

Introduction to Information Systems

Summary by Martin Rubli

This is version 1.5 from 2005-07-06.

Table of Contents

1 SQL	1
1.1 Basics	1
1.2 Data types	1
1.3 Query syntax	1
1.4 Joins	2
1.5 GROUP BY clause	2
1.6 HAVING clause (filtering the query results)	2
1.7 Uniting queries	3
1.8 Intersection and difference	3
1.9 NULL values	3
1.10 Indexes	3
1.11 Views	4
1.12 Data warehouses	4
2 Relational data model	5
2.1 Notation	5
2.2 Functional dependencies	5
2.3 Inference rules for functional dependencies	5
2.4 Keys	5
2.5 Constraints	6
2.6 Closure	7
2.7 Normal forms	7
2.8 Relation decomposition	8
2.9 Data anomalies	9
2.10 Relational algebra	10
3 Entity/Relationship modeling	11
3.1 Basics	11
3.2 Entity/Relationship (E/R) diagrams	11
4 Concurrency	12
4.1 Transactions	12
4.2 Concurrency problems	12
4.3 Schedules and Serializability	14
4.4 Locking	15
4.5 Transactions in SQL	16
4.6 Recovery	17
5 JDBC	20
5.1 Basics	20
5.2 Examples	20
6 XML	21
6.1 Terminology	21
6.2 XML Namespaces	22
6.3 Well-formedness	22
6.4 Document type definition (DTD/DocType)	23
7 XPath	25
7.1 Location paths	25
8 XQuery	27
8.1 FLWOR („Flower“) expressions	27
8.2 Conditional expressions	28
8.3 Quantified expressions	28
8.4 Grouping and functions	29
8.5 Nesting	29
8.6 Joins	30

1 SQL

1.1 Basics

SQL: Structured Query Language

DDL: Data Definition Language (CREATE, ALTER, DROP, ...)

DML: Data Manipulation Language (SELECT)

Atomic type/Data type: one of the data types available for the attributes (see next section)

Attribute/Column: field in a row. Has a data type and a unique name.

Record/Tuple/Row/Entity: Unordered set of attributes

Table: Set of records. Has a unique name.

Schema: Table name and its attributes. Representation: `TableName (Attribute1, Attribute2, ...)`

Key: Attribute whose values are unique. Representation: `TableName (KeyAttribute, ...)`

1.2 Data types

Different implementations of SQL support many different data types. These are a few of the most frequently used:

INTEGER	32-bit integer value	CHAR (<i>n</i>)	fixed-length string, <i>n</i> characters, automatic padding with blanks
SMALLINT	16-bit integer value		
BIGINT	64-bit integer value	VARCHAR (<i>n</i>)	variable-length string, max <i>n</i> chars
NUMERIC (<i>i</i> , <i>j</i>)	floating point: <i>i</i> = precision, <i>j</i> = scale	BYTEA	byte array (binary data)
FLOAT	real number		
DATE	date (yyyy-mm-dd)		
TIME	time (hh:mm:ss.sss...)		
TIMESTAMP	date/time		

1.3 Query syntax

Syntax was taken from PostgreSQL and slightly simplified.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
      * | expression [ AS output_name ] [, ...]
      [ FROM from_item [, ...] ]
      [ WHERE condition ]
      [ GROUP BY expression [, ...] ]
      [ HAVING condition [, ...] ]
      [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
      [ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
      [ LIMIT { count | ALL } ]
```

where `from_item` can be one of:

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] |
column_definition [, ...] ) ]
function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [,
...]) ]
```

```
INSERT INTO table [ ( column [, ...] ) ]
      { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) | query }
```

```
DELETE FROM table [ WHERE condition ]
```

```

UPDATE table SET column = { expression | DEFAULT } [, ...]
  [ FROM fromlist ]
  [ WHERE condition ]

CREATE TABLE table_name (
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ ... ] ]
  | table_constraint
  | LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ] } [, ... ]
)

ALTER TABLE [ ONLY ] name [ * ]
  ADD [ COLUMN ] column type [ column_constraint [ ... ] ]

ALTER TABLE [ ONLY ] name [ * ]
  DROP [ COLUMN ] column [ RESTRICT | CASCADE ]

ALTER TABLE [ ONLY ] name [ * ]
  ALTER [ COLUMN ] column { SET DEFAULT expression | DROP DEFAULT }

ALTER TABLE [ ONLY ] name [ * ]
  RENAME [ COLUMN ] column TO new_column

ALTER TABLE name
  RENAME TO new_name

CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ] AS query

DROP TABLE name [, ...] [ CASCADE | RESTRICT ]

```

1.4 Joins

A `JOIN` clause combines two `FROM` items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, `JOINS` nest left-to-right. In any case `JOIN` binds more tightly than the commas separating `FROM` items.

`INNER JOIN` produces a simple Cartesian product, the same result as you get from listing the two items at the top level of `FROM`, but restricted by the join condition (if any). This join type is just a notational convenience, since it does nothing that couldn't be done with plain `FROM` and `WHERE`.

`LEFT [OUTER] JOIN` returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns.

Conversely, `RIGHT [OUTER] JOIN` returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a `LEFT JOIN` by switching the left and right inputs.

1.5 GROUP BY clause

`GROUP BY` will condense into a single row all selected rows that share the same values for the grouped expressions. The expression can be an input column name, or the name or ordinal number of an output column, or an arbitrary expression formed from input-column values.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without `GROUP BY`, an aggregate produces a single value computed across all the selected rows). When `GROUP BY` is present, it is not valid for the `SELECT` list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

1.6 HAVING clause (filtering the query results)

`HAVING` eliminates group rows that do not satisfy the condition. `HAVING` is different from `WHERE`: `WHERE` filters individual rows *before* the application of `GROUP BY`, while `HAVING` filters group rows created by `GROUP BY`.

1.7 Uniting queries

The given select statements are `SELECT` statements without `ORDER BY`, `LIMIT`, or `FOR UPDATE` clauses. (`ORDER BY` and `LIMIT` can be attached to a subexpression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the `UNION`, not to its right-hand input expression.)

The `UNION` operator computes the set union of the rows returned by the involved `SELECT` statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two `SELECT` statements that represent the direct operands of the `UNION` must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of `UNION` does not contain any duplicate rows unless the `ALL` option is specified. `ALL` prevents elimination of duplicates.

Multiple `UNION` operators in the same `SELECT` statement are evaluated left to right, unless otherwise indicated by parentheses.

1.8 Intersection and difference

The `INTERSECT` and `EXCEPT` operators are of similar syntax as the `UNION` operator.

The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

The `EXCEPT` operator computes the set of rows that are in the result of the left `SELECT` statement but not in the result of the right one.

1.9 NULL values

Meaning: „value does not exist“, „value exists but is unknown“, „value not applicable“, etc.

Testing for NULL values: `expr IS [NOT] NULL`

We can't use `=` or `<>` operators because SQL sees every NULL value as different from every other NULL value!

In JOINS, rows that have NULL values for the join attributes are not included in the result (except for OUTER JOINS).

1.10 Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

Indexes are created using the following SQL statement:

```
CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [, ...] )
    [ WHERE predicate ]
```

Indexes are usually implemented using search trees or hash tables.

