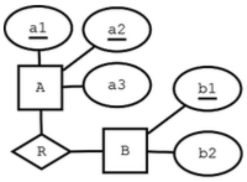
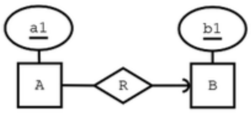
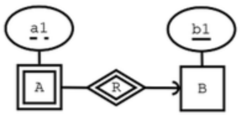
E-R diagrams and database schemas



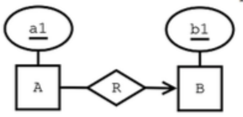
```
A(a1,a2,a3)
B(b1,b2)
R(a1,a2,b)
    (a1,a2) -> A.(a1,a2)
    b       -> B.b1
```



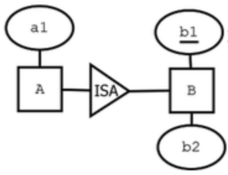
```
A(a1,b)
    b -> B.b1
B(b1)
```



```
A(a1,b)
    b -> B.b1
B(b1)
```



```
ER-Approach:
 A(a1)           Null Approach:
 B(b1)            A(a1,b (or Null))
 R(a,b)             b -> B.b1
   a -> A.a1      B(b1)
   b -> B.b1
```



```
                  Null-Approach:
ER-Approach:       B(b1,b2,a1(or Null))
 B(b1,b2)
 A(b1,a1)         OO-Approach:
  b1 -> B.b1       A(b1,b2,a1)
                   B(b1,b2)
```

Functional dependencies

Definition (tuple, attribute, value). A **tuple** has the form

$$\{A_1 = v_1, \ldots, A_n = v_n\}$$

where $A_1, \ldots, A_n$ are **attributes** and $v_1, \ldots, v_n$ are their **values**.

Definition (signature, relation). The **signature** of a tuple, S, is the set of all its attributes, $\{A_1, \ldots, A_n\}$. A **relation** R of signature S is a set of tuples with signature S. But we will sometimes also say "relation" when we mean the signature itself.

Definition (projection). If t is a tuple of a relation with signature S, the **projection** $t.A_i$ computes to the value v_i.

Definition (simultaneous projection). If X is a set of attributes $\{B_1, \ldots, B_m\} \subseteq S$ and t is a tuple of a relation with signature S, we can form a simultaneous projection,

$$t.X = \{B_1 = t.B_1, \ldots, B_m = t.B_m\}$$

Definition (functional dependency, FD). Assume X is a set of attributes and A an attribute, all belonging to a signature S. Then A is **functionally dependent** on X in the relation R, written $X \rightarrow A$, if

- for all tuples t,u in R, if $t.X = u.X$ then $t.A = u.A$.

If Y is a set of attributes, we write $X \rightarrow Y$ to mean that $X \rightarrow A$ for every A in Y.

Definition (multivalued dependency, MVD). Let X,Y,Z be disjoint subsets of a signature S such that $S = X \cup Y \cup Z$. Then Y has a **multivalued dependency** on X in R, written $X \twoheadrightarrow Y$, if

- for all tuples t,u in R, if $t.X = u.X$ then there is a tuple v in R such that
  - $v.X = t.X$
  - $v.Y = t.Y$
  - $v.Z = u.Z$

Definition. An attribute A **follows** from a set of attributes Y, if there is an FD $X \to A$ such that $X \subseteq Y$.
Definition (closure of a set of attributes under FDs). The **closure** of a set of attributes $X \subseteq S$ under a set FD of functional dependencies, denoted $X+$, is the set of those attributes that follow from X.
Definition (trivial functional dependencies). An FD $X \to A$ is **trivial**, if $A \in X$.
Definition (superkey, key). A set of attributes $X \subseteq S$ is a **superkey** of S, if $S \subseteq X+$.
A set of attributes $X \subseteq S$ is a **key** of S if

- X is a superkey of S
- no proper subset of X is a superkey of S

Definition (Boyce-Codd Normal Form, BCNF violation). A functional dependency $X \to A$ **violates BCNF** if

