Carnegie Mellon University

Embedded Database Logic

Lecture #15

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #3 is due Monday October 19th

Project #4 is due Monday December 10th

Homework #4 is due Monday November 12th

UPCOMING DATABASE EVENTS

BlazingDB Tech Talk

→ Thursday October 25th @ 12pm

→ CIC - 4th floor (ISTC Panther Hollow Room)

Brytlyt Tech Talk

→ Thursday November 1st @ 12pm

→ CIC - 4th floor (ISTC Panther Hollow Room)

OBSERVATION

Until now, we have assumed that all of the logic for an application is located in the application itself.

The application has a "conversation" with the DBMS to store/retrieve data.

→ Protocols: JDBC, ODBC

CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

Parser
Planner
Optimizer
Query Execution

CONVERSATIONAL DATABASE API

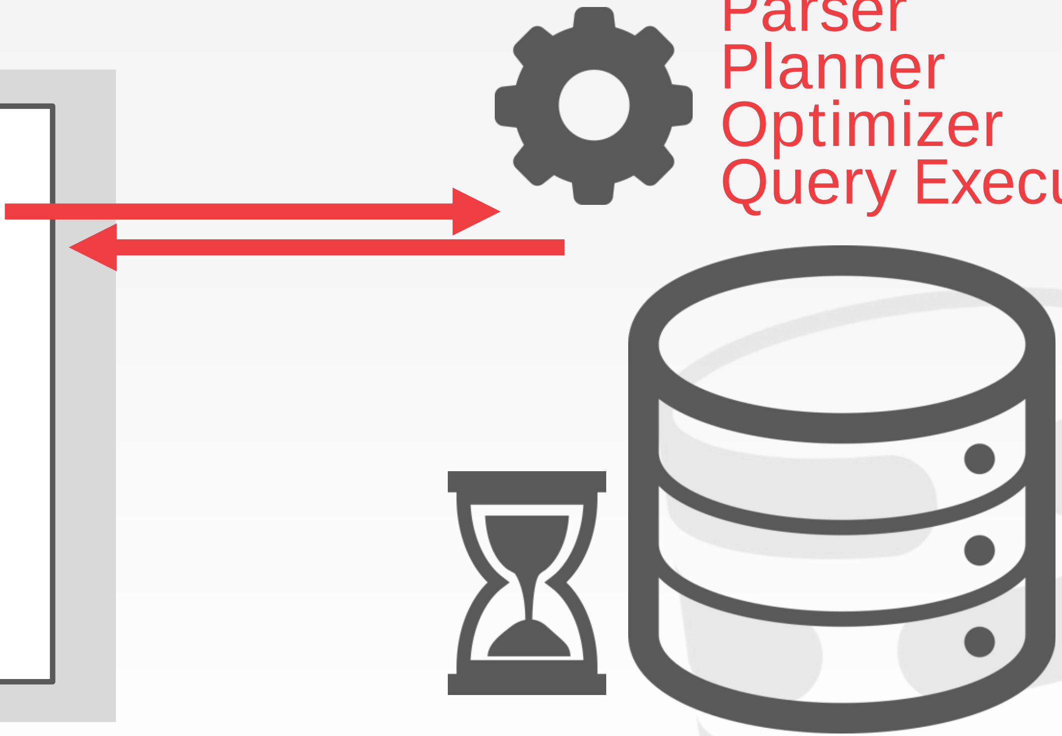
Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

