SQL

SQL

General Remarks

- SQL: Structured Query Language
- Former name was SEQUEL
- Standardized query language for relational DBMSs: SQL-89, SQL-92, SQL-99
- SQL is a declarative query language

Parts of SQL

- Four major parts:
  - DRL: Data Retrieval Language
  - DML: Data Manipulation Language
  - DDL: Data Definition Language
  - DCL: Data Control Language

- The DRL contains the commands for posing queries
- A simple query consists of the three clauses **select**, **from**, and **where**

```
select list of attributes
from   list of relations
where  predicate;
```

A Simple Example

Students

StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Query: “Give me all information about all students”

```
select *
from   Students;
```

Result

StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Query: “Give me the student number and the name of all students”

```
select StudNr, Name
from   Students;
```

StudNr	Name
24002	Xenokrates
25403	Jonas
26120	Fichte
26830	Aristoxenos
27550	Schopenhauer
28106	Carnap
29120	Theophrastos
29555	Feuerbach

Duplicate Elimination

- In contrast to relational algebra, SQL does not eliminate duplicates
- If duplicate elimination is desired, the keyword **distinct** must be used

Example

```
select Semester
from   Students;
```

Semester
18
12
10
8
6
3
2
2

```
select distinct Semester
from   Students;
```

Semester
18
12
10
8
6
3
2

Query: “Give me all information about students in a semester greater than 10”

```
select *
from   Students
where  Semester > 10;
```

StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12

Predicates

- Predicates in the where clause can be combined logically with: AND, OR, NOT
- The following comparison operators can be used: $=, <, \leq, >, \geq$, between, like

Example for Between

Query: “Give me the names of all students who are in their second to fourth semester”

```
select Name
from   Students
where  Semester between 2 and 4;
```

is equivalent to

```
select Name
from   Students
where  Semester >= 2
and    Semester <= 4;
```

Comparison with a List of Values

Comparisons with multiple values can be done using `in`

```
select *
from   Students
where  Semester in (1, 2, 3, 4);
```

String Comparisons

- String constants must be enclosed in single quotes

  Query: “Give me all information about the student with the name Feuerbach”

```
select *
from   Students
where  Name = 'Feuerbach';
```

Search with Wildcards

Query: “Give me all information about students whose name starts with an S”

```
select *
from   Students
where  Name like 'S%';
```

Possible Wildcards

- _ stands for any single character
- % stands for any string (also of length 0)

Null Values

- SQL has a special value **NULL**
- This value exists for all the different data types and represents unknown, unavailable, or inapplicable values
- NULL is tested as follows:

```
select *
from   Professors
where  Room is null;
```

Null Values(2)

- Null values are propagated in arithmetic expressions: if at least one operand is NULL, the result is NULL as well
- SQL has a three-valued logic: **true**(t), **false**(f), and **unknown**(u):

not	
t	f
u	u
f	t

and	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

or	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

- The result of an SQL query only contains tuples for which the where clause evaluates to true

Multiple Relations

- If multiple relations appear in the from clause, they are combined using a cross product
- Example:

Query: “Output all lectures and professors”

```
select *
from   Lectures, Professors;
```

- Cross products are mostly not meaningful, joins are more interesting
- Join predicates are specified in the where clause:

```
select *
from   Lectures, Professors
where  GivenBy = PersNr;
```

Joins(2)

- Arbitrarily many relation names may appear in the from clause
- If no cross products are desired, all of them should be joined in the where clause
- The different join variants from relational algebra are also available in SQL:

```
select *
from   R1 [cross | inner | natural | left outer | right outer | full outer]
       join R2 [on R1.A = R2.B];
```

- Another problem: name clashes (attributes with the same name in different relations) must be resolved
- Example: join of
  - `Students(StudNr, Name, Semester)`
  - `attend(StudNr, LectureNr)`
  - `Lectures(LectureNr, Title, Credits, GivenBy)`

Qualified Attribute Names

- In this example query we have to specify where StudNr and LectureNr should come from
- To do so, the relation name is written before the attribute name

```
select *
from   Students, attend, Lectures
where  Students.StudNr = attend.StudNr
and    attend.LectureNr = Lectures.LectureNr;
```

Short Form

- To save typing, the relations can also be renamed

```
select *
from   Students S, attend A, Lectures L
where  S.StudNr = A.StudNr
and    A.LectureNr = L.LectureNr;
```

Set Operations

- SQL also offers the usual set operations: union, intersection, and difference
- As in relational algebra, they require the combined relations to have the same schema

Union

Prof1

PersNr	Name
2125	Sokrates
2126	Russel

Prof2

PersNr	Name
2126	Russel
2137	Kant

Query: “Unite both lists”

```
select * from Prof1
union
select * from Prof2;
```

PersNr	Name
2125	Sokrates
2126	Russel
2137	Kant

Duplicate Elimination

- In contrast to **select**, **union** eliminates duplicates automatically
- If duplicates are desired in the result, the **union all** operator must be used

Intersection

Prof1

PersNr	Name
2125	Sokrates
2126	Russel

Prof2

PersNr	Name
2126	Russel
2137	Kant

Query: “Which professors are on both lists”

```
select * from Prof1
intersect
select * from Prof2;
```

PersNr	Name
2126	Russel

Set Difference

Prof1

PersNr	Name
2125	Sokrates
2126	Russel

Prof2

PersNr	Name
2126	Russel
2137	Kant

Query: “Which professors are on the first, but not on the second list?”

```
select * from Prof1
except
select * from Prof2;
```

PersNr	Name
2125	Sokrates

Sorting

- Tuples in a relation are not (automatically) sorted
- The result of a query can be sorted using the **order by** clause
- Sorting can be ascending or descending (the default is ascending)

Example

```
select *
from   Students
order by Semester desc, Name;
```

StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29555	Feuerbach	2
29120	Theophrastos	2

Nested Queries

- Queries can be nested inside other queries, i.e., there can be more than one select clause
- A nested select can appear in the where clause, in the from clause, and even in a select clause itself
- In principle, the “inner” query computes an intermediate result that is used in the “outer” one

Select in Where Clause

- Two different kinds of subqueries: correlated and uncorrelated
- uncorrelated: subquery refers only to its “own” attributes
- correlated: subquery also references attributes of the outer query

Query: “Give me the names of all students who attend lecture 5001”

```
select S.Name
from   Students S
where  S.StudNr in
       (select A.StudNr
        from   attend A
        where  A.LectureNr = 5001);
```

