CSCI-620

Data Querying

DML

- Language to access and manipulate data (a query language)

- Two classes of query language
  - Pure
    - Relational Algebra
    - Tuple relational calculus
  - Practical
    - SQL
    - GraphQL

Roadmap

1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing

Roadmap

1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing

Relational Algebra

- Formal procedural query language
- Takes one or two relations as input and produces a relation as output
- Enables us to write queries that we will express in SQL and other languages

Procedural

Doctor			
ssn	lastName	salary	...
1	Smith	$97K	
2	James	$67K	
4	Smith	$120K	

"Result"			
ssn	lastName	salary	...
1	Smith	$97K	

$$\sigma_{lastName="\text{Smith}" \wedge\ salary<\$100\text{K}}(Doctor)$$

Restriction (Selection)

Doctor			
ssn	lastName	salary	...
1	Smith	$97K	
2	James	$67K	
4	Smith	$120K	

"Result"	
lastName	salary
Smith	$97K
James	$67K
Smith	$120K

$$\pi_{lastName,salary}(Doctor)$$

Projection

Doctor			
ssn	lastName	salary	...
1	Smith	$97K	
2	James	$67K	
4	Smith	$120K	

"Result"
lastName
Smith

$$\pi_{lastName}(\sigma_{lastName="\text{Smith}" \wedge\ salary<\$100\text{K}}(Doctor))$$

Composition

$$Doctor \times Patient$$

(Doctor.ssn, Doctor.firstName,
Doctor.lastName, Doctor.dob,
Patient.ssn, Patient.firstName,
Patient.lastName, Patient.dob,
middleName, salary, primaryDoctor)

Cartesian Product

$$\sigma_{Doctor.ssn=primaryDoctor}(Patient\ x\ Doctor)$$

Joins

$$\text{Supervisor} \bowtie \text{Employee}$$

$$\pi_{\text{SupervisorID,Department,EmployeeID,Name}}\Big(\rho_{\text{Supervisor.SupervisorID/SupervisorID}}\Big(\sigma_{\text{Supervisor.SupervisorID=Employee.SupervisorID}}\big(\text{Supervisor} \times \text{Employee}\big)\Big)\Big)$$

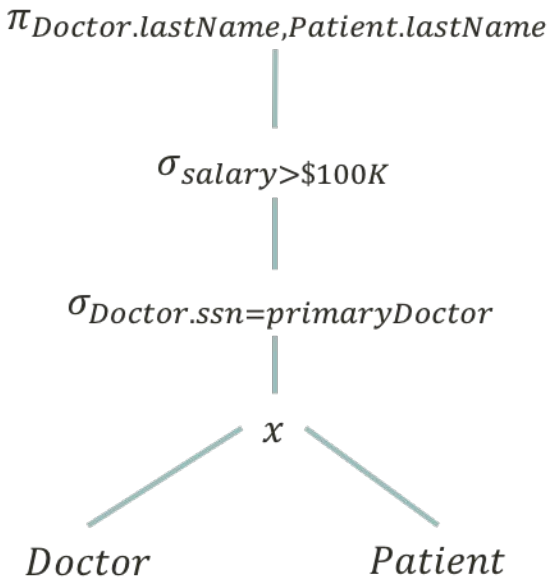
Natural Join

$$\text{Supervisor} \bowtie \text{Employee}$$

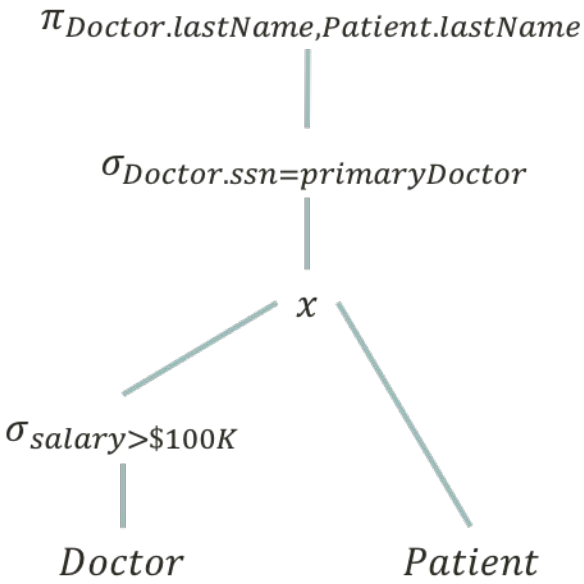
$$\pi_{\text{SupervisorID,Department,EmployeeID,Name}}\Big(\rho_{\text{Supervisor.SupervisorID/SupervisorID}}\Big(\sigma_{\text{Supervisor.SupervisorID=Employee.SupervisorID}}\big(\text{Supervisor} \times \text{Employee}\big)\Big)\Big)$$

Natural Join

Important for query optimization!



