CS480

Database Systems

4 - SQL

Course webpage

Boris Glavic

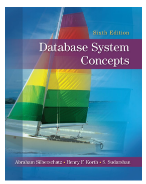
bglavic@uic.edu

SQL

SQL Overview

SQL Overview

Textbook



Textbook: Chapter 3

History

- IBM Sequel language developed as part of the **System R** project at IBM Research
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL
  - SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016, SQL:2019-2020, SQL:2023
- Advanced systems implement all (most of) SQL-92 and selected features from later standards
- Many systems use **non-standard syntax** for some language features / implement their own **proprietary features**

Language Structure

DDL

- The Data Definition Language (DDL) part of the language is for managing the **schema** of a database

DML

- The Data Manipulation Language (DML) part of the language is for changing and querying the database **instance**

Bag vs. Set Semantics

Set semantics

- The formal relational model is typically defined using relations that are sets (set semantics)

Bag semantics

- SQL uses a model of relations called bag semantics where relations are **bags** (**multisets**) of tuples
  - we allow **duplicates**

Bag vs. Set Semantics Example

Set Semantics

Orders

Item	Quantity
Lawnmower	3
Lawnmower	2
Shovel	1

Bag Semantics

Orders

Item	Quantity
Lawnmower	3
Lawnmower	3
Lawnmower	2
Shovel	1
Shovel	1
Shovel	1

SQL Overview

Queries

SELECT-FROM-WHERE

- SQL queries are structured into blocks
- The clauses of a block are identified through English language keywords (e.g., WHERE)

Queries Example

Find names of students majoring in CS

```sql
SELECT name
  FROM student
 WHERE deptname = 'Comp. Sci.';
```

name
Zhang
Shankar
Williams
Brown
X Y
Lazy Bert

SQL Overview

DDL

- New tables are created using the CREATE TABLE statement

```sql
CREATE TABLE instructor (
  ID char(5) PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  deptname VARCHAR(20),
  salary NUMERIC(8,2),
);
```


DML

Insert

- add new rows into a table

Update

- modify rows that fulfill a condition (WHERE)

Delete

- delete rows that fulfill a condition (WHERE)

DML Examples

Insert

```sql
INSERT INTO instructor
VALUES (333,'Peter Petersen', 'Comp. Sci.', 40000);
```

Update

```sql
UPDATE student SET deptname = 'CS'
 WHERE deptname = 'Computer Science';
```

Delete

```sql
DELETE FROM instructor WHERE name = 'Peter Petersen';
```


SQL Overview

Type System

Domain Types

- int - integer (size is machine / system dependent)
- char(n) - fixed length character string (exactly n characters)
- varchar(n) - variable length string (up to n characters)
- date - a date
- numeric(p,d)
  - — - fixed point number with up to p digits and d digits precision (after the dot)
  - — /e.g., numeric(7,4) can encode 100.0005, but not 1000.003 or 100.00005 /

Strict Typing

- SQL employs a **strict** type systems
- Functions and operators have fixed input types and return a fixed output type
- Functions overloaded is supported (same name, different types)
  - — *e.g., + for integers and + for floats*

```sql
1 + 1 -- returns int
1.0 + 1.0 -- return float
```


Manual Casting

CAST

- CAST (expr AS type)

```sql
CAST (1 AS NUMERIC(3,2))
```

Postgres Casting Syntax

- expr::type (postgres specific casting syntax)

```sql
1::float
```

Automatic Casting

- If the user applies a function or operator to input types for which no version of the function exists, then most databases try to cast the input tuples such that an existing function can be used

```
SELECT pg_typeof(1) AS typ1,
       pg_typeof(1.0) AS type10,
       pg_typeof(1::int + 1.0::float) AS typeplus;
```

typ1	type10	typeplus
integer	numeric	double precision

SQL Overview

Postgres Documentation

- We can only cover a small (but important) part of SQL in this course
- To lookup all the details about a language construct, you can use the excellent Postgres documentation:
  - https://www.postgresql.org/docs/16/sql.html

Queries

Queries

Query Blocks

- Queries are organized into blocks
- Blocks are in turn divided into clauses
- The **order of clauses** within a block is **fixed**
- Many clauses are **optional**
- Clauses typically start with a descriptive English keyword, e.g., `WHERE`

Query Block Structure

```
SELECT [DISTINCT] <expression_list>
[FROM <relation / subquery list>]
[WHERE <condition>]
[GROUP BY <expression_list>]
[HAVING <condition>]
[ORDER BY <expression list + directions>]
[LIMIT <n>] [OFFSET <n>]
```


Execution Order

- `FROM` - compute cross product of from clause items
- `WHERE` - filter rows based on condition
- `GROUP BY` - group on expressions
- `HAVING` - if present filter aggregation results
- `SELECT` - for each remaining row compute expressions (generalized projection)
- `DISTINCT` - remove duplicate rows
- `ORDER-BY` - sort on the result of a list of expressions
- `LIMIT` / `OFFSET` - keep `LIMIT` rows after skipping `OFFSET` rows

Execution Order

Remark

- The execution order is important for understanding the semantics of SQL, but database optimizers will often choose alternative equivalent execution orders if they are estimated to be faster.

FROM

- the FROM clauses determines which tables are accessed by the query

```
SELECT *
FROM student
LIMIT 3;
```

id	name	deptname	totcred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80

FROM Example - Multiple tables

- if multiple tables are listed, then this is treated like a cross product

```
SELECT *
FROM instructor, department
LIMIT 4;
```

id	name	deptname	salary	deptname	building	budget
10101	Srinivasan	Comp. Sci.	65000.00	Biology	Watson	90000.00
10101	Srinivasan	Comp. Sci.	65000.00	Comp. Sci.	Taylor	100000.00
10101	Srinivasan	Comp. Sci.	65000.00	Elec. Eng.	Taylor	85000.00
10101	Srinivasan	Comp. Sci.	65000.00	Finance	Painter	120000.00

FROM Alias

- Each table can be assigned an alias in the `FROM` clause
- Tables may appear more than once (with different aliases)

```
SELECT * FROM student s, instructor i;
SELECT * FROM student s1, student s2, instructor i;
```

Alias with / without AS

- in some systems you can also alias with `AS`

```
SELECT * FROM student AS s;
```


FROM Alias

- Aliases in `FROM` also allow for renaming of attributes

```
SELECT * FROM department d(name,build,moneystuff)
 LIMIT 1;
```

name	build	moneystuff
Biology	Watson	90000.00

Attribute References

- Attributes are referenced by name, e.g., `deptname`
- Optionally quantified by alias / table name, e.g., `student.name`

```
SELECT s.name
FROM student s LIMIT 1;
```

name
Zhang

Joins

- SQL supports multiple types of joins:
  - CROSS JOIN - cross product
  - [INNER] JOIN - a theta join
    - join condition `ON`: boolean condition
    - join condition `USING`: specify common columns to join on equality
  - NATURAL JOIN
  - LEFT / RIGHT / FULL OUTER JOIN

Joins Example

```
SELECT s.name, s.deptname, t.courseid, t.secid
FROM student s JOIN takes t ON (s.id = t.id)
LIMIT 3;
```

name	deptname	courseid	secid
Zhang	Comp. Sci.	CS-101	1
Zhang	Comp. Sci.	CS-347	1
Shankar	Comp. Sci.	CS-101	1

Joins Example

```
SELECT s.name, s.deptname, t.courseid, t.secid
FROM student s NATURAL JOIN takes t
LIMIT 3;
```

name	deptname	courseid	secid
Zhang	Comp. Sci.	CS-101	1
Zhang	Comp. Sci.	CS-347	1
Shankar	Comp. Sci.	CS-101	1

Joins Example

```
SELECT s.name, s.deptname, t.courseid, t.secid
FROM student s JOIN takes t USING (id)
LIMIT 3;
```

name	deptname	courseid	secid
Zhang	Comp. Sci.	CS-101	1
Zhang	Comp. Sci.	CS-347	1
Shankar	Comp. Sci.	CS-101	1

