W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 4: ER, Relational, SQL (IV)

W4111 – Introduction to Databases
Section 002, Spring 2021

Lecture 4: ER, Relational, SQL (IV)

We will start in a couple of minutes.

Contents

Contents

- Introduction: comments, questions and answers
  Some *secret advanced SQL.*

- Intermediate-Advanced SQL:
  - Data modification operations: basics, advanced
  - Common Table Expressions (WITH, CTE)
  - Constraints
  - Views
  - Indexes
  - More Set Like Concepts
  - A Little More on Subquery
  - Functions
  - Metadata and Catalog

Comments and Observations

YOU MAY NOT USE GROUP BY … …

Secret Advanced SQL – Window Function

```
SELECT
   *, round(h/team_hits,3)  AS percentage_of_team_hits,
   round(rbi/team_rbis,3) AS percentage_of_team_rbis
FROM
   (SELECT playerid, nameLast, nameFirst FROM people) AS simple_people
LEFT JOIN
   (SELECT
      playerid, yearid, stint, teamid, h, rbi,
      sum(h) OVER (PARTITION BY teamid, yearid) AS team_hits,
      sum(rbi) OVER (PARTITION BY teamid, yearid) AS team_rbis
    FROM batting) AS relative_batting
USING(playerid)
ORDER BY yearid DESC, teamid ASC, stint ASC, playerid ASC;
```

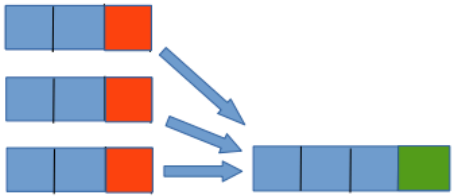
playerid	nameLast	nameFirst	yearid	stint	teamid	h	rbi	team_hits	team_rbis	percentage_of_team_...	percentage_of_team_rbis
ahmedni01	Ahmed	Nick	2019	1	ARI	141	82	1419	778	0.099	0.105
almonab01	Almonte	Abraham	2019	1	ARI	9	4	1419	778	0.006	0.005
andrima01	Andriese	Matt	2019	1	ARI	0	0	1419	778	0.000	0.000
avilaal01	Avila	Alex	2019	1	ARI	34	24	1419	778	0.024	0.031
bradlar01	Bradley	Archie	2019	1	ARI	0	0	1419	778	0.000	0.000
chafian01	Chafin	Andrew	2019	1	ARI	0	0	1419	778	0.000	0.000
clarkta01	Clarke	Taylor	2019	1	ARI	6	2	1419	778	0.004	0.003
crichst01	Crichton	Stefan	2019	1	ARI	0	0	1419	778	0.000	0.000
cronke01	Cron	Kevin	2019	1	ARI	15	16	1419	778	0.011	0.021
duplajo01	Duplantier	Jon	2019	1	ARI	1	0	1419	778	0.001	0.000
dysonja01	Dyson	Jarrod	2019	1	ARI	92	27	1419	778	0.065	0.035
escobed01	Escobar	Eduardo	2019	1	ARI	171	118	1419	778	0.121	0.152
florewi01	Flores	Wilmer	2019	1	ARI	84	37	1419	778	0.059	0.048
ginkeke01	Ginkel	Kevin	2019	1	ARI	0	0	1419	778	0.000	0.000
godleza01	Godley	Zack	2019	1	ARI	3	2	1419	778	0.002	0.003
greinza01	Greinke	Zack	2019	1	ARI	13	8	1419	778	0.009	0.010
hiranyo01	Hirano	Yoshihisa	2019	1	ARI	0	0	1419	778	0.000	0.000
hollagr01	Holland	Greg	2019	1	ARI	0	0	1419	778	0.000	0.000
jonesad01	Jones	Adam	2019	1	ARI	126	67	1419	778	0.089	0.086

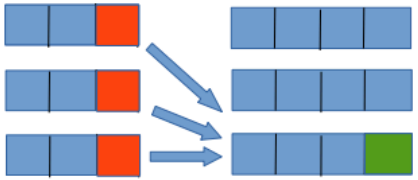
- ”As a fan, I want a table that has player information, a player’s H and RBI per year, stint and team, **and shows the player’s H and RBI as a percentage of the team’s totals**.
- I cannot easily do this with a GROUP BY
  - I want a row for each playerID, yearID, stint and team showing the individual performance.
  - GROUP BY reduces the number of rows to the distinct groups.
  - What I want is the individual rows that also has columns containing data produced by GROUP BY.
- This is a combination of Window functions (like aggregation) and partition (like GROUP BY).

(Switch to Notebook)

Window functions: what are they?

- A window function performs a calculation across a set of rows that are related to the current row, similar to what can be done with an aggregate function.
- But unlike traditional aggregate functions, a window function does not cause rows to become grouped into a single output row.
- So, similar to normal function, but can access values of other rows "in the vicinity" of the current row

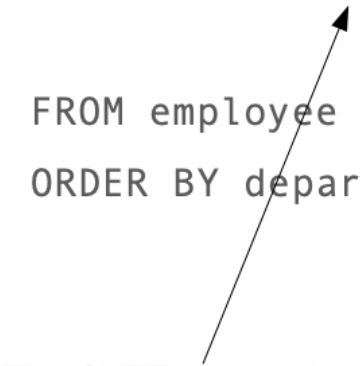




ORACLE

Window function example, no frame

```
SELECT name, department_id, salary,
       SUM(salary) OVER (PARTITION BY department_id)
                         AS department_total
       FROM employee
       ORDER BY department_id, name;
```



ORACLE

Partitions: 10, 20, 30

name	department_id	salary	department_total
Newt	NULL	75000	75000
Dag	10	NULL	370000
Ed	10	100000	370000
Fred	10	60000	370000
Jon	10	60000	370000
Michael	10	70000	370000
Newt	10	80000	370000
Lebedev	20	65000	130000
Pete	20	65000	130000
Jeff	30	300000	370000
Will	30	70000	370000