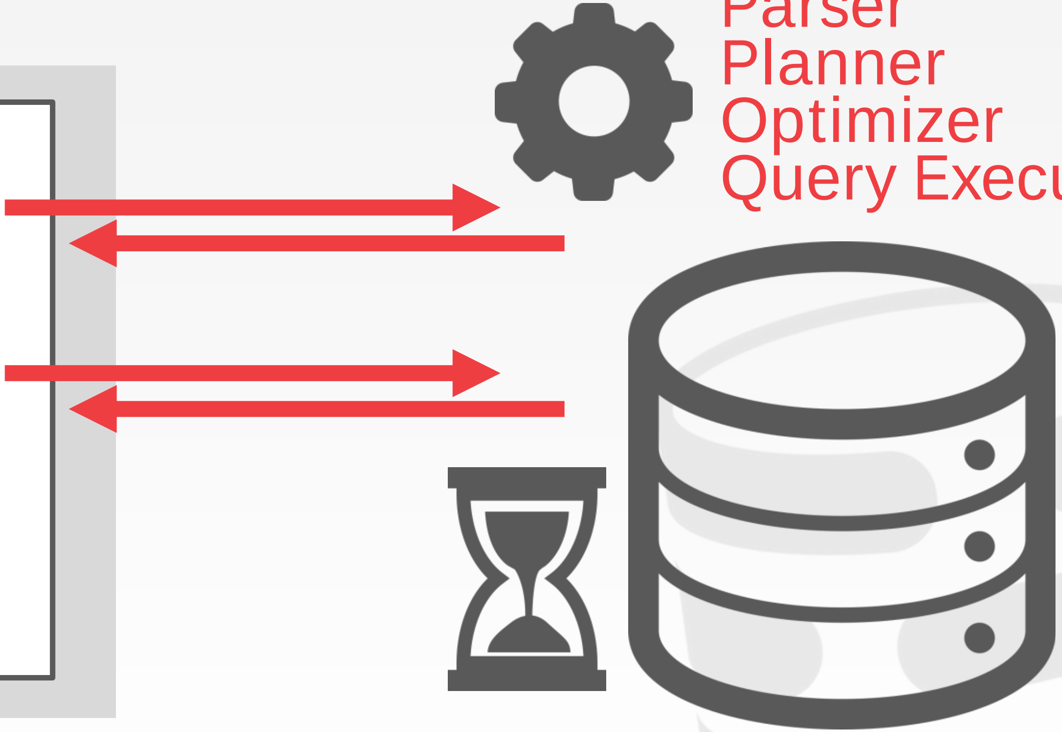


Parser
Planner
Optimizer
Query Execution

CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

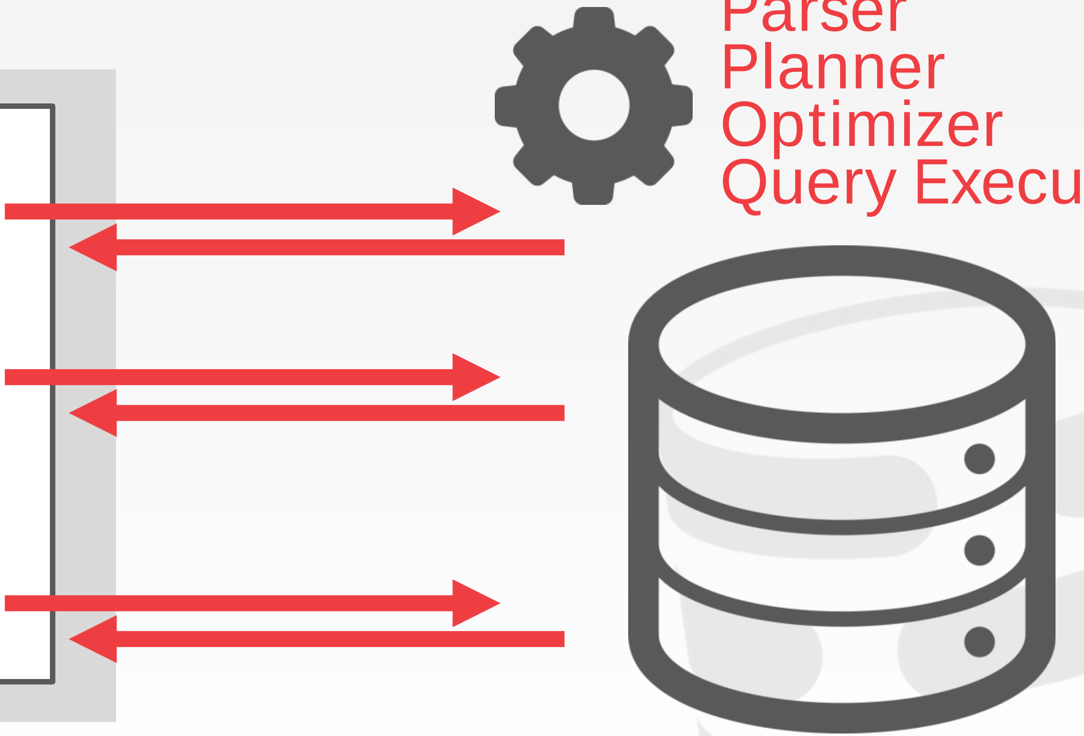


Parser
Planner
Optimizer
Query Execution

CONVERSATIONAL DATABASE API

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```



EMBEDDED DATABASE LOGIC

Move application logic into the DBMS to avoid multiple network round-trips.