Outer Joins Example

```
SELECT s.name, s.deptname, t.courseid, t.secid, s.totcred
FROM student s LEFT OUTER JOIN takes t ON (s.id = t.id)
ORDER BY totcred ASC LIMIT 3;
```

name	deptname	courseid	secid	totcred
Snow	Physics			0
X Y	Comp. Sci.			0
Lazy Bert	Comp. Sci.			0

SELECT

- the SELECT clause consists of a **list** of **projection** expressions and optional **renaming** (`AS`)
- determines what will be returned by the query
- also handles aggregation (more on that later)

```
SELECT name AS n, age / 10 AS decades, ...
```

SELECT Example

```
SELECT name
  FROM student
 LIMIT 3;
```

name
Zhang
Shankar
Brandt

SELECT Example

```
SELECT credits * 12 AS morecred, title
  FROM course
 LIMIT 3;
```

morecred	title
48	Intro. to Biology
48	Genetics
36	Computational Biology

DISTINCT

- if DISTINCT is specified in the SELECT clause, then duplicate results are eliminated

```
SELECT DISTINCT deptname FROM student;
```

deptname
Physics
Biology
Elec. Eng.
Finance
Comp. Sci.
History
Music

WHERE

- the WHERE clause specifies a **selection condition**
- as in relational algebra selection is an expression consisting of …
  - — logical connectives AND, OR, NOT
  - — comparisons, e.g., <, =, <=, >=, …
  - — references to attributes and constants
- final result of a WHERE clause condition has to be **Boolean**

WHERE Example

```
SELECT * FROM student
 WHERE deptname = 'Comp. Sci.' OR  deptname = 'Music';
```

id	name	deptname	totcred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
76543	Brown	Comp. Sci.	58
00000	X Y	Comp. Sci.	0
99999	Lazy Bert	Comp. Sci.	0

GROUP BY + Aggregation

- The GROUP BY clause specifies which expressions to group on

Restrictions

- If a query block contains a GROUP BY clause, then only group-by expressions and aggregation functions can be used in the SELECT clause
- If the SELECT clause mentions an aggregation function, but there is no GROUP BY clause then no non-aggregated attribute references are allowed

SQL Aggregation Functions

- count
- sum
- min
- max
- avg
- and several more

Aggregation Example

```
SELECT count(*) FROM student;
```

count
15

GROUP BY + Aggregation Example

```
SELECT deptname, count(*)
  FROM student
GROUP BY deptname
LIMIT 3;
```

deptname	count
Physics	3
Biology	1
Elec. Eng.	2

GROUP BY + Aggregation Example

- group-by on expressions is allowed

```
SELECT count(*),
       (((end_hr * 60) + end_min) - (start_hr * 60 + start_min)) AS
       ↪  length
  FROM time_slot
 GROUP BY (((end_hr * 60) + end_min) - (start_hr * 60 + start_min));
```

count	length
4	75
15	50
1	150

HAVING

- the HAVING clause specifies a selection condition over group-by and aggregation results
  - not all HAVING aggregation functions have to occur in the SELECT clause

```
SELECT deptname
FROM student
GROUP BY deptname
HAVING count(*) > 3;
```

deptname
Comp. Sci.

ORDER BY

- the ORDER BY clause specifies a sort order for the results
- list of order-by expressions each optionally with a sort direction (ASC, DESC)

Remark

- For most parts, SQL treats relations as **bags**
  - ORDER BY introduces an ordering over the elements in a bag
- If two rows are incomparable wrt. the sort order, their order in the result is implementation / data dependent

ORDER BY Example

```
SELECT *
FROM student
WHERE deptname = 'Biology' OR deptname = 'Comp. Sci.'
ORDER BY deptname ASC, name DESC;
```

id	name	deptname	totcred
98988	Tanaka	Biology	120
00128	Zhang	Comp. Sci.	102
00000	X Y	Comp. Sci.	0
54321	Williams	Comp. Sci.	54
12345	Shankar	Comp. Sci.	32
99999	Lazy Bert	Comp. Sci.	0
76543	Brown	Comp. Sci.	58

ORDER BY Example (non deterministic)

```
SELECT *
FROM student
WHERE deptname = 'Biology' OR deptname = 'Comp. Sci.'
ORDER BY deptname ASC;
```

id	name	deptname	totcred
98988	Tanaka	Biology	120
12345	Shankar	Comp. Sci.	32
54321	Williams	Comp. Sci.	54
00128	Zhang	Comp. Sci.	102
00000	X Y	Comp. Sci.	0
99999	Lazy Bert	Comp. Sci.	0
76543	Brown	Comp. Sci.	58

LIMIT / OFFSET

- OFFSET specifies a number of rows to skip
- LIMIT specifies a maximal number of rows to return
  - if the query returns less rows, then only these are returned

Ordering and LIMIT / OFFSET

- If no ORDER BY clause is specified, then it is implementation / data dependent what rows are returned!
- If ORDER BY is specified, then rows are first sorted before computing LIMIT
  - top-k queries

LIMIT / OFFSET Examples

3 Departments with the most students

```
SELECT deptname, count(*) AS headcnt
FROM student
GROUP BY deptname ORDER BY headcnt DESC
LIMIT 3;
```

deptname	headcnt
Comp. Sci.	6
Physics	3
Elec. Eng.	2

LIMIT / OFFSET Examples

Department with the 2nd most number of students

```
SELECT deptname, count(*) AS headcnt
FROM student
GROUP BY deptname ORDER BY headcnt DESC
OFFSET 1 LIMIT 1;
```

deptname	headcnt
Physics	3

Queries

Set Operations in SQL

Sets vs. Bags

- In SQL each set operation comes in a **set** and a **bag** flavor:
  - A version that treats that inputs as sets
  - A version that treats the inputs as bags (indicated by appending ALL to the operation)
- SQL set operations are applied to two query blocks (or results of other set operations)

Supported operations

- UNION [ALL] - union
- EXCEPT [ALL] - set difference
- INTERSECT [ALL] - intersection

UNION Examples

Set union

```
(SELECT name FROM student
 WHERE deptname = 'Biology')
 UNION
(SELECT name FROM student
  WHERE deptname = 'Biology');
```

name
Tanaka

Bag union

```
(SELECT name FROM student
 WHERE deptname = 'Biology')
 UNION ALL
(SELECT name FROM student
  WHERE deptname = 'Biology');
```

name
Tanaka
Tanaka

UNION (set) Example

Tables

```
SELECT * FROM u;
```

a
1
1
1
2
3
3
3
4

```
SELECT * FROM v;
```

b
1
2
2
3
3

Set union

```
(SELECT * FROM u)
UNION
(SELECT * FROM v);
```

a
1
4
2
3

UNION (bag) Example

Tables

```
SELECT * FROM u;
```

a
1
1
1
2
3
3
3
4

```
SELECT * FROM v;
```

b
1
2
2
3
3

bag union

```
(SELECT * FROM u WHERE a =
↪ 2)
UNION ALL
(SELECT * FROM v);
```

a
2
1
2
2
3
3

INTERSECT (set) Example

Tables

```
SELECT * FROM u;
```

a
1
1
1
2
3
3
3
4

```
SELECT * FROM v;
```

b
1
2
2
3
3

Set intersection

```
(SELECT * FROM u)
INTERSECT
(SELECT * FROM v);
```

a
1
3
2

INTERSECT (bag) Example

Tables

```
SELECT * FROM u;
```

a
1
1
1
2
3
3
3
4

```
SELECT * FROM v;
```

b
1
2
2
3
3

bag intersection

```
(SELECT * FROM u)
INTERSECT ALL
(SELECT * FROM v);
```

a
1
3
3
2

EXCEPT (set) Example

Tables

```
SELECT * FROM u;
```

a
1
1
1
2
3
3
3
4

```
SELECT * FROM v;
```

b
1
2
2
3
3

Set difference

```
(SELECT * FROM u)
EXCEPT
(SELECT * FROM v);
```

a
4

EXCEPT (bag) Example

Tables

```
SELECT * FROM u;
```

a
1
1
1
2
3
3
3
4

```
SELECT * FROM v;
```

b
1
2
2
3
3

bag difference

```
(SELECT * FROM u)
EXCEPT ALL
(SELECT * FROM v);
```

a
1
1
3
4

Queries

Queries with Multiple Blocks

- Subqueries allow us to use query blocks inside the `FROM` clause
- Subqueries always have to have an alias