- The subquery is evaluated once; for each tuple of the outer query we check whether its StudNr occurs in the result of the subquery

Query: “Find all professors for whom assistants with different areas work”

```
select distinct P.Name
from   Professors P, Assistants A
where  A.Boss = P.PersNr
and    exists
       (select *
        from   Assistants B
        where  B.Boss = P.PersNr
        and    A.Area <> B.Area);
```

- For each tuple of the outer query, the inner query has different values; the exists predicate is true if the subquery contains at least one tuple

Existential and Universal Quantifiers in SQL

- `exists` and `not exists` correspond to semi- and anti-join
- a negated anti-join can be used as universal quantifier
- Example: students who attended all 4-credit lectures

```
select s.*
from   Students s
where  not exists
       (select *
        from   Lectures l
        where  l.Credits = 4 and not exists
               (select *
                from   attend a
                where  a.LectureNr = l.LectureNr and a.StudNr = s.StudNr));
```

Quantified Comparisons

- `=some`/`=all`/`>some`/`>all` etc. are a generalization of `in`
- Example: students in the highest semester

```
select Name
from   Students
where  Semester >= all
       (select Semester
        from   Students);
```

- Example: students not in the highest semester

```
select Name
from   Students
where  Semester < some
       (select Semester
        from   Students);
```

Other Nested Selects

- When nesting a select in a select clause, one must make sure that only one tuple with one attribute is returned
- When nesting in a from clause, correlated subqueries are often not allowed (depending on the DBMS)

Aggregate Functions

- Attribute values (or entire tuples) can be summarized in different ways
  - Counting: count()
  - Summing up: sum()
  - Computing the average: avg()
  - Finding the maximum: max()
  - Finding the minimum: min()

Example

Students

StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

```
select count(*)
from   Students;
```

1
8

Example(2)

Students

StudNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

```
select count(distinct Semester)
from   Students;
```

1
7

Query: “Give me the student with the highest StudNr”

```
select Name, max(StudNr)
from   Students;
```

- Does not work like this!!!

Min/Max(2)

- Aggregate functions reduce all values of a column to a **single** value
- For the attribute `StudNr` we tell the DBMS that the maximum should be taken
- For the attribute `Name` we give the DBMS no information at all on how all the different names should be reduced to one

Min/Max(3)

- How to do it correctly?
- Using a nested query:

```
select StudNr, Name
from   Students
where  StudNr =
       (select max(StudNr)
        from   Students);
```

- Sometimes one wants to split tuples into different groups and aggregate these groups separately

Query: “For each lecture, count the number of attending students”

```
select LectureNr, count(*) as Number
from   attend
group by LectureNr;
```

Result

attend

StudNr	LectureNr
27550	4052
26120	5001
27550	5001
29120	5001
29555	5001
25403	5022
29555	5022
28106	5041
29120	5041
29120	5049
28106	5052
28106	5216
28106	5259

$\rightarrow$

Result

LectureNr	Number
4052	1
5001	4
5022	2
5041	2
5049	1
5052	1
5216	1
5259	1

Grouping(2)

- All attributes that do not appear in the group by clause may only appear aggregated in the select clause
- E.g., the following query is not correct (for the same reason as the first max query):

```
select GivenBy, Title, count(*) as Number
from   Lectures
group by GivenBy;
```

Having

- The where clause is evaluated before grouping
- If further filtering is required after grouping, the **having** clause must be used

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

```
select GivenBy, count(LectureNr) as NumLectures
from   Lectures
group by GivenBy
having count(*) > 3;
```

Views

- Actually belong to the DDL
- But are often used to make queries easier to read, therefore we discuss them here
- Represent a kind of “virtual relation”
- Show a part of the database

Views(2)

- Advantages
  - Simplify access for certain user groups
  - Can be used to restrict access to the data
- Disadvantages
  - Not all views support update operations

Query: “Find the names of all professors who give lectures with more than the average number of credits and who employ more than three assistants”

- We do not do everything at once, but break it down into smaller, more manageable parts
- These parts are realized using views

Complex Query(2)

- Find all lectures with an above-average number of credits:

```
create view AboveAvgCredits as
select LectureNr, GivenBy
from   Lectures
where  Credits >
       (select avg(Credits)
        from   Lectures);
```

Complex Query(3)

- Find (the PersNr of) all professors with more than three assistants:

```
create view ManyAssistants as
select Boss
from   Assistants
group by Boss
having count(*) > 3;
```

Complex Query(4)

- Now everything is put together (views can be accessed like a regular relation)

```
select Name
from   Professors
where  PersNr in
       (select GivenBy
        from   AboveAvgCredits)
and    PersNr in
       (select Boss
        from   ManyAssistants);
```

Local Views

- Views are globally visible, view names must be unique
- often impractical for formulating queries
- instead: views for a single query only

```
with AboveAvgCredits as (select ...),
     ManyAssistants as (select ...)
select Name
from   Professors
where  PersNr in
       (select GivenBy
        from   AboveAvgCredits)
and    PersNr in
       (select Boss
        from   ManyAssistants);
```

Case Distinctions with CASE

- simple case

```
select case Rank when 'C4' then 'full professor'
                 when 'C3' then 'associate professor'
                 else 'other' end
from   Professors
```

- general case

```
select case when Semester > 10 then 'long-term student'
            when Semester < 4 then 'beginner'
            else 'intermediate' end
from   Students
```

Working with Text

- Concatenation

```
select 'a' || 'b';
```

- Extraction

```
select substring('abcfoo' from
3 for 2);
```

Type Conversion with CAST

- explicit type conversion with `CAST(value AS TYPE)`
- Example

```
select cast(StudNr as text) ||
' ' || Name
from   Students
```

- Caution: error if the conversion is not possible

NULL Handling

NULL values often have to be handled separately

- Comparisons with NULL yield NULL
- Test with `IS NULL`
- explicit comparison with `x IS NOT DISTINCT FROM y`
- Masking: `COALESCE(x, -1)`
- Generation: `NULLIF(x, -1)`

Problem: find all prerequisites of a lecture

```
select Predecessor
from   require, Lectures
where  Successor = LectureNr and
       Title = 'The Vienna Circle'
```

Does not find the indirect prerequisites

We can go one step further

```
with
  step0 as (
    select Predecessor
    from   require, Lectures
    where  Successor = LectureNr and
           Title = 'The Vienna Circle'),
  step1 as (
    select r.Predecessor
    from   step0 s, require r
    where  s.Predecessor = r.Successor)
select * from step0 union all select * from step1
```

We want to do this as often as necessary

Recursion in SQL (3)

WITH RECURSIVE statements can read themselves

```
with recursive rec as (
  select ...                -- base case
  union all
  select ... from rec ...   -- extension
)
select * from rec
```

The extension is evaluated as long as the extension finds tuples

Recursion in SQL (4)

```
with recursive transreq as (
  select Predecessor
  from   require, Lectures
  where  Successor = LectureNr and
         Title = 'The Vienna Circle'
  union all
  select r.Predecessor
  from   transreq s, require r
  where  s.Predecessor = r.Successor)
select * from transreq
```

Corresponds to the query from Recursion (2), generalized to transitive steps.