Potential Benefits

→ Efficiency

→ Reuse

TODAY'S AGENDA

User-defined Functions
Stored Procedures
Triggers
Change Notifications
User-defined Types
Views

USER-DEFINED FUNCTIONS

A **user-defined function** (UDF) is a function written by the application developer that extends the system's functionality beyond its built-in operations.

→ It takes in input arguments (scalars)
→ Perform some computation
→ Return a result (scalars, tables)

UDF DEFINITION

Return Types:

→ Scalar Functions: Return a single data value
→ Table Functions: Return a single result table.

Computation Definition:

→ SQL Functions
→ External Programming Language

UDF – SQL FUNCTIONS

A SQL-based UDF contains a list of SQL statements that the DBMS executes in order when the UDF is invoked.

→ The function returns whatever the result is of the last query executed;

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE FUNCTION get_foo(int)
                RETURNS foo AS $$
  SELECT * FROM foo WHERE foo.id = $1;
$$ LANGUAGE SQL;
```

UDF – SQL FUNCTIONS

A SQL-based UDF contains a list of SQL statements that the DBMS executes in order when the UDF is invoked.

→ The function returns whatever the result is of the last query executed;

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE FUNCTION get_foo(int)  Input Args
                RETURNS foo AS $$
  SELECT * FROM foo WHERE foo.id = $1;
$$ LANGUAGE SQL;
```

UDF – SQL FUNCTIONS

A SQL-based UDF contains a list of SQL statements that the DBMS executes in order when the UDF is invoked.

→ The function returns whatever the result is of the last query executed;

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE FUNCTION get_foo(int)
        Return Args RETURNS foo AS $$
  SELECT * FROM foo WHERE foo.id = $1;
$$ LANGUAGE SQL;
```

UDF – SQL FUNCTIONS

A SQL-based UDF contains a list of SQL statements that the DBMS executes in order when the UDF is invoked.

→ The function returns whatever the result is of the last query executed;

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE FUNCTION get_foo(int)
                RETURNS foo AS $$
  SELECT * FROM foo WHERE foo.id = $1;
$$ LANGUAGE SQL;
```

Function Body

UDF – EXTERNAL PROGRAMMING LANGUAGE

Some DBMSs support writing UDFs in languages other than SQL.

→ **SQL Standard**: SQL/PSM
→ **Oracle/DB2**: PL/SQL
→ **Postgres**: PL/pgSQL
→ **MSSQL/Sybase**: Transact-SQL

Other systems support more common programming languages:

→ Sandbox vs. non-Sandbox

PL/PGSQL EXAMPLE

```
CREATE OR REPLACE FUNCTION get_foo(int)
                   RETURNS SETOF foo AS $$
  BEGIN
    RETURN QUERY SELECT * FROM foo
                         WHERE foo.id = $1;
  END;
$$ LANGUAGE plpgsql;
```

PL/PGSQL EXAMPLE (2)

```
CREATE OR REPLACE FUNCTION sum_foo(i int)
                         RETURNS int AS $$
  DECLARE foo_rec RECORD;
  DECLARE out INT;
  BEGIN
    out := 0;
    FOR foo_rec IN SELECT id FROM foo
                    WHERE id > i LOOP
      out := out + foo_rec.id;
    END LOOP;
    RETURN out;
  END;
$$ LANGUAGE plpgsql;
```

PostgreSQL

STORED PROCEDURES

A **stored procedure** is a self-contained function that performs more complex logic inside of the DBMS.

→ Can have many input/output parameters.
→ Can modify the database table/structures.
→ Not normally used within a SQL query.

Some DBMSs distinguish UDFs vs. stored procedures, but not all.

