

# Data Management

## 05 Query Languages (SQL)

**Matthias Boehm**

Graz University of Technology, Austria  
Computer Science and Biomedical Engineering  
Institute of Interactive Systems and Data Science  
BMK endowed chair for Data Management



# Announcements/Org

## ■ #1 Video Recording

- Link in **TeachCenter** & **TUbe** (lectures will be public)
- Currently via <https://tugraz.webex.com/meet/m.boehm>



## ■ #2 Exercise 1

- Deadline: **Nov 02 + 7 late days** in TeachCenter
- Grading starts Nov 09

**Q&A**

## ■ #3 Exercise 2

- Task description published last weekend, discussed today
- Remaining data cleaning until Wednesday → **simplification**
- Deadline: **Nov 30 + 7 late days** in TeachCenter

# Agenda

- **Structured Query Language (SQL)**
- **Other Query Languages (XML, JSON)**
- **Exercise 2: Query Languages and APIs**

# Structured Query Language (SQL)

# What is a(n) SQL Query?

```

SELECT Firstname, Lastname, Affiliation, Location
FROM Participant AS R, Locale AS S
WHERE R.LID = S.LID
        AND Location LIKE '%, GER'
    
```

#1 **Declarative:**  
what not how



Firstname	Lastname	Affiliation	Location
Volker	Markl	TU Berlin	Berlin, GER
Thomas	Neumann	TU Munich	Munich, GER

#2 **Flexibility:**  
closed → composability

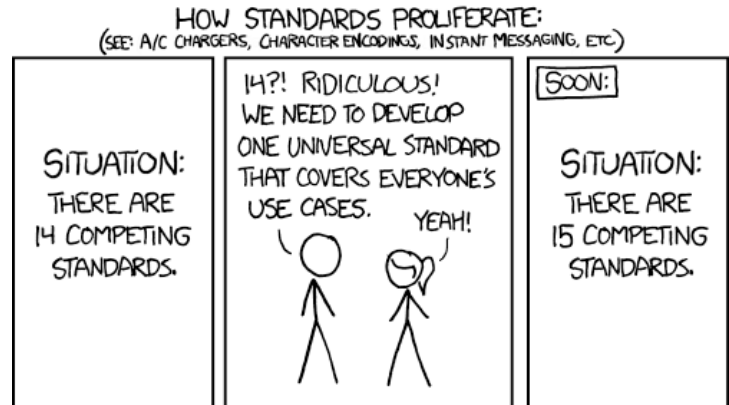
#3 **Automatic  
Optimization**

#4 **Physical Data  
Independence**

# Why should I care?

## SQL as a Standard

- Standards ensure **interoperability**, avoid **vendor lock-in**, and protect **application investments**
- Mature standard** with heavy industry support for decades
- Rich eco system** (existing apps, BI tools, services, frameworks, drivers, design tools, systems)



<https://xkcd.com/927/>

## SQL is here to stay

- Foundation of mobile/server **application data management**
- Adoption of existing standard** by new systems (e.g., SQL on Hadoop, cloud DBaaS)
- Complemented by NoSQL abstractions, see lecture **10 NoSQL (key-value, document, graph)**



# Overview SQL

- **Structured Query Language (SQL)**
  - Current Standard: ISO/IEC 9075:2016 (SQL:2016)
  - **Data Definition Language (DDL)** → Manipulate the database schema
  - **Data Manipulation Language (DML)** → Update and query database
  - **Data Control Language (DCL)** → Modify permissions

- **Dialects**

- Spectrum of system-specific dialects for **non-core features**
- Data types and size constraints
- Catalog, builtin functions, and tools
- Support for new/optional features
- Case-sensitive identifiers

Name	Examples
T-SQL	Microsoft, Sybase
PL/SQL	Oracle, (IBM)
PL/pgSQL	PostgreSQL, derived
Unnamed	<b>Most systems</b>

# The History of the SQL Standard

[C. J. Date: A Critique of the  
SQL Database Language.  
SIGMOD Record 1984]