Partition == disjoint set of rows in result set

Here: all rows in partition are peers

ORACLE

Window function example, frame

```
SELECT name, department_id, salary,
       SUM (salary) OVER (PARTITION BY department_id
                          ORDER BY name
                          ROWS 2 PRECEDING) total
       FROM employee
       ORDER BY department_id, name;
```

ORACLE

Concepts (https://www.slideshare.net/DagHWanvik/sql-window-functions-for-mysql)

Partitions: 10, 20, 30

ORDER BY name within each partition

name	department_id	salary	total
Newt	NULL	75000	75000
Dag	10	NULL	NULL
Ed	10	100000	100000
Fred	10	60000	160000
Jon	10	60000	220000
Michael	10	70000	190000
Newt	10	80000	210000
Lebedev	20	65000	65000
Pete	20	65000	130000
Jeff	30	300000	300000
Will	30	70000	370000

moving window frame:

```
SUM (salary)
...
ROWS 2 PRECEDING
```

a frame is a subset of a partition

ORACLE

Some "Nastily Exhausting Wizarding Tests (N.E.W.T.)" Level Stuff

Observations:

- This is some serious wizard stuff.
- It is not on the core, recommended department syllabus for SQL.
- The feature is new in MySQL 8.0, and relatively new in all SQL products.
- There are inconsistencies between the various SQL products.
- And, Pandas also has similar functions.

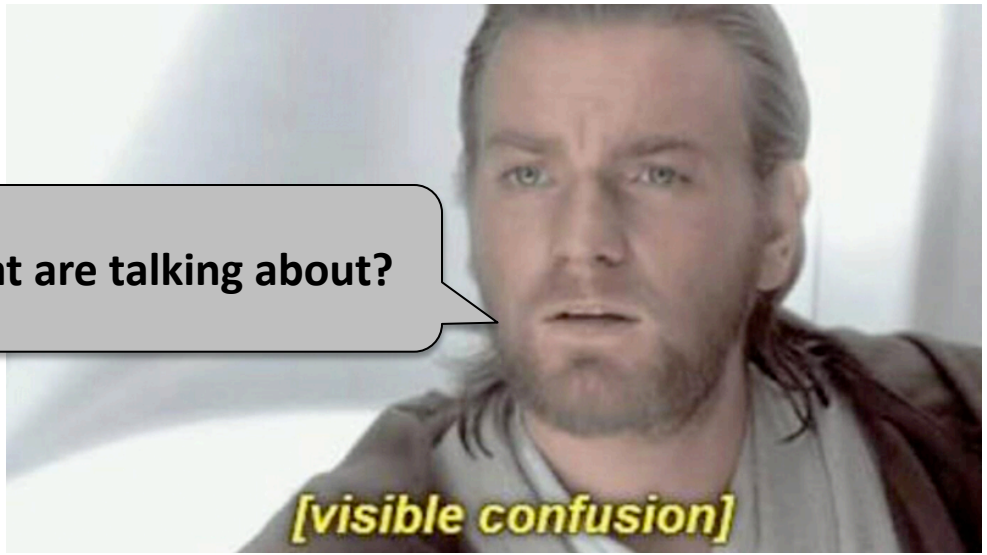
Why do we cover this ...?

- It is really cool and useful.
- The capability is extremely useful for data science, machine learning, etc.
- The TAs love to ask questions on this stuff ...

__get_access_path__(self, fields)

__get_access_path__(self, fields)

Implement
__get_access_path__



Access Path

- Every relational operator accepts one or more tables as input. The operator "accesses the tuples" in the tables.
- There are two "ways" to retrieve the relevant tuples
  - Scan the relation (via blocks)
  - Use an index and matching condition.
- ~~A selection (WHERE) condition is in *conjuctive normal form* if~~
  - ~~Each term is column_name op value~~
  - ~~op is one of <, <=, =, >, >=, <> (or =, != for Hash Index)~~
  - ~~The terms are ANDed, e.g. (nameLast = 'Ferguson') AND (ab > 500)~~
  - ~~CNF allows using a matching index for the access path.~~
- The engine can select a *matching index*
  - A Hash Index on (c1,c2,c3) if the condition is c1=x AND c2=y AND c3=z.
  - ~~A Tree Index if the condition is (can be ordered as)~~
    - ~~c1 op value~~
    - ~~c1 op x AND/OR c2 op y~~
    - ~~... ...~~
  - Many indexes may match, and the engine chooses the *most selective match* (number of tuples or blocks) that match.

- HW 2 implements simple indexes using dictionaries.
- Dictionaries use hash indexes.

Some More on Changing Data

Basic INSERT, UPDATE, DELETE

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

  - Problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

- The base format is INSERT INTO table_name (c1, c2, ..., cn) values (v11, v12, ..., v1n), (v21, v22, ..., v2n), ...
  - c1, c2, ... is a list of column names in any order.
  - (vi1, vi2, ... is a list of the corresponding column values.
- There is also INSERT INTO table_name (c1, c2, ..., cn) SELECT ... ..., where the select statement produces a table compatible with the column list.

Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem

Updates

- Give a 5% salary raise to all instructors

```
update instructor
  set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
  set salary = salary * 1.05
 where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
   set salary = salary * 1.03
   where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
set salary = case
                when salary <= 100000 then salary * 1.05
                else salary * 1.03
             end
```