STORED PROCEDURES

Application

```
BEGIN
  SQL
  Program Logic
  SQL
  Program Logic
  ⋮
COMMIT
```

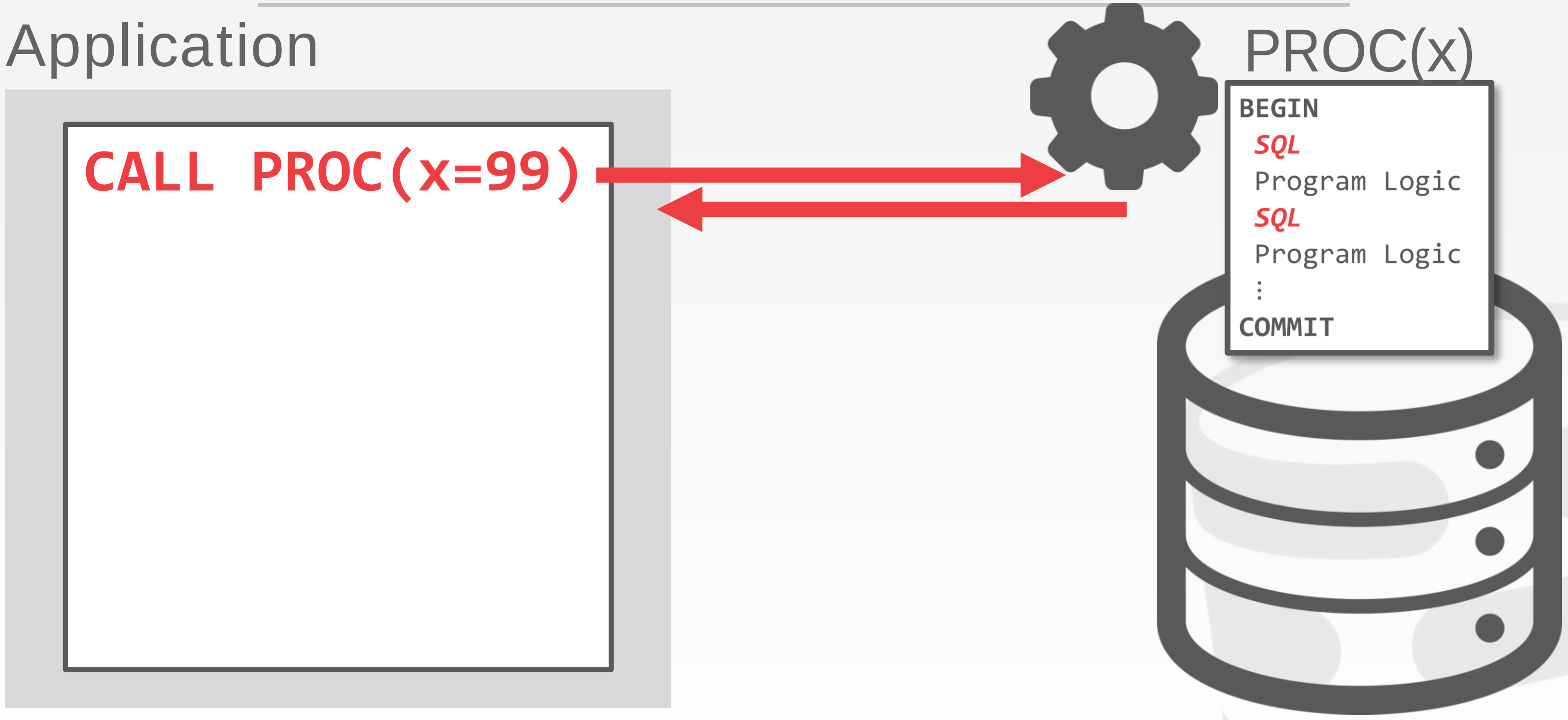
STORED PROCEDURES

Application

```
CALL PROC(x=99)
```

PROC(x)

STORED PROCEDURES



STORED PROCEDURE VS. UDF

A UDF is meant to perform a subset of a read-only computation within a query.

A stored procedure is meant to perform a complete computation that is independent of a query.

DATABASE TRIGGERS

A **trigger** instructs the DBMS to invoke a UDF when some event occurs in the database.