Imagine we have a table called `test1` with many entries and we execute a query of the following form:

```
SELECT content FROM test1 WHERE id = constant;
```

With no advance preparation, the system would have to scan the entire `test1` table, row by row, to find all matching entries. If there are a lot of rows in `test1` and only a few rows (perhaps only zero or one) that would be returned by such a query, then this is clearly an inefficient method. But if the system has been instructed to maintain an index on the `id` column, then it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

When an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are non-essential or do not get used at all should be removed. Note that a query or data manipulation command can use at most one index per table.

1.11 Views

There are two different types of views:

- **Virtual views:** computed only on demand, therefore always up to date, but slow at runtime
- **Materialized views:** Precomputed offline, fast at runtime (comparable to tables), may have stale data

Databases usually employ virtual views. Materialized are used in data warehouses.

1.12 Data warehouses¹

A data warehouse comprises a computing system used to store information regarding an organization's activities in a database. The database design favours reporting on and analysing the data in order to gain strategic information and to facilitate decision making.

Data warehouses may hold large amounts of information, sometimes in smaller logical units called Data marts. Often the schemas of data marts are stored in what are known as „Star Schemas“, or Dimensional Modelling form; however there is no industry standard requiring that the schemas of data marts be in any particular form. There is, in fact, some controversy about the most useful form of data mart schemas.

Conventional database systems use highly normalized data formats to ensure consistency of data and minimal use of space. However this often means that transactions and queries against a fully normalized database perform slowly. Data warehouses often use a more de-normalized (relaxed) format. This speeds up queries, and has the additional benefit that the schema will be more intuitive to non-administrative users as they are exploring it. For example, rather than having a single record in a table contain customer information, that information may be replicated across a whole series of tables.

Source: http://en.wikipedia.org/wiki/Data_warehouse

¹ not part of the course, but still interesting :-)

2 Relational data model

2.1 Notation

Relational schema: RelationName (AttributeName, Attribute2Name, ...)

Database schema: set of relational schemas

```
Relation1 (Attribute11, Attribute12, ...),
Relation2 (Attribute21, Attribute22, ...),
...
```

Functional dependency: $X \rightarrow Y$ or $X_1, X_2, \dots, X_n \rightarrow Y_1, Y_2, \dots, Y_n$

2.2 Functional dependencies

The concept of functional dependencies is the most important one in relational schema design.

Functional dependency/FD/f.d.: constraint between two sets of attributes from the database. If $X \rightarrow Y$, then the values of Y are uniquely determined by the values of X.

More formally: If $X \rightarrow Y$, then $t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$

In an FD $X \rightarrow Y$ the values of X are said to **functionally determine** the values of Y. X is called the **left-hand side** of the FD, Y is called the **right-hand side**.

Example: "ASID \rightarrow Title, Price" but "Category $\not\rightarrow$ Price"²

Test: To verify $X \rightarrow Y$, erase all other columns. The remaining relation must be 1:1.

$X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency doesn't hold anymore. Formal: $(X - \{A \in X\}) \not\rightarrow Y$

$X \rightarrow Y$ is a **partial dependency** if some attribute A can be removed from X and the dependency still holds.

2.3 Inference rules for functional dependencies

Splitting and Combining rule	$X_1, X_2, \dots, X_n \rightarrow Y_1, Y_2, \dots, Y_n$ is equivalent to $X_1, X_2, \dots, X_n \rightarrow Y_1$ $X_1, X_2, \dots, X_n \rightarrow Y_2$... $X_1, X_2, \dots, X_n \rightarrow Y_n$
Trivial rule	$X_1, X_2, \dots, X_n \rightarrow X_i$
Transitive closure rule	If $X_1, X_2, \dots, X_n \rightarrow Y_1, Y_2, \dots, Y_n$ and $Y_1, Y_2, \dots, Y_n \rightarrow Z_1, Z_2, \dots, Z_n$ then: $X_1, X_2, \dots, X_n \rightarrow Z_1, Z_2, \dots, Z_n$
Augmentation	If $X_1, X_2, \dots, X_n \rightarrow Y$ then $X_1, X_2, \dots, X_n, Z_1, Z_2, \dots, Z_m \rightarrow Y$

2.4 Keys

Keys are used to uniquely identify rows (attribute values) in a table (schema).

Superkey: subset of attributes for which no two distinct rows can have the same subset values (default superkey: the entire set of attributes)

Key: minimal superkey: a superkey that, if we remove any one attribute, loses its uniqueness property and therefore would not be a superkey any more

² The symbol „ $\not\rightarrow$ “ denotes „no functional dependency“ in this summary. Apparently the strike through arrow used before didn't print correctly on some systems.

Candidate key: each of the (possibly many) keys a schema can have

Primary key: a candidate key arbitrarily designated (i.e. every one would do) to uniquely identify each table row

Entity: a collection (set) of information that share the same attributes (and therefore attribute types), but have different values

Entity type: a collection of entities that have the same attributes

Entity set: the collection of all entities of a particular entity type in the database at any point in time

Schema: described by an entity type

Strong entity type: entity type that has a key attribute

Weak/Child/Subordinate entity type: an entity type that doesn't have key attributes of its own

Identifying/Owner/Parent/Dominant entity type: the entity type containing the entities to which specific entities from a weak entity type T are related to by some of T's attribute values

Identifying relationship: the relationship type that relates a weak entity type to its owner

Partial key/Discriminator: the set of attributes that can uniquely identify weak entities that are related to the same owner entity

Example: Think of an *Employee* table and a *Children* table. Every employee can have many children, so there is an 1:n relationship between the two tables. Now, the child records in the *Children* table don't have keys on their own, but can only be uniquely identified after determining the particular employee to which each child belongs. The *Children* table is the weak entity type, while the *Employee* table is the owner entity type.

2.5 Constraints

Relationships in relational databases create constraints on the records in the corresponding tables. These constraints are called **referential integrity (RI) constraints**.

Domain: a set of atomic values (e.g. for INTEGER this might be { ..., -1, 0, 1, ... }, for CHAR(1) { 'a', 'b', 'c', ... })

A set of attributes FK in relation R1 is a **foreign key** of R1 that references relation R2 if it satisfies the following two rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R2. (The attributes FK are said to **reference** or **refer to** the relation R2.)
2. A value of FK in a row t1 of R1 either occurs as a value of PK for some row t2 of R2 or is null. (In the former case we say that the row t1 **references** or **refers to** the row t2. R1 is called the **referencing relation** and R2 is the **referenced relation**).

When updating the referencing or referenced relations, the referential integrity constraints must be enforced. SQL has three policies for maintaining RI:

- **Reject:** The modifications causing constraint violations are rejected.
- **Cascade:** When rows in the referenced table are deleted, all corresponding rows in the referencing table are automatically deleted, too („cascade delete“). When the value of a primary key value is modified, all values in the corresponding foreign key fields are also modified („cascade update“).
- **Nullify:** When rows in the referenced table are deleted or primary key values are modified, all corresponding foreign key values are set to NULL.

2.6 Closure

Given the set of functional dependencies F , the set of all attributes that are dependent on X are called **closure X^+ of X under F** .