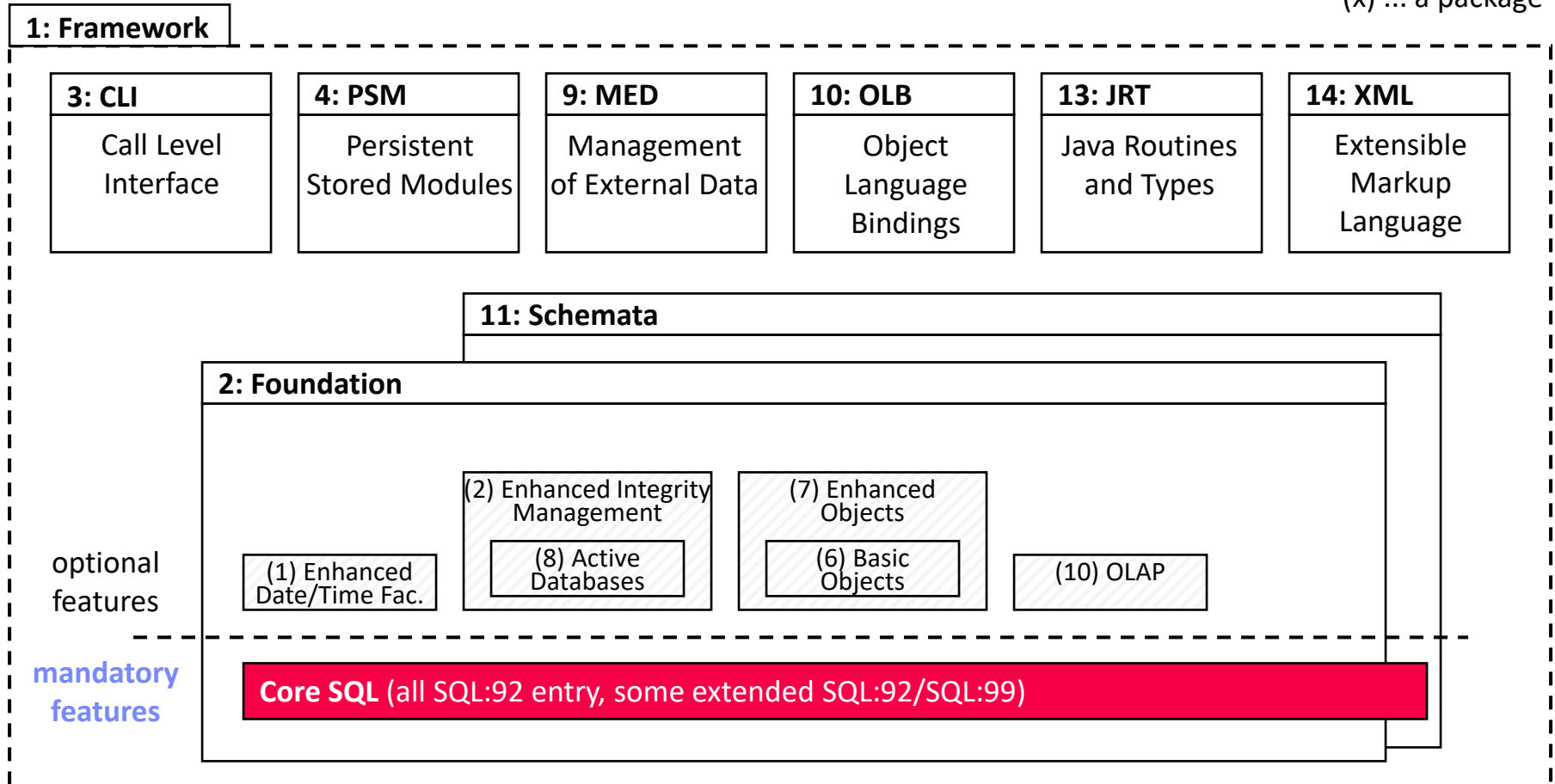
- **SQL:1986**
  - **Database Language SQL**, ANSI X3.135-1986, ISO-9075-1987(E)
  - '87 international edition
- **SQL:1989 (120 pages)**
  - **Database Language SQL with Integrity Enhancements**, ANSI X3.135-1989, ISO-9075-1989(E)
- **SQL:1992 (580 pages)**
  - **Database Language SQL**, ANSI X3-1992, ISO/IEC-9075 1992, DIN 66315
  - '95 SQL/CLI (part 3), '96 SQL/PSM (part 4)
- **SQL:1999 (2000 pages)**
  - **Information Technology – Database Language – SQL**, ANSI/ISO/IEC-9075 1999
  - Complete reorg, '00 OLAP, '01 SQL/MED, '01 SQL/OLB, '02 SQL/JRT
- **SQL:2003 (3764 pages)**
  - **Information Technology – Database Language – SQL**, ANSI/ISO/IEC-9075 2003



# The History of the SQL Standard, cont.

## Overview SQL:2003

x: ... a part  
(x) ... a package



# The History of the SQL Standard, cont.

Since SQL:2003 overall structure remained unchanged ...

- **SQL:2008** (???? pages)
  - **Information Technology – Database Language – SQL**, ANSI/ISO/IEC-9075 2003
  - E.g., **XML** XQuery extensions, case/trigger extension
- **SQL:2011** (4079 pages)
  - **Information Technology – Database Language – SQL**, ANSI/ISO/IEC-9075 2011
  - E.g., time periods, temporal constraints, time travel queries
- **SQL:2016** (???? pages)
  - **Information Technology – Database Language – SQL**, ANSI/ISO/IEC-9075 2016
  - E.g., **JSON** documents and functions (optional)

➔ **Note:** We can only discuss common primitives

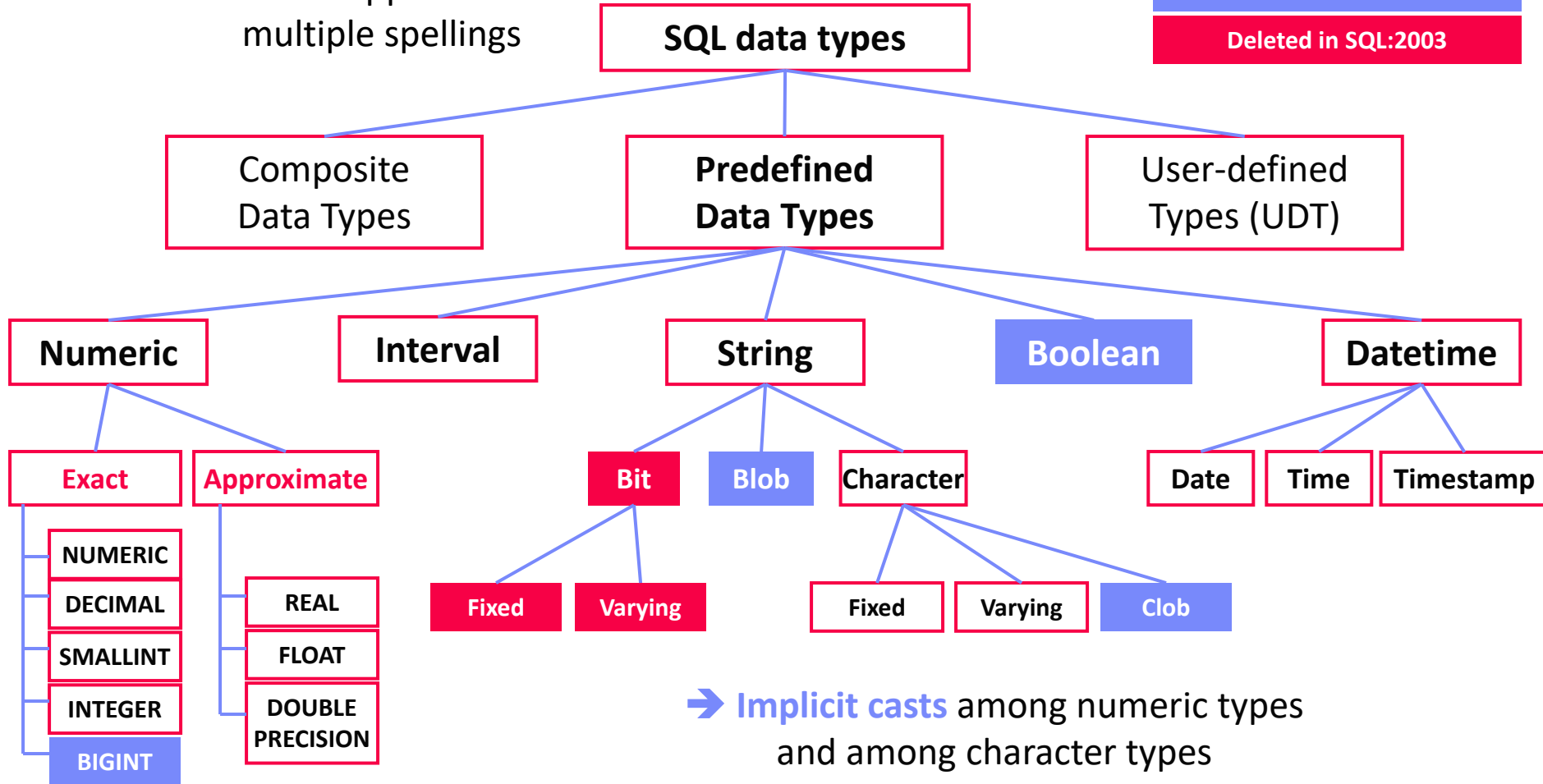
[Working Draft SQL:2011:  
[https://www.wiscorp.com/  
SQLStandards.html](https://www.wiscorp.com/SQLStandards.html)]