Original

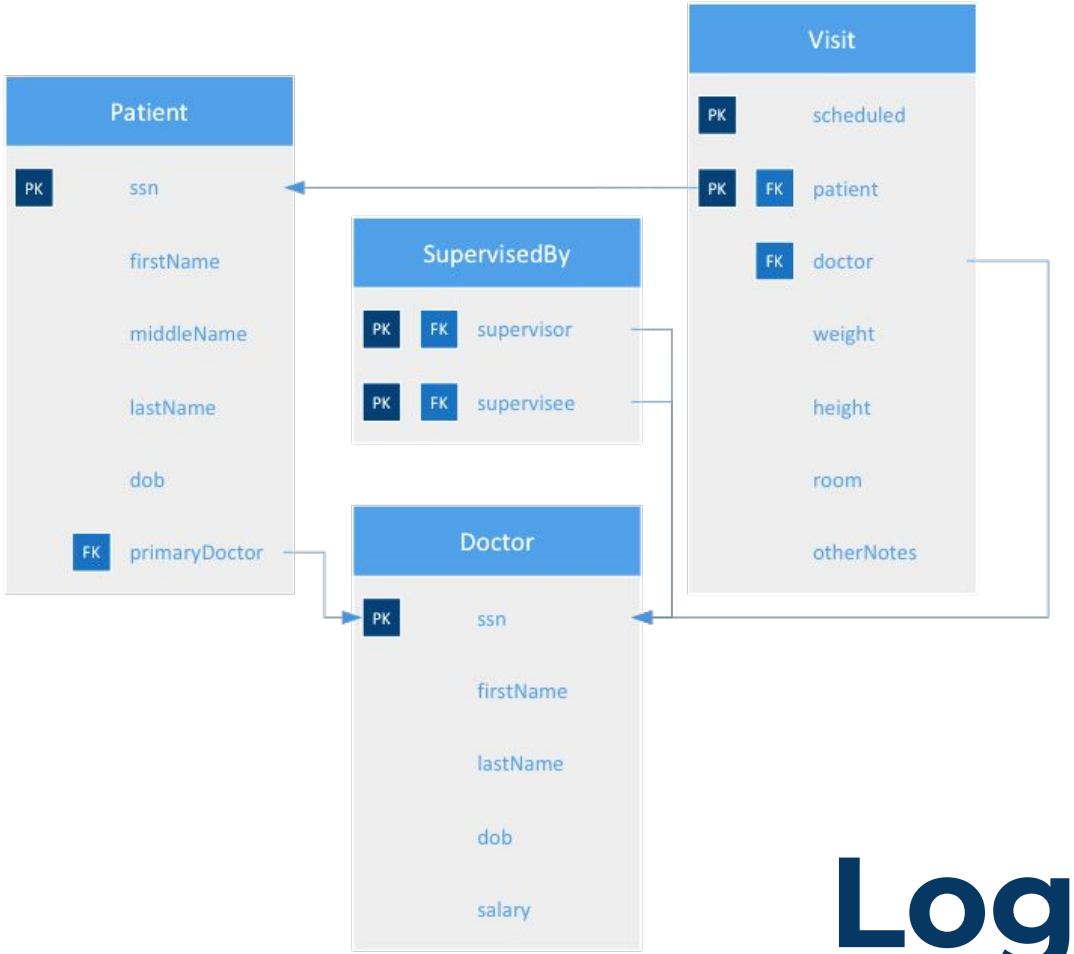


Equivalent

Equivalent Expressions

Other Operators

- Union **(∪)**
- Set difference **(−)**
- Rename **(ρ)** $\rho_{x(A_1, A_2, \ldots, A_n)}(E)$



Logical Model

Structured Query Language (SQL)

- Data Definition Language
  - CREATE
  - DROP
  - ALTER
- Data Manipulation Language
  - INSERT
  - DELETE
  - UPDATE
  - SELECT

Technology we are using

Connect to Postgres

```
$ psql -h reddwarf.cs.rit.edu
Password:

[mmior] #
```

Creating Tables

```
CREATE TABLE Patient (
    ssn CHARACTER(9),
    firstName VARCHAR(75) NOT NULL,
    middleName VARCHAR(75),
    lastName VARCHAR(75) NOT NULL,
    primaryDoctor CHARACTER(9),
    PRIMARY KEY (ssn),
    FOREIGN KEY (primaryDoctor)
    REFERENCES Doctor(ssn)
);
```

Test Data

Meaningful is key!