The following **algorithm** can be used to determine X^+ of X under F :

```

X+ := X;
do
  oldX+ := X+;
  for each functional dependency Y → Z in F do
    if (X+ ⊇ Y) then          // if the left-hand side Y is contained in X+
      X+ := X+ ∪ Z;          // add the right-hand side Z to the closure
until (X+ == oldX+);        // until nothing was added during a complete cycle

```

2.7 Normal forms

Normal forms are used to determine whether a given relational schema satisfies certain criteria. They can be thought of as rules that describe the „goodness“ or „badness“ of a relational schema. Primary goals of the normalization process are **minimal redundancy** and **minimal insertion, deletion, and update anomalies**. The normal form of a relation refers to the highest normal form condition that it meets. So if a given schema satisfies 3NF, it implicitly also satisfies 2NF and 1NF.

It is *not* necessary that every database satisfies all normal forms! Relations may be left in lower normal forms for performance (and simplicity) reasons. The process of storing a join of higher normal form relations as a new base relation is known as **denormalization**. However, relations should usually satisfy 3NF or BCNF.

Two necessary definitions: a **prime attribute** is an attribute that is part of any candidate key. a **nonprime attribute** is therefore an attribute that is not a member of any candidate key.

2.7.1 First Normal Form (1NF)

Formal: Relations are flat

Informal: The 1NF disallows „tables within tables“ or „relations as row attributes“. All attributes must be atomic and cannot be composed of multiple attributes or other complex structures.

Test: Relation should have no nonatomic attributes or nested relations.

Remedy: Form new relations for each nonatomic attribute or nested relation.

Example of a violation:

Departments		
Name	Number	Locations
Research	1	{ London, Berlin, Paris }
Administration	2	{ Lausanne }

The violation lies in the multiplicity of location values in the first row.

2.7.2 Second Normal Form (2NF)

Formal: A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R .

Informal: Every (non-key) attribute must depend on the *entire* primary key.

Test: For relations where the primary key contains multiple attributes, no non-key attribute should be functionally dependent on (only) a part of the primary key.

Remedy: Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.

Example of a violation:

EmployeeProjects(EmployeeID, ProjectID, Hours, EmployeeName, ProjectName, ProjectLocation)

FD1: EmployeeID, ProjectID \rightarrow Hours

FD2: EmployeeID \rightarrow EmployeeName

FD3: ProjectID \rightarrow ProjectName, ProjectLocation

The schema is not 2NF because of the nonprime attribute *EmployeeName* and FD2. FD2 makes *EmployeeName* partially dependent on the primary key (*ProjectID* could be removed from the primary key and *EmployeeName* would still be uniquely dependent of the new primary key).

The same problem is caused by FD3. The *EmployeeID* is not necessary to uniquely determine *ProjectName* and *ProjectLocation*.

2.7.3 Third Normal Form (3NF)

A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there is a set of non-key attributes Z and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

Formal: No nonprime attribute of R is transitively dependent on the primary key.

Informal: Non-key attributes should not be transitively dependent on key-attributes (i.e. functionally dependent on other non-key attributes).

Test: A relation should not have a non-key attribute functionally determined by another non-key attribute (or by a set of non-key attributes).

Remedy: Decompose and set up a relation that includes the non-key attribute(s) that functionally determine(s) another non-key attribute(s).

Example of a violation:

EmployeeDepartments(EmployeeID, EmployeeName, DepartmentID, DepartmentName)

FD1: EmployeeID \rightarrow EmployeeName, DepartmentID

FD2: DepartmentID \rightarrow DepartmentName

DepartmentName is transitively dependent on *EmployeeID* through *DepartmentID*, but *DepartmentID* is neither a key itself nor a subset of the key of *EmployeeDepartments*.

2.7.4 Boyce-Codd Normal Form (BCNF)

The BCNF is stricter than the 3NF – every relation in BCNF is also in 3NF.

A functional dependency $X \rightarrow Y$ is **trivial** if $X \supseteq Y$; otherwise, it is **nontrivial**.

Formal: Whenever a nontrivial functional dependency $X \rightarrow A$ holds in R , then X is a superkey of R .

Informal: Whenever a set of attributes (X) of R determines another attribute (A), it (X) should determine all attributes of R .

Remedy: Find a dependency $X \rightarrow Y$ that violates BCNF (choose Y as large as possible). Decompose the relation R into $R_1(A, \{\text{attributes of } R\} - X - Y)$ and $R_2(X, Y)$. Continue the process for all relations (including the newly created R_1 and R_2) until no violations are left.

2.8 Relation decomposition

Decomposition algorithms decompose a **universal** (meaning that every attribute name is unique) relation schema R into a set of relation schemas $D = \{ R_1, R_2, \dots, R_m \}$. D is called the **decomposition** of R .

Given a set of dependencies F on R , the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i (R_i is a subset of R) is called $\pi_{R_i}(F)$ (**projection of F on R_i**). This means that all left- and right-hand-side attributes of F^+ are in R_i .

A decomposition is called **lossless/dependency-preserving** if no dependencies are lost in the decomposition. Formally, if $(\pi_{R_1}(F) \cup \pi_{R_2}(F) \cup \dots \cup \pi_{R_m}(F))^+ = F^+$

2.9 Data anomalies

For the anomaly examples appearing below, the following table will be used. As visible from the primary key EmployeeID, the table primarily lists employees. However, department information is also stored in every row.

EmployeesDepartments					
<u>EmployeeID</u>	EmployeeName	Birthday	DepartmentID	DepartmentName	DepartmentBossID
SMI04	Smith	1965-01-09	4	Research	ZEL01
WON01	Wong	1955-12-08	4	Research	ZEL01
ZEL01	Zelaya	1968-07-19	7	Administration	NAR02
NAR02	Narayan	1972-04-30	7	Administration	NAR02
WAF01	Wafo	1985-06-20	13	Chaos & Confusion	NAR02

2.9.1 Redundancy

Redundancy: Superfluous repetition of data.

Redundancy has different bad effects on database applications. On the one hand it can significantly increase the storage space the stored information needs. On the other hand it creates difficulties when the database information is updated.

2.9.2 Update anomalies

Update anomalies are classified into insertion anomalies, deletion anomalies, and modification anomalies.

Insertion anomalies: There are two subtypes of insertion anomalies:

- On insertion of a new row into a table, certain fields have to be filled with specific data in order to be consistent with the existing data in that table. (**Example:** If a new employee from department 4 is added to our sample table, we would have to fill the *DepartmentName* field with „Research“ because otherwise, the new row would not be consistent with the first two rows.)
- On insertion of a new row into a table, certain fields have to be filled in with Null because the rows are not atomic and only partial information is available. (**Example:** If a new department „Headquarters“ is created, the employee fields, i.e. *EmployeeID*, *EmployeeName*, and *Birthday*, would have to be set to Null unless we have the employee data for the first „Headquarters“ employee available at the same time. This is a problem especially since the *EmployeeID* is a primary key.)

Deletion anomalies: If a table row contains non-atomic data, deleting a row (in the intention of deleting only some data) could inadvertently delete too much information. (**Example:** If employee „Wafo“ is fired and we were to delete the whole 5th row, we would also implicitly delete the entire „Chaos & Confusion“ department.)

Modification anomalies: If a foreign key field of a single row is modified, it is not enough to modify that field in only one row because other rows that share partial data with the modified row will still have the old information, making the rows contradicting. (**Example:** If the „Research“ department gets a new boss, the *DepartmentBossID* of every employee of that department has to be changed, otherwise the table would be inconsistent, possibly showing different bosses for the same department.)