# Data Types in SQL:2003

- Large Variety of Types

- With support for multiple spellings

Added in SQL:1999 / SQL:2003  
Deleted in SQL:2003



# Data Types in PostgreSQL

Appropriate, Brief, Complete

## ■ Strings

- **CHAR(n)** → fixed-length character sequence (padded to n)
- **VARCHAR(n)** → variable-length character sequence (n max)
- TEXT → variable-length character sequence

## ■ Numeric

- SMALLINT → 2 byte integer (signed short)
- **INT/INTEGER** → 4 byte integer (signed int)
- SERIAL → INTEGER w/ auto increment
- **NUMERIC(p, s)** → exact real with p digits and s after decimal point

## ■ Time

- **DATE** → date
- **TIMESTAMP/TIMESTAMPTZ** → date and time, timezone-aware if needed

## ■ JSON

- JSON → text JSON representation (requires reparsing)
- JSONB → binary JSON representation

# Create, Alter, and Delete Tables

Templates in SQL  
Examples in PostgreSQL

## ■ Create Table

- Typed attributes
- Primary and foreign keys
- **NOT NULL**, **UNIQUE** constraints
- **DEFAULT** values
- **CHECK** constraints

```
CREATE TABLE Students (
  SID INTEGER PRIMARY KEY,
  Fname VARCHAR(128) NOT NULL,
  Lname VARCHAR(128) NOT NULL,
  Mtime DATE DEFAULT CURRENT_DATE
);
```

```
CREATE TABLE Students AS SELECT ...;
```

## ■ Alter Table

- **ADD/DROP** columns
- **ALTER** data type, defaults, constraints, etc

```
ALTER TABLE Students ADD DoB DATE;
```

```
ALTER TABLE Students ADD CONSTRAINT
  PKStudent PRIMARY KEY(SID);
```

## ■ Delete Table

- Delete table
- **Note:** order of tables matters due to referential integrity

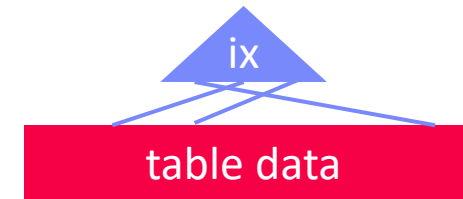
```
DROP TABLE Students; -- sorry
DROP TABLE Students CASCADE;
DROP TABLE IF EXISTS Countries,
  Cities, Airports, Airlines,
  Routes, Planes, Routes_Planes;
```

# Create and Delete Indexes

## ■ Create Index

- Create a secondary (nonclustered) index on a set of attributes
- **Clustered:** tuples sorted by index
- **Non-clustered:** sorted attribute with tuple references
- Can specify uniqueness, order, and indexing method
- **PostgreSQL methods:** btree, hash, gist, and gin
  - ➔ see lecture **07 Physical Design and Tuning**

```
CREATE INDEX ixStudLname
ON Students USING btree
(Lname ASC NULLS FIRST);
```



## ■ Delete Index

- Drop indexes by name

```
DROP INDEX ixStudLname;
```

## ■ Tradeoffs

- Indexes often automatically created for **primary keys** / **unique** attributes
- **Lookup/scan performance** vs **insert performance**

# Database Catalog

[Meikel Poess: **TPC-H**. Encyclopedia of Big Data Technologies 2019]