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
set tot_cred = (select sum(credits)
                from takes, course
                where takes.course_id = course.course_id  and
                      S.ID= takes.ID.and
                                takes.grade <> 'F' and
                       takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
    when sum(credits) is not null then sum(credits)
    else 0
 end
```

Summary

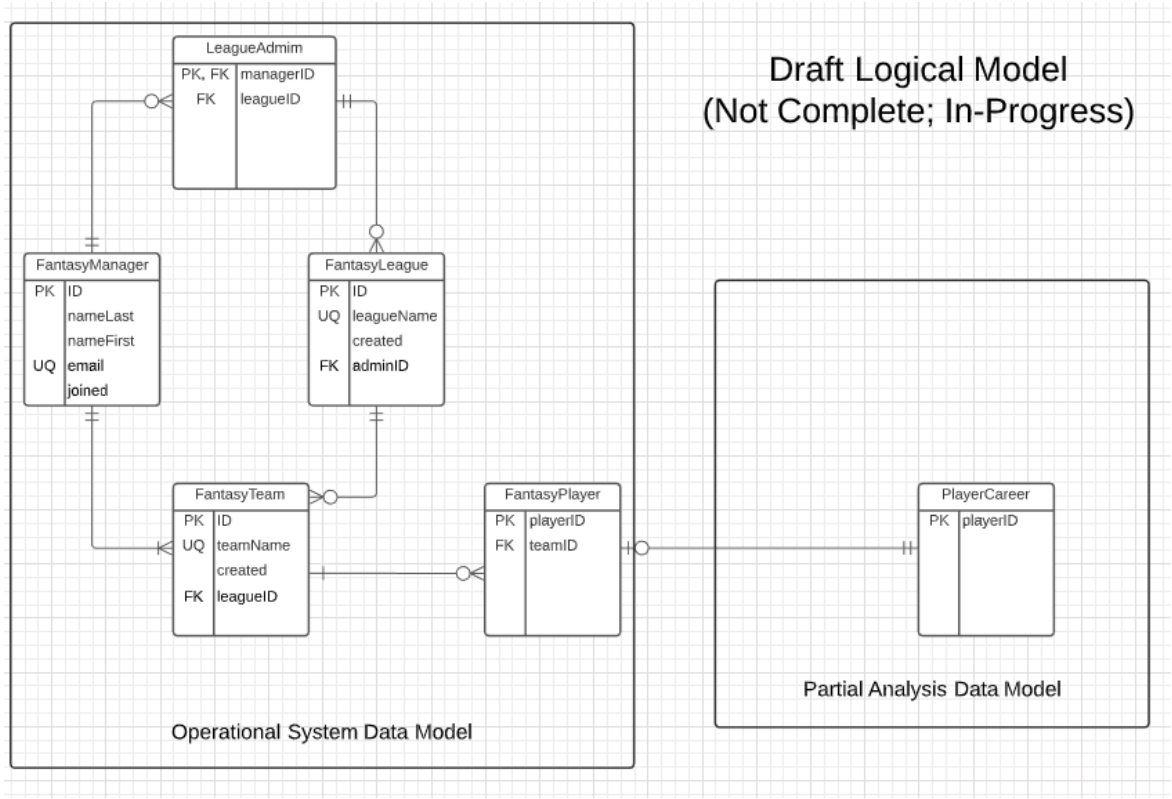
- INSERT, UPDATE and DELETE are pretty straightforward.
- UPDATE and DELETE are very similar to SELECT
  - WHERE clause specifies which rows are affected.
  - The SELECT choose the columns to return.
  - The SET clause chooses and changes columns.
  - DELETE just removes the specified rows.
- INSERT, UPDATE and DELETE changes must not violate constraints, e.g.
  - INSERT a row that causes a duplicate key.
  - DELETE a referenced (target) foreign key.
  - UPDATE columns that create a duplicate key.
  - INSERT values do not include all NOT NULL columns.
- I am not going to do example now, but you have seen and will see me do examples in the context of larger examples.

Examples

Pretend to Play Fantasy Baseball

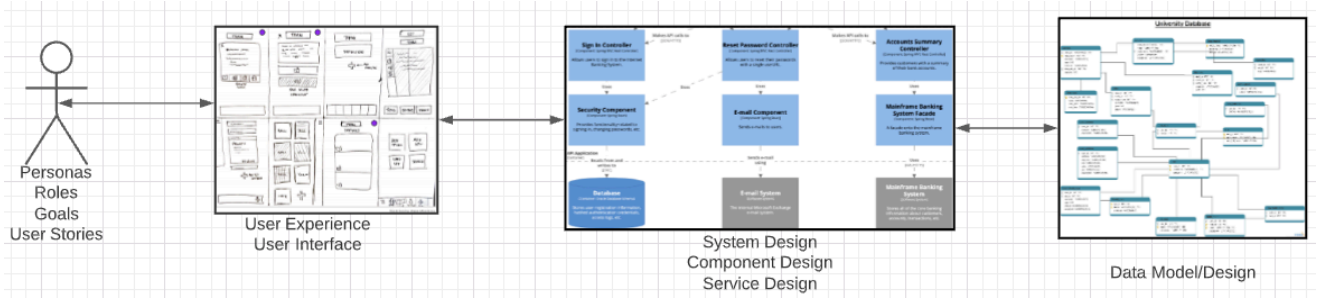
- My example projects for the two tracks will be a **simple** fantasy baseball solution.
  - "**Fantasy baseball** is a game in which people manage rosters of league baseball players, either online or in a physical location, using fictional fantasy baseball team names. The participants compete against one another using those players' real life statistics to score points."
- My solution will have two subsystems:
  - *Analysis system*:
    - Read only data that enables people to select players based on performance, and to estimate the performance of their fantasy team.
    - Event based update system that changes analysis data based on new data.
  - *Operational system* that manages create, retrieve update, delete, etc. of their fantasy teams and leagues.
- INSERT, UPDATE, DELETE primarily apply to the operational system.

In-Progress Operation System Data Model



Problem Statement – Modified Reminder