- X is not a superkey
- the dependency is not trivial

A relation **is in Boyce-Codd Normal Form** (BCNF) if it has no BCNF violations.
Definition (prime). An attribute A is prime if it belongs to some key.
Definition (Third Normal Form, 3NF violation). A functional dependency $X \to A$ **violates 3NF** if

- X is not a superkey
- the dependency is not trivial
- A is not prime

Definition (trivial multivalued dependency). A multivalued dependency $X \twoheadrightarrow A$ is trivial if $Y \subseteq X$ or $X \cup Y = S$.
Definition (Fourth Normal Form, 4NF violation). A multivalued dependency $X \twoheadrightarrow A$ **violates 4NF** if

- X is not a superkey
- the MVD is not trivial.

Algorithm (BCNF decomposition). Consider a relation R with signature S and a set F of functional dependencies. R can be brought to BCNF by the following steps:

1. If R has no BCNF violations, return R
2. If R has a violating functional dependency $X \to A$, decompose R to two relations
   - R_1 with signature $X \cup \{A\}$
   - R_2 with signature $S - \{A\}$
3. Apply the above steps to R_1 and R_2 with functional dependencies projected to the attributes contained in each of them.

Algorithm (4NF decomposition). Consider a relation R with signature S and a set M of multivalued dependencies. R can be brought to 4NF by the following steps:

1. If R has no 4NF violations, return R
2. If R has a violating multivalued dependency $X \twoheadrightarrow Y$, decompose R to two relations
   - R_1 with signature $X \cup \{Y\}$
   - R_2 with signature $S - Y$
3. Apply the above steps to *R1* and *R2*

Concept (minimal basis of a set of functional dependencies; not a rigorous definition). A **minimal basis** of a set F of functional dependencies is a set F- that implies all dependencies in F. It is obtained by first weakening the left hand sides and then dropping out dependencies that follow by transitivity. Weakening an LHS in $X \to A$ means finding a minimal subset of X such that A can still be derived from F-.
Algorithm (3NF decomposition). Consider a relation R with a set F of functional dependencies.

1. If R has no 3NF violations, return R.
2. If R has 3NF violations,
   - compute a minimal basis of F- of F
   - group F- by the left hand side, i.e. so that all depenencies $X \to A$ are grouped together
   - for each of the groups, return the schema $XA_1 \ldots A_n$ with the common LHS and all the RHSs
   - if one of the schemas contains a key of R, these groups are enough; otherwise, add a schema containing just some key

Relational algebra

relation ::=

relname	**name of relation (can be used alone)**
| $\sigma_{\text{condition}}$ relation	**selection (sigma)** `WHERE`
| $\pi_{\text{projection+}}$ relation	**projection (pi)** `SELECT`
| $\rho_{\text{relname (attribute+)?}}$ relation	**renaming (rho)** `AS`
| $\gamma_{\text{attribute*,aggregationexp+}}$ relation	**grouping (gamma)** `GROUP BY, HAVING`
| $\tau_{\text{expression+}}$ relation	**sorting (tau)** `ORDER BY`
| δ relation	**removing duplicates (delta)** `DISTINCT`
| relation $\times$ relation	**cartesian product** `FROM, CROSS JOIN`
| relation $\cup$ relation	**union** `UNION`
| relation $\cap$ relation	**intersection** `INTERSECT`
| relation $-$ relation	**difference** `EXCEPT`
| relation $\bowtie$ relation	`NATURAL JOIN`
| relation $\bowtie_{\text{condition}}$ relation	**theta join** `JOIN ON`
| relation $\bowtie_{\text{attribute+}}$ relation	`INNER JOIN`
| relation $\bowtie^{o}_{\text{attribute+}}$ relation	`FULL OUTER JOIN`
| relation $\bowtie^{oL}_{\text{attribute+}}$ relation	`LEFT OUTER JOIN`
| relation $\bowtie^{oR}_{\text{attribute+}}$ relation	`RIGHT OUTER JOIN`

