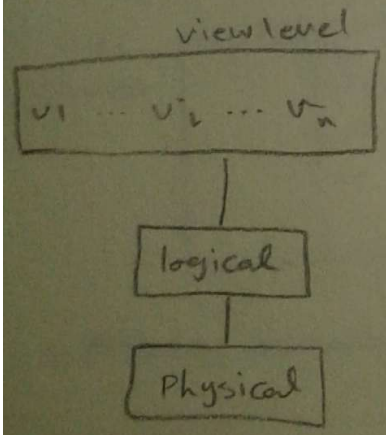
Data
- Structured
- Unstructured
- Storage
- Retrieval

DBMS: Data Base Management System → Providing users with an abstract view of data with some details hidden!
- connection of interrelated data
- Programs to access the data
- convinient & efficient

* Cons of using file systems → data redundancy and inconsistancy
  - difficulty in accessing data: need a new program for each task
  - data isolation → multiple files and formats
  - integrity Problems → constraints
  - atomicity of updates → inconsistancy Problems
    - some OPS should either be compeletely done or not at all happen
  - Concurrent access → uncontrolled concurrent access
  - Security Problems

* Levels of abstraction
  - Physical → how the data is stored
  - logical → describes data and relation
  - view → show data with hiding some details



* Data model: collection of tools for describing → data, semantics, relationships, constraints

types → relational model → data in tables
- XML
- entity-relationship
- object-oriented

* Schema ↔ ~~variables~~ Types Instances ↔ variable

Physical level → overall Physical structure of database
logical level → ~ logical ~ ~ ~ time

Instance → actuall content of the database at a Particular Point in

* Physical data independence: change Physical schema without having to change logical schema → must be well defined

(DDL) compiler
* Data Definition Language: for defining database schema →
set of table templates in a data dictionary
↳ contains meta data (data about data)

→ can only be changed by the DBMS not users
→ must contain constraints for data

→ Domain Constraints
→ Referential Integrity → table relations
→ assertion
→ authorization

can be tested
minimal
overhead

* Data Manipulation Language (DML): for accessing and manipulating
also called query language

~~possessing~~
→ deletion
→ modification
→ insertion
→ retrieval

DML types → Procedural: what and how to get data?
→ Declarative: what data?

DML classes → Pure: Provides Properties about computational Power & optimization
→ Relational Algebra
→ tuple relational ~~Calculus~~ Calculus
→ domain ~ ~
→ Commercial: for commercial systems → SQL

* Entity relationship model: describes the ~~logical~~ overall structure of the DB
logical structure

==> UML

* Normalization: design appropriate schemas → easy retrieval with minimal redundancy

* Database Engine → Storage Manager
→ Query Processing
→ Transaction Manager

* Storage Manager: a module that provides the interface between low level data & submitted queries
→ interaction with the O.S. file manager
→ efficient storing, retrieving and updating of data

Components → authorization & integrity manager
→ transaction manager
→ file manager
→ buffer manager

data structures → data files
→ data dictionary
→ indices

* Query Processing → 1. Parsing and translation
2. optimization → based on data statistics
3. evaluation
⇒ COST!

* Transaction Manager
↳ a collection of operations that perform a single logical function in a
→ ensures that the database remains in a consistent state despite system failures (Power and...) and transaction failures.

types → Recovery Manager
Concurrency-control manager: controls the interaction among concurrent operations

* Database Architecture → centralized
↳ client-server
↳ Parallel (multi-Processor)
↳ Distributed

* Intro to relational model *

* attribute (columns) → allowed values = domain → null is a member of every domain (unknown ⇒ complexity)
  → must be atomic

$A_1, A_2, \dots, A_n$ → attributes ⇒ $R = (A_1, A_2, \dots, A_n)$: relation schema

$D_1, D_2, \dots, D_n$ sets ⇒ subset of $D_1 \times \dots \times D_n$ = relation

$(a_1, \dots, a_n) \mid a_i \in D_i$

order not important ← (rows) tuples

current values = relation's instance

$K \subseteq R$

K = SuperKey ⟺ values for K are sufficient to identify a unique tuple of each possible relation r(R)

minimal K ⇒ candidate key (selected) ⇒ primary key

* Relational Algebra Operations (for relations r & s)