2.10 Relational algebra

2.10.1 Operations

In the last column a 'C' stands for 'commutative' and an 'A' for 'associative'. Note that both properties don't make sense for all operations.

Symbol	Name	Description	
$R \cup S$	Union	The result includes all tuples that are in R or in S or in R and in S . Duplicates are eliminated. R and S must be union compatible , i.e. have the same number of equally typed attributes. SELECT * FROM R UNION SELECT * FROM S;	CA
$R \cap S$	Intersection	The result includes all tuples that are in both R and S . SELECT * FROM R INTERSECT SELECT * FROM S;	CA
$R - S$	(Set) Difference	The result includes all tuples that are in R but not in S . SELECT * FROM R EXCEPT SELECT * FROM S;	
$\sigma_{\Theta}(R)$	Selection	Selects all tuples that are in R and meet the condition(s) Θ . The conditions can have the following types: =, <, ≤, ≥, >, <> SELECT * FROM R WHERE Θ ;	C
$\Pi_A(R)$	Projection	Selects the attributes A from R and removes resulting duplicates. SELECT DISTINCT A FROM R;	A ³
$R \bowtie S$	Join	Combines related (having equal names) tuples of R and S. SELECT DISTINCT R.*, S.* FROM R, S WHERE R.a = S.a; Tuples whose join attributes are Null do <i>not</i> appear in the result.	CA
$R \Join_{\Theta} S$	Theta join	Combines tuples of R and S explicitly using Θ as the relation (join condition). SELECT DISTINCT ... FROM R JOIN S On Θ ; If Θ is an equality, the join is called an equijoin .	C
$R * S$	Natural join ⁴	Combines related tuples of R and S, but removes the duplicated (joined) columns. SELECT DISTINCT R.*, S.s ₁ , ..., S.s _{k-1} , S.s _{k+1} , ..., S.s _m FROM R, S WHERE R.r _k = S.s _k ;	CA
$R \Join S$	Semijoin	Selects all tuples from R that have equal related tuples in R and S. SELECT R.* FROM R NATURAL JOIN S;	CA
$R \times S$	Cartesian product, Cross/Product join	Combines each tuple of R with each tuple of S. SELECT * FROM R, S;	CA
$\rho_{S(A_1, \dots)}(R)$	Renaming	Renames the attributes of the relation R to A, ... and (if a new relation name is given) the result to S. CREATE VIEW S [(A1 [, ...])] AS SELECT * FROM R ⁵	

3 If projections are cascaded as in $\Pi_{\langle list1 \rangle}(\Pi_{\langle list2 \rangle}(R)) = \Pi_{\langle list1 \rangle}(R)$, then $\langle list2 \rangle$ must contain all attributes in $\langle list1 \rangle$, otherwise the left-hand side is an incorrect expression because $\langle list1 \rangle$ would reference non-existent attributes.

4 Note that in the course notes \bowtie is used as the symbol for the natural join instead. The join operator does not appear specifically.

5 The SQL example only holds if S is given. For an expression like $\rho_{A_1, \dots}(R)$ the AS clause of a SELECT statement would be the corresponding SQL element.

2.10.2 Equivalence expressions

- $R \cap S = R - (R - S)$
- $R \bowtie S = \Pi_{\langle \text{unique attributes} \rangle}(\sigma_{\langle \text{equality of related attributes} \rangle}(R \times S))$
- $R \bowtie S = \sigma_{\theta}(R \times S)$, e.g. equijoin: $R \bowtie_{A=B} S = \sigma_{A=B}(R \times S)$
- $R \sqcap S = \Pi_{\langle \text{attributes in } R \rangle}(R \bowtie S)$

2.10.3 Bags

Bag/Multiset: a set with repeated elements

Relational engines work on bags, not on sets, so all operations need to be defined clearly on bags:

- Union: $\{a, b, b, c\} \cup \{a, b, b, b, e, f, f\} = \{a, a, b, b, b, b, b, c, e, f, f\}$
- Difference: $\{a, b, b, b, c, c\} - \{b, c, c, c, d\} = \{a, b, b\}$
- Selection: preserves the number of occurrences
- Projection, cartesian product, join: do not remove duplicate elements

3 Entity/Relationship modeling

3.1 Basics

E/R models represent the entities of a database and their relationships, usually in a graphic way. They help depicting the database in an implementation independent way, that is often oriented at real-world objects and relationships. Objects are called *entities*, classes are called *entity sets*.

3.2 Entity/Relationship (E/R) diagrams

Entity	Rectangle	1:1 relation	double arrow
Attribute	Ellipse	1:n relation	simple arrow
Key attribute	Ellipse, <u>underlined name</u>	n:n relation	line
Relationship	Trapezoid		

4 Concurrency

4.1 Transactions

Transaction: A unit of interaction with a database management system that is treated in a coherent and reliable way independent of other transactions.

Transactions must be resistant against system failures, e.g. if a transaction cannot be completed because of a power outage, the *entire* transaction will be rolled back, i.e. all of its changes discarded.

Transactions have the **ACID properties**: A = atomicity, C = consistency, I = isolation, D = durability

Atomicity: the entire operation sequence is either executed or not at all

Consistency: the operation sequence transforms the database from one consistent state into another

Isolation: running transactions (i.e. their intermediate states) are transparent to other transactions

Durability: effects of completed transactions are not lost due to hardware or software failures

Transactions work on so-called **elements/data items**: an element can be a record, a number of records (**block**) or an entire relation. The size of such a data item defines the **granularity**.

4.2 Concurrency problems

When multiple transactions are being run concurrently, different problems can occur. These problems can lead to inconsistent data or lost changes.

4.2.1 Lost update problem

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

Example:

T ₁	T ₂
read_item(X); X := X - C;	
	read_item(X); X := X + D; write_item(X); commit;
write_item(X); commit;	

In the above example the changes of transaction T₂ are lost because T₁ overwrites the value X + D with X - C. The situation is comparable to *race conditions* in concurrent programming.

4.2.2 Dirty read (or Temporary update) problem

This problem occurs when a transaction relies on the temporary data of an already running transaction that later does further changes to that data.

Example 1:

T ₁	T ₂
read_item(X); X := X + C; write_item(X);	
	read_item(X); Y := Y + X; write_item(Y); commit;
X := X + D; write_item(X); commit;	

The value of Y will be incorrect because T₂ used an intermediate state of X (i.e. $x + c$) for its calculation. T₁ later makes another change to X, rendering the value of Y useless.

Example 2:

T ₁	T ₂
read_item(X); X := X + C; write_item(X);	
	read_item(X); X := X + D; write_item(X); commit;
rollback;	

The value of X read by T₂ is called **dirty data** because it has been created by a transaction that has not completed yet. T₂ reads a value of X that will be changed back later when T₁ rolls back.

4.2.3 Unrepeatable read problem

This problem occurs when a transaction reads an item twice in a row and receives different values each time.

Example:

T ₁	T ₂
read_item(X);	
	read_item(X); X := X + C; write_item(X); commit;
read_item(X);	

When X is read the first time, the original value of X results, however on the second read attempt, the new value stored by T₂ suddenly appears.

4.2.4 Incorrect summary problem

This problem occurs if one transaction is calculating an aggregate summary function on a number of records that are being updated at the same time by a second transaction.

The problem applies to different aggregate functions as well.