- DML contains commands to
  - insert data
  - delete data
  - modify data

Inserting Data

- Data is inserted with the **insert** command
- Inserting constant values
  - Specifying all attribute values:

```
insert into Professors
values (2138, 'Kossmann', 'C4', 123);
```

  - Omitting attribute values:

```
insert into Professors(PersNr, Name, Rank)
values (2138, 'Kossmann', 'C4');
```

Inserting Data(2)

- Copying data from other relations

```
insert into Professors(PersNr, Name, Rank)
select PersNr, Name, 'C3'
from   Assistants
where  PersNr = 3002;
```

- Modifications are made with the **update** command

```
update Professors
set    Room = 121
where  PersNr = 2138;
```

Deleting Data

- The **delete** command deletes data

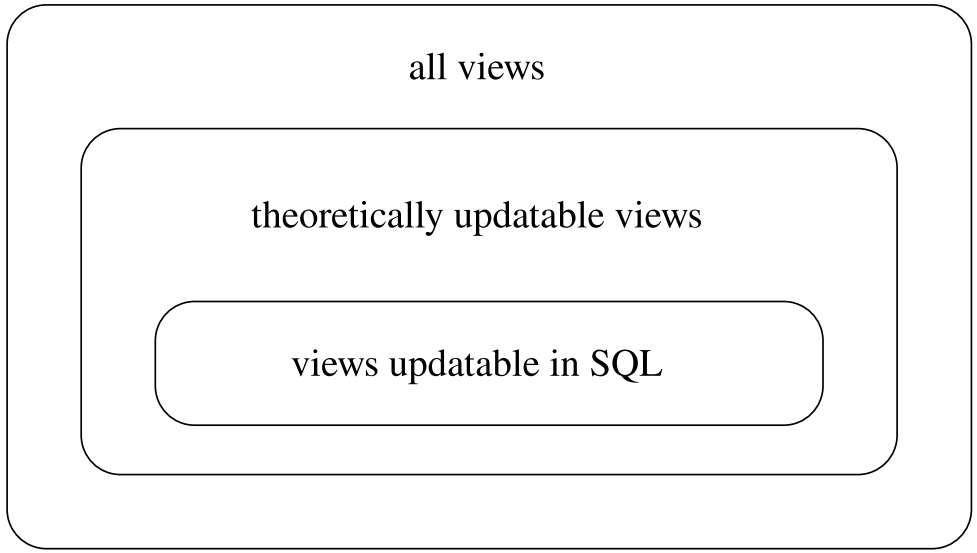
```
delete from Professors
where PersNr = 2138;
```

- Caution! Omitting the where clause deletes the contents of the entire relation

```
delete from Professors;
```

Updatability of Views

- In SQL
  - only one base relation
  - key must be present
  - no aggregate functions, grouping, or duplicate elimination
- In general:



DDL

- The DDL is used to define the schema of a database
- Also contains commands to control access to data

Creating Relations

- Relations are created with the **create table** command

```
create table Professors (
  PersNr integer,
  Name   varchar(30),
  Rank   char(2),
  Room   integer
);
```

Data Types

- *Integer*: `smallint` (2 bytes), `integer` (4 bytes), `bigint` (8 bytes)
- *fixed-point numbers*: `numeric(precision,scale)` (*precision* is the total number of decimal digits, *scale* is the number of digits after the decimal point)
- *floating point numbers*: `float` (4 bytes), `double precision` (8 bytes)
- *strings*: `varchar(n)` (maximum length *n*), `char(n)` (maximum length *n*, padded with spaces, strange semantics), `text` (arbitrary length)
- *other types*: `bytea` (binary data), `timestamp` (8 bytes), `date` (4 bytes), `interval` (16 bytes), `boolean` (1 byte)
- all types can be `NULL`, unless `NOT NULL` is specified
- smallint/bigint/text/bytea are extensions, but often supported (e.g., in PostgreSQL)

Defining Keys

- A primary key can be defined for each relation

```
create table Professors (
  PersNr integer,
  Name   varchar(30),
  Rank   char(2),
  Room   integer,
  primary key (PersNr)
);
```

Integrity Constraints

- One of the tasks of a DBMS is to ensure the consistency of the data
- Semantic integrity constraints describe properties of the modeled mini-world
- Using constraints, the DBMS can check these conditions automatically

- Besides primary keys there is a whole range of other integrity constraints:
  - not null
  - unique
  - check clauses

Not Null Constraint

- Enforces that certain attribute values must be specified when inserting tuples
- Mandatory for keys

```
create table Professors (
  PersNr integer not null primary key,
  Name   varchar(30) not null,
  Rank   char(2) not null,
  Room   integer
);
```

Check Clauses

- **check** clauses can restrict the domain of attributes

```
create table Professors (
  PersNr integer not null primary key,
  Name   varchar(30) not null,
  Rank   char(2) not null,
  Room   integer,
  check (Room > 0 and Room < 99999)
);
```

Check Clauses(2)

- Check clauses can contain complete SQL queries

```
create table attend (
  StudNr integer,
  LectureNr integer,
  check (StudNr not in
         (select P.StudNr
          from   test P
          where  P.LectureNr = attend.LectureNr
          and    P.Grade < 5)),
  primary key (StudNr, LectureNr)
);
```

Referential Integrity

- R and S are two relations with the schemas $\mathcal{R}$ and $\mathcal{S}$, respectively
- κ is the primary key of R
- Then $\alpha \subset \mathcal{S}$ is a foreign key if for all tuples $s \in S$:
  - $s.\alpha$ contains either only null values or only values not equal to null
  - If $s.\alpha$ contains no null values, there exists a tuple $r \in R$ with $s.\alpha = r.\kappa$
- Compliance with these properties is called *referential integrity*