## ■ Catalog Overview

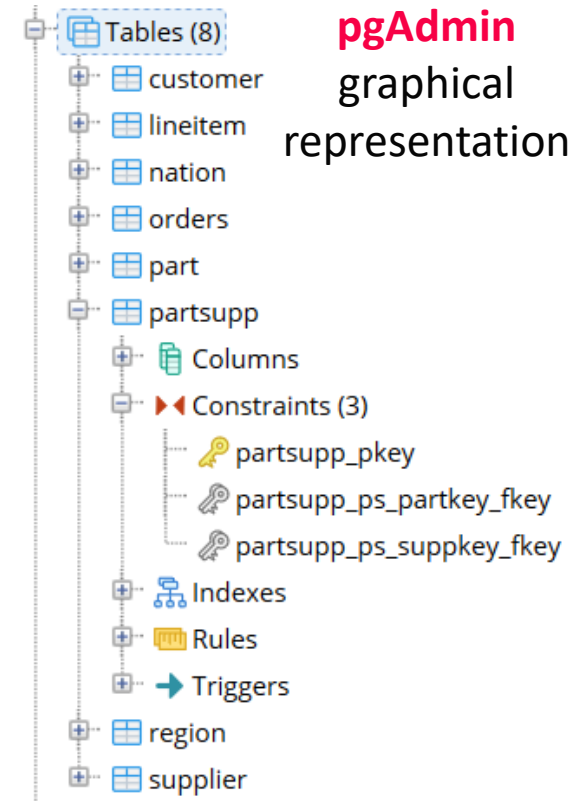
- **Meta data** of all database objects (tables, constraints, indexes) → **mostly read-only**
- **Accessible through SQL**
- Organized by schemas (**CREATE SCHEMA tpch;**)

## ■ SQL Information\_Schema

- Schema with tables for all tables, views, constraints, etc
- **Example:** check for existence of accessible table

```
SELECT 1 FROM information_schema.tables
WHERE table_schema = 'tpch'
      AND table_name = 'customer'
```

(defined as views over PostgreSQL catalog tables)



# Insert

## ■ Insert Tuple

- **Insert a single tuple** with implicit or explicit attribute assignment

```
INSERT INTO Students (SID, Lname, Fname, MTime, DoB)
VALUES (7, 'Boehm', 'Matthias', '2002-10-01', '1982-06-25');
```

- Insert attribute key-value pairs to use auto increment, defaults, NULLs, etc

```
INSERT INTO Students (Lname, Fname, DoB) SERIAL SID,
VALUES ('Boehm', 'Matthias', '1982-06-25'), DEFAULT MTime
(...), (...);
```

## ■ Insert Table

- **Redirect query result into INSERT** (append semantics)

```
INSERT INTO Students
SELECT * FROM NewStudents;
```

**Analogy Linux redirect (append):**  
 cat NewStudents.txt >> Students.txt



# Update and Delete

## ▪ Update Tuple/Table

- **Set-oriented update** of attributes
- Update single tuple via predicate on **primary key**

```
UPDATE Students
  SET MTime = '2002-10-02'
  WHERE LName = 'Boehm';
```

## ▪ Delete Tuple/Table

- **Set-oriented delete** of tuples
- Delete single tuple via predicate on **primary key**

```
DELETE FROM Students
  WHERE extract(year
    FROM mtime) < 2010;
```

## ▪ **Note:** Time travel and multi-version concurrency control

- Deleted tuples might be just **marked as inactive**
- See lecture **09 Transaction Processing and Concurrency**

# Basic Queries

Basic Query Template

- Select-From-Where
- Grouping and Aggregation
- Having and ordering
- Duplicate elimination

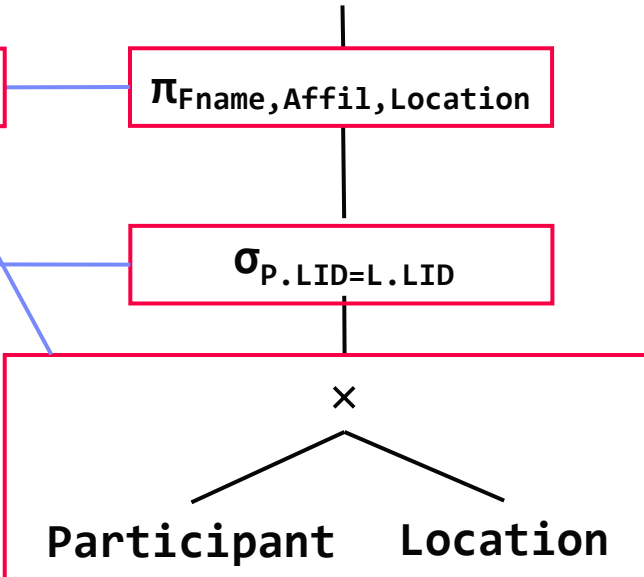
```

SELECT [DISTINCT] <column_list>
FROM [<table_list> |
<table1> [RIGHT | LEFT | FULL] JOIN
<table2> ON <condition>]
[WHERE <predicate>]
[GROUP BY <column_list>]
[HAVING <grouping predicate>]
[ORDER BY <column_list> [ASC | DESC]]
    
```

Example

```

SELECT Fname, Affil, Location
FROM Participant AS P,
     Locale AS L
WHERE P.LID = L.LID;
    
```



# Basic Queries, cont.

## ▪ Distinct and All

- **Distinct and all** alternatives
- Projection w/ **bag semantics** by default

```
SELECT DISTINCT Lname, Fname
FROM Students;
```

## ▪ Sorting

- Convert a **bag** into a **sorted list** of tuples; order lost if used in other ops
- Single order: (Lname, Fname) **DESC**
- Evaluated last in a query tree

```
SELECT * FROM Students
ORDER BY Lname DESC,
         Fname DESC;
```

## ▪ Set Operations

- See **04 Relational Algebra and Calculus**  
→ **UNION, INTERSECT, EXCEPT**
- Set operations **set semantics** by default  
→ **DISTINCT** (set) vs **ALL** (bag)

```
(SELECT Firstname, Lastname
FROM Participant2018)
UNION DISTINCT
(SELECT Firstname, Lastname
FROM Participant2013)
```

# Grouping and Aggregation

## Grouping and Aggregation

- **Grouping:** determines the distinct groups
- **Aggregation:** compute aggregate  $f(B)$  per group
- Column list can only contain **grouping columns**, **aggregates**, or **literals**
- **Having:** selection predicate on groups and aggregates

## Example

- Sales (Customer, Location, Product, Quantity, Price)
- **Q: Compute number of sales sumQ and revenue per product sumQP**

```

SELECT Product,
       sum(Quantity) AS SumQ,
       sum(Quantity*Price) AS SumQP
FROM Sales
GROUP BY Product
    
```