Example:

T₁ (update query)	T₂ (aggregate function)
	<pre>sum := 0; read_item(row1); sum := sum + row1; read_item(row2); sum := sum + row2;</pre>
<pre>read_item(row5); row5 := row5 - C; write_item(row5); commit;</pre>	<pre>read_item(row3); sum := sum + row3; read_item(row4); sum := sum + row4;</pre>
	<pre>read_item(row5); sum := sum + row5; read_item(row6); sum := sum + row6;</pre>
<pre>read_item(row6); row6 := row6 + C; write_item(row6); commit;</pre>	

T₁ deducts C from row 5 and then adds C to row 6, keeping the sum of all rows constant (think of bank accounts for instance). In the mean time, T₂ reads the value of row 5 *after* the change and the value of row 6 *before* the change. The calculated sum therefore turns out to be incorrect.

4.3 Schedules and Serializability

4.3.1 Definitions

Schedule/History: An ordering of the operations from a set of transactions where the operations of each single transaction are in the original order. Note that operations from different transactions may interleave without violating the given constraint.

Complete schedule: A schedule with each of its transactions having either a commit or abort operation at the end.

Serial schedule: A schedule in which each transaction is executed in its entirety, i.e. a schedule with no interleaved transactions. The order of the transaction amongst each other is *not* important.

Serializable schedule: Some schedule with interleaved transactions that produces the same result as *any* serial schedule.

Conflicting operations: Two operations belonging to different transactions that both access the same item X and at least one of the operations being a write operation on X.

Two schedules are said to be **conflict equivalent** if the order of any two conflicting operations is the same in both schedules.

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule.

4.3.2 Testing for serializability

Serializability of a schedule can be checked for using a **precedence or serialization graph**. This is a directed graph that consists of nodes T_i (transactions) and edges e_i (certain operations).

Algorithm:

1. For each transaction T_i in schedule S draw a node labeled T_i .
2. Draw an edge from T_j to T_k if one of the operations in T_j appears before some conflicting operation in T_k .
 - a) *Read-after-write*: For each case where T_k does a `read_item(X)` after T_j does a `write_item(X)` create an edge $T_j \rightarrow T_k$.
 - b) *Write-after-read*: For each case where T_k does a `write_item(X)` after T_j does a `read_item(X)` create an edge $T_j \rightarrow T_k$.
 - c) *Write-after-write*: For each case where T_k does a `write_item(X)` after T_j does a `write_item(X)` create an edge $T_j \rightarrow T_k$.
3. The schedule S is serializable if and only if the precedence graph has no cycles.

There are two approaches to serializability:

- *Optimistic*: Check serializability after every transaction that is executed using the above method. If the check fails, abort the transactions.
- *Pessimistic*: Make sure that no non-serializable schedule can occur while a transaction is executed („locking“).

In reality most concurrency control methods don't actually test for serializability but guarantee serializability by enforcing certain rules.

4.4 Locking

4.4.1 Definitions

Shared/S-/Read lock: A lock where other transactions are allowed to read but not write the locked item.

Exclusive/X-/Write lock: A lock where other transactions are not allowed access to the locked item. The locking transaction has exclusive access to the item.

Overview:

		requested lock	
		S	X
held lock	-	✓	✓
	S	✓	x
	X	x	x

4.4.2 Two-phase locking (2PL)

A transaction follows the **two-phase locking protocol** if *all* locking operations precede the *first* unlock operation (i.e. no lock can follow an unlock). Before accessing an object, a lock must be acquired. Only one lock per object is allowed. At the end of the transaction all locks must be released.

Theorem: 2PL ensures conflict serializability.

Expanding/Growing/First phase: The phase of a transaction during which new locks on items are acquired but none are released.

Shrinking/Second phase: The phase of a transaction during which existing locks are released but no new locks are acquired.

Conservative/static 2PL: A transaction must lock *all* the items it accesses *before* it begins execution. (Once the transaction starts, it is in its shrinking phase.)

There exist more rigid variations of 2PL that guarantee **strict schedules** (transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted):

Strict 2PL: A transaction does not release any of its *exclusive* locks until after it commits or aborts.

Rigorous 2PL: A transaction does not release any of its *exclusive or shared* locks until after it commits or aborts. (The transaction is in its expanding phase until it ends.)

2PL can lead to **deadlocks**; each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set.

Example:

T ₁	T ₂
read_lock(Y); read_item(Y);	
	read_lock(X); read_item(X);
write_lock(X);	
	write_lock(Y);

After the second write_lock T₁ waits for T₂ to release X and T₂ waits for T₁ to release Y. Neither of the transactions can continue.

Deadlock situations can be checked for by analyzing the **wait-for graph**. If it contains cycles, there is a deadlock situation. It can be resolved by aborting one of the blocked transactions.

4.4.3 The phantom problem

Example: A transaction T₁ reads a set of rows from a table based on a WHERE clause condition. Now suppose a transaction T₂ inserts a new row that also satisfies the above WHERE condition into the table. If T₁ is repeated, then it will see a phantom (a row that previously didn't exist).

Definition: If insertion of new rows is not prevented by locks, this can lead to different results if a reading transaction repeats the same operation; new rows can suddenly and unexpectedly turn up.

4.5 Transactions in SQL

The following syntax can be used to set transaction properties in SQL:

```
SET TRANSACTION
  [ ISOLATION LEVEL { REPEATABLE READ | READ [UN]COMMITTED | SERIALIZABLE } ]
  [ READ WRITE | READ ONLY ]
```

4.5.1 Access mode

Specifies the data access mode for the transaction.

READ WRITE: Allows update, insert, delete and create commands to be executed

READ ONLY: Allows data retrieval only

4.5.2 Isolation level

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently.

READ UNCOMMITTED: The current transaction can see uncommitted changes of other transactions.

Problems: Dirty read, Unrepeatable read, Phantom

Implementation: No shared locks

READ COMMITTED:	A statement can only see rows committed before it began. This is the default. Problems: Unrepeatable read, Phantom Implementation: Shared locks can be released anytime; exclusive locks strict 2PL
REPEATABLE READ:	Locks will be placed on all data that is used in a query, and other transactions cannot update the data. Problems: Phantom Implementation: Strict 2PL on all data
SERIALIZABLE:	The current transaction can only see rows committed before first query or data-modification statement was executed in this transaction. Problems: none Implementation: Strict 2PL on all data and indices

4.6 Recovery

As required by the atomicity propriety of transactions, a transaction must either succeed as a whole or fail as a whole. It is therefore not allowed for some operations of a transaction to be applied while other operations of the same transaction are not. This however happens on different occasions, e.g. if a transaction is rolled back before its execution has completed or if a system failure occurs.

4.6.1 The system log

Log: The system log keeps track of all transaction operations that affect the values of database items in an append-only manner.

Log record: A single entry in the log. There are different types of log records:

Notation	Action
[start, T]	Execution of transaction T has started (T is a unique ID).
[write_item, T, X, old_value, new_value]	Transaction T has changed the value of item X from <i>old_value</i> to <i>new_value</i> . Depending on the log mechanism used, the <i>old_value</i> or <i>new_value</i> attributes may be missing.
[read_item, T, X]	Transaction T has read the value of database item X.
[commit, T]	Transaction T has completed successfully and affirms that its effect can be committed to the database.
[abort, T]	Transaction T has been aborted.