Referential Integrity(2)

- In SQL, referential integrity can be enforced:

```
create table Professors (
  PersNr integer primary key,
  …
);

create table Lectures (
  LectureNr     integer primary key,
  …
  GivenBy integer not null,
  foreign key (GivenBy)
  references Professors(PersNr)
);
```

Referential Integrity(3)

- Changes to key attributes can be propagated automatically
- **set null**: all foreign key values that point to a key that is changed or deleted are set to NULL
- **cascade**: all foreign key values that point to a key that is changed or deleted are changed to the new value or deleted as well

Referential Integrity(4)

```
create table Lectures (
  LectureNr     integer primary key,
  …
  GivenBy integer not null,
  foreign key (GivenBy) references Professors(PersNr)
    [on delete {set null | cascade}]
    [on update {set null | cascade}]
);
```

Default Values

- If an attribute value is not specified during insertion, a default value is used
- If no specific default is given, NULL is the default value

```
create table Assistants (
  PersNr     integer not null primary key,
  Name       varchar(30) not null,
  Area varchar(30) default 'Theory of Ideas',
  Boss       integer
);
```

Indexes

- Indexes speed up access to relations (but slow down update operations)
- Most DBMSs automatically create an index on the primary key (to check uniqueness quickly)
- More details on indexes follow later

```
create [unique] index IndexName
on table Relation (
  Attribute [asc | desc],
  Attribute [asc | desc], …)
```

Removing Objects

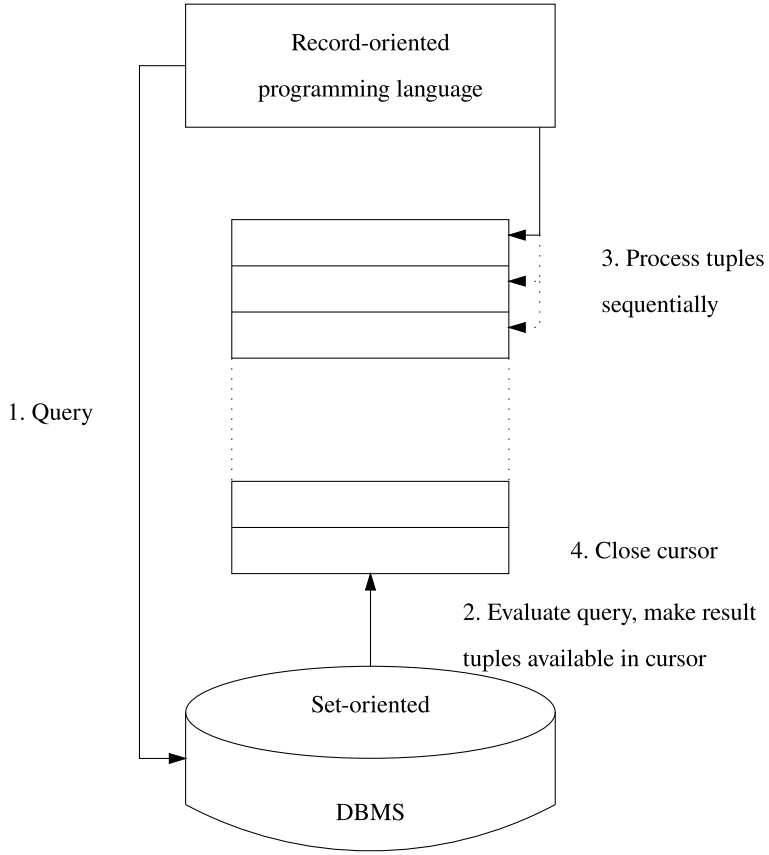
- Relations, views, and indexes can be removed again with the **drop** command:
  - **drop table** *Relation*;
  - **drop view** *View*;
  - **drop index** *Index*;

- Contains commands to control the flow of transactions
- A transaction is a set of interactions between application/user and the DBMS
- Covered later in the context of transaction management

Variants of SQL

- A database can be used not only interactively
- SQL can be embedded into other programming languages
- Problem: SQL is set-oriented, most programming languages are not

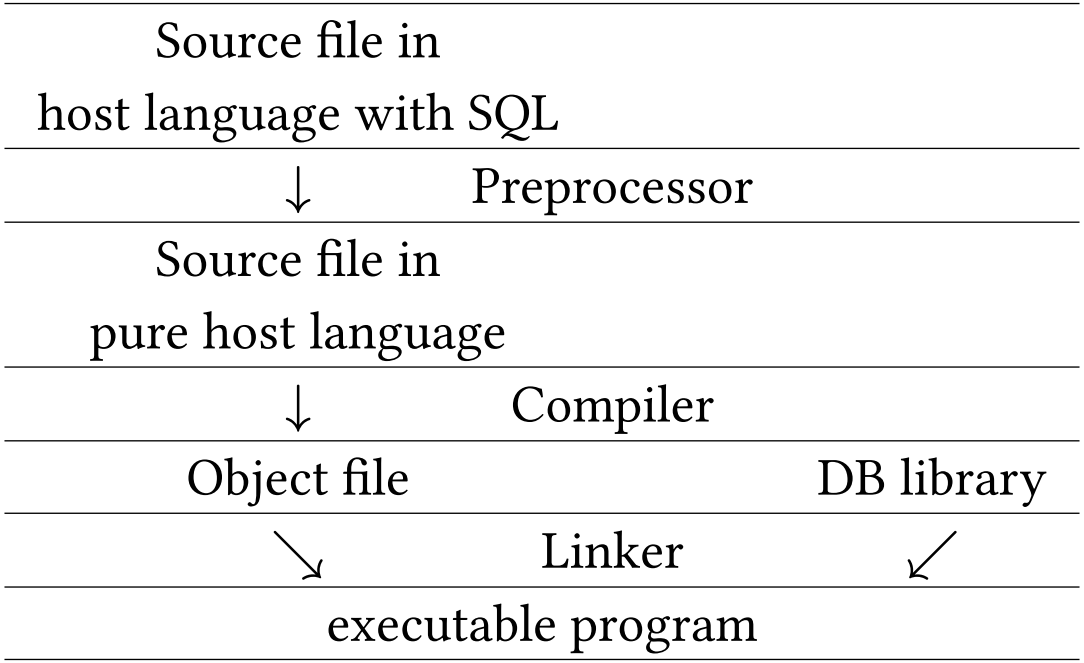
Queries in Applications



Embedded SQL

- Here, SQL commands are embedded directly into the respective host language (e.g., C, C++, Java, etc.)
- SQL commands are marked by a preceding **EXEC SQL**
- They are replaced by the preprocessor with constructs of the respective language