Requirements

- Non-functional
  (performance, scalability, etc.)
- Information - data to process
- Use cases - tasks users must perform

Devise Queries

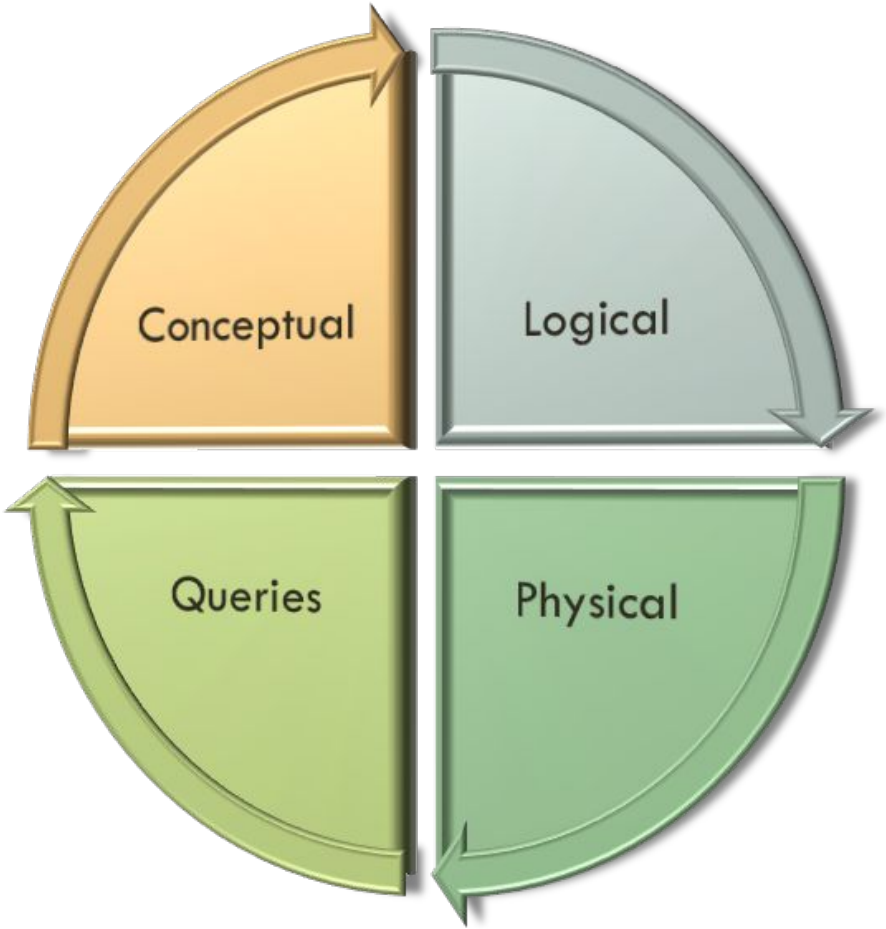
Understanding

- Retrieve all majors of students that are 30 years old or older or have no advisor
- Retrieve doctors that have attended the largest number of patients from 2014 to 2016 in a row
- Retrieve car models that have been manufactured between 1998 and 2013