The developer has to define:

→ What type of **event** will cause it to fire.
→ The **scope** of the event.
→ When it fires **relative** to that event.

TRIGGER EXAMPLE

```sql
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```sql
CREATE TABLE foo_audit (
    id SERIAL PRIMARY KEY,
    foo_id INT REFERENCES foo (id),
    orig_val VARCHAR,
    cdate TIMESTAMP
);
```

TRIGGER EXAMPLE

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE TABLE foo_audit (
    id SERIAL PRIMARY KEY,
    foo_id INT REFERENCES foo (id)
```

```
CREATE OR REPLACE FUNCTION log_foo_updates()
                   RETURNS trigger AS $$
  BEGIN
    IF NEW.val <> OLD.val THEN
      INSERT INTO foo_audit
                   (foo_id, orig_val, cdate)
            VALUES (OLD.id, OLD.val, NOW());
    END IF;
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

Tuple Versions

TRIGGER EXAMPLE

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  val VARCHAR(16)
);
```

```
CREATE TABLE foo_audit (
    id SERIAL PRIMARY KEY,
    foo_id INT REFERENCES foo (id)
```

```
CREATE OR REPLACE FUNCTION log_foo_updates()
                   RETURNS trigger AS $$
  BEGIN
    IF NEW.val <> OLD.val THEN
      INSERT INTO foo_audit
                   (foo_id, orig_val, cdate)
            VALUES (OLD.id, OLD.val, NOW());
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER foo_updates
   BEFORE UPDATE ON foo FOR EACH ROW
  EXECUTE PROCEDURE log_foo_updates();
```

TRIGGER DEFINITION

Event Type:

- → INSERT
- → UPDATE
- → DELETE
- → TRUNCATE
- → CREATE
- → ALTER
- → DROP

Event Scope:

- → TABLE
- → DATABASE
- → VIEW
- → SYSTEM

Trigger Timing:

- → Before the statement executes.
- → After the statement executes
- → Before each row that the statement affects.
- → After each row that the statement affects.
- → Instead of the statement.

CHANGE NOTIFICATIONS

A **change notification** is like a trigger except that the DBMS sends a message to an external entity that something notable has happened in the database.

→ Think a "pub/sub" system.

→ Can be chained with a trigger to pass along whenever a change occurs.

SQL standard: `LISTEN` + `NOTIFY`

NOTIFICATION EXAMPLE

```
CREATE OR REPLACE FUNCTION notify_foo_updates()
                    RETURNS trigger AS $$
  DECLARE notification JSON;
  BEGIN
    notification = row_to_json(NEW);
    PERFORM pg_notify('foo_update',
                      notification::text);
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

Notification Payload

NOTIFICATION EXAMPLE

```
CREATE OR REPLACE FUNCTION notify_foo_updates()
                  RETURNS trigger AS $$
  DECLARE notification JSON;
  BEGIN
    notification = row_to_json(NEW);
    PERFORM pg_notify('foo_update',
                      notification::text);
    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;
```

Notification Payload

```
CREATE TRIGGER foo_notify
  AFTER INSERT ON foo_audit FOR EACH ROW
  EXECUTE PROCEDURE notify_foo_updates();
```

OBSERVATION

All DBMSs support the basic primitive types in the SQL standard. They also support basic arithmetic and string manipulation on them.

But what if we want to store data that doesn't match any of the built-in types?

```
coordinate (x, y, label)
```

COMPLEX TYPES

Approach #1: Attribute Splitting

→ Store each primitive element in the complex type as its own attribute in the table.

```
INSERT INTO locations
   (x, y, label)
VALUES
   (10, 20, "OTB");
```

Approach #2: Application Serialization

→ Java serialize, Python pickle
→ Google Protobuf, Facebook Thrift
→ JSON / XML

```
CREATE TABLE locations (
   coord JSONB NOT NULL
);
```

```
INSERT INTO location (coord)
VALUES (
 '{x:10, y:20, label:"OTB"}'
);
```

USER-DEFINED TYPES

A **user-defined type** is a special data type that is defined by the application developer that the DBMS can stored natively.

→ First introduced by Postgres in the 1980s.
→ Added to the SQL:1999 standard as part of the "object-relational database" extensions.