Semantics

- The database evaluates queries bottom-up
- Once the result of a subquery has been evaluated, we can (conceptually) treat it just like a table in the database

Subquery Example

Number of departments with a certain number of students

```
SELECT count(*) AS numdep, numst
FROM (SELECT count(*) AS numst, deptname
      FROM student
      GROUP BY deptname) AS ns
GROUP BY numst;
```

numdep	numst
1	6
4	1
1	3
1	2

Subquery Example

Number of students and other information for departments

```
SELECT d.deptname, numst, building, budget
FROM (SELECT count(*) AS numst, deptname
      FROM student
      GROUP BY deptname) AS ns,
     department d
WHERE ns.deptname = d.deptname
```

deptname	numst	building	budget
Biology	1	Watson	90000.00
Comp. Sci.	6	Taylor	100000.00
Elec. Eng.	2	Taylor	85000.00
Finance	1	Painter	120000.00

Queries

What Are Nested Subqueries?

Nested subqueries

- Nested subqueries allow queries to be nested inside **scalar expressions**
  - *e.g., inside a `WHERE` clause condition or `SELECT` clause expression*
  - The most common use is in `WHERE` clause conditions

Types of Subqueries - Scalar Subqueries

Scalar subqueries

- The query is required to return a single row

Note!

- returning more than one row is a runtime error!

Semantics

- The result of the subquery is substituted into the expression
- Then the expression is evaluated as usual

```
SELECT *
  FROM student
 WHERE totcred = (SELECT max(totcred) FROM student);
```


Correlations

What are correlated attributes

- Referencing attributes from the **outer** query within the subquery

Semantics of correlated references

- For each row returned by the `FROM` clause of the outer query:
  - Substitute correlated attribute reference with values from that row
  - Evaluate the subquery

Correlation Example

```
SELECT name, deptname
FROM student s
WHERE totcred = (SELECT max(totcred)
                   FROM student o
                  WHERE s.deptname = o.deptname)
LIMIT 3;
```

name	deptname
Zhang	Comp. Sci.
Brandt	History
Chavez	Finance

Correlation Example

```
SELECT name, deptname
FROM student s
WHERE totcred = (SELECT max(totcred)
                   FROM student o
                  WHERE s.deptname
                        = o.deptname)
LIMIT 3;
```

replace `s.deptname` with `'Comp.  Sci.'`

```
(SELECT max(totcred)
   FROM student o
  WHERE 'Comp. Sci.' = o.deptname)
```

...	Zhang	Comp. Sci.	...
...	Shankar	Comp. Sci.	...
...	Brandt	History	...
...	....	...	...

EXISTS Subquery

Exists subqueries

- `EXISTS q` for a query `q`

Semantics

- Returns `true` if the query returns a non-empty result

EXISTS Subquery Example

```
SELECT *
  FROM student s
 WHERE EXISTS (SELECT * FROM takes t WHERE t.id = s.id)
LIMIT 5;
```

id	name	deptname	totcred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56

IN Subqueries

IN subqueries

- `e in q` for a query `q` that returns a single column and expression `e`

Semantics

- returns `true` if **any** of the answers of `q` is equal to `e`

IN Subquery Example

```
SELECT *
  FROM student s
 WHERE s.id IN (SELECT id FROM takes)
LIMIT 3;
```

id	name	deptname	totcred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80

ANY / ALL Subqueries

ANY / ALL subqueries

- `e ANY/ALL op q` where
  - `q` is a query returning a single column
  - `e` is an expression
  - `op` is a comparison operator, e.g., <

Semantics

- **ANY** returns **true** if the comparison evaluates to **true** for **at least one** result of `q`
  - **IN** is equivalent to `= ANY`
- **ALL** returns **true** if the comparison evaluates to **true** for **all** results of `q`

ANY / ALL handling `null` values

- for **ALL**
  - if at least one comparison returns **false**, the result if **false**
  - if all comparisons return **true**, the result is **true**
  - otherwise (all **null** or some **true** and some **null**) the result is **null**

Where Can We Use Nested Subqueries?

- subqueries can be used anywhere an expression is allowed

```
SELECT EXISTS(SELECT * FROM student);
```

exists
t

```
SELECT count(*) FROM student s1
 GROUP BY (SELECT count(*)
            FROM student s2
           WHERE s1.deptname = s2.deptname);
```

count
6

Queries

What Are Window Functions?

Window functions

- window functions are aggregation functions that are applied to subsets of table called **windows**
- in contrast to "regular" aggregation, one result is returned for each `FROM` clause tuple
- the OVER clause specifies which tuples belong to a window

Semantics

- for each row `r` from the `FROM` clause determine the subset of the `FROM` clause tuples that belong `r`'s window
- calculate the aggregation function over the window

Window Function Example

```
SELECT name,
       count(*) OVER (PARTITION BY deptname) AS depheadcnt
FROM student
LIMIT 4;
```

name	depheadcnt
Tanaka	1
Shankar	6
Zhang	6
Williams	6

OVER Clause - Syntax

Syntax

```
OVER ([PARTITION BY attrs] [ORDER BY orderexprs] [windowspec])
```


OVER Clause - Semantics

Semantics

- PARTITION BY works like `GROUP BY` for regular aggregation
  - the window is restricted to rows with the same `PARTITION BY` values as the current row
- ORDER BY sorts the rows
  - if no `windowspec` is provided then all rows <= the current row are included in the window
- windowspec determines which rows to included based on their sort order position (based on `ORDER BY`) relative to the current row

Window Specification

ROWS BETWEEN lower bound AND upper bound

- provides a number of rows smaller than (**lower bound**) and larger than (**upper bound**) the current row to include in the window
- keywords `UNBOUNDED PRECEDING` and `UNBOUNDED FOLLOWING` are used to include all smaller / larger rows

RANGE BETWEEN lower bound AND upper bound

- provides a range of values to include in the window
- all rows that have values within the range are included

Window Specification

GROUPS BETWEEN lower bound AND upper bound

- provides a number of values to include in the window
- all rows that have values within the range are included

Window Function Examples

```sql
SELECT name, count(*) OVER (PARTITION BY deptname)
FROM student
LIMIT 3;
```

name	count
Tanaka	1
Shankar	6
Zhang	6

Window Function Examples

```sql
SELECT name, count(*) OVER (ORDER BY name)
FROM student
LIMIT 3;
```

name	count
Aoi	1
Bourikas	2
Brandt	3

Window Function Examples

```sql
SELECT name, deptname, count(*) OVER
         (PARTITION BY deptname ORDER BY name)
FROM student
ORDER BY deptname LIMIT 3;
```

name	deptname	count
Tanaka	Biology	1
Brown	Comp. Sci.	1
Lazy Bert	Comp. Sci.	2

Window Function Examples

```sql
SELECT name, count(*) OVER
         (ORDER BY name
         ROWS BETWEEN 1 PRECEDING AND UNBOUNDED FOLLOWING) AS cnt
FROM student
ORDER BY cnt DESC LIMIT 4;
```

name	cnt
Bourikas	15
Aoi	15
Brandt	14
Brown	13

Window Function Examples

```sql
SELECT day, starthr, startmin, count(*) OVER
         (ORDER BY "day", starthr, startmin
         ROWS BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING) AS cnt
FROM timeslot
ORDER BY day, starthr, startmin LIMIT 4;
```

day	starthr	startmin	cnt
F	8	0	0
F	9	0	1
F	11	0	2
F	13	0	3

Queries

Common Table Expressions

- A Common Table Expression (CTE) assigns a name to a query using the `WITH` clause

```sql
WITH query1 AS [query],
     query2 AS [query],
     ...
     queryn AS [query]
SELECT ...
```

Caveats

- In contrast to views, CTEs are only valid within the scope of a query
- Query Q_i can refer to any query Q_j for $j < i$
- The final `SELECT` statement can refer to any Q_i

CTE Example

```sql
WITH numst AS
    (SELECT count(*) AS nums, deptname
       FROM student GROUP BY deptname)
SELECT * FROM numst WHERE nums = (SELECT max(nums) FROM numst);
```

nums	deptname
6	Comp. Sci.

