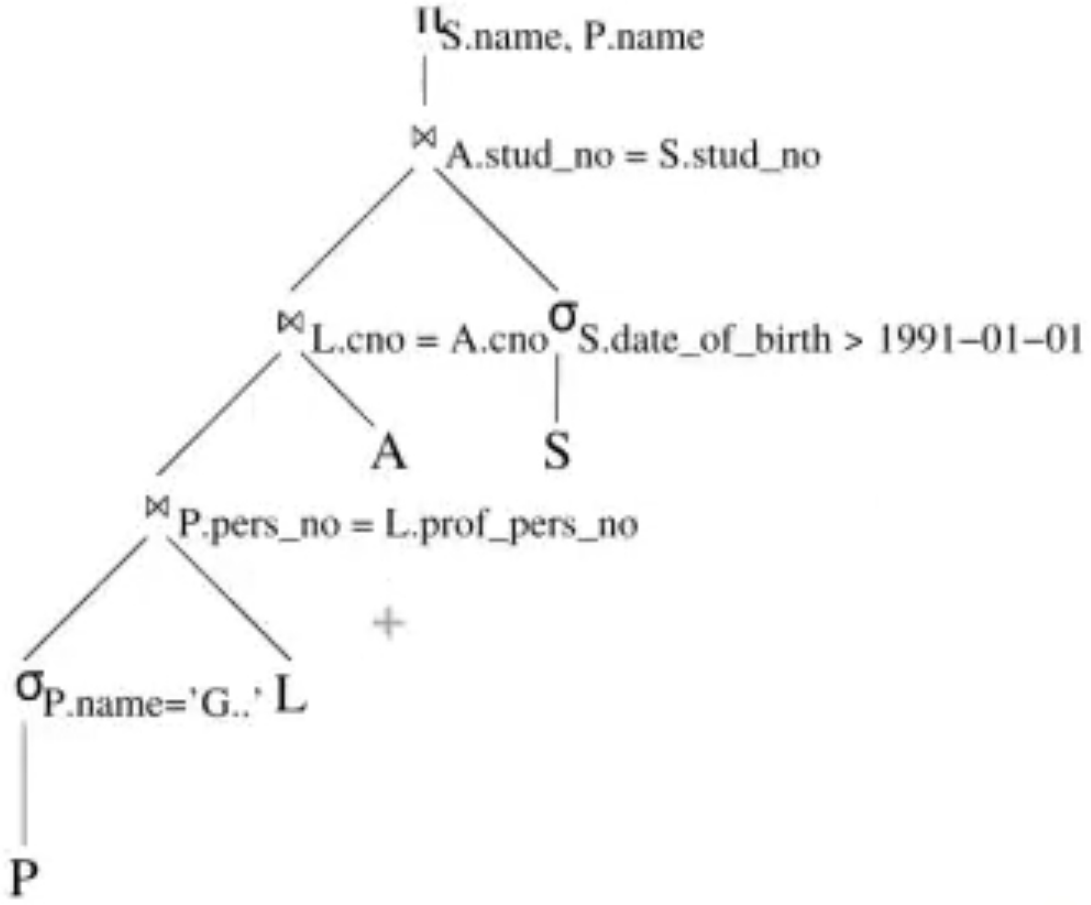
Logical Level (2)

- Basic techniques
  - Breaking up selections
  - Pushing “down” selections
  - Converting selection and Cartesian products into joins
  - Determining join order
  - Inserting projections
  - Pushing “down” projections

- Let’s play through some options with a concrete query:

```
select  S.name, P.name
from    student S, attends A, lecture L, professor P
where   S.stud_no = A.stud_no
and     A.course_no = L.course_no
and     L.prof_pers_no = P.pers_no
and     S.date_of_birth > 1991-01-01
and     P.name = 'Gamper';
```

(Optimized) Query Plan



(A Selection of) Rewrite Rules

$$
\begin{aligned}
R_1 \bowtie R_2 &= R_2 \bowtie R_1 \\
R_1 \cup R_2 &= R_2 \cup R_1 \\
R_1 \cap R_2 &= R_2 \cap R_1 \\
R_1 \times R_2 &= R_2 \times R_1 \\
R_1 \bowtie (R_2 \bowtie R_3) &= (R_1 \bowtie R_2) \bowtie R_3 \\
R_1 \cup (R_2 \cup R_3) &= (R_1 \cup R_2) \cup R_3 \\
R_1 \cap (R_2 \cap R_3) &= (R_1 \cap R_2) \cap R_3 \\
R_1 \times (R_2 \times R_3) &= (R_1 \times R_2) \times R_3 \\
\sigma_{p_1 \wedge p_2 \wedge \cdots \wedge p_n}(R) &= \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots)) \\
\Pi_{l_1}(\Pi_{l_2}(\ldots(\Pi_{l_n}(R))\ldots)) &= \Pi_{l_1}(R) \\
&\text{with} \quad l_1 \subseteq l_2 \subseteq \cdots \subseteq l_n \subseteq \mathcal{R} = \mathit{schema}(R) \\
\Pi_l(\sigma_p(R)) &= \sigma_p(\Pi_l(R)), \text{ if } \mathit{attr}(p) \subseteq l
\end{aligned}
$$

Rewritten rules...