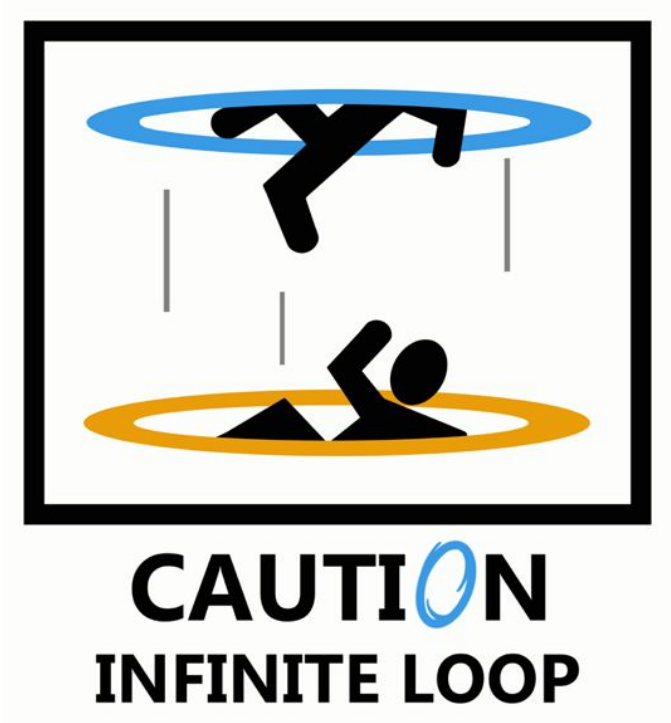
No questions during exams

Devise Queries

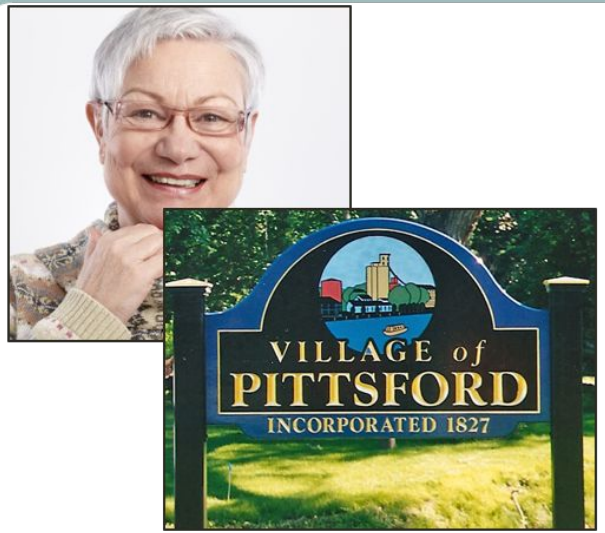
Don’t Know?



Refine



Everything else must still be valid!