$\sigma_{\text{conditions}}(r)$: row selection (rows) — exp → $\sigma_{A \le B \wedge D > 5}(r)$ (doesn't remove duplicates)

$\Pi_{\text{columns \& conditions}}(r)$: column selection — exp → $\Pi_{A,C}(r)$ (doesn't remove duplicate tuples)

$\cup$: union of two relations — exp → $r \cup s$

$-$: set difference of two relations — exp → $r - s$

$\cap$: intersection ~ ~ ~ — exp → $r \cap s$

$\times$: Cartesian Product (join) — exp → $r \times s$ (doesn't remove duplicate)

naming attributes for relation r ⇒ r.A or r.B

$\rho_x(E)$: returns the expression E under the name x (renaming tables)

$\bowtie$: natural join of r & s → keep the same attributes, spread the different ones

* relational language → input & output = table
  → all data in output table appears in one of the input tables

SQL ⟹ case sensetive

- Domain Types:

  char(n) → fixed length n

  varchar(n) → variable length ≤ n

  int → machine dependent

  smallint

  numeric(p,d) → fixed point
  - → p digits on the left of the decimal point
  - → d ~ ~ ~ right ~ ~ ~

  real → floating point
  - → machine dependent

  float(n) → floating point
  - → max n digits

- Create Table:

```
create table r (
    A1 D1,
    ⋮
    An Dn,
    (integrity constraint1),
    ⋮
    (integrity constraint n));
```

not null

not null
Primary key(A_1, ...)
Foreign key(A_m, ...)
references s

← (integrity constraint1),

- Update Tables:

  insert into r values (...);

  delete from r → removes all tuples

  drop table r

  alter table r add A D → add attribute A with domain D
  - → all values are set to null

  alter table r drop A → drop attribute A

- Basic Query ⟹ result Relation

  select $A_1, \dots, A_n$ → like Projection oReration in relational algebra

  from $r_1, \dots, r_m$ → ~ Cartesian Product ~ ~

  where P → Predicate → renames common attributes

  select distinct A_1 → force the elimination of duplicates

  from ...

  ~ all ~ → all duplicates are included

  select * → all attributes

- literal attribute with no from clause ⟹ select '437'

Result → | col | → naming the col → select '437' as 'col-name'
| 437 |

select '437' from r ⟹ | 437 | } N → number of tuples in r
| ⋮ |
| 437 |

- select with operations ⟹ select $A_1/2, A_2-3, A_3*4, A_4+1$ from r

renaming → select $A_1/2$ as B_1, A_2+2 as B_2 ⟹ from r
{ can be used to rename relations too in that case as is optional

- comparison i 'where' clause → and, or, not → can be applied to the result of arithmetic expressions.

- (%) → matches any substring (_) → matches any character

⟹ strings are compared using like exp → where name like '%dar_'

(\) → escape character → { like '%' matches any string
like '\%' matches only %

case sensetive

_ "||" operator is used for string concatenation

- order by attribute → list in alphabetic order
  - desc → order by name desc
  - asc (default)

sort on multiple attributes ⟹ order by A_1, A_2

- where A_1 between x and y
- tuple comparison → $(A_1, A_2) = (A_1', A_2')$
- Set Operations

union ≡ ∪ intersect ≡ ∩ except ≡ − ⟹ eliminate duplicates

intersect all, except all, union all ← keep duplicates

- the result of any arthmetic expression involving null is null

where A_1 is null ⟹ this predicate checks for nulls

- three valued logic → true, false, unknown → result of any comparison with null → treated as false as predicate

check for unknowns: "P is unknown"

- Aggregate Functions

'avg, min, max, sum' (ignores null), count ⟹ select aggregate-function(A_i)

- Group by → obvious!