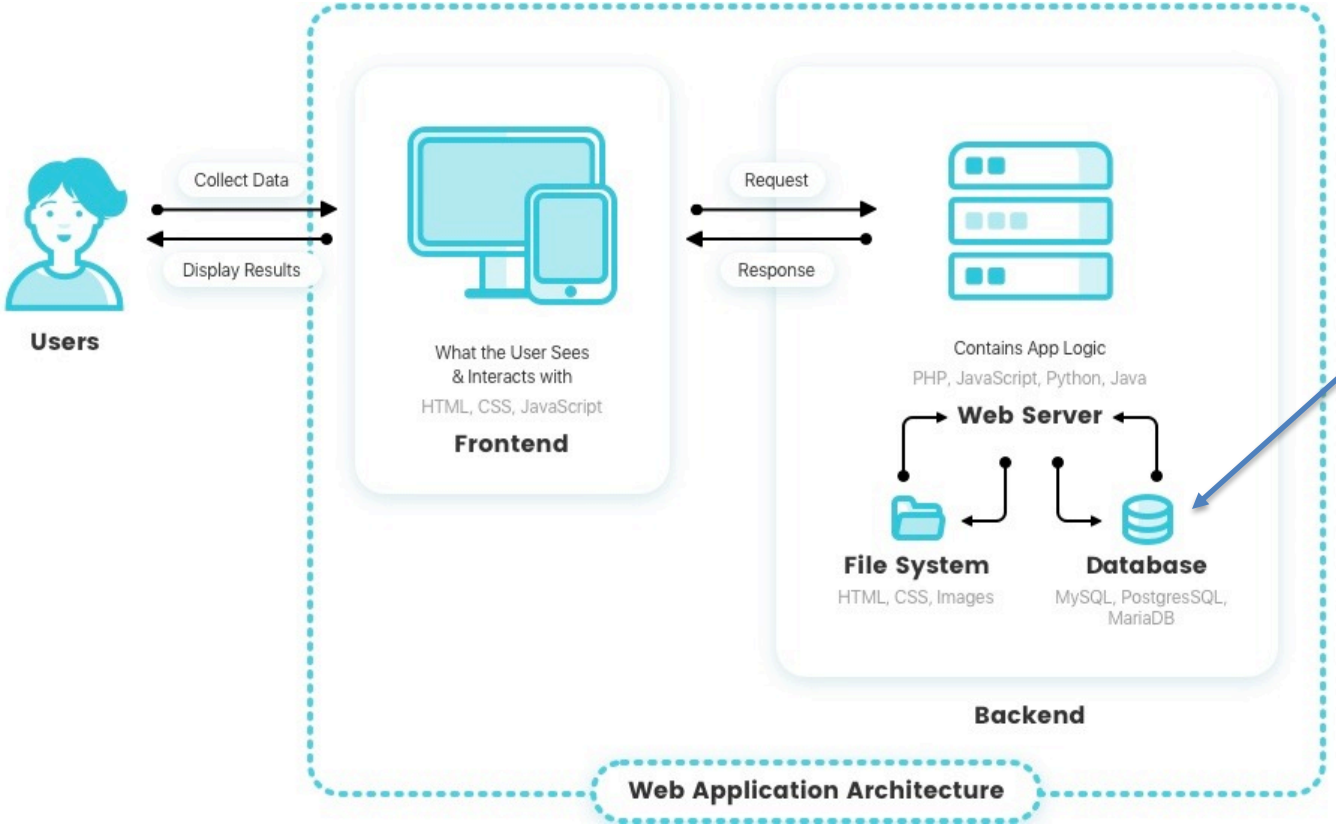
- We must build a system that supports academic operations in a university.
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

- The processes are iterative, with continuous extension and details.
- We will start implementing various *user stories.* Implementation requires:
  - Web UI
  - Paths
  - Data model and operations. (Will only cover this now).

Web Application

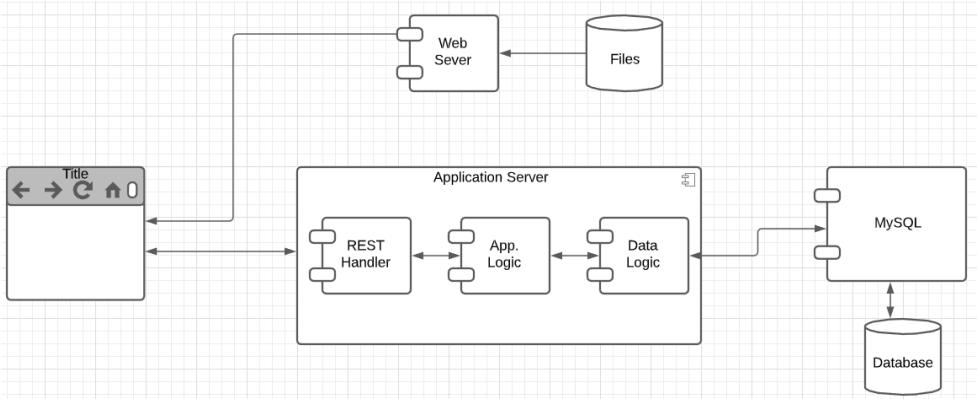


IMDB Database

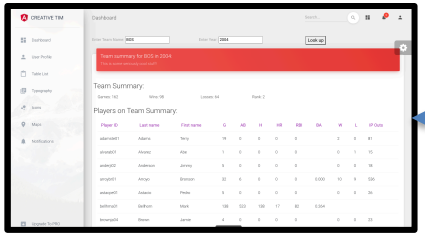
It would be “cool” if we could look at the data directly!

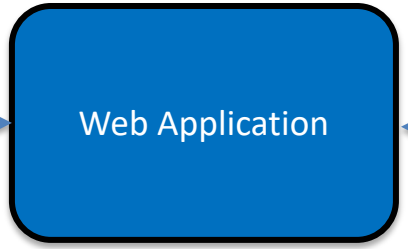
User Story

- "In software development and product management, a user story is an informal, natural language description of one or more features of a software system. User stories are often written from the perspective of an end user or user of a system. " (https://en.wikipedia.org/wiki/User_story)
- Example user stories that I need to implement for the operational system:
  - "As a fantasy team manager, I want to search for players based on career stats."
  - "As a fantasy team manager, I want to add a player to my fantasy team.
  - etc.
- I need to implement:
  - UI
  - Application logic
  - Database tables.



Simple Example





Two HTTP/REST calls:

- GET /api/team_summary?team_id=BOS&year_id=2004
- GET /api/teams?teamID=BOS&yearID=2004

Two SQL Queriess:

- SELECT to Teams table.
- SELECT to team_summary, which is a view. We cover views later.

- "As a baseball fan, I want to enter a teamID and yearID and see:
  - Team wins and losses.
  - Summary of player performance for players on the team and year."
- Built:
  - Dashboard page.
  - REST handlers and application logic.
  - Database view.

Tracks will build:

- Programming: Mix of CRUD and dashboard.
- Non-programming: Dashboards, Data transformation.

Will see more details as we move forward.

Switch to Notebook and Code.

- Angular project
- OpenAPI
- Application code

Common Table Expressions

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
        (select max(budget)
         from department)
select department.name
from department, max_budget
where department.budget = max_budget.value;
```