Queries

What are views?

- Views enable us to **assign a name** to a **query**
- Views can be referenced in queries just like tables
- Views can be ...
  - **non-materialized** or **virtual**: the database does not store the result of the query, but only the definition of the view
  - **materialized**: the database stores the result of the query

Non-materialized Views

How do non-materialized views work?

- The DBMS just stores the definition (the query) in its catalog
- Whenever the view is referenced in a query, we replace it with its definition

Advantages

- We do not need to keep the query result up to date

Disadvantages

- Whenever we reference the view in a query it has to be computed from scratch

Non-materialized Views Example

```
SELECT * FROM numstud;
```

deptname	numst
Physics	3
Biology	1
Elec. Eng.	2
Finance	1
Comp. Sci.	6
History	1
Music	1

Non-materialized Views Example

```
EXPLAIN SELECT * FROM numstud;

QUERY PLAN
HashAggregate  (cost=1.23..1.30 rows=7 width=17)
  Group Key: student.deptname
  ->  Seq Scan on student  (cost=0.00..1.15 rows=15 width=9)
```


Materialized Views

How do materialized views work?

When the view is defined, the database system evaluates the query and stores the query result in the database as a table

Advantages

- The database can read the stored query results instead of having to reevaluate the query

Materialized Views

Disadvantages

- When the tables accessed by the view are updated, then the stored query result becomes stale
  - The stored result is no longer the same as evaluating the view's query over the current state of the database
- Materialized views have to be refreshed manually by running

```
REFRESH MATERIALIZED VIEW viewname;
```


Materialized Views Example

```
CREATE MATERIALIZED VIEW numst AS
(SELECT deptname, count(*) AS numst FROM student GROUP BY deptname);
```


Materialized Views Example

```
SELECT * FROM numst WHERE deptname = 'Comp. Sci.';
```

deptname	numst
Comp. Sci.	6

```
INSERT INTO student
VALUES ('55555', 'Peter Petersen', 'Comp. Sci.', 45);
SELECT * FROM numst WHERE deptname = 'Comp. Sci.';
```

deptname	numst
Comp. Sci.	6

Materialized Views Example

```
REFRESH MATERIALIZED VIEW numst;
SELECT * FROM numst WHERE deptname = 'Comp. Sci.';
```

deptname	numst
Comp. Sci.	7

Queries

What are Recursive Queries?

- Recursive queriesallows us to express recursive computations in SQL
- Recursive queries consist of:
  - a initialization part that returns the initial state of the query result table
  - a recursive step that takes as input the state of the query result table computed in the previous iteration and computes new tuples to be added to the query result

Syntax

- Recursive queries are defined as common table expressions

```
WITH RECURSIVE myrec AS (
  [q-init] -- intialization query
  UNION [ALL]
  [q-recursive-step] -- recursive step
  )
...
```


Fix Points

Definition (Fix point)

Consider a function $f : \mathbb{D} \rightarrow \mathbb{D}$ from some domain $\mathbb{D}$ to itself. We call $x \in \mathbb{D}$ a fix point for f iff:

$$x = f(x)$$

Fix Point Iteration

Definition (Fix point iteration)

Consider an **initial state** x_0 and **function** f we define the following **iteration sequence** for $n > 0$:

$$x_n = f(x_{n-1})$$

If this sequence has a fix point, i.e., $x_{n+1} = x_n$ for some $n \geq 0$, then we call $x_* = x_n$ the fix point of the iteration.

Existence Of Fix Points

Existence of fix points

- Some sequences **do not** have fix points
- Some sequences take **infinitely** many steps to reach a fix point

Diverging sequence

$$x_0 = 1$$
$$f(x) = x + 1$$

- diverges (no fix point)

[1, 2, 3, 4, 5, 6, 7]

Existence Of Fix Points

Infinite convergence

$$x_0 = 1$$
$$g(x) = 1 - \frac{x}{2}$$

- reaches fix point after infinitely many steps

[1, 0.5, 0.75, 0.625, 0.6875, 0.65625, 0.671875]

Existence Of Fix Points

Periodic

$$x_0 = 1$$
$$h(x) = 1 - x$$

- periodic (no fix point)

[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]

Recursive Queries Semantics

Fix point iteration

- Database D, initialization query Q_{init}, recursive step query Q_{rec}

$$D_0 = Q_{init}(D) \quad (1)$$
$$D_n = Q_{rec}(D_{n-1}) \cup D_{n-1} \quad (2)$$

Recursive Query Example

Indirect Prerequisites

```
WITH RECURSIVE inpre AS (
  (SELECT * FROM prereq) -- direct prereqs
  UNION
  (SELECT i.courseid, p.prereqid -- recursive step
     FROM inpre i, -- reference to previous iteration result
          prereq p WHERE i.prereqid = p.courseid))
SELECT * FROM inpre;
```

courseid	prereqid
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101

Recursive Query Restrictions

Linear Queries

- In Q_{rec} the result from the previous iteration can only be referenced once!
- This is called linear recursion

```
WITH RECURSIVE q AS (
  SELECT ... -- init
  UNION
  (SELECT ... FROM q q1, q q2 -- not allowed!
    WHERE ...))
SELECT * FROM q;
```


Queries

Common Functions

String concatenation

- || concatenates two strings

```
SELECT 'a' || 'b' AS str;
```

str
ab

Conditions with CASE

- return THEN expression result for first WHEN that evaluates to true
- if none evaluate to true, then return the ELSE expression result (or NULL if ELSE is omitted)

```
SELECT CASE WHEN (1 < 1) THEN 1
            WHEN (2 < 3) THEN 2
            ELSE 3
       END;
```

case
2

CASE Examples

```
SELECT name,
       CASE WHEN totcred > 100 THEN 'ready'
            ELSE 'notready'
       END AS cangraduate
FROM student
LIMIT 3;
```

name	cangraduate
Zhang	ready
Shankar	notready
Brandt	notready

IN with constant

- check whether value appears in a list of values

```
SELECT 2 IN (1,2,5,6) AS havetwo;
```

havetwo
t

Queries

Nulls In Expression

- scalar operations and comparisons with NULL return NULL

```
SELECT 1 + NULL AS x;
```

x

Operations Targeting Nulls

COALESCE

- COALESCE returns the first non-null input (or null if all inputs are null)

```
SELECT COALESCE(NULL,3,NULL) AS firstnonnull;
```

firstnonnull
3

Operations Targeting Nulls

IS NULL

- IS NULL returns true if its input is null

```
SELECT (1 + NULL) IS NULL AS isnull;
```

isnull
t

Nulls In WHERE / HAVING and Other Condition

WHERE / HAVING

- if the condition evaluates to NULL or false the tuple is removed

CASE

- the same applies to the condition of the CASE construct

Nulls And Aggregation

Aggregation Functions

- NULL values are ignored

Group-by values

- NULL is treated as a regular value

DDL

DDL

`CREATE TABLE` statement

- the CREATE TABLE statement creates a new table

```
create_table := CREATE TABLE <table_name> (
  <table_item>+
);

table_item := <column_def> | <constraint>

column_def :- <name> <data_type> [<constraints>]
constraint := NOT NULL | UNIQUE | PRIMARY KEY | CHECK <cond> | DEFAULT <valu
```