Sometimes called **structured user-defined types** or **structured types.**

USER-DEFINED TYPES

Each DBMS exposes a different API that allows you to create a UDT.

→ Oracle supports PL/SQL.
→ DB2 supports creating types based on built-in types.
→ MSSQL/Postgres only support type definition using external languages (.NET, C)

```
CREATE TYPE coordinates AS OBJECT (
  x INT NOT NULL,
  y INT NOT NULL,
  label VARCHAR(32) NOT NULL
);
```

ORACLE

VIEWS

Creates a "virtual" table containing the output from a **SELECT** query. The view can then be accessed as if it was a real table.

This allows programmers to simplify a complex query that is executed often.

→ Won't make it faster though.

Often used as a mechanism for hiding a subset of a table's attributes from certain users.

VIEW EXAMPLE (1)

Create a view of the CS student records with just their id, name, and login.

```
CREATE VIEW cs_students AS
  SELECT sid, name, login
    FROM student
   WHERE login LIKE '%@cs';
```

Original Table

sid	name	login	age	gpa
53666	Kanye West	kw@cs	40	3.5
53677	Justin Bieber	jb@ece	23	2.25
53688	Tone Loc	tloc@isr	51	3.8
53699	Andy Pavlo	pavlo@cs	36	3.0

VIEW EXAMPLE (1)

Create a view of the CS student records with just their id, name, and login.

```
CREATE VIEW cs_students AS
  SELECT sid, name, login
    FROM student
   WHERE login LIKE '%@cs';
```

Original Table

sid	name	login	age	gpa
53666	Kanye West	kw@cs	40	3.5
53677	Justin Bieber	jb@ece	23	2.25
53688	Tone Loc	tloc@isr	51	3.8
53699	Andy Pavlo	pavlo@cs	36	3.0

```
SELECT * FROM cs_students;
```

sid	name	login
53666	Kanye West	kw@cs
53699	Andy Pavlo	pavlo@cs

VIEW EXAMPLE (1)

Create a view of the CS student records with just their id, name, and login.

```
CREATE VIEW cs_students AS
  SELECT sid, name, login
    FROM student
   WHERE login LIKE '%@cs';
```

Original Table

sid	name	login	age	gpa
53666	Kanye West	kw@cs	40	3.5
53677	Justin Bieber	jb@ece	23	2.25
53688	Tone Loc	tloc@isr	51	3.8
53699	Andy Pavlo	pavlo@cs	36	3.0

```
SELECT * FROM cs_students;
```

sid	name	login
53666	Kanye West	kw@cs
53699	Andy Pavlo	pavlo@cs

VIEW EXAMPLE (2)

Create a view with the average age of all of the students.

```
CREATE VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

VIEWS VS. SELECT INTO

VIEW

→ Dynamic results are only materialized when needed.

SELECT...INTO

→ Creates static table that does not get updated when student gets updated.

```
CREATE VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

```
SELECT AVG(gpa) AS avg_gpa
  INTO cs_gpa
  FROM student
 WHERE login LIKE '%@cs';
```

VIEWS VS. SELECT INTO

VIEW

→ Dynamic results are only materialized when needed.

SELECT...INTO

→ Creates static table that does not get updated when student gets updated.

```
CREATE VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

```
SELECT AVG(gpa) AS avg_gpa
  INTO cs_gpa
  FROM student
 WHERE login LIKE '%@cs';
```

UPDATING VIEWS

The SQL-92 standard specifies that an application is allowed to modify a **VIEW** if it has the following properties:

→ It only contains one base table.
→ It does not contain grouping, distinction, union, or aggregation.

MATERIALIZED VIEWS

Creates a view containing the output from a **SELECT** query that is automatically updated when the underlying tables change.

```
CREATE MATERIALIZED VIEW cs_gpa AS
  SELECT AVG(gpa) AS avg_gpa
    FROM student
   WHERE login LIKE '%@cs';
```

CONCLUSION

Moving application logic into the DBMS has lots of benefits.

→ Better Efficiency
→ Reusable across applications

But it has problems:

→ Not portable
→ DBAs don't like constant change.
→ Potentially need to maintain different versions.

NEXT CLASS

TRANSACTIONS!!!