projection ::=

expression	**expression, can be just an attribute**
| expression → attribute	**rename projected expression** `AS`

aggregationexp ::=

aggregation(*|attribute)	**without renaming**
| aggregation(*|attribute) → attribute	**with renaming** `AS`

expression, condition, aggregation, attribute ::=

as in SQL, but excluding subqueries

SQL

```
statement ::=
      CREATE TABLE tablename (
    * attribute type inlineconstraint*
    * [CONSTRAINT name]? constraint
    ) ;
  |
      DROP TABLE tablename ;
  |
      INSERT INTO tablename tableplaces? values ;
  |
      DELETE FROM tablename
    ? WHERE condition  ;
  |
      UPDATE tablename
      SET setting+
    ? WHERE condition ;
  |
      query ;
  |
      CREATE VIEW viewname
      AS ( query ) ;
  |
      ALTER TABLE tablename
   +  alteration ;
  |
      COPY tablename FROM filepath ;
        ## postgresql-specific, tab-separated

query ::=
      SELECT DISTINCT? columns
    ? FROM table+
    ? WHERE condition
    ? GROUP BY attribute+
    ? HAVING condition
    ? ORDER BY attributeorder+
  |
      query setoperation query
  |
      query ORDER BY attributeorder+
        ## no previous ORDER in query
  |
      WITH localdef+ query

table ::=
      tablename
  |   table AS? tablename  ## only one iteration allowed
  |   ( query ) AS? tablename
  |   table jointype JOIN table ON condition
  |   table jointype JOIN table USING (attribute+)
  |   table NATURAL jointype JOIN table

condition ::=
      expression comparison compared
  |   expression NOT? BETWEEN expression AND expression
  |   condition boolean condition
  |   expression NOT? LIKE 'pattern*'
  |   expression NOT? IN values
  |   NOT? EXISTS ( query )
  |   expression IS NOT? NULL
  |   NOT ( condition )
```

```
type ::=
     CHAR ( integer ) | VARCHAR ( integer ) | TEXT
   | INT | FLOAT

inlineconstraint ::=     ## not separated by commas!
     PRIMARY KEY
   | REFERENCES tablename ( attribute ) policy*
   | UNIQUE | NOT NULL
   | CHECK ( condition )
   | DEFAULT value

constraint ::=
     PRIMARY KEY ( attribute+ )
   | FOREIGN KEY ( attribute+ )
       REFERENCES tablename ( attribute+ ) policy*
   | UNIQUE ( attribute+ ) | NOT NULL ( attribute )
   | CHECK ( condition )

policy ::=
     ON DELETE|UPDATE CASCADE|SET NULL
              ## alternatives: CASCADE and SET NULL

tableplaces ::=
     ( attribute+ )

values ::=
     VALUES ( value+ )  ## VALUES only in INSERT
   | ( query )

setting ::=
     attribute = value

alteration ::=
     ADD COLUMN attribute type inlineconstraint*
   | DROP COLUMN attribute

localdef ::=
     WITH tablename AS ( query )

columns ::=
     *         ## literal asterisk
   | column+

column ::=
     expression
   | expression AS name

attributeorder ::=
     attribute (DESC|ASC)?

setoperation ::=
     UNION | INTERSECT | EXCEPT

jointype ::=
     LEFT|RIGHT|FULL OUTER?
   | INNER?

comparison ::=
     = | < | > | <> | <= | >=
expression ::=
      attribute
  |   tablename.attribute
  |   value
  |   expression operation expression
  |   aggregation ( DISTINCT? *|attribute)
  |   ( query )

value ::=
      integer | float | string ## string in single quotes
  |   value operation value
  |   NULL

boolean ::=
      AND | OR

## triggers

functiondefinition ::=
  CREATE FUNCTION functionname() RETURNS TRIGGER AS $$
  BEGIN
*   triggerstatement
  END
  $$ LANGUAGE 'plpgsql'
  ;

triggerdefinition ::=
  CREATE TRIGGER triggernane
    whentriggered
    FOR EACH ROW|STATEMENT
  ? WHEN ( condition )
    EXECUTE PROCEDURE functionname
    ;

whentriggered ::=
    BEFORE|AFTER events ON tablename
  | INSTEAD OF   events ON viewname

events ::= event | event OR events
event  ::= INSERT | UPDATE | DELETE

triggerstatement ::=
    IF ( condition ) THEN statement+ elsif* END IF ;
  | RAISE EXCEPTION 'message' ;
  | statement ;  ## INSERT, UPDATE or DELETE
  | RETURN NEW|OLD|NULL ;

elsif ::= ELSIF ( condition ) THEN statement+
```

```
compared ::=
     expression
   | ALL|ANY values

operation ::=
     "+" | "-" | "*" | "/" | "%"
   | "||"

pattern ::=
    % | _ | character  ## match any string/char
  | [ character* ]
  | [^ character* ]

aggregation ::=
    MAX | MIN | AVG | COUNT | SUM


## privileges

statement ::=
   GRANT  privilege+ ON object TO user+ grantoption?
 | REVOKE privilege+ ON object FROM user+ CASCADE?
 | REVOKE GRANT OPTION FOR privilege
     ON object FROM user+ CASCADE?
 | GRANT rolename TO username adminoption?

privilege ::=
   SELECT | INSERT | DELETE | UPDATE | REFERENCES
 | ALL PRIVILEGES ## | ...

object ::=
   tablename (attribute+)+ | viewname (attribute+)+
 | trigger ## | ...

user ::= username | rolename | PUBLIC

grantoption ::= WITH GRANT OPTION

adminoption ::= WITH ADMIN OPTION

## transactions

statement ::=
  START TRANSACTION mode* | BEGIN | COMMIT | ROLLBACK

mode ::=
   ISOLATION LEVEL level
 | READ WRITE | READ ONLY | NOT? DEFERRABLE

level ::=
   SERIALIZABLE | REPEATABLE READ | READ COMMITTED
 | READ UNCOMMITTED

## indexes

statement ::=
   CREATE INDEX indexname ON tablename (attribute+)?
```