Product	Quantity	Price
A	1	10
B	3	20
A	2	10
B	1	20



Product	SumQ	SumQP
A	3	30
B	4	80

# BREAK (and Test Yourself)

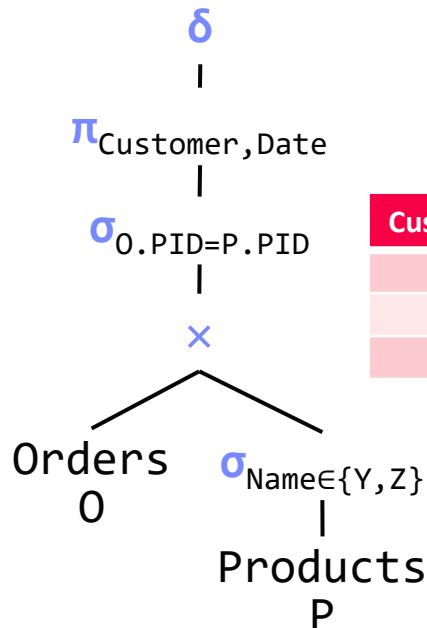
- Task: SQL queries for the following query trees.

Orders

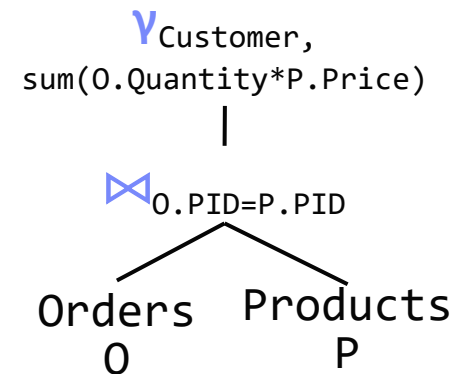
OID	Customer	Date	Quantity	PID
1	A	'2019-06-22'	3	2
2	B	'2019-06-22'	1	3
3	A	'2019-06-22'	1	4
4	C	'2019-06-23'	2	2
5	D	'2019-06-23'	1	4
6	C	'2019-06-23'	1	1

Products

PID	Name	Price
1	X	100
2	Y	15
4	Z	75
3	W	120



Customer	Date
A	'2019-06-22'
C	'2019-06-23'
D	'2019-06-23'



Customer	Sum
A	120
B	120
C	130
D	75

```

SELECT DISTINCT Customer, Date
FROM Orders O, Products P
WHERE O.PID = P.PID
AND Name IN('Y', 'Z')
    
```

```

SELECT Customer,
       sum(O.Quantity * P.Price)
FROM Orders O, Products P
WHERE O.PID = P.PID
GROUP BY Customer
    
```

# Subqueries

## ■ Subqueries in Table List

- Use a subquery result like a base table
- Modularization with **WITH C AS (SELECT ...)**

```
SELECT S.Fname, S.Lname, C.Name
FROM Students AS S,
     (SELECT CID, Name FROM Country
      WHERE ...) AS C
WHERE S.CID=C.CID;
```

## ■ Subqueries w/ IN

- Check containment of values in result set of sub query

```
SELECT Product, Quantity, Price
FROM Sales
WHERE Product NOT IN(
     SELECT Product FROM Sales
     GROUP BY Product
     HAVING sum(Quantity*Price)>1e6)
```

## ■ Other subqueries

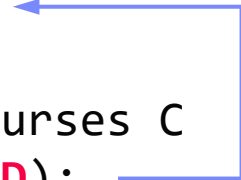
- **EXISTS**: existential quantifier  $\exists x$  for correlated subqueries
- **ALL**: comparison (w/ universal quantifier  $\forall x$ )
- **SOME/ANY**: comparison (w/ existential quantifier  $\exists x$ )

# Correlated and Uncorrelated Subqueries

## ■ Correlated Subquery

- **Evaluated subquery for every tuple** of outer query
- Use of attribute from table bound in outer query inside subquery

```
SELECT P.Fname, P.Lname
FROM Professors P,
WHERE NOT EXISTS(
    SELECT * FROM Courses C
    WHERE C.PID=P.PID);
```



## ■ Uncorrelated Subquery

- Evaluate subquery just once
- No attribute correlations between subquery and outer query

```
SELECT P.Fname, P.Lname
FROM Professors P,
WHERE P.PID NOT IN(
    SELECT PID FROM Courses);
```

## ■ Query Unnesting (de-correlation)

- Rewrite during query compilation
- See lecture [08 Query Processing](#)

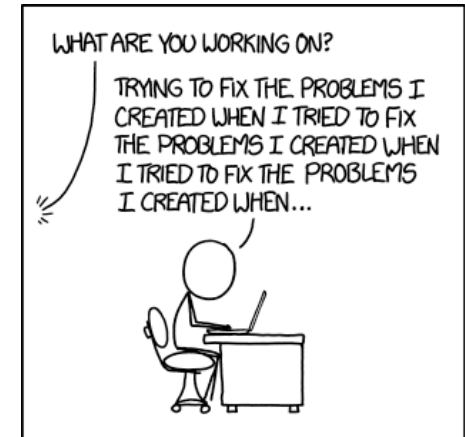
[Thomas Neumann, Alfons Kemper: Unnesting Arbitrary Queries. **BTW 2015**]



# Recursive Queries

## Approach

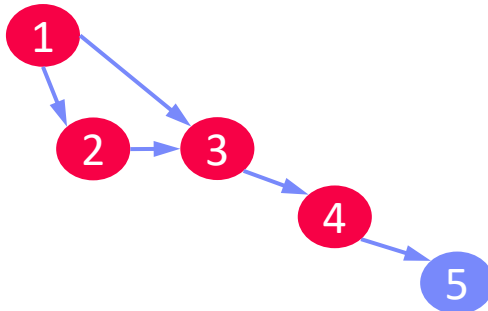
- **WITH RECURSIVE** <name> (<arguments>)
- Compose recursive table from **non-recursive term**, **union all/distinct**, and **recursive term**
- Terminates when recursive term yields empty result