Embedded SQL(2)



Dynamic SQL

- Used when the queries are not yet known at compile time of the program
- Standardized interfaces
  - ODBC (Open Database Connectivity)
  - JDBC (for Java)
- More flexible, but usually somewhat slower than Embedded SQL

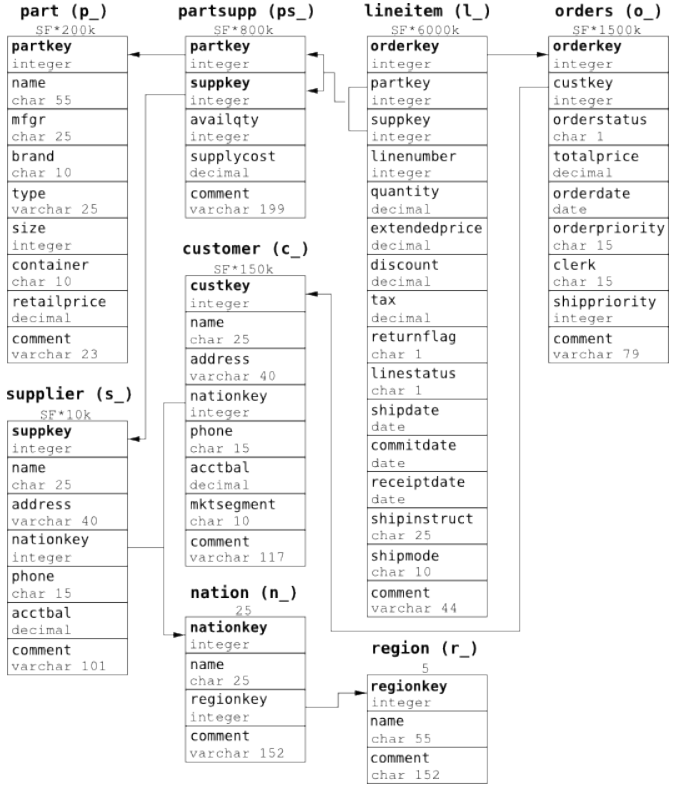
Summary

- SQL is *the* standard language for working with relational systems
- SQL contains commands for retrieving, modifying, inserting, and deleting data
- There are further commands to define a schema, control access, and manage transactions

The next slides go beyond the introductory material, but contain some useful constructs

TPC-H Data Set

- TPC-H is an ad-hoc decision support benchmark
- randomly generated data set, generator is freely available
- size is configurable (*scale factor* 1 corresponds to about 1 GB)



Schema Definition

```
create table lineitem (
   l_orderkey integer not null,
   l_partkey integer not null,
   l_suppkey integer not null,
   l_linenumber integer not null,
   l_quantity decimal(12,2) not null,
   l_extendedprice decimal(12,2) not null,
   l_discount decimal(12,2) not null,
   l_tax decimal(12,2) not null,
   l_returnflag char(1) not null,
   l_linestatus char(1) not null,
   l_shipdate date not null,
   l_commitdate date not null,
   l_receiptdate date not null,
   l_shipinstruct char(25) not null,
   l_shipmode char(10) not null,
   l_comment text not null
);
```

Loading CSV-like Data

```
$ head -n 1 lineitem.tbl
1|15519|785|1|17| ... |egular courts above the|
$ psql tpch
tpch=# \copy lineitem from lineitem.tbl delimiter '|'
ERROR:  extra data after last expected column
CONTEXT:  COPY lineitem, line 1: "1|15519|785|1|17|24386.67|0.04|0.02|N|O|
1996-03-13|1996-02-12|1996-03-22|DELIVER IN PERSON|TRUCK|egular courts above
the|"
tpch# \q
$ sed -i 's/|$//' lineitem.tbl
$ psql
tpch=# \copy lineitem from lineitem.tbl delimiter '|'
COPY 600572
```

- https://www.postgresql.org/docs/current/static/sql-copy.html

```
select l_orderkey, -- single line comment
    sum(l_extendedprice * (1 - l_discount)) revenue,
    o_orderdate, o_shippriority
from customer, orders, lineitem /* this is a
multi-line comment */
where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15'
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 10;
```

- How many items were shipped by German suppliers in 1995?
- What are the names and account balances of the 10 customers from EUROPE in the FURNITURE market segment with the highest account balance?

Subqueries

- Subqueries can appear (almost) anywhere:

```
select n_name,
       (select count(*) from region)
from nation,
     (select *
      from region
      where r_name = 'EUROPE') region
where n_regionkey = r_regionkey
and exists (select 1
            from customer
            where n_nationkey = c_nationkey);
```

Correlated Subqueries

```
select avg(l_extendedprice)
from lineitem l1
where l_extendedprice =
      (select min(l_extendedprice)
       from lineitem l2
       where l1.l_orderkey = l2.l_orderkey);
```

- a subquery is correlated if it refers to tuples of the outer query (`l1.l_orderkey`)
- naive execution: evaluate the inner query for each tuple of the outer query (quadratic runtime, unless there is an index on `l_orderkey`)

Decorrelation of Queries

- Queries can be rewritten so that the correlation disappears (some systems do this automatically):

```
select avg(l_extendedprice)
from lineitem l1,
     (select min(l_extendedprice) m, l_orderkey
      from lineitem
      group by l_orderkey) l2
where l1.l_orderkey = l2.l_orderkey
and l_extendedprice = m;
```

- Decorrelate the following query

```
select c1.c_name
from customer c1
where c1.c_mktsegment = 'AUTOMOBILE'
or c1.c_acctbal >
      (select avg(c2.c_acctbal)
       from customer c2
       where c2.c_mktsegment = c1.c_mktsegment);
```

Set Operators

- UNION, EXCEPT, and INTERSECT remove duplicates

```
select n_name from nation where n_regionkey = 2
union
select n_name from nation where n_regionkey in (1, 2)
intersect
select n_name from nation where n_regionkey < 3
except
select n_name from nation where n_nationkey = 21;
```