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept _total (dept_name, value) as
        (select dept_name, sum(salary)
         from instructor
         group by dept_name),
dept_total_avg(value) as
       (select avg(value)
       from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

Common Table Expressions

- CTE: (https://www.essentialsql.com/introduction-common-table-expressions-ctes/)
  - "The Common Table Expressions or CTE's for short are used within SQL (...) to simplify complex joins and subqueries, ..."
  - "A CTE (Common Table Expression) defines a temporary result set which you can then use in a SELECT statement. It becomes a convenient way to manage complicated queries."
  - There are two types of CTEs:
    - Non-recursive.
    - Recursive (note, recursive SQL weirds me out).
  - The benefits are clarity and improved quality through incremental development.

- Basic syntax by example.
  - There may be several CTEs.
  - A CTE can reference other CTEs.

```
With Employee_CTE (EmployeeNumber, Title)
AS
(
SELECT NationalIDNumber,
       JobTitle
FROM   HumanResources.Employee
)
SELECT EmployeeNumber,
       Title
FROM   Employee_CTE
```

CTE
(Common Table Expression)

Query Using CTE

Common Table Expressions – Example

```
WITH career_batting (playerid, h, ab, bb, hr, rbi) AS
  (
    SELECT playerid, sum(h) AS h, sum(ab) AS ab, sum(bb) AS bb,
        sum(hr) AS hr, sum(rbi) AS rbi
      FROM batting GROUP BY playerid
  ),
  career_pitching (playerid, w, l, ipouts, er) AS
  (
    SELECT playerid, sum(w) AS w, sum(l) AS l, sum(ipouts) as ipouts, sum(er) AS er
      FROM pitching GROUP BY playerid
  ),
  career_summary (playerid, h, ab, bb, hr, rbi, bavg, obp, w, l, ipouts, er, era) AS
  (
    SELECT career_batting.playerid, h, ab, bb, hr, rbi,
      IF(ab<500, NULL, round(h/ab, 3)) AS bavg,
      IF(ab<500, NULL, round((h+bb)/(ab + bb), 3)) AS obp,
      w, l, ipouts, er, round((er/(ipouts/3))*9, 3) AS era
    FROM
      career_batting join career_pitching using(playerid)
  )
SELECT
  playerid, nameLast, nameFirst, career_summary.*
FROM
  people JOIN career_summary using(playerid);
```

- Producing a career summary requires several tasks:
  1. Computing batting totals
  2. Computing pitching totals
  3. Applying formulas to produce derived averages and metrics.
  4. Joining (1), (2) and (3) into a career summary.
  5. Joining with people to bring in personal information.
- Producing the table is possible with a single SELECT statement, but ... ...
  - Developing incrementally one simpler query at a time is easier.
  - The resulting query is easier to understand and maintain.

Views

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

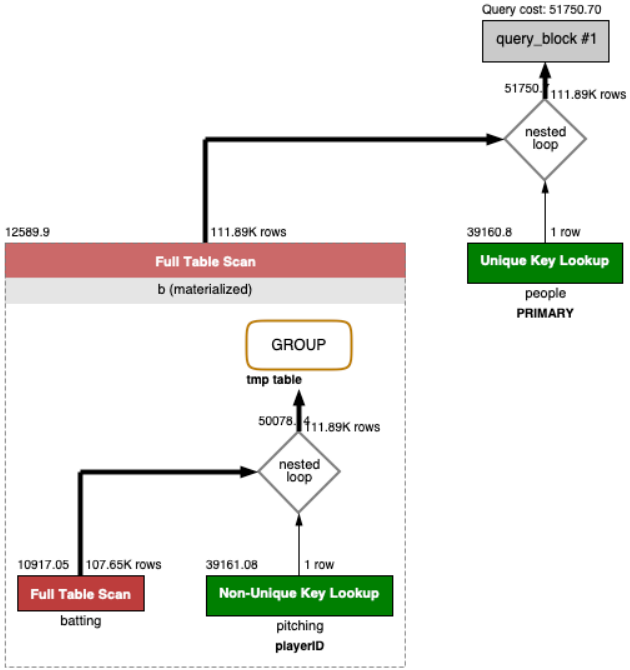
  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

- I created a view that summarizes player performance by team and year.
- This requires GROUP BY to eliminates stints and JOIN.

Example

- SELECT * FROM lahmansbaseballdb.team_player_year_summary;
- Looks like a simple query, but ... is pretty complicated under the covers.