<https://xkcd.com/1739/>

## Example

- Courses(CID, Name),  
Precond(pre REF CID, suc REF CID)
- Dependency graph (pre → suc)



```

WITH RECURSIVE rPrereq(p,s) AS(
    (SELECT pre, suc
     FROM Precond WHERE suc=5)
    UNION DISTINCT
    (SELECT B.pre, B.suc
     FROM Precond B, rPrereq R
     WHERE B.suc = R.p)
)
SELECT DISTINCT p FROM rPrereq
    
```

4  
3  
1  
2



# Procedures and Functions

## Overview Procedures and Functions

- Stored programs, written in **PL/pgSQL** or other languages
- Control flow (loops, branches) and SQL queries**

## (Stored) Procedures

- Can be called standalone via **CALL** <proc\_name>( <args> );
- Procedures return no outputs

```

CREATE PROCEDURE prepStud(a INT)
LANGUAGE PLPGSQL AS $$
BEGIN
    DELETE FROM Students;
    INSERT INTO Students
        SELECT * FROM NewStudents;
END; $$;
    
```

## Functions

- Can be called standalone or inside queries
- Functions are value mappings
- Table functions can return sets of records with multiple attributes

```

CREATE FUNCTION sampleProp(FLOAT)
RETURNS FLOAT
AS 'SELECT $1 * (1 - $1);'
LANGUAGE SQL;
    
```

# Triggers

## Overview Trigger

- Similar to stored procedure but register ON INSERT, DELETE, or UPDATE
- Allows complex check constraints and active behavior such as replication, auditing, etc (good and bad)

## Trigger **CREATE TRIGGER** <triggername>

### Template

```

BEFORE | AFTER | INSTEAD OF
INSERT | DELETE | (UPDATE OF <column_list>)
ON <tablename>
    
```

```

[REFERENCING <old_new_alias_list>]
    
```

```

[FOR EACH {ROW | STATEMENT}]
    
```

```

[WHEN (<search condition>)]
    
```

```

<SQL procedure statement> |
    
```

```

BEGIN ATOMIC
    
```

```

    {<SQL Procedure statement>;}...
    
```

```

END
    
```

Event

Condition

Action

Not supported in  
PostgreSQL  
(need single UDF)

# Views and Authorization

## ■ Creating Views

- **Create a logical table from a query**
- Inserts can be propagated back to base relations only in special cases
- **Allows authorization** for subset of tuples

```
CREATE VIEW TeamDM AS  
SELECT * FROM  
    Employee E, Employee M  
WHERE E.MgrID = M.EID  
AND M.login = 'mboehm';
```

## ■ Access Permissions Tables/Views

- **Grant** query/modification rights on database objects for specific users, roles
- **Revoke** access rights from users, roles (recursively revoke permissions of dependent views via **CASCADE**)

```
GRANT SELECT  
ON TABLE TeamDM  
TO mboehm;
```

```
REVOKE SELECT  
ON TABLE TeamDM  
FROM mboehm;
```

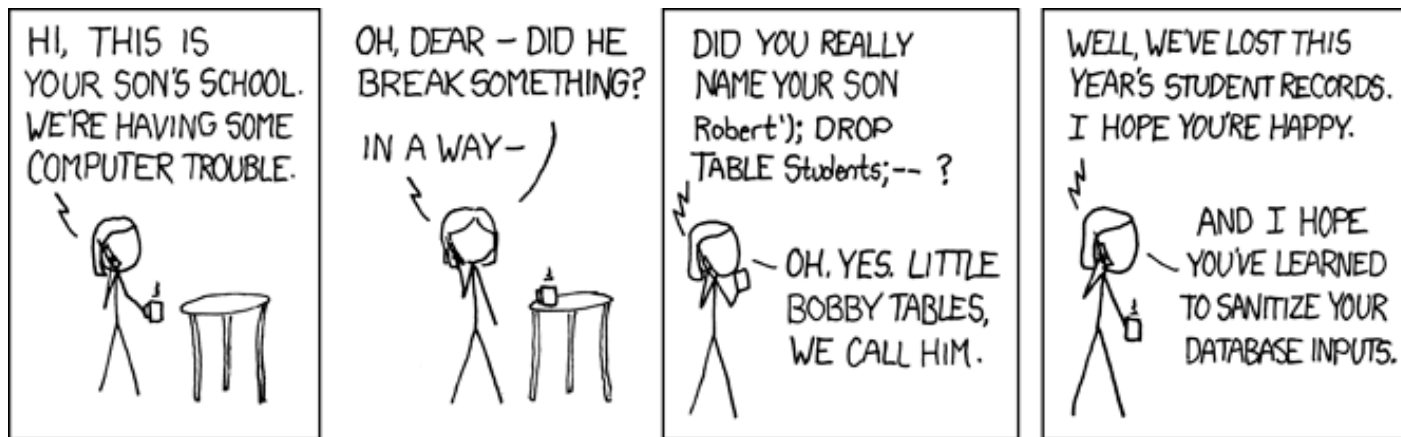
# Beware of SQL Injection



## ■ Problematic SQL String Concatenation

```
INSERT INTO Students (Lname, Fname)
VALUES ('"+ @lname +"', '"+ @fname +"' );";
```

## ■ Possible SQL-Injection Attack



<https://xkcd.com/327/>

```
INSERT INTO Students (Lname, Fname) VALUES ('Smith', 'Robert');
DROP TABLE Students; --');
```

# Other Query Languages (XML, JSON)

# No really, why should I care?

- **Semi-structured XML and JSON**
  - **Self-contained documents** for representing nested data
  - **Common data exchange formats** without redundancy of flat files
  - Human-readable formats → often used for SW configuration
- **Goals**
  - **Awareness of XML and JSON** as data models
  - Query languages and embedded querying in SQL