“Set” Operators

- UNION ALL: $l + r$
- EXCEPT ALL: $\max(l - r, 0)$
- INTERSECT ALL (very exotic): $\min(l, r)$

```
select n_name from nation where n_regionkey = 2
union all
select n_name from nation where n_regionkey in (1, 2)
intersect all
select n_name from nation where n_regionkey < 3
except all
select n_name from nation where n_nationkey = 21;
```

Other Useful Constructs

- `case` (conditional expressions)

```
select case when n_nationkey > 5
            then 'big' else 'small' end
from nation;
```

- `coalesce(a, b)`: replace NULL with some other value
- `cast` (explicit type conversion)
- `generate_series(begin,end)`
- `random` (random float between 0 and 1):

```
select cast(random()*6 as integer)+1
from generate_series(1,10);  -- 10 dice rolls
```

Working with Strings

- Concatenation:

```
select 'a' || 'b';
```

- simple pattern matching:

```
select 'abcfoo' like 'abc%';
```

- regular expression matching:

```
select 'abcabc' ~ '(abc)*';
```

- extract substring:

```
select substring('abcfoo' from 3 for 2);
```

- regular expression replacement: `(str, pattern, replacement, flags)`

Working with Strings

```
select regexp_replace('ababfooab', '(ab)+', 'xy', 'g');
```

Sampling

- sampling modes: `bernoulli` (pick random tuples) or `system` (pick random pages)
- the random seed can optionally be set with `repeatable`
- supported by PostgreSQL $\geq$ 9.5:

```
select *
from nation tablesample bernoulli(5) -- 5 percent
     repeatable (9999);
```

- one can also request arbitrary tuples:

```
select *
from nation
limit 10; -- 10 arbitrary rows
```

- Compute the average of `o_totalprice` using a sample of 1% of all orders

Recursive Common Table Expressions

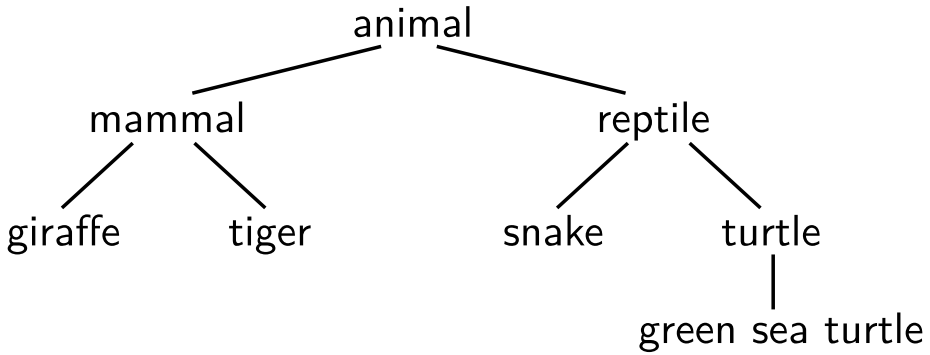
- called `recursive`, but is actually iteration
- traverses hierarchical data of *arbitrary* depth (joins only allow a constant number of steps)

```
with recursive r (i) as (
  select 1                          -- non-recursive part
  union all
  select i+1 from r where i < 5)  -- recursive part
select * from r;
```

- Algorithm:

$workingTable = evaluateNonRecursive()$
output $workingTable$
while $workingTable$ is not empty
  $workingTable = evaluateRecursive(workingTable)$
  output $workingTable$

WITH RECURSIVE ... UNION ALL




```
with recursive
 animals (id, name, parent) as (values (1, 'animal', null),
         (2, 'mammal', 1), (3, 'giraffe', 2), (4, 'tiger', 2),
         (5, 'reptile', 1), (6, 'snake', 5), (7, 'turtle', 5),
         (8, 'green sea turtle', 7)),
  r as (select * from animals where name = 'turtle'
        union all
        select animals.*
        from r, animals
        where animals.id = r.parent)
select * from r;
```

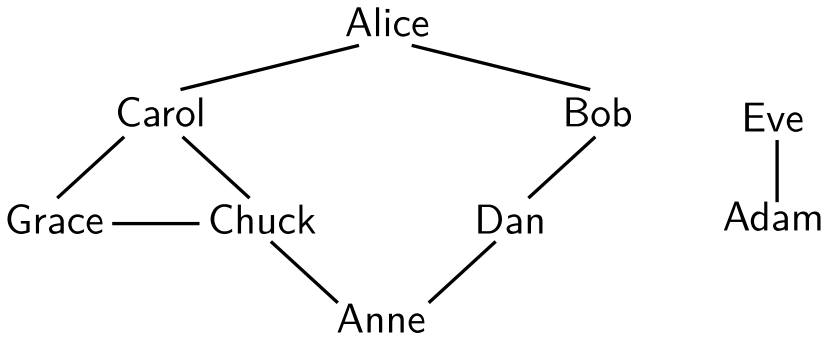
- Compute all descendants of ‘reptile’
- Compute 10! using recursion
- Compute the first 20 Fibonacci numbers ($F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$)

Recursive Common Table Expressions with UNION

- with graph-like data, UNION ALL may not terminate
- **`with recursive`** *`[non-recursive]`* **`union`** *`[recursive]`*
- allows traversing cyclic structures
- Algorithm:

$workingTable = unique(evaluateNonRecursive())$
$result = workingTable$
while $workingTable$ is not empty
  $workingTable = unique(evaluateRecursive(workingTable)) \setminus result$
  $result = result \cup workingTable$
output $result$