4.6.2 Checkpointing

Checkpoints enable the recovery manager to stop replaying the journal early. The DBMS periodically writes a [checkpoint] log record to the log after making sure that no transactions are running at that time (stop accepting new transactions and wait for running transactions to finish).

Problem: The database freezes during checkpointing.

4.6.3 Nonquiescent/fuzzy checkpointing

Nonquiescent⁶/fuzzy checkpointing allows the DBMS to continue executing transactions during checkpointing. Ordinarily the database is blocked during checkpointing.

The implementation of nonquiescent checkpointing depends on the logging method being used.

⁶ From *Merriam-Webster's Unabridged Dictionary*: quiescent: marked by a state of inactivity or repose

4.6.4 Undo logging (Undo/No-redo logging)

Logging rules:

1. If T modifies X, then [write_item, T, X, old_value] must be written to disk before X is output to the disk.
2. If T commits, then [commit, T] must be written to disk only after all changes by T are output to the disk.

Recovery idea:

- Decide for each transaction T whether it is completed or not (e.g. whether there is a commit or abort).
- Undo all modifications by incomplete transactions.

Nonquiescent checkpointing:

Periodically a [start checkpoint, T₁, ..., T_k] record is written with T₁, ..., T_k being all currently active transactions. Normal operation can then continue. As soon as T₁, ..., T_k all have completed, an [end checkpoint] record is written.

Recovery:

Read log from the end until the first [start checkpoint] of a completed checkpoint (i.e. a corresponding [end checkpoint] was also found) is encountered:

- [commit, T]: mark T as completed
- [abort, T]: mark T as completed
- [write_item, T, X, old_value]: if T is not completed, then write X = old_value to disk, else ignore
- [start, T]: ignore

Properties:

- Disk outputs are done early (before transaction commits).
- Undo commands are idempotent
- Disadvantage: Committed transactions must be flushed to the disk before [commit] can be written. This makes this method somewhat inefficient.

4.6.5 Redo logging (Redo/No-undo logging)

Logging rule:

1. If T modifies X, then [write_item, T, X, new_value] and [commit, T] must be written to disk before X is output to the disk.

Recovery idea:

- Decide for each transaction T whether it is completed or not (e.g. whether there is a commit or abort).
- Redo all updates of committed transactions.

Nonquiescent checkpointing:

Periodically a [start checkpoint, T₁, ..., T_k] record is written with T₁, ..., T_k being all currently active transactions. All dirty blocks (i.e. blocks of committed transactions) are flushed to the disk while normal operation continues. As soon as all blocks have been written, an [end checkpoint] record is written.

Recovery:

Look for the last [end checkpoint] and go to its corresponding [start checkpoint, T_1, \dots, T_k]. Go back to the [start] of the oldest transaction of T_1, \dots, T_k . From there on continue forward and redo all write operations for committed transactions (ignoring transactions committed before the [start checkpoint, T_1, \dots, T_k]).

- [commit, T]: mark T as completed (committed)
- [abort, T]: ignore
- [write_item, T, X, new_value]: if T is committed, then write $X = \text{new_value}$ to disk, else ignore
- [start, T]: ignore

Properties:

- Disk outputs are done late (after transaction has committed).
- If [commit, T] is not found, T hasn't written any data to the disk and there is no need to undo.
- Disadvantage: Inflexible on *when* to output to disk; output can only happen after the transaction has completed, so large transactions may require a lot of buffer space.

4.6.6 Undo/Redo logging**Logging rule:**

1. If T modifies X, then [write_item, T, X, old_value, new_value] must be written to the disk before X is output to the disk.

Recovery idea:

- Redo all committed transactions from the beginning to the end.
- Undo all uncommitted transactions from the end to the beginning.

Recovery:

- [commit, T]: mark T as completed
- [abort, T]: mark T as completed
- [write_item, T, X, old_value, new_value]: if T is completed, then write $X = \text{new_value}$ to disk, else write $X = \text{old_value}$ to disk
- [start, T]: ignore

5 JDBC

5.1 Basics

JDBC: Java DataBase Connectivity

JDBC consists of a multi-layered architecture. The **JDBC driver manager** provides the **JDBC API** to the Java application. To access the actual database, the JDBC driver manager uses different drivers for different database backends.

5.2 Examples

JDBC example for PostgreSQL:

```
try {
    Class.forName("org.postgresql.Driver");    // load the driver
} catch (ClassNotFoundException e) {
    return;
}
Connection conn = DriverManager.getConnection("jdbc:postgresql:DB", "user",
"password");    // open a new connection
ResultSet rs = conn.createStatement().executeQuery(sql);
while(!rs.next()) {    // true as long as there are rows
    // do stuff with rs.getInt(n), rs.getString("name"), etc.
}
conn.close();
```

Modification of records with JDBC cursors:

```
rs.updateFloat("price", 10.99f);
if(userIsSure)
    rs.updateRow();
else
    rs.cancelRowUpdates();
```

Executing data modification queries:

```
rs.executeUpdate("DELETE * FROM iis_assistants WHERE clue <= 0");
```

Error handling:

```
rs.last();
try {
    rs.next();
} catch (SQLException e) {
    System.err.println("Couldn't go beyond EOF. Duh!");
}
```

The `ResultSetMetaData` interface can be used to get information about the types and properties of the columns in a `ResultSet` object. The following code fragment creates the `ResultSet` object `rs`, creates the `ResultSetMetaData` object `rsmd`, and uses `rsmd` to find out how many columns `rs` has and whether the first column in `rs` can be used in a `WHERE` clause.

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
ResultSetMetaData rsmd = rs.getMetaData();
int numberOfColumns = rsmd.getColumnCount();
boolean b = rsmd.isSearchable(1);
```

6 XML

6.1 Terminology

Name	Definition/Syntax/Example(s)
Document	A data object is an XML document if it is well-formed. (<i>Colloq</i> : a single root element)
Name	A name is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters. Name ::= (Letter '_' ':') (NameChar)*
Element	Each XML document contains one or more elements , the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its „generic identifier“ (GI), and <i>may</i> have a set of attribute specifications. element ::= EmptyElemTag STag content Etag
Start tag	The beginning of every non-empty XML element is marked by a start-tag . STag ::= '<' Name (S ⁷ Attribute)* S? '>' <termdef id="dt-dog" term="dog">
End tag	The end of every element that begins with a start-tag <i>must</i> be marked by an end-tag containing a name that echoes the element's type as given in the start-tag. ETag ::= '</' Name S? '>' </termdef>
Empty element	An element with no content is said to be empty . The representation of an empty element is either a start-tag immediately followed by an end-tag, or an empty-element tag . An empty-element tag takes a special form: EmptyElemTag ::= '<' Name (S Attribute)* S? '/>' </br>
Attributes	The Name-AttValue pairs are referred to as the attribute specifications of the element, with the Name in each pair referred to as the attribute name and the content of the AttValue (the text between the ' or " delimiters) as the attribute value . Attribute ::= Name Eq AttValue
Content	The text between the start-tag and end-tag is called the element's content .
ID	An ID is a special type of attribute with AttName „id“. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must <i>uniquely</i> identify the elements which bear them.
IDREF (References)	An IDREF is a special type of attribute with AttName „idref“. IDREF values must match the value of some ID attribute.

⁷ S stands for one or multiple whitespace characters, e.g. blanks or tabs

CDATA	<p>CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string <code><![CDATA[</code> and end with the string <code>]]></code>.</p> <pre><![CDATA[<greeting>Hello, world!</greeting>]]></pre> <p>„<greeting>“ and „</greeting>“ are recognized as character data, not markup.</p>
Entity reference	<p>An entity reference refers to the content of a named entity. References to parsed general entities use ampersand (&) and semicolon (;) as delimiters.</p> <pre>&#x3C;</pre> <pre>&quot;</pre>
Comments	<p>Comments may appear anywhere in a document outside other markup. They are not part of the document's character data. The string <code>--</code> (double-hyphen) must not occur within comments.</p> <pre>Comment ::= '<!--' ((Char - '-') ('-' (Char - '-')))* '-->'</pre> <pre><!-- declarations for <head> & <body> --></pre>
Processing instructions	<p>Processing instructions (PIs) allow documents to contain instructions for applications.</p> <pre>PI ::= '<?' PITarget (S (Char* - (Char* '?' Char*)))? '>'</pre> <pre>PITarget ::= Name - (('X' 'x') ('M' 'm') ('L' 'l'))</pre>

XML is so-called **semistructured data**. This means that attributes can be missing or repeated and multiple attributes with different types can exist in different objects. In database language, XML violates the 1NF because it makes nested and heterogeneous collections possible.

6.2 XML Namespaces

An **XML namespace** is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names.

The URI is used as a unique identifier and has no other function. It doesn't even have to exist.

A namespace is declared using a family of reserved attributes. Such an attribute's name must either be „xmlns“ or have „xmlns:“ as a prefix.

Example: Namespace declaration, which associates the namespace prefix „edi“ with the namespace name „http://ecommerce.org/schema“:

```
<x xmlns:edi='http://ecommerce.org/schema'>
  <!-- the "edi" prefix is bound to http://ecommerce.org/schema
        for the "x" element and contents -->
</x>
```

6.3 Well-formedness

A textual object is a **well-formed** XML document if:

1. Taken as a whole, it matches the document production: `document ::= prolog element Misc*`
2. It meets all the well-formedness constraints given in the specification.
3. Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

A few sample well-formedness constraints:

- The Name in an element's end-tag must match the element type in the start-tag.
- An attribute name must not appear more than once in the same start-tag or empty-element tag.
- A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

6.4 Document type definition (DTD/DocType)

The XML document type declaration contains (or points to) constraints on the structure of the document. This grammar is known as a **document type definition**, **DTD**, or **DocType**. The DTD can point to an external entity or can contain the markup declarations directly.

DTD has a dedicated successor called XML Schema (XML Schema Definition, XSD).

6.4.1 Declaration

Syntax:

```
doctypedekl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' intSubset ']' S?)? '>'
intSubset ::= (markupdecl | DeclSep)*
DeclSep ::= PReference | S
markupdecl ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment
```

Example: external DTD

```
<?xml version="1.0"?>
<!DOCTYPE greeting SYSTEM "hello.dtd">
<greeting>Hello, world!</greeting>
```

Example: internal DTD

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

6.4.2 DTD syntax

A DTD mostly consists of ELEMENT and ATTLIST definitions.

Example:

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate?, socialsecuritynumber?)>
<!ELEMENT name (#PCDATA) >
<!ELEMENT birthdate (#PCDATA) >
<!ELEMENT socialsecuritynumber (#PCDATA) >
```

The characters *, + and ? are so-called **quantifiers**. They act on the preceding element:

*	0, 1, or many elements („any number of“)
+	1 or more elements („at least one“)
?	0 or 1 times the element („optional“)

Element content:

The **content specification**, i.e. everything after the element name can contain a combination of the following:

<i>children element</i>	the specified element (like <code>person</code> in the above example)
#PCDATA	parsed character data („text“)
EMPTY	must be empty
ANY	anything (no further specification)

Attribute-list declaration:

Attribute-list declarations define what attributes each element can/must have to comply to the DTD. They consist of a name, a type and a default declaration each. Some of the possible values of each of these tokens are listed in the following table:

Name	a valid attribute name
Type	one of the following attribute types (list incomplete): <ul style="list-style-type: none">• ID (value is a unique id)• IDREF (value is the id of another element)• CDATA (character data)• (enum1 enum2 ...) (exactly one of the elements in the enumeration)
Default declaration	<ul style="list-style-type: none">• value (default value of the attribute)• #REQUIRED (the element must have this attribute)• #IMPLIED (no default value is provided: „optional“)• #FIXED value (the attribute must always have the given value)

7 XPath

The purpose of XPath is to address parts of an XML document and navigate through its hierarchical structure.

The **context node** can be thought of as a node matched by the partial expression up to the most recently parsed position. In the table below the context node is anything matched by what is to the left to the given element.

The following treats XPath, version 1.0. Version 2.0 is still a working draft.

7.1 Location paths

A **location path** is an expression that selects a set of nodes relative to the context node. The result of evaluating such an expression is the node-set containing the matched nodes (the nodes selected by the location path).

XPath has two syntaxes for location paths; a default, straightforward syntax which is rather verbose however, and an abbreviated syntax which allows for common cases to be expressed in a more compact form.

A **location step** selects a set of nodes and has three parts:

- an **axis**: specifies the tree relationship between the nodes selected and the context node
- a **node test**: specifies the node type and expanded-name of the nodes selected
- zero or more **predicates**: arbitrary expressions to further refine the set of nodes selected

Location step syntax:

```
Step ::= AxisSpecifier NodeTest Predicate*
      | AbbreviatedStep
```

```
AxisSpecifier ::= AxisName '::'
              | AbbreviatedAxisSpecifier
```

```
AxisName ::= 'ancestor'           // parents of the context node (recursive)
            | 'ancestor-or-self'  // the context node and its ancestors
            | 'attribute'         // the attributes of the context node
            | 'child'             // immediate children of the context node
            | 'descendant'        // children of the context node (recursive)
            | 'descendant-or-self' // the context node and its descendants
            | 'following'         // all nodes that are after the context node
                                 // in document order, excluding children
            | 'following-sibling' // all following siblings of the context node
            | 'namespace'        // the namespace nodes of the context node
            | 'parent'            // the parent of the context node (if any)
            | 'preceding'         // like 'following' but reversed
            | 'preceding-sibling' // all preceding siblings of the context node
            | 'self'              // just the context node itself
```

```
Predicate ::= '[' Expr ']'
```

```
NodeTest ::= NameTest
          | NodeType '(' ')'
          | 'processing-instruction' '(' Literal ')'
```

```
NodeType ::= 'comment'
           | 'text'
           | 'processing-instruction'
           | 'node'
```

A **relative location path** consists of a sequence of one or more location steps separated by /. Each step in turn selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The set of nodes identified by the composition of the steps is the resulting set.

An **absolute location path** consists of / optionally followed by a relative location path. A / by itself selects the root node of the document containing the context node.