# XML (Extensible Markup Language)

## ■ XML Data Model

- Meta language to define specific **exchange formats**
- Document format for **semi-structured data**
- Well formedness
- XML schema / DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <student id="1">
    <course id="INF.01017UF" name="DM"/>
    <course id="706.550" name="AMLS"/>
  </student>
  <student id="5">
    <course id="706.520" name="DIA"/>
  </student>
</data>
```

## ■ XPath (XML Path Language)

- Query language for **accessing collections of nodes** of an XML document
- Axis specifies for ancestors, descendants, siblings, etc

```
/data/student[@id='1']/course/@name
```

↓  
"DM"  
"AMLS"

## ■ XSLT (XML Stylesheet Language Transformations)

- Schema mapping (transformation) language for XML documents

## ■ XQuery

- Query language to extract, transform, and analyze XML documents

# XML in PostgreSQL, cont.

## ■ Overview XML in PostgreSQL

- Data types **TEXT** or **XML** (well-formed, type-safe operations)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

## ■ Creating XML

- Various **built-in functions** to parse documents, and create elements/attributes
- XMLPARSE(<xml\_document>) → **XML type**
- XMLELEMENT / XMLATTRIBUTES

```
INSERT INTO Students
(Fname, Lname, Doc)
VALUES ('John', 'Smith',
xmlparse(<source_doc>));
```

## ■ Processing XML

- Execute **XPath** expressions on XML types
- XMLEXIST with **XPath instead of XQuery**
- XPATH with optional namespace handling

```
SELECT Fname, Lname,
xpath('/student/@id', Doc)
FROM Students
```



# JSON (JavaScript Object Notation)

## ■ JSON Data Model

- Data exchange format for **semi-structured data**
- **Not as verbose as XML** (especially for arrays)
- Popular format (e.g., Twitter)



```
{“students:”[
  {“id”: 1, “courses”:[
    {“id”:“INF.01017UF”, “name”:“DM”},
    {“id”:“706.550”, “name”:“AMLS”}]}],
  {“id”: 5, “courses”:[
    {“id”:“706.520”, “name”:“DIA”}]}],
]}
```

## ■ Query Languages

- **Most common: libraries** for tree traversal and data extraction
- **JSONiq**: XQuery-like query language
- **JSONPath**: XPath-like query language

### JSONiq Example:

```
declare option jsoniq-version “...”;
for $x in collection(“students”)
  where $x.id lt 10
  let $c := count($x.courses)
  return {“sid”:$x.id, “count”:$c}
```

[<http://www.jsoniq.org/docs/JSONiq/html-single/index.html>]

# JSON in PostgreSQL, cont.

## Overview JSON in PostgreSQL

- Alternative data types: **JSON** (text), **JSONB** (binary, with restrictions)
- Implements RFC 7159, built-ins for conversion and access

## Creating JSON

- Built-in functions for creating JSON from tables and tables from JSON input

```
SELECT row_to_json(t) FROM
    (SELECT Fname, Lname
     FROM Students) t
```

## Processing JSON

- Specialized operators for **tree traversal and data extraction**
- > operator**: get JSON array element/object
- >> operator**: get JSON array element/object as text
- Built-in functions for extracting json (e.g., json\_each)

```
SELECT Fname, Lname,
    Doc->students->>id
    FROM Students
```

# Exercise 2:

## Query Languages and APIs

[https://mboehm7.github.io/teaching/ws2122\\_dbs/  
02\\_ExerciseQueriesAPIs.pdf](https://mboehm7.github.io/teaching/ws2122_dbs/02_ExerciseQueriesAPIs.pdf)

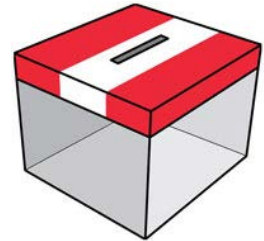
Published: **Nov 06, 2021**

(data cleaning until Nov 09 publish date)

Deadline: **Nov 30, 2021**

# Exercises: Austrian National Elections

New



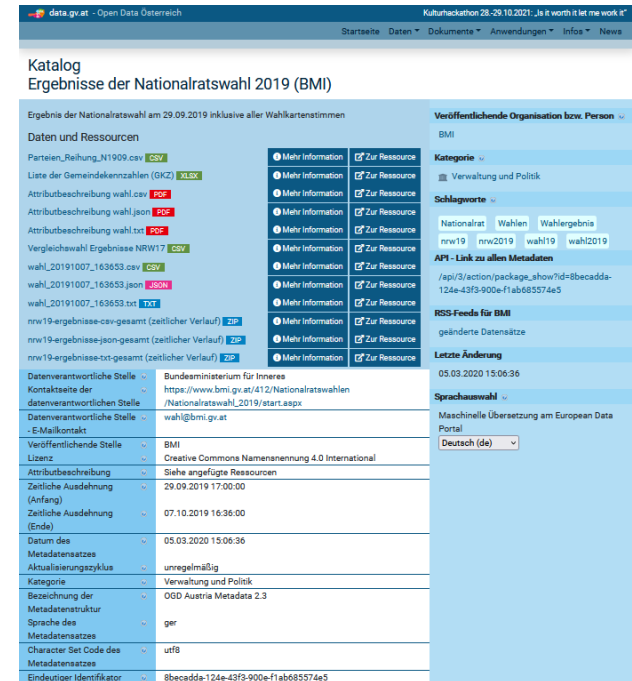
[www.offenewahlen.at/](http://www.offenewahlen.at/)  
[www.data.gv.at](http://www.data.gv.at)

## Dataset

- Austrian National Elections 2017 / 2019 with results over time and Graz districts (**still being cleaned/prepared** → Ex 02)
- Clone or download your copy from <https://github.com/tugraz-isds/datasets.git>
- Find CSV files in <datasets>/elections\_at

## Exercises

- **01** Data modeling (relational schema)
- **02** Data ingestion and SQL query processing
- **03** Physical design tuning, query processing, and transaction processing
- **04** Large-scale data analysis (distributed query processing and ML model training)



The screenshot shows the 'Katalog Ergebnisse der Nationalratswahl 2019 (BMI)' page on the Open Data Austria website. It lists various data resources such as 'Parteien\_Reihung\_N1909.csv', 'Liste der Gemeindeganzzahlen (GIZ)', and 'Vergleichswahl Ergebnisse NRW17'. Each resource includes a file type icon (CSV, JSON, TXT, ZIP) and links for 'Mehr Information' and 'Zur Ressource'. The right sidebar shows metadata for the selected resource, including the publishing organization (BMI), category (Verwaltung und Politik), and a list of related datasets.

# Task 2.1: Schema Creation via SQL (3/25 points)

## ■ Schema creation via SQL

- Relies on lectures [04 Relational Algebra](#) and [05 Query Languages \(SQL\)](#)
- Setup DBMS PostgreSQL, and start pgAdmin (UI), or psql (terminal)
- Docker container w/ basic setup in next days
- Create database db<studentID> and setup relational schema, including primary keys, foreign keys, NOT NULL, UNIQUE

## ■ Recommended Schema

- Feel free to use and submit the provided schema
- [https://mboehm7.github.io/teaching/ws2122\\_dbs/CreateSchema.sql](https://mboehm7.github.io/teaching/ws2122_dbs/CreateSchema.sql)

## ■ Partial Results

- CreateSchema.sql