Creating Tables - Example

- `REFERENCES` defines a single column `FOREIGN KEY`
- `CHECK` defines a boolean condition over values of a single row that has to be fulfilled
- `NOT NULL` disallows `NULL` values in a column

```
CREATE TABLE courserating (
  studentid VARCHAR(5) NOT NULL REFERENCES student,
  courseid VARCHAR(8) NOT NULL REFERENCES course,
  rating NUMERIC(2,1) CHECK (rating BETWEEN 0.0 AND 5.0),
  PRIMARY KEY(courseid, studentid)
);
```


DDL

`ALTER TABLE` statement

- the ALTER TABLE statement changes the definition of a table
- many different changes are possible
- here we just review a few
- `https://www.postgresql.org/docs/current/sql-altertable.html`

Altering Columns

Adding / deleting columns

- newly added columns will be populated with `NULL` values or the `DEFAULT` value of the new column if specified

```
ALTER TABLE student ADD COLUMN age INT;
ALTER TABLE student DROP COLUMN age;
```


Altering Columns

Renaming columns

- Changing the name of a column

```
ALTER TABLE student RENAME COLUMN name TO fullname;
SELECT * FROM student LIMIT 1;
```

id	fullname	deptname	totcred
00128	Zhang	Comp. Sci.	102

Altering Constraints

Dropping Constraints

- drop a constraint by name (have to lookup system-generated names if need be)

```
ALTER TABLE courserating
  DROP CONSTRAINT courserating_pkey;
```


Altering Constraints

Adding Constraints

- add a new named constraint

```
ALTER TABLE courserating
  ADD CONSTRAINT courserating_key
  PRIMARY KEY (courseid, studentid);
```

DML

DML

The Data Manipulation Language (DML) part of SQL provides language constructs for inserting, deleting, and updating rows of a table.

DML

Insert With VALUES

Inserting Rows

- In this form of the INSERT statement, one or more rows are provided to be inserted into the table

```
INSERT INTO <table_name> VALUES <row_list>;
```

```
INSERT INTO courserating
VALUES
('00128', 'BIO-101', 3.5),
('00128', 'BIO-301', 3.7);
```


Insert Query Results

Inserting Query Results

- In this form of the INSERT statement, the result of a query is inserted into a table
- The query has to return the same number of columns as the table and data types have to be compatible

```
INSERT INTO courserating
  (SELECT s.id, c.courseid, 2.0 AS rating
     FROM student s, course c
    WHERE s.deptname = c.deptname AND s.deptname = 'Comp. Sci.');
```


DML

DELETE Statement

Deletion

- The DELETE statement removes all rows that fulfill the WHERE clause condition of the statement

```
DELETE FROM courserating
 WHERE courseid = 'CS-101';
```


Nested subqueries

DELETE statements can use nested subqueries in the WHERE clause

```
DELETE FROM courserating
 WHERE studentid IN (SELECT id FROM student WHERE deptname = 'Comp.
  ↪ Sci');
```


DML

UPDATE Statement

- The UPDATE statement modifies the values of all rows for which the WHERE clause evaluates to true
- The SET specifies how to update rows using a list of statements of the form:

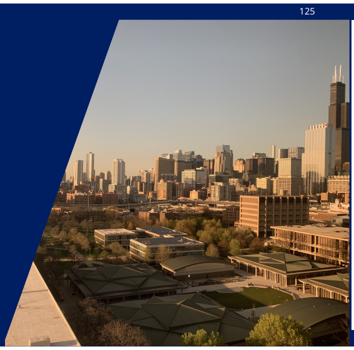
```
attr = expr
```

- expr is an expression over the attributes of the table and is evaluated using the values of the current row
- Nested subqueries can be used in both the SET and WHERE clause

Update Examples

```sql
UPDATE courserating
   SET rating = CASE WHEN rating < 3.0
                     THEN rating + 1.0
                     ELSE 4.0
                END
WHERE courseid = 'CS-101';
```


Database Catalog



Database Catalog

What is the database catalog?

- The DBMS stores schema information in the database catalog
- Most DBMS make the catalog available for querying as tables (or views)
- In postgres, every database has the information_schema and the pg_catalog schemata
  - The default schema is public

Querying the catalog

```sql
SELECT * FROM pg_tables WHERE tablename = 'student';
```

schemaname	tablename	tableowner	tablespace	hasindexes	hasrules	hastriggers	rowsecurity
public	student	postgres		t	f	t	f

Querying the catalog

```sql
SELECT table_schema AS schema,
       table_name AS tablename,
       table_type AS ttype
  FROM information_schema.tables LIMIT 3;
```

schema	tablename	ttype
public	x	BASE TABLE
public	department	BASE TABLE
public	course	BASE TABLE

Further Reading

- **catalog tables**: https://www.postgresql.org/docs/16/catalogs.html
- **catalog views**: https://www.postgresql.org/docs/16/views.html

Query Execution, Optimization & Explain

Query Execution, Optimization & Explain

What is Query Execution?

- The DBMS features multiple implementations for each relational algebra operator
  - these differ in resource requirements (memory, I/O cost, CPU cost)
  - may only be applicable under certain conditions
- The execution engine takes a plan (a tree of operators that implement a query) and evaluates the plan to produce query results

Query Execution, Optimization & Explain

Some Important Operator Implementations

- **table access**
  - sequential scan - scan through all rows of the table
  - index scan - retrieve rows from a table fulfilling a condition using an available index
- **joins**
  - nested loop join - for each row from the left table scan through all rows of the right table
  - hash join - for an equality join, build a hash table over one of the tables (with join attributes as key) and for each row of the other table probe the hash table to find matches
  - merge join - sort both input tables on the join attributes and then simultaneously scan through the table

Some Important Operator Implementations

- **aggregation**
  - group-by with hashing - store partial results for each group in a hashtable indexed by group. For each row update the group's aggregation result in the hash table (or create a new one if we do not have an entry for the group yet)
  - group-by with sorting - sort the input table on the group-by attributes, then scan through the input table once maintaining a partial aggregation result for the current group. Once we observe a new group, output the result for the current group and reinitialize the aggregation result for the next group.

What is Query Optimization?

- As mentioned before, the database optimizer ...
  - generates multiple plans for a query
  - estimates the execution cost for each plan
  - selects the plan with the lowest estimated cost
- query optimization **101**
  - The database translates the query into relational algebra (or something every similar)
  - For each logical operator in a query we choose an implementation (a physical operator)
  - The database also applies equivalence preserving transformations to transform the query into equivalent query with a different operator tree

Query Execution, Optimization & Explain

What Is an Index?

Indexes

- An index is a data structure that enables **fast access** to the **rows of a table** based on the **values** of the rows in **one or more attributes**
- Index structures in databases are:
  - **disk-based**: the index is materialized on disk and can be larger than available main memory
  - **optimized to minimize I/O**: data structures are designed to reduce the amount of I/O needed to access data

In-memory Index Structures

You probably already know several in-memory index structures:

- Balanced search trees (e.g., red-black tree, AVL-tree)
- Hash tables (e.g., Python dictionary)

Common Disk-based Index Structures

- B-tree
  - a balanced search tree with large fan out and nodes sized to be a multiple of disk page size
- Extensible hashing
  - hash tables with buckets that are multiple of the disk page size large and can grow without full reorganization

Index Trade-offs

Faster Access

- In terms of O-notation:
  - tree-based indexes are have logarithmic look-up runtime ($O(\log n)$)
  - hash-based indexes have expected constant time look-up ($O(1)$)
- Without an index we have to scan through the whole table ($O(n)$)

Overhead For updates

- When we modify the database, then indexes have to be updated too
  - This slows down updates

Storage overhead

- Indexes take up extra storage on disk and in memory

Index Trade-offs

Supported predicates

- Not all conditions can be checked using index structures
- **B-tree**: order-based and equality comparisons (<, <, =, >, >)
- **Hash-index**: equality comparisons

Higher Constant Factors

- The cost per row we are accessing (the constant factor) is significantly higher for indexes than for just scanning through a table
- Details depend on machine / DBMS / size of rows / ...
  - realistic numbers: if the query needs more than **0.1% of the table**, then the index would be slower

Index Definition in SQL

Defining & Dropping Indexes

- `CREATE INDEX ..` and `DROP INDEX ...`
- Postgres documentation:
  https://www.postgresql.org/docs/current/sql-createindex.html

```
CREATE INDEX <name> ON <table> (<col_or_expr_list>);
```

```sql
CREATE INDEX student_name ON student (name);
```


Query Execution, Optimization & Explain

What is EXPLAIN?