XML

```
document ::= header? dtd? element

header ::= "<?xml version=1.0 encoding=utf-8 standalone=no?>"
  ## standalone=no if with DTD

dtd ::= <! DOCTYPE ident [ definition* ]>

definition ::=
    <! ELEMENT ident rhs >
  | <! ATTLIST ident attribute* >

rhs ::=
   EMPTY | #PCDATA | ident
  | rhs"*" | rhs"+" | rhs"?"
  | rhs , rhs
  | rhs "|" rhs

attribute ::= ident type #REQUIRED|#IMPLIED

type ::= CDATA | ID | IDREF

element   ::= starttag element* endtag | emptytag
```

```
starttag  ::= < ident attr* >
endtag    ::= </ ident >
emptytag  ::= < ident attr* />

attr ::= ident = string ## string in double quotes

## XPath

path ::=
    axis item cond? path?
  | path "|" path

axis ::= / | //

item ::= "@"? (ident*) | ident :: ident

cond ::= [ exp op exp ] | [ integer ]

exp  ::=  "@"? ident | integer | string

op   ::= = | != | < | > | <= | >=
```

Grammar conventions

- CAPITAL words are SQL or XML keywords, to take literally
- small character words are names of syntactic categories, defined each in their own rules
- | separates alternatives
- + means one or more, separated by commas in SQL, by white space in XML
- * means zero or more, separated by commas in SQL, by white space in XML
- ? means zero or one
- in the beginning of a line, + * ? operate on the whole line; elsewhere, they operate on the word just before
- ## start comments, which explain unexpected notation or behaviour
- text in double quotes means literal code, e.g. "*" means the operator *
- other symbols, e.g. parentheses, also mean literal code (quotes are used only in some cases, to separate code from grammar notation)
- parentheses can be added to disambiguate the scopes of operators, in both SQL and XML