WITH RECURSIVE ... UNION

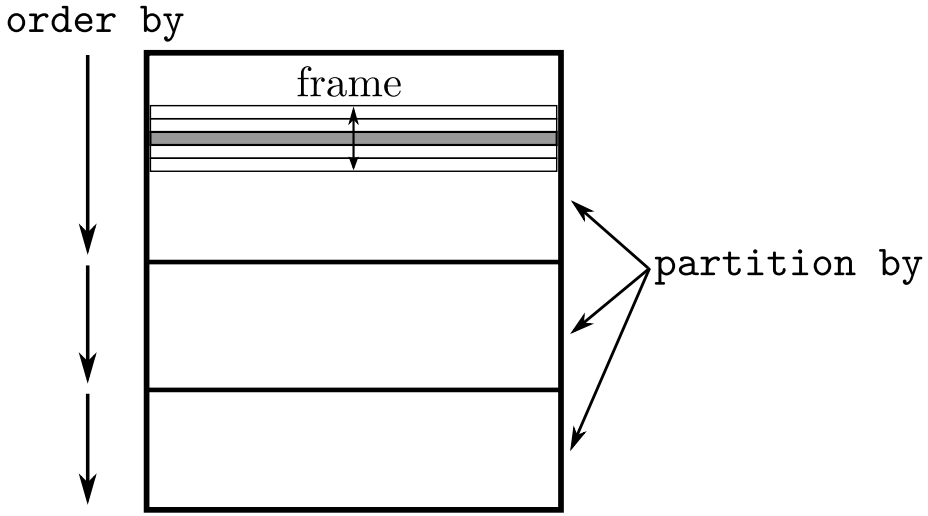



```
with recursive
 friends (a, b) as (values ('Alice', 'Bob'), ('Alice', 'Carol'),
   ('Carol', 'Grace'), ('Carol', 'Chuck'), ('Chuck', 'Grace'),
   ('Chuck','Anne'),('Bob','Dan'),('Dan','Anne'),('Eve','Adam')),
 friendship (name, friend) as -- friendship is symmetric
    (select a, b from friends union all select b, a from friends),
 r as (select 'Alice' as name
       union
       select friendship.name from r, friendship
       where r.name = friendship.friend)
select * from r;
```

- very versatile feature: time series analysis, ranking, top-k, percentiles, moving averages, cumulative sums
- window functions are evaluated after most other clauses (including `group by`), but before `order by`
- in contrast to aggregation, window functions do not change the input, they only compute additional columns

Window Functions: Concepts

```
select o_custkey, o_orderdate,
  sum(o_totalprice) over       -- window function
    (partition by o_custkey    -- partitioning clause
     order by o_orderdate      -- ordering clause
     range between unbounded preceding
      and current row)         -- framing clause
from customer;
```



Window Functions without Framing

- ranking:
  - `rank()`: rank of the current row with gaps
  - `dense_rank()`: rank of the current row without gaps
  - `row_number()`: row number of the current row
  - `ntile(n)`: distribute evenly over buckets (returns integer from 1 to `n`)
- distribution:
  - `percent_rank()`: relative rank of the current row ((rank - 1) / (number of rows - 1))
  - `cume_dist()`: relative rank of the peer group[1] ((number of rows preceding or peer with current row) / (number of rows))
- navigation in partition:
  - `lag(expr, offset, default)`: evaluate `expr` on the preceding row in the partition
  - `lead(expr, offset, default)`: evaluate `expr` on the following row in the partition

[1] Rows with equal partitioning and ordering values are called *peers*.

- Determine medals based on the number of orders, example output:

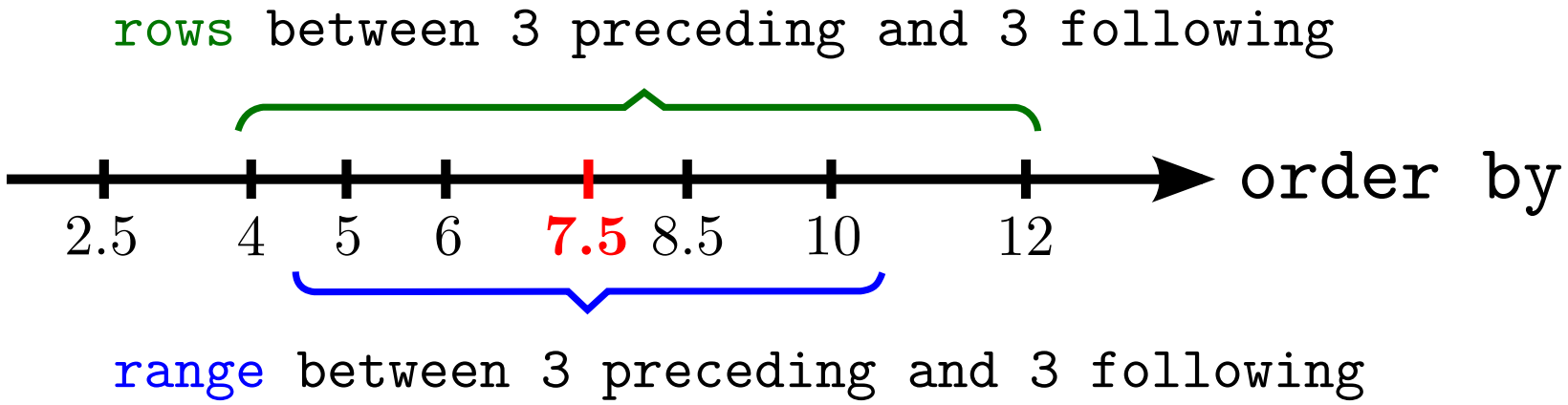
```
 custkey | count | medal
---------+-------+--------
    8761 |    36 | gold
   11998 |    36 | gold
    8362 |    35 | bronze
    4339 |    35 | bronze
     388 |    35 | bronze
    3151 |    35 | bronze
    9454 |    35 | bronze
```

- Compute the yearly (`extract(year from o_orderdate)`) percentage change in revenue (`sum(o_totalprice)`), example output:

```
  y   |    revenue     | pctchange
------+----------------+-----------
 1992 | 3249822143.71 |
 1993 | 3186680293.06 |     -1.94
 1994 | 3276391729.79 |      2.82
 1995 | 3269894993.32 |     -0.20
 1996 | 3227878999.30 |     -1.28
 1997 | 3212138221.07 |     -0.49
 1998 | 1933789650.38 |    -39.80
```

Window Functions: Framing

- `current row`: the current row (in `range` mode including all peers)
- `unbounded preceding`: first row in the partition
- `unbounded following`: last row in the partition



- default frame (when an `order by` clause is present): `range between unbounded preceding and current row`
- default frame (when no `order by` clause is present): `range between unbounded preceding and unbounded following`
- complex frame specifications are not yet supported by PostgreSQL