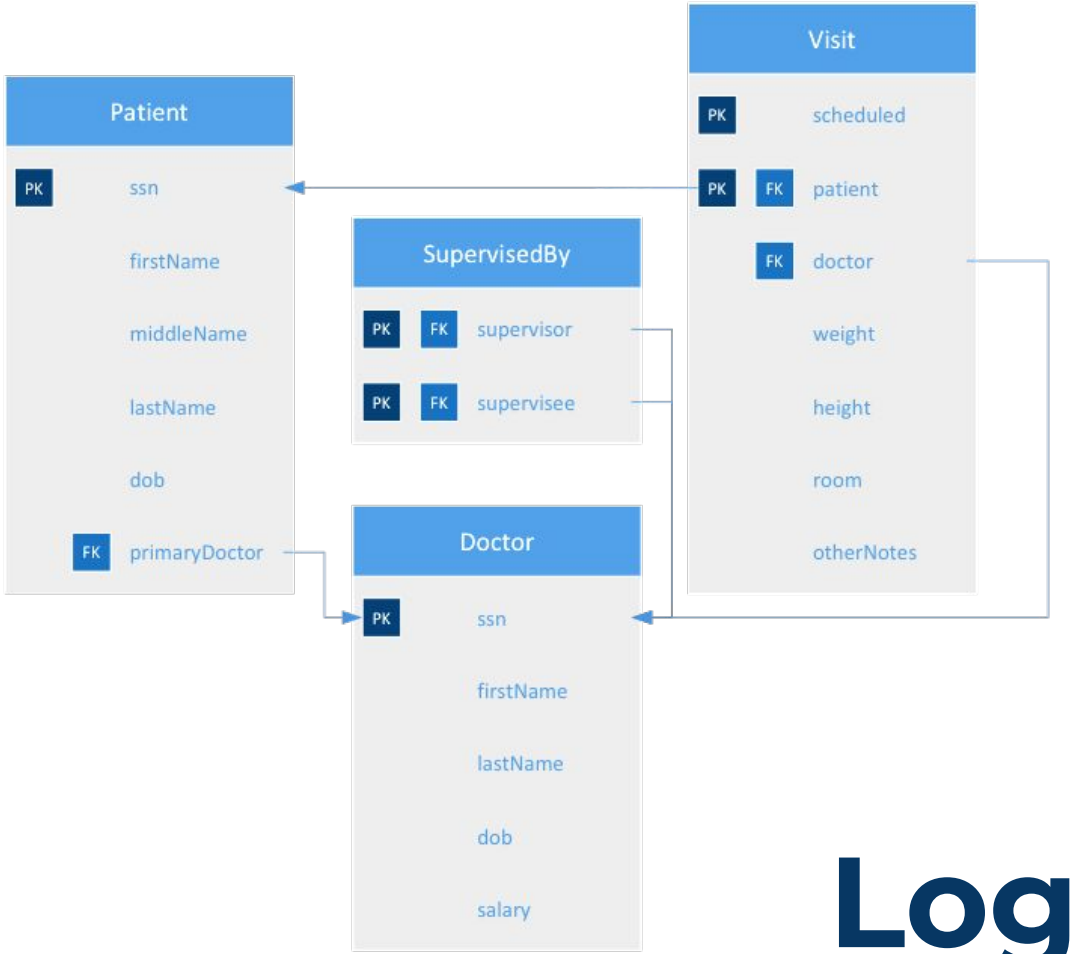


1. Go to the Patient folder
2. Open the file index
3. For each file
   3.1. If city = ‘Pittsford’
      3.1.1. Store patient
4. Return result

What

How

What but never how



Logical Model

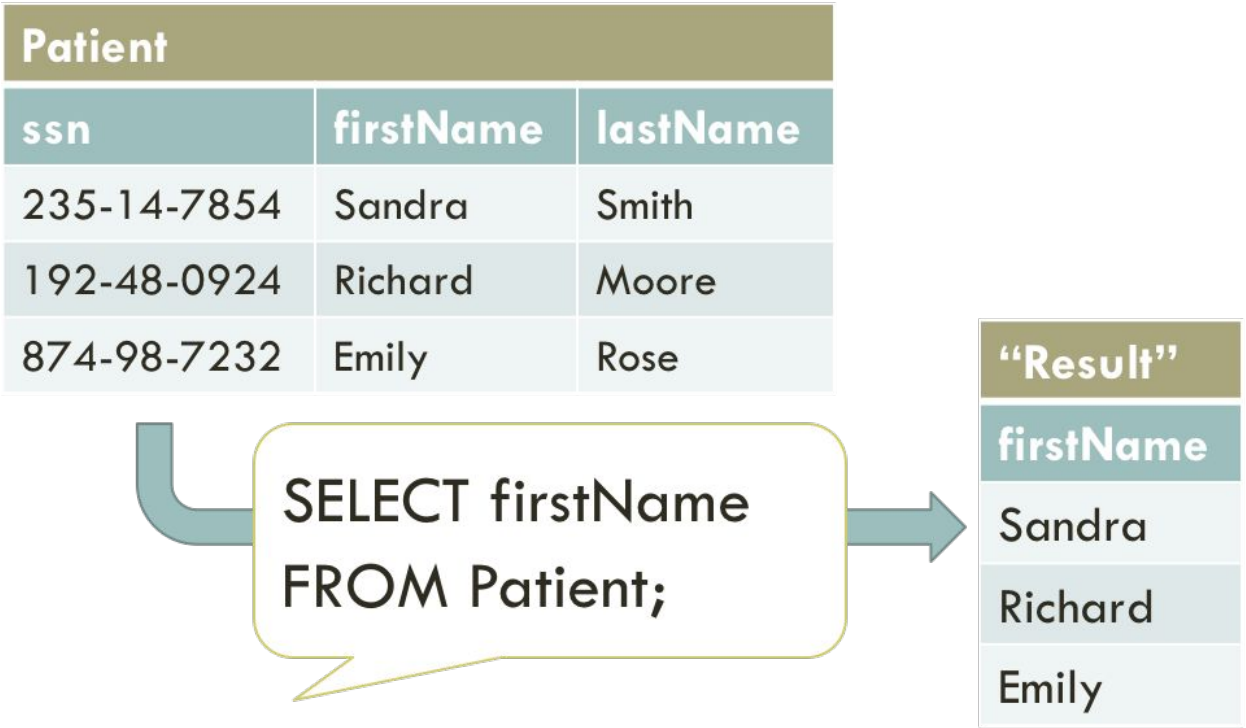
```
CREATE TABLE Patient (
  ssn CHAR(9),
  firstName VARCHAR(75) NOT NULL,
  middleName VARCHAR(75),
  lastName VARCHAR(75) NOT NULL,
  primaryDoctor CHARACTER(9),
  PRIMARY KEY (ssn),
  FOREIGN KEY (primaryDoctor)
REFERENCES Doctor(ssn)
);
```