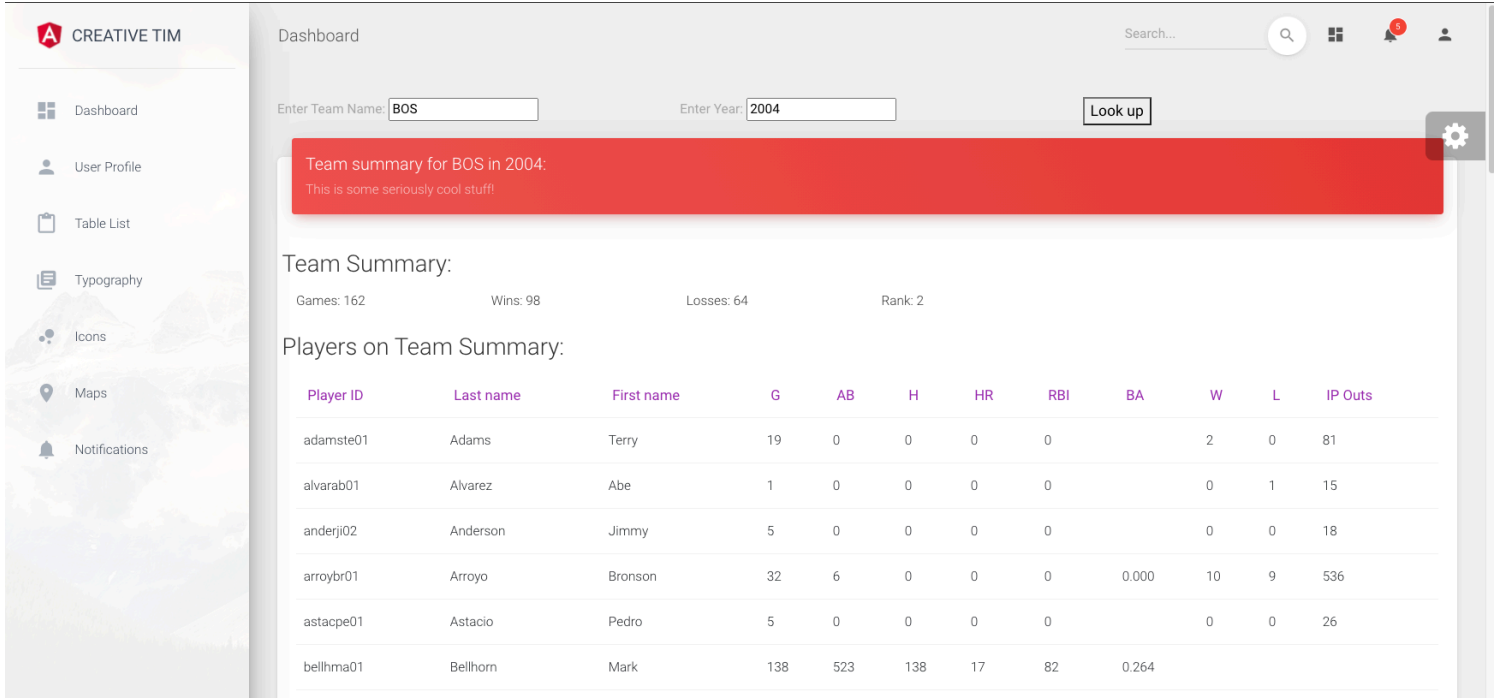
playerID	nameLast	nameFirst	teamID	yearID	G	AB	H	R	HR	RBI	BA	L	W	IPOuts
abercda01	Abercrombie	Frank	TRO	1871	1	4	0	0	0	0	0.000	NULL	NULL	NULL
addybo01	Addy	Bob	RC1	1871	25	118	32	30	0	13	0.271	NULL	NULL	NULL
allisar01	Allison	Art	CL1	1871	29	137	40	28	0	19	0.292	NULL	NULL	NULL
allisdo01	Allison	Doug	WS3	1871	27	133	44	28	2	27	0.331	NULL	NULL	NULL
ansonca01	Anson	Cap	RC1	1871	25	120	39	29	0	16	0.325	NULL	NULL	NULL
armstbo01	Armstrong	Robert	FW1	1871	12	49	11	9	0	5	0.224	NULL	NULL	NULL
barkeal01	Barker	Al	RC1	1871	1	4	1	0	0	2	0.250	NULL	NULL	NULL
barnero01	Barnes	Ross	BS1	1871	31	157	63	66	0	34	0.401	NULL	NULL	NULL
barrebi01	Barrett	Bill	FW1	1871	1	5	1	1	0	1	0.200	NULL	NULL	NULL
barrofr01	Barrows	Frank	BS1	1871	18	86	13	13	0	11	0.151	NULL	NULL	NULL
bassjo01	Bass	John	CL1	1871	22	89	27	18	3	18	0.303	NULL	NULL	NULL
battijo01	Battin	Joe	CL1	1871	1	3	0	0	0	0	0.000	NULL	NULL	NULL
bealsto01	Beals	Tommy	WS3	1871	10	36	7	6	0	1	0.194	NULL	NULL	NULL
beaveed01	Beavens	Edward	TRO	1871	3	15	6	7	0	5	0.400	NULL	NULL	NULL
bechtge01	Bechtel	George	PH1	1871	20	94	33	24	1	21	0.351	2	1	78
bellast01	Bellan	Steve	TRO	1871	29	128	32	26	0	23	0.250	NULL	NULL	NULL
berkena01	Berkenstock	Nate	PH1	1871	1	4	0	0	0	0	0.000	NULL	NULL	NULL
berryto01	Berry	Tom	PH1	1871	1	4	1	0	0	0	0.250	NULL	NULL	NULL
berthha01	Berthrong	Harry	WS3	1871	17	73	17	17	0	8	0.233	NULL	NULL	NULL
biermch01	Bierman	Charles	FW1	1871	1	2	0	0	0	0	0.000	NULL	NULL	NULL
birdge01	Bird	George	RC1	1871	25	106	28	19	0	13	0.264	NULL	NULL	NULL
birdsda01	Birdsall	Dave	BS1	1871	29	152	46	51	0	24	0.303	NULL	NULL	NULL
brainas01	Brainard	Asa	WS3	1871	30	134	30	24	0	21	0.224	15	12	792
brannmi01	Brannock	Mike	CH1	1871	3	14	1	2	0	0	0.071	NULL	NULL	NULL
burrohe01	Burroughs	Henry	WS3	1871	12	63	15	11	1	14	0.238	NULL	NULL	NULL
careyto01	Carey	Tom	FW1	1871	19	87	20	16	0	10	0.230	NULL	NULL	NULL



(Show view creating in notebook)

Views

- Defining the view enables separation of roles/concerns.
- Someone building a web application does not also need to be a SQL expert.
- SQL expert provides views that simply development of other components.