! ⟹ attributes in "select" clause outside of aggregate functions must appear in "group by" list

```
select dept, avg(salary) as avg-sal
from instructor
group by dept
"having" (avg-sal > 1000);
```

applied after forming groups unlike "where"

runtime error ↑

- Subquery

select ..., A_i, ... → replace by a subquery that generates a single value.

from ..., r_j, ... → replace by a subquery that's valid.

where ..., P_t, ... → A <operation> (subquery)

↳ no need for "having" clause

uses → set membership, set cardinality, set comparisons

- some (subquery)
- exists(subquery) or not exists(subquery)
- unique(subquery) → checks for duplicates
- Case

when ~~

else ~~

end

* Intermediate SQL *

* Join: takes two relations and returns a new one.
=> a cartesian product that requires tuples in the two relations match under certain rules and specifies the output's attributes!

==> typically used as subqueries in the from clause.

types → outer join → left, right, full
→ inner join

- outer join: avoids loss of info → uses null values
→ after computing the join, adds tuples that don't match from one relation.

Syntax: r_1 natural left outter join r_2 (left)
r_1 ~ right ~ ~ r_2 (right)
r_1 ~ full ~ ~ r_2 (full → union)

- inner join ⟷ natural join (default)

- JoinCondition: defines → matching tuple in 2 relation
→ present attributes in the output

- Join Type: defines how non-matching tuples are treated

Join Condition: natural
on <Predicate>
using $(A_1, \dots, A_n)$ → can cause duplicate attributes

* if instead of "on" we use "using", we define how and by which attribute the relations are joind and can control ~~multi~~ duplicate attributes.

* View: hide certain data from certain users
Def: any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a view.

==> create view V as <query expression> ==> generates a virtual view
==> causes the saving of the expression into the queries!

view relations: creat view V_1 as $\langle \dots V_2 \dots \rangle$ → V_1 depends directly on V_2
~ ~ ~ ~ $\langle \dots \langle \dots V_2 \dots \rangle \dots \rangle$ → " ~ on V_2 (can be direct)
~ ~ V " $\langle \dots V \dots \rangle$ → recursive

* view v_1 defined by expression e_1 that may contain other views =>

repeat
  find any view relation v_i in e_1
  replace v_i by the expression defining it
until there are no more view relations in e_1

↳ if there are recursive views, this loop will not terminate.

View Expansion

* View updating: modification in terms of a view must be translated to a modification to the actuall relations in the logical model.

* ~~Some~~ Some updates can not be translated uniquly!

=> Some implementations ~~do~~ allow updates only on simple views

→ Simple view: 1. from: only one database relation
2. select: * attribute names
3. select: not listed *s can be set to null
4. where: no "having" or "group by" clause

* Some updates can not be translated at all.

* Materializing a view: create a physical table for the view

→ if query relations are changed, materialized view becomes out of date. → must maintain the view!

materialized view maitenance

↙ Periodically ↓ lazily ↘ immediately

* Transactions: sequence of query and/or update statements
begin implicitly

→ atomic

Transaction Ending → commit work → permanent change → a new transaction starts afterwards

↳ rollback work → undo all the performed updates

by default most databases commit each sql statement as a transaction.

* Integrity Constraints: ensure authorized changes to the database
→ guard against accidental damage to the database that cause inconsistency

for a single relation:
- not null
- primary key
- unique → unique($A_1, \dots, A_n$) defines a candidate key (can be null)
- check(P) → check(A in (...))

* Referential Integrity ⟶ foreign key (...) references (...)
on delete cascade
on update cascade
set null } alternative
set default }

check assertion <assertion-name> check <predicate>;
↳ not supported by almost any database

* Built-in datatypes in SQL
date ⟶ '2005-7-27' time ⟶ '09:00:30.75' + ⟹ time stamp

interval ⟶ period of time
↳ arithmetic operation on date/time/timestamp

* Syntax Time!-*

Default values: A, D, default v_i;

create index ___ on r(A)

* Large Object types ⟶ for storing large files (Photo, CAD, etc)
↳ types ⟶ blob ⟶ binary large object
↳ uninterpreted binary data
↳ interpretation done outside of the DB
⟶ clob ⟶ character large object

exp: book clob(10KB)
music blob(10 MB)

the result of a query returning a large object is a locater
⟶ can be used to fetch data in pieces
↳ similar to 'read' function call in O.S.

* User defined Datatypes ⟶ distinct
↳ structured: nested complex data

create type dollar as numeric(12,2) final;

⟹ results to strong type checking!
type casting: cast(A to numeric(12,2))

numeric ↙
obviously!