The following table shows a few location path examples:

Description	Abbreviated syntax	Default syntax
selects the context node	.	self
selects the parent of the context node	..	parent::node()
selects the <code>para</code> element children of the context node	para	child::para
selects all element children of the context node	*	child::*
selects all text node children of the context node	text()	child::text()
selects all the children of the context node, whatever their node type	node()	child::node()
selects the <code>name</code> attribute of the context node	@name	attribute::name
selects all the attributes of the context node	@*	attribute::*
selects all <code>para</code> elements with a <code>length</code> attribute smaller than 100	para[@length<"100"]	child::para[attribute::length<"100"]
selects all <code>para</code> elements that contain a child element with name "title" and value "Once upon a time"	para[title="Once upon a time"]	child::para[child::title="Once upon a time"]
selects the first <code>para</code> child of the context node	para[1]	child::para[position()=1]
selects the last <code>para</code> child of the context node	para[last()]	child::para[position()=last()]
selects all <code>para</code> grandchildren of the context node	*/para	child::*/child::para
selects the second <code>section</code> of the fifth <code>chapter</code> of the <code>doc</code>	/doc/chapter[5]/section[2]	/child::doc/child::chapter[position()=5]/child::section[position()=2]
selects the <code>para</code> element descendants of the <code>chapter</code> element children of the context node	chapter//para	child::chapter/descendant::para
selects all the <code>para</code> descendants of the document root (i.e. all <code>para</code> elements in the same document as the context node)	//para	/descendant::para
selects the <code>para</code> element descendants of the context node	./para	descendant::para
selects the context node if it is a <code>para</code> element, and otherwise selects nothing		self::para

8 XQuery

XPath is a powerful method to extract sets of XML documents. However, it has its limitations:

- no join queries possible
- read-only (no changes possible)
- no quantifiers available (ALL, DISTINCT, TOP, ANY, SOME, ...)
- no aggregate functions available (Sum, Max, Avg, ...)

XQuery is designed as a subset of XPath 2.0, i.e. any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages.

The following treats XQuery, version 1.0.

8.1 FLWOR („Flower“) expressions

„FLWOR“ stands for the following five keywords:

- for
- let
- where
- order by
- return

They allow iteration and binding of variables to results. Let's start with an example combining all five clauses:

```
for $d in fn:doc("depts.xml")//deptno
let $e := fn:doc("emps.xml")//emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
  <big-dept>
    {
      $d,
      <headcount>{fn:count($e)}</headcount>,
      <avgsal>{fn:avg($e/salary)}</avgsal>
    }
  </big-dept>
```

For clause: evaluates an expression and iterates over the items in the result set binding one or more variables to the elements.

Let clause: binds one or more variables to the results of an expression, but does not iterate like the 'for' clause does. 'let' returns a single list of all result elements, whereas 'for' returns a single element per variable in each iteration.

Example: for vs. let

Query using 'let':

```
let $s := (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

Output:

```
<out>
  <one/>
  <two/>
  <three/>
</out>
```

The 'return' command was only executed once returning the entire list.

Query using 'for':

```
for $s in (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

Output:

```
<out>
  <one/>
</out>
<out>
  <two/>
</out>
<out>
  <three/>
</out>
```

Here the 'return' command was executed once for each element of the list.

Where clause: filters the results generated by the 'for' and 'let' clauses.

Return clause: adds elements to the so-called *tuple stream*, the „collection of results“.

Order By clause: determines the order in which result tuples are output by the 'return' clauses, i.e. orders the tuple stream.

8.2 Conditional expressions

XQuery supports conditional expressions with the 'if ... then ... else' construct:

```
IfExpr ::= "if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
```

Example: comparison

```
if ($widget1/unit-cost < $widget2/unit-cost)
then $widget1
else $widget2
```

Example: test for existence of an attribute

```
if ($part/@discounted)
then $part/wholesale
else $part/retail
```

The else part is mandatory, so it can't be left out. Instead, it is possible to return an empty value:

```
if ($widget1/unit-cost < $widget2/unit-cost)
then $widget1
else ()
```

8.3 Quantified expressions

Quantified expressions test a number of elements against given expressions and always return true or false in the end.

```
QuantifiedExpr ::= (("some" "$") | ("every" "$")) VarName TypeDeclaration?
  "in" ExprSingle ("," "$" VarName TypeDeclaration? "in" ExprSingle)*
  "satisfies" ExprSingle
```

If the quantifier is *some*, the quantified expression is true if at least one evaluation of the test expression has yields true; otherwise the quantified expression is false.

If the quantifier is *every*, the quantified expression is true if every evaluation of the test expression has the value true; otherwise the quantified expression is false. If the 'in' clause generates zero tuples, the value is true.

Examples:

- `every $part in //part satisfies $part/@discounted`
true if every part element has a discounted attribute (regardless of their values)

- `some $emp in //employee satisfies ($emp/bonus > 0.25 * $emp/salary)`
true if at least one employee element satisfies the given comparison expression
- `some $x in (1, 2, 3), $y in (2, 3, 4) satisfies $x + $y = 4`
yields true
- `every $x in (1, 2, 3), $y in (2, 3, 4) satisfies $x + $y = 4`
yields false

8.4 Grouping and functions

In XQuery braces `{}` are used to force evaluation of the enclosed expression. If variable names are used without enclosing them in braces, they will not be interpolated.

The following example demonstrates evaluation forcing and also introduces the use of different functions. The query finds the part number and average price for parts that have at least 3 suppliers:

```
for $pn in fn:distinct-values(fn:doc("catalog.xml")//partno)
let $i := fn:doc("catalog.xml")//item[partno = $pn]
where fn:count($i) >= 3
order by $pn
return
  <well-supplied-item>
    <partno> {$pn} </partno>
    <avgprice> {fn:avg($i/price)} </avgprice>
  </well-supplied-item>
```

First a few words on the used functions:

- ***fn:distinct-values*** eliminates duplicate elements, in the above example duplicate part numbers.
- ***fn:count*** gives the number of element in a set
- ***fn:avg*** calculates the average of a number of values
- ***fn:doc*** returns the document node of an XML document given by the URI

There also exists a ***fn:sum*** function which sums up the values in a set.

8.5 Nesting

XQuery is a functional language, which implies that expressions can be nested with full generality.

Example: subquery

```
<authlist>
{
  for $a in fn:distinct-values($books)//author
  order by $a
  return
    <author>
      <name>
        { $a/text() }
      </name>
      <books>
        {
          for $b in $books//book[author = $a]
          order by $b/title
          return $b/title
        }
      </books>
    </author>
}
</authlist>
```

8.6 Joins

Join queries are accomplished by using more than one variables in a 'for' clause.

Example: join

```
for $b in document("big.xml")/big,
    $a in $b/book[publisher/text()='Addison Wesley']/author
return
  <result>
  {
    $a,
    for $t in $b/book[author/text()=$a/text()]/title
    return $t
  }
</result>
```

The above example gives a list of all authors under contract with Addison Wesley with the titles of the books they wrote.

References

- Introduction to Information Systems, lecture notes; K. Aberer, 2004
- Fundamentals of Database Systems, 3rd edition; R. Elmasri, S. Navathe, 2000
- PostgreSQL 7.4.2 Documentation; The PostgreSQL Global Development Group, 2003
- Extensible Markup Language (XML) 1.0 (Third Edition); W3C, 2004
- XML Path Language (XPath) 1.0; W3C, 1999
- XQuery 1.0: An XML Query Language; W3C, 2003
- Wikipedia: The Free Encyclopedia; <http://en.wikipedia.org/wiki/>

Disclaimer

The author cannot be held responsible for your failure in any exam or life in general. All information here corresponds to the best of the author's knowledge which is poorer than you might think. If you find errors, please contact the author.