View Definition and Use

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be ***recursive*** if it depends on itself.

Views Defined Using Other Views

- **create view** ***physics_fall_2017*** **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2017';

- **create view** ***physics_fall_2017***_*watson* **as**
  select *course_id*, *room_number*
  from ***physics_fall_2017***
  where *building*= 'Watson';

View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty*

  values ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.
- Two approaches
  - Reject the insert
  - Inset the tuple

    ('30765', 'Green', 'Music', null)

    into the *instructor* relation

Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name= department.dept_name*;
- **insert into** *instructor_info*
  values ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';
- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

View Summary

- Realization of one of Codd's Rules: "**Rule 5:** The *comprehensive data sublanguage rule*:

  A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items:

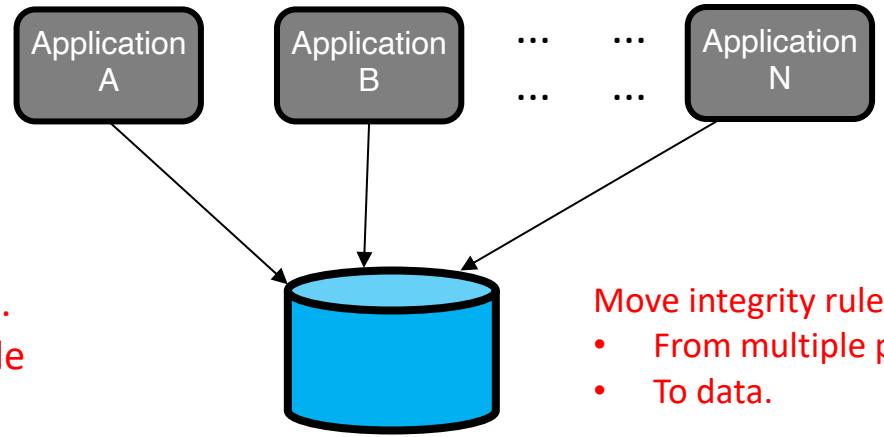
  - Data definition.
  - View definition.
  - Data manipulation (interactive and by program).
  - Integrity constraints.
  - Authorization.
  - Transaction boundaries (begin, commit and rollback)."

- Views provide several benefits. Three core benefits are:
  1. Enabling queries without exposing sensitive information.
  2. Encapsulation: The underlying data model can evolve without changing apps using views.
  3. Separation of concerns/Bounded complexity: Developers can be productive without being SQL experts. The focus on UI, web application, ... expertise.

Constraints

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number



Note:
- Integrating constraints as part of data definition is one of the major benefits.
- Eliminates the need to coordinate code in multiple applications.

Move integrity rules
- From multiple programs.
- To data.

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**

Unique Constraints

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
    (course_id varchar (8),
     sec_id varchar (8),
     semester varchar (6),
     year numeric (4,0),
     building varchar (15),
     room_number varchar (7),
     time slot id varchar (4),
     primary key (course_id, sec_id, semester, year),
     check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Referential Integrity (Cont.)

- Foreign *keys can be* specified as part of the SQL **create table** statement

  foreign key (*dept_name*) **references** *department*

- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  foreign key (*dept_name*) **references** *department* (*dept_name*)

Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
        (…
         dept_name varchar(20),
         foreign key (dept_name) references department
             on delete cascade
             on update cascade,
          . . .)
```

- Instead of cascade we can use :
  - **set null**,
  - **set default**

Note:

- I do not use cascading, but that is just my preference.
- I believe that changes should be explicit, not implemented behind the scenes.

Integrity Constraint Violation During Transactions

- Consider:

  create table *person* (
  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

Note: Do not worry about transaction for now.

Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

  check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

  The check condition states that the time_slot_id in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.

  - The condition has to be checked not only when a tuple is inserted or modified in *section* , but also when the relation *time_slot* changes

Note:

- Not all databases support complex check conditions.
- You can use triggers, which we will cover, to implement more complex check constraints.

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

  create assertion <assertion-name> **check** (<predicate>);

Note: Assertions are inconsistently supported and not that common.

User-Defined Types

- **create type** construct in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final**

- Example:

  create table *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Note:
Domains and UDTs are an important concept but not as well supported as they should be.

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Defining Indexes

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

  create index <name> **on** <relation-name> (attribute);

Note:

- MySQL conflates and makes obscure concepts like Key, Constraint and Index.
- MySQL Indexes:
  - Primary
  - Unique

  } These are actually constraints. Indexes are necessary efficiently enforcing constraints.
  - Index
  - Full Index

Index Creation Example

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- The query:

  ```
  select *
  from   student
  where  ID = '12345'
  ```

  can be executed by using the index to find the required record, without looking at all records of *student*

Some More Set-Like Concepts

Set Comparison – “some” Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept name = 'Biology');
```

Definition of "some" Clause

- F <comp> **some** $r \Leftrightarrow \exists\ t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = 'Biology');
```

Definition of "all" Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≠ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester"

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists  (select *
                 from section as T
                 where semester = 'Spring' and year= 2018
                         and S.course_id = T.course_id);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of “not exists” Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2017

```
select T.course_id
 from course as T
 where unique ( select R.course_id
                from section as R
                where T.course_id= R.course_id
                      and R.year = 2017);
```

Completing Subqueries

- A subquery is a query inside a query.
- Logically, the engine calls the subquery when evaluating every row in processing.
- Correlation means properties from the outer row being evaluated are inputs to the inner query.
- Subqueries:
    - Return tables if in the FROM clause.
    - Scalars in the SELECT clause.
    - Tables or scalars depending on the comparison operator in WHERE.

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
from ( select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from ( select dept_name, avg (salary)
        from instructor
        group by dept_name)
        as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,
        ( select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

- Runtime error if subquery returns more than one result tuple

Functions
(Required for HW 2)

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Note:

- The programming language, runtime and tools for functions, procedures and triggers are not easy to use.
- My view is that calling external functions is an anti-pattern (bad idea).
  - External code degrades the reliability, security and performance of the database.
  - Databases are often mission critical and the heart of environments.

Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count  integer;
              select count (* ) into d_count
              from instructor
              where instructor.dept_name = dept_name
        return d_count;
end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
        returns table  (
          ID varchar(5),
              name varchar(20),
           dept_name varchar(20),
              salary numeric(8,2))
      return table
              (select ID, name, dept_name, salary
               from instructor
               where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *
from table (instructor_of ('Music'))
```