Window Functions with Framing

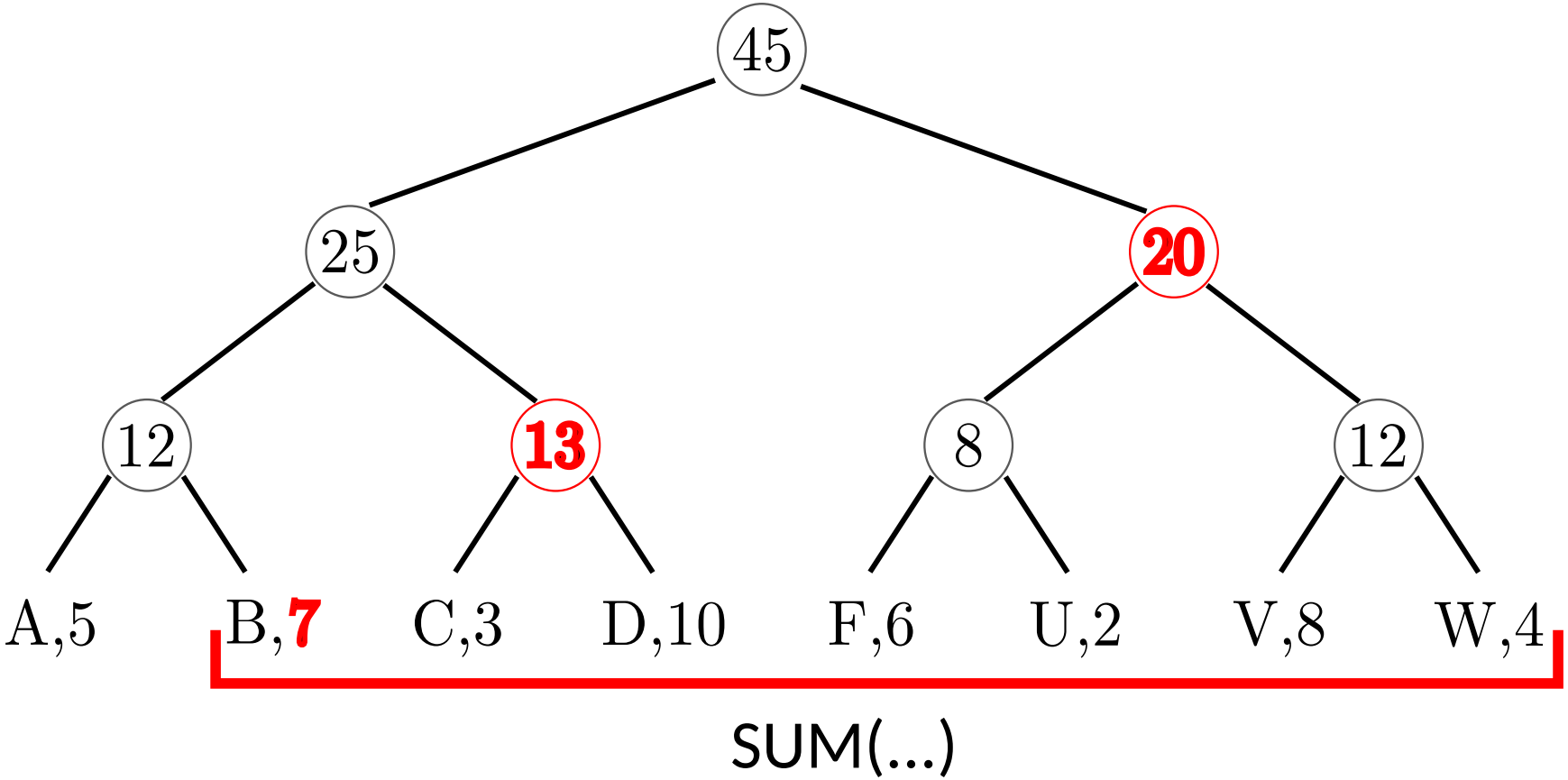
- aggregates (`min`, `max`, `sum`, ...):
  compute aggregate over all tuples in the current frame
- navigation in frame:
  `first_value(expr)`, `last_value(expr)`, `nth_value(expr, nth)`: evaluate `expr` on the first/last/`nth` row of the frame

- Compute the cumulative customer spending (`sum(o_totalprice)`) over time (`o_orderdate`)
- For each customer from GERMANY, compute the cumulative spending (`sum(o_totalprice)`) by year (`extract(year from o_orderdate)`), example output:

```
 custkey |  yr  | running_sum
---------+------+-------------
      62 | 1992 |   169991.32
      62 | 1993 |   344376.79
      62 | 1994 |   433638.98
      62 | 1995 |   960047.31
      62 | 1996 |  1372061.28
      62 | 1997 |  1658247.25
      62 | 1998 |  2055669.94
      71 | 1992 |   403017.41
      71 | 1993 |   751256.86
```

```
71 | 1994 |  1021446.72
71 | 1995 |  1261012.10
```

Efficient Evaluation with Segment Trees



Statistical Aggregates

- `stddev_samp(expr)`: standard deviation
- `corr(x, y)`: correlation
- `regr_slope(y, x)`: linear regression slope
- `regr_intercept(y, x)`: linear regression intercept

Ordered-Set Aggregates

- aggregate functions that require materialization/sorting and have special syntax
- `mode()`: most frequent value
- `percentile_disc(p)`: compute discrete percentile ($p \in [0, 1]$)
- `percentile_cont(p)`: compute continuous percentile ($p \in [0, 1]$), interpolates if necessary, only for numeric data types

```
select percentile_cont(0.5)
    within group (order by o_totalprice)
from orders;
```

```
select o_custkey,
 percentile_cont(0.5) within group (order by o_totalprice)
from orders
group by o_custkey;
```

- aggregate across multiple dimensions, e.g., revenue by year, by customer, by supplier
- specify multiple groupings:
  ```
  group by grouping sets ((a, b), (a), ())
  ```
- hierarchical groupings:
  ```
  group by rollup (a, b)
  ```
- both are equivalent to:

```
select a, b, sum(x) from r group by a, b
union all
select a, null, sum(x) from r group by a
union all
select null, null, sum(x) from r;
```

- all (2^n) groupings:
  `group by cube (a, b)` is equivalent to
  ```
  group by grouping sets ((a, b), (a), (b), ())
  ```

- Aggregate revenue (`sum(o_totalprice)`): total, by region (`r_name`), by nation (`n_name`), example output:

```
     revenue    |        region        |       nation
----------------+----------------------+----------------------
   836330704.31 | AFRICA               | ALGERIA
   902849428.98 | AFRICA               | ETHIOPIA
   784205751.27 | AFRICA               | KENYA
   893122668.52 | AFRICA               | MOROCCO
   852278134.31 | AFRICA               | MOZAMBIQUE
  4268786687.39 | AFRICA               |
...
 21356596030.63 |                      |
```