- Postgres allows us to inspect which execution plan its optimizer selected for a query from within SQL using the EXPLAIN statement

```
EXPLAIN SELECT * FROM student;

QUERY PLAN
Seq Scan on student  (cost=0.00..1.15 rows=15 width=25)
```

```
EXPLAIN SELECT * FROM largestudent WHERE id = '00128';

QUERY PLAN
Index Scan using largestudent_pkey on largestudent  (cost=0.42..8.44 rows=1 width=38)
  Index Cond: (id = '00128'::text)
```


EXPLAIN Examples

```
EXPLAIN SELECT * FROM student s, takes t WHERE s.id = t.id;

QUERY PLAN
Hash Join  (cost=1.34..2.63 rows=22 width=53)
  Hash Cond: ((t.id)::text = (s.id)::text)
  ->  Seq Scan on takes t  (cost=0.00..1.22 rows=22 width=28)
  ->  Hash  (cost=1.15..1.15 rows=15 width=25)
        ->  Seq Scan on student s  (cost=0.00..1.15 rows=15 width=25)
```


EXPLAIN Examples

```
EXPLAIN SELECT courseid, count(*) numreg FROM student s, takes t
        WHERE s.id = t.id GROUP BY courseid;

QUERY PLAN
HashAggregate  (cost=2.74..2.86 rows=12 width=15)
  Group Key: t.courseid
  ->  Hash Join  (cost=1.34..2.63 rows=22 width=7)
        Hash Cond: ((t.id)::text = (s.id)::text)
        ->  Seq Scan on takes t  (cost=0.00..1.22 rows=22 width=13)
        ->  Hash  (cost=1.15..1.15 rows=15 width=6)
              ->  Seq Scan on student s  (cost=0.00..1.15 rows=15 width=6)
```


Maintaining Statistics

- The optimizer relies on statistics about the number of rows and value distributions of attributes in a table
- You can force postgres to update its statistics using the `ANALYZE` statement

```sql
ANALYZE;
```


EXPLAIN ANALYZE

Issues with EXPLAIN

- `EXPLAIN` does **not** execute the query
  - if the optimizer's estimations are off, you will not know!

EXPLAIN ANALYZE

- The `EXPLAIN ANALYZE` version of `EXPLAIN` executes the query and shows actual numbers in addition to the estimates

EXPLAIN ANALYZE Examples

```
EXPLAIN ANALYZE SELECT * FROM student;

QUERY PLAN
Seq Scan on student  (cost=0.00..1.15 rows=15 width=25) (actual time=0.005..0.007 rows=15 loops=1)
Planning Time: 0.622 ms
Execution Time: 0.047 ms
```

```
EXPLAIN ANALYZE SELECT * FROM student WHERE id = '00128';

QUERY PLAN
Seq Scan on student  (cost=0.00..1.19 rows=1 width=25) (actual time=0.007..0.010 rows=1 loops=1)
  Filter: ((id)::text = '00128'::text)
  Rows Removed by Filter: 14
Planning Time: 0.582 ms
Execution Time: 0.037 ms
```


EXPLAIN ANALYZE Examples

```
EXPLAIN ANALYZE SELECT * FROM student s, takes t WHERE s.id = t.id;

QUERY PLAN
Hash Join  (cost=1.34..2.63 rows=22 width=53) (actual time=0.037..0.048 rows=22 loops=1)
  Hash Cond: ((t.id)::text = (s.id)::text)
  ->  Seq Scan on takes t  (cost=0.00..1.22 rows=22 width=28) (actual time=0.003..0.005 rows=22 loops=1)
  ->  Hash  (cost=1.15..1.15 rows=15 width=25) (actual time=0.016..0.016 rows=15 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  Seq Scan on student s  (cost=0.00..1.15 rows=15 width=25) (actual time=0.005..0.007 rows=15 loops=1)
Planning Time: 1.612 ms
Execution Time: 0.087 ms
```


EXPLAIN ANALYZE Examples

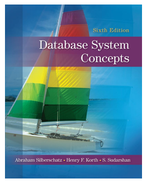
```
EXPLAIN ANALYZE SELECT courseid, count(*) numreg
        FROM student s, takes t
        WHERE s.id = t.id GROUP BY courseid;

QUERY PLAN
HashAggregate  (cost=2.74..2.86 rows=12 width=15) (actual time=0.050..0.054 rows=12 loops=1)
  Group Key: t.courseid
  Batches: 1  Memory Usage: 24kB
  ->  Hash Join  (cost=1.34..2.63 rows=22 width=7) (actual time=0.029..0.039 rows=22 loops=1)
        Hash Cond: ((t.id)::text = (s.id)::text)
        ->  Seq Scan on takes t  (cost=0.00..1.22 rows=22 width=13) (actual time=0.003..0.005 rows=22 loops=1)
        ->  Hash  (cost=1.15..1.15 rows=15 width=6) (actual time=0.011..0.012 rows=15 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Seq Scan on student s  (cost=0.00..1.15 rows=15 width=6) (actual time=0.003..0.005 rows=15 loops=1)
Planning Time: 1.730 ms
Execution Time: 0.120 ms
```

Access Control

Access Control

Textbook



Textbook: Chapter 4.6 (authorization)

Why Access Control?

- Most organizations store several types of data
- Not all users of the database should …
  - — get access to all data
  - — should be allow to update data
  - — should be allowed to change the database's schema
- The solution is access control (part of the SQL standard)

Access Control Permissions

Common permissions

- select - query (read) the data (no modifications)
- insert - insert new data (no delete or update)
- update - updates, but no deletion of data
- delete - delete data
- all priviledges - all applicable privileges

Access Control

Grant

- grant gives privileges to users (or roles as described later).

```
GRANT <priviledges> ON <database_object> TO <user/role_list>;
```

```
CREATE USER hrstaff_peter;
CREATE USER hrstaff_bob;
GRANT select, update ON student TO hrstaff_peter, hrstaff_bob;
```


Grant with Grant Option

- A user having role **X** for object **O** can grant this to any user **U**
  - — This does not give **U** the privilege to grant **X** on **O** to user users
- To allow **U** to bestow this priviledge to other users, we have to use specify this explicitly using `WITH GRANT OPTION`

```
GRANT select ON student TO testuser WITH GRANT OPTION;
```


Revoke

- revoke removes privileges

```
REVOKE <priviledge_list> ON <database_object> FROM <user/role_list>;
```

```
CREATE USER testuser;
GRANT select ON student TO testuser;
REVOKE select ON student FROM testuser;
```


Access Control

Users and Permissions

Users

- Most DBMS allow users to be defined (typically independent of OS users)
- Creating users:
  https://www.postgresql.org/docs/current/sql-createuser.html

```
CREATE USER name WITH <options>

options := PASSWORD <password> | SUPERUSER | ...
```

Superusers

- In Postgres, any users created with SUPERUSER has all permissions!
- In real production environments use this with extreme care
- … but quite useful for our purpose

Roles

Role

- roles allow priviledges to be grouped
  - grant privileges to a role
  - grant role to a user

Role Example

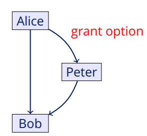
- Graduate affairs (GA) personal should have update access to student and enrollment information and read access to courses
- instead of giving these privileges to each individual HR use, grant them to an HR role and then just grant the role to the user

```
CREATE USER peter;
CREATE USER bob;
CREATE USER alice;
CREATE ROLE GA;
GRANT GA TO peter, bob, alice;
GRANT all privileges ON student, takes TO GA;
GRANT select ON course TO GA;
```