(Core) Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
for r as
      select budget from department
          where dept_name = 'Music'
do
                  set n = n + r.budget
end for
```

Note:

- There are various other looping constructs.

(Core) Language Constructs – if-then-else

- Conditional statements (**if-then-else**)

if *boolean expression*
then *statement or compound statement*
elseif *boolean expression*
then *statement or compound statement*
else *statement or compound statement*
end if

Switch to Notebook

Metadata and Catalog

Metadata and Catalog

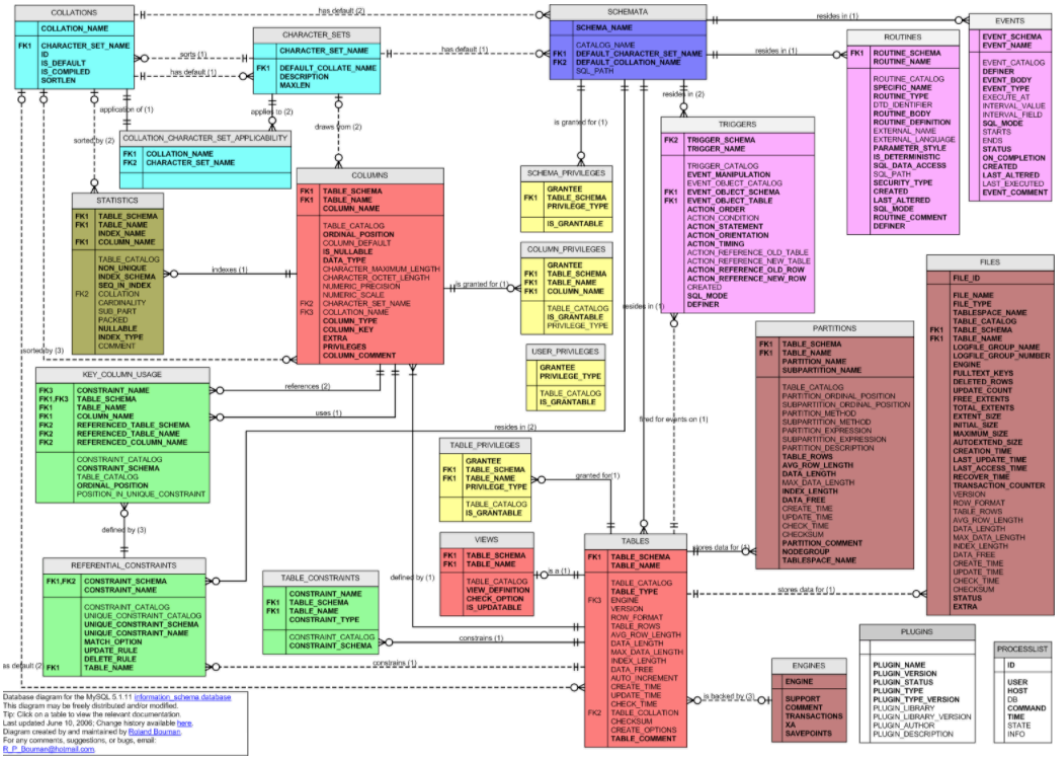
- ‘Metadata is "data that provides information about other data". In other words, it is "data about data". Many distinct types of metadata exist, including descriptive metadata, structural metadata, administrative metadata, reference metadata and statistical metadata.’ (https://en.wikipedia.org/wiki/Metadata)

- “The database catalog of a database instance consists of metadata in which definitions of database objects such as base tables, views (virtual tables), synonyms, value ranges, indexes, users, and user groups are stored. ... ...

  The SQL standard specifies a uniform means to access the catalog, called the INFORMATION_SCHEMA, but not all databases follow this ...” (https://en.wikipedia.org/wiki/Database_catalog)

- Codd’s Rule 4: Dynamic online catalog based on the relational model:
  - The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

MySQL Catalog (Information_Schema)



Some of the MySQL Information Schema Tables:

- 'ADMINISTRABLE_ROLE_AUTHORIZATIONS'
- 'APPLICABLE_ROLES'
- 'CHARACTER_SETS'
- 'CHECK_CONSTRAINTS'
- 'COLUMN_PRIVILEGES'
- 'COLUMN_STATISTICS'
- 'COLUMNS'
- 'ENABLED_ROLES'
- 'ENGINES'
- 'EVENTS'
- 'FILES'
- 'KEY_COLUMN_USAGE'
- 'PARAMETERS'
- 'REFERENTIAL_CONSTRAINTS'
- 'RESOURCE_GROUPS'
- 'ROLE_COLUMN_GRANTS'
- 'ROLE_ROUTINE_GRANTS'
- 'ROLE_TABLE_GRANTS'
- 'ROUTINES'
- 'SCHEMA_PRIVILEGES'
- 'STATISTICS'
- 'TABLE_CONSTRAINTS'
- 'TABLE_PRIVILEGES'
- 'TABLES'
- 'TABLESPACES'
- 'TRIGGERS'
- 'USER_PRIVILEGES'
- 'VIEW_ROUTINE_USAGE'
- 'VIEW_TABLE_USAGE'
- 'VIEWS'

- CREATE and ALTER statements modify the data.
- DBMS reads information:
  - Parsing
  - Optimizer
  - etc.

Switch to Notebook

More Complex Entity-Relationship Concepts

Backup