```
CREATE TABLE Locations(  
  LKey INT PRIMARY KEY,  
  LocationID VARCHAR(32) UNIQUE NOT NULL,  
  Name VARCHAR(128) NOT NULL,  
  ParentLKey INT REFERENCES Locations  
);
```

## Task 2.2 Data Ingestion via CLI (10/25 points)

### ■ Data Ingestion Program via ODBC/JDBC

- Relies on lectures [05 Query Languages \(SQL\)](#) and [06 APIs \(ODBC, JDBC\)](#)
- Write a program that performs **deduplication and data ingestion**
- Programming language of your choosing (Python, Java, C#, C++ recommended)

### ■ Data Ingestion Process

- Data: [https://github.com/tugraz-isds/datasets/tree/master/elections\\_at](https://github.com/tugraz-isds/datasets/tree/master/elections_at)
- Invoke your ingestion program as follows → script to compile and run

```
./runIngestData.sh ./Locations.csv ./Parties.csv ./Votes.csv \  
./Elections.csv <host> <port> <database> <user> <password>
```

(e.g., localhost 5432 db1234567 postgres postgres)

### ■ Partial Results

- Source code `IngestData.*`, and
- Script `runIngestData.sh`

## Task 2.3: SQL Query Processing (10/25 points)

### ■ SQL Query Processing

- Relies on lecture [05 Query Languages \(SQL\)](#)
- Expected results: [https://mboehm7.github.io/teaching/ws2122\\_dbs/Results.zip](https://mboehm7.github.io/teaching/ws2122_dbs/Results.zip)

### ■ List of Queries

- **Q01:** What is the ID of location Graz(Stadt)? (return LocationID)
- **Q02:** Select all parties of the election NR2017. (return ShortName, LongName, Ballot Position, sorted ascending by Ballot Position with NULLs last)
- **Q03:** Compute the voter turnout rate (total-votes/eligible) for all districts of Graz(Stadt) in election NR2019. (return location name, turnout rate, sorted descending by turnout)
- **Q04:** Compute the top 10 locations of election NR2019 by voter turnout rate. (return name, turnout, sorted descending by turnout)
- **Q05:** Which parties from the election NR2019 did not participate in NR2017? (return ShortName, LongName, sorted ascending by ShortName)

## Task 2.3: SQL Query Processing (10/25 points)

### ■ List of Queries, cont.

- **Q06:** Compute the support (fraction of received votes) in NR2019 of all parties that received more than 4% of votes.  
 (return ShortName, support; sorted descending by support)
- **Q07:** Find the parties that won (with highest support rate) at least one location in NR2019.  
 (return ShortName, count of won locations, total Austrian support rate; sorted descending by won locations)
- **Q08:** Compare for each state of Österreich (e.g., Steiermark) the total number of votes with the sum of votes in all last-level child locations.  
 (return the state name, total votes, sum of votes in child locations, difference in votes, sorted ascending by state name)

### ■ Partial Results

- SQL Script for each query: Q01.sql, Q02.sql, ..., Q08.sql



## Task 2.4: Query Plans (2/25 points)

### ■ Explain Query Plans

- Relies on lecture [04 Relational Algebra](#) and [05 Query Languages \(SQL\)](#)
- Obtain and **analyze execution plans** of Q05

### ■ Example

#### EXPLAIN VERBOSE

```
SELECT L.location, count(*)
  FROM Participant P,
       Locale L
 WHERE P.lid = L.lid
 GROUP BY L.location
 HAVING count(*)>1
```

```
"HashAggregate (...)" // grouping
" Output: l.location, count(*)"
" Group Key: l.location"
" Filter: (count(*) > 1)" // selection
" -> Hash Join (...)" // join
"   Output: l.location" // projection
"   Hash Cond: (l.lid = p.lid)"
"   -> Seq Scan on Locale l (...)"
"     Output: l.lid, l.location"
"   -> Hash (...)"
"     Output: p.lid" // projection
"   -> Seq Scan on Participant p (...)"
"     Output: p.lid"
```

### ■ Partial Results

- ExplainQ06.sql

# Conclusions and Q&A

## ■ Summary

- History and fundamentals of the **Structured Query Language (SQL)**
- Awareness of **XML and JSON** (data model and querying)

## ■ Exercise Submissions

- **Exercise 1:** Nov 02 + 7 late days, grading in progress
- **Exercise 2:** Nov 30, published Nov 09 (preview Nov 06)

## ■ Next Lectures (Part A)

- **06 APIs (ODBC, JDBC, OR frameworks)** [Nov 15]
- **07 Physical Design and Tuning** [Nov 22]
- **08 Query Processing** [Nov 29]
- **09 Transaction Processing and Concurrency** [Dec 06]