Access Control

Modeling Indirect Privileges As Graphs

- **Alice** grants a right to **Peter** with the **grant option**
- **Alice** grants the same right to **Bob**
- **Peter** also grants the right to **Bob**



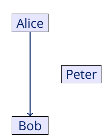
RESTRICT and CASCADE

- Indirect privileges are handled by REVOKE based on whether RESTRICT or CASCADE is used
  - RESTRICT - if indirect privileges would be affected, the database rejects the statement
  - CASCADE - if indirect privileges are affected, then they are revoked too

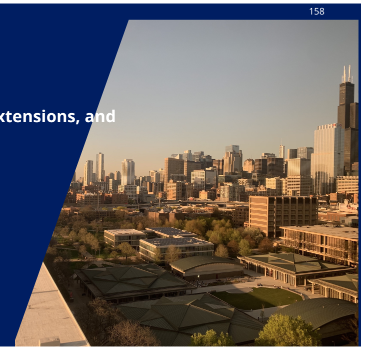
```
REVOKE select ON student FROM testuser RESTRICT;
REVOKE select ON student FROM testuser CASCADE;
```


Example RESTRICT and CASCADE

- **Alice** revokes the privilege from **Peter**
- This cascades to **Bob** if the CASCADE option is used for REVOKE (RESTRICT would fail instead)
- **Bob** still retains the right as it was also directly granted by **Alice**



Triggers, Procedural Extensions, and UDFs



Triggers, Procedural Extensions, and UDFs

Overview

Issues & Inconveniences in SQL Queries

- No **modularity** apart from views
  - but views do not have parameters → they are inflexible
- No **state** and **procedural constructs** (looping, conditional execution)

SQL Procedures & Functions

Basic Syntax

- functions

```
CREATE FUNCTION <func_name>(<arg_list>)
RETURNS <return_type> AS <code> LANGUAGE SQL;
```

- procedures

```
CREATE FUNCTION <func_name>(<arg_list>)
AS <code> LANGUAGE SQL;
```


SQL Procedures & Functions

Remarks

- the function's **code** has to be provided as a **string**
  - to avoid having to heavily escape the code, Postgres supports and alternative string syntax: `$somestring$ code $somestring$` where `somestring` should be a string that is unlikely to appear in the code

SQL Function Example

```
CREATE FUNCTION one() RETURNS INTEGER AS
$$
  SELECT 1;
$$
LANGUAGE SQL;
SELECT one();
```

one
1

SQL Function Example

```
CREATE FUNCTION myadd(a int, b int) RETURNS INTEGER AS
$$
  SELECT a + b;
$$
LANGUAGE SQL;
SELECT myadd(10,11);
```

myadd
21

SQL Function Example

```
CREATE FUNCTION insertstud(id CHAR(5),
                           name VARCHAR(20),
                           deptname VARCHAR(20))
  RETURNS CHAR(5) AS
$$
  INSERT INTO student VALUES (id,name,deptname,0);
  SELECT id;
$$
LANGUAGE SQL;
```


SQL Function Example

```
SELECT insertstud('00000', 'X Y', 'Comp. Sci.');
SELECT * FROM student WHERE name = 'X Y';
```

id	name	deptname	totcred
00000	X Y	Comp. Sci.	0

Table-valued Functions

What are table-valued functions

- SQL functions can return tables
- Such functions are called in the `FROM` clause
- If the function takes input parameters then they may come from prior `FROM` clause items
- In Postgres the return type of these functions is
  - `SET OF RECORD` or
  - `<tablename>`: if the function returns rows with the same schema as table `tablename`

Table-valued Functions Example

```
CREATE FUNCTION csstud() RETURNS SETOF student AS
$$
  SELECT * FROM student WHERE deptname = 'Comp. Sci.';
$$
LANGUAGE SQL;
SELECT * FROM csstud();
```

id	name	deptname	totcred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
54321	Williams	Comp. Sci.	54
76543	Brown	Comp. Sci.	58
00000	X Y	Comp. Sci.	0

Triggers, Procedural Extensions, and UDFs

What is PL/pgSQL?

- PL/pgSQL is Postgres's procedural language embedding SQL
  - `https://www.postgresql.org/docs/current/plpgsql.html`
- Standard **imperative language constructs**
  - **Variables** and **assignment** (using `:=`)
    - Can assign the results of queries to variables
  - **Looping** constructs
  - **Function calls**
  - **Cursors** allow looping through query results
  - **Blocks** - enclosed by `BEGIN` and `END`

Anatomy Of A PL/pgSQL Function

```
CREATE FUNCTION <name> (<parameters>) AS $$
[DECLARE
  <declaration_list>
]
BEGIN
<statement_list>
END;
$$ LANGUAGE plpgsql;
```


Basic Function Example

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
SELECT sales_tax(100.0) AS salestax;
```

salestax
6

Assignments & Variables

Assignment Examples

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int)
↪  AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
SELECT * FROM sum_n_product(2, 4);
```

sum	prod
6	8

Assignment Examples With Queries

```
CREATE FUNCTION query_f(dept VARCHAR, OUT cnt INT) AS $$
BEGIN
    cnt := (SELECT count(*) FROM student WHERE deptname = dept);
END;
$$ LANGUAGE plpgsql;
SELECT * FROM query_f('Comp. Sci.');
```

cnt
6

Conditional Execution (If Statement)

- If statements allow conditional execution as in imperative programming languages

```
IF <condition>
THEN
   <code>
[ELSIF <condition>
THEN
   <code>
]+
[ELSE
   <code>
]
END IF
```


If Statement Examples

```
CREATE FUNCTION opt_sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
  IF subtotal > 100.0 THEN
    tax := subtotal * 0.06;
  ELSE
    tax := 0;
  END IF;
END;
$$ LANGUAGE plpgsql;
SELECT opt_sales_tax(50.0) AS opttax;
```

opttax
0

Looping Constructs - While

- The while loop iterates as long as its condition evaluates to true

```
WHILE <condition> LOOP
  <code>
END LOOP;
```


While Example

```
CREATE FUNCTION mypower(n int, m int) RETURNS INT AS
$$
DECLARE
  result integer := 1;
BEGIN
  WHILE m > 0 LOOP
    result := result * n;
    m := m - 1;
  END LOOP;
  RETURN result;
END;
$$ LANGUAGE plpgsql;
```


While Example

```
SELECT mypower(10,3);
```

mypower
1000

Looping Constructs - For

- The for loop iterates over a set of results assigning each result to `<varname>`
- Iterating over query results: the variable has to be of type RECORD

```
FOR <varname> IN <expression> LOOP
  <code>
END LOOP;
```


For Example

```
CREATE FUNCTION totstud_cred()
  RETURNS INT AS
$$
DECLARE
  totalcredits integer := 0;
  credrec RECORD;
BEGIN
  FOR credrec IN (SELECT totcred FROM student) LOOP
    totalcredits := totalcredits + credrec.totcred;
  END LOOP;
  RETURN totalcredits;
END;
$$ LANGUAGE plpgsql;
```


For Example

```
SELECT totstud_cred() AS total;
```

total
854

Triggers, Procedural Extensions, and UDFs

External Functions

Functions in C

- Postgres has build in support for functions written in C
- Functions have to be compiled into a dynamically linked library
- Postgres has to be instructed to load such a library before the function can be used

Other Procedural Languages

- Postgres supports a larger number of procedural languages
- Some come with the base distributions and other require building extensions
  - `https://wiki.postgresql.org/wiki/PL_Matrix`

Triggers, Procedural Extensions, and UDFs

What is a Trigger?

- Triggers are functions that are executed when a table is accessed
- For DML statements, triggers may change the result of DML statements
- Triggers have conditions that determines when they fire (the trigger's function is called)
- Trigger functions are executed either BEFORE, AFTER, or INSTEAD OF the statement that triggers them
- Triggers can be executed **once per statement** or **for every row**
- `https://www.postgresql.org/docs/16/triggers.html`

Trigger Syntax

- `https://www.postgresql.org/docs/16/sql-createtrigger.html`

```
CREATE TRIGGER <name> <when_executed> <event> ON <table>
<alias_rows_table>
<per_row_or_statement>
[WHEN <condition>]
EXECUTE { FUNCTION | PROCEDURE } <func_name> (<arguments>)

when_executed := BEFORE | AFTER | INSTEAD OF
event := INSERT | UPDATE | DELETE
per_row_or_statement := FOR EACH ROW | FOR EACH STATEMENT
alias_rows_table :=
```


Trigger Functions in PL/pgSQL

- Trigger functions in PL/pgSQL have to return `trigger`
- The old and new row / table is available as variables **OLD** and **NEW**
- The return value is the updated row / table

PL/pgSQL Example

Disallow Updates to Student ID

```
CREATE FUNCTION lockid() RETURNS trigger AS
$$
  BEGIN
    IF OLD.id != NEW.id THEN
      RAISE EXCEPTION 'cannot modify student ids';
    END IF;
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL Example

Disallow Updates to Student ID

```
CREATE TRIGGER lockid BEFORE UPDATE ON student
  FOR EACH ROW
EXECUTE FUNCTION lockid();

UPDATE student SET id = '11111';

:2: ERROR:  cannot modify student ids
CONTEXT:  PL/pgSQL function lockid() line 4 at RAISE
```


PL Access to SQL

PL Access to SQL

Database architectures

- Server-based: clients connect to the database over a network protocol
- Embedded: the database is embedded into the application and accessed through an API from the programming language

Postgres

- **Postgres** is server-based system
- Client libraries exist for many programming languages that implement the Postgres network protocol
- `https://wiki.postgresql.org/wiki/Client_Libraries`
- `https://www.postgresql.org/download/products/2-drivers-and-interfaces/`
- We will look at two common examples (Java and Python)
  - example code for Java, Python, and JS is available in the git repos: `https://github.com/lordpretzel/cs480`

PL Access to SQL

JDBC

- JDBC is a Java SPI for communicating with SQL databases
- Different DBMS are supported through drivers (Java libraries)
- Provides a standardized interface for all supported databases

Import Relevant Classes

- Import relevant JDBC classes

```java
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Connection;
```


Loading the Driver

- To connect to a database you need to have the **jar file** for the **driver** for your DBMS
  - the jar has to be in your **class path**
- You need to load the driver using the classloader

```java
String JDBC_DRIVER = "org.postgresql.Driver";

// load the driver based on the drivers class name
Class.forName(JDBC_DRIVER);
```


Connections & Statements

- Connection represent network connections to the database
- Statement objects are used to execute SQL
- ResultSet objects are handles to query results and can be used to iterate over query results

Opening & Closing Connections

```
public static final String JDBC_DB = "university";
public static final String JDBC_PORT = "5450";
public static final String JDBC_HOST = "127.0.0.1";
public static final String JDBC_URL = "jdbc:postgresql://" + JDBC_HOST
↪  + ":" + JDBC_PORT + "/" + JDBC_DB;
public static final String DBUSER = "postgres";
public static final String DBPASSWD = "test";

Connection c = DriverManager.getConnection(JDBC_URL, DBUSER,
↪  DBPASSWD);
c.close();
```


Creating & Closing Statements

- Statement objects are used to execute SQL statements
  - Inspite of the name

```
Statement s = c.createStatement();
s.close();
```


Executing Queries and Processing Results

- The `Statement` classes `executeQuery` method runs a query and returns a `ResultSet` object
- The `ResultSet` object is used to iterate over result rows and retrieve attribute values of the current row as Java objects

```
r = s.executeQuery("SELECT id, tot_cred, name FROM student ORDER BY
↪  name ASC;");
while(r.next()) {
        String id = r.getString("id");
        String name = r.getString("name");
        int tot_cred = r.getInt("tot_cred");
        System.out.println(id + "," + name + "," + tot_cred);
}
r.close();
```


SQL Injection

- SQL passed as a string to the execute methods
- If such a string is dynamically constructed from user input, then this represents a security thread
- Attackers may craft responses that change the executed SQL code's semantics to …
  - Retrieve data they should not have access to
  - Modify the database

SQL Injection Example

- Consider a web form where the user inputs a student UIN and gets back student information
- This web interface may construct a query like this where `uin` is the UIN submitted by the user through the webform

```
sql = "SELECT * FROM student WHERE id = '" + uin + "';";
```

- Now an attacker can craft a uin value that includes quotes to change the statements where condition

```
111111' OR 'a' = 'a
```

- Substituting this value the resulting query is (which returns all students)

```
SELECT * FROM student WHERE id = '111111' OR 'a' = 'a';
```


SQL Injection

- We cannot cover SQL injection in depth here. Here are some resources if you want to know more:
  - `https://portswigger.net/web-security/sql-injection`
  - `https://owasp.org/www-project-mutillidae-ii/`

Prepared Statements

- Prepared statements are statements with parameters
  - The statement is created once
  - The statement can be executed many times with different parameters

Preventing SQL Injection

Prepared statements prevent SQL injection as the user input is only assigned to parameters and there is no way to change what statement is executed

Prepared Statements in JDBC

Creating Prepared Statements

- Prepared statements are regular SQL statements that can contain parameters (represented using ?)
- In JDBC prepared statements are created by calling the `prepareStatement` method of the `Connection` class

```
PreparedStatement p = conn.prepareStatement("SELECT * FROM student
↪  WHERE id = ?");
```


Prepared Statements in JDBC

Executing Prepared Statements

- First set values for the parameters using the type-specific `set` methods of `PreparedStatement`
- Then call `executeUpdate` or `executeQuery`

```
p.setString(1,"11111"); // set first parameter to value "11111"
ResultSet rs = p.executeQuery();
```


Advantages of Prepared Statements

- Typically the database only **optimizes the query / update once** if it is a **prepared statement**
  - sophisticated systems may generate multiple plans for different selectivity (caused by the choice of parameter values), but still will not parse and optimize the prepared statement every time it is executed
- This is useful for fast queries where query optimization can become a bottleneck
  - e.g., simple updates

Cleanup

- Note that `Connection`, `Statement`, and `ResultSet` objects need to be explicitly closed to release resources

```
resultsset.close();
statement.close();
connection.close();
```


Metadata Access

- JDBC provides an API for accessing the database catalog in a system-independent way
- You get a `DatabaseMetaData` object by calling the `Connection` classes `getMetaData()` method

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
↪  and Column-Pattern
// Returns: One row for each column
while(rs.next()) {
        System.out.println(rs.getString("COLUMN_NAME"),
                                          rs.getString("TYPE_NAME"));
}
```


PL Access to SQL

`psycopg` library

- Most common (but not only) Python library
- Wraps Postgre's C library (can lead to installation problems)

Connections

- To communicate with the database you first have to open a connection

```
import psycopg2

# define connection parameters
connection = { 'dbname': 'cs480_slides',
               'user': 'postgres',
               'host': '127.0.0.1',
               'password': 'test',
               'port': 5450 }

# open connection
conn = psycopg2.connect(**connection)
print(conn)
```


Cursor

- Cursors allow execution of SQL code

```
cur = conn.cursor()

# run query
cur.execute("SELECT name, deptname FROM student")

# fetch all results into a list of tuples
rows = cur.fetchall()
print(rows[0])
```

```
('Zhang', 'Comp. Sci.')
```


Cleanup

- After being done with a cursor / connection, you should close them to release resources

```
cur.close()
conn.close()
```


Recap

Recap

- DDL - Data definition language
  - Create & modify the **database schema**
- DML - Updates & Queries
  - **Inserts**, **updates**, and **deletes**
  - **Query blocks**
  - **(Nested) Subqueries**
  - **Window functions**
  - **Views** and **CTEs**
  - **Recursive queries**
- Database Catalog
  - stores **schema information** accessible as tables / views
- Access Control

Recap

- Triggers and Procedural Extensions
  - **triggers** are function that are executed conditional on DML operations on tables
- SQL from a Programming Language
  - access **SQL** using client libraries for a PL
  - **JDBC** and **ODBC**