SQL

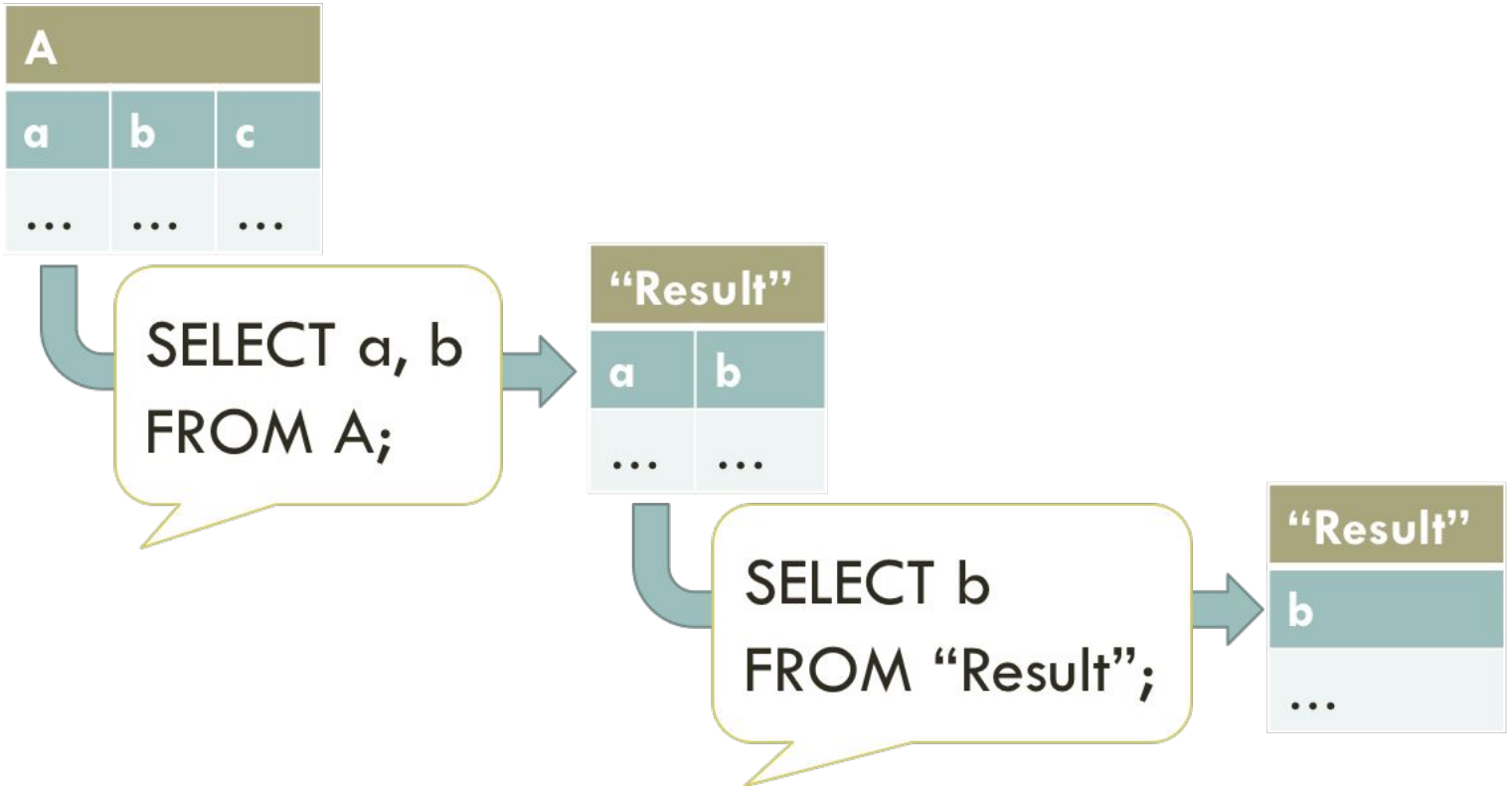
```
INSERT INTO Patient(ssn, firstName,
middleName, lastName, primaryDoctor)
VALUES ('134879381', 'Clark',
'Jerome', 'Kent', '984761647');
```


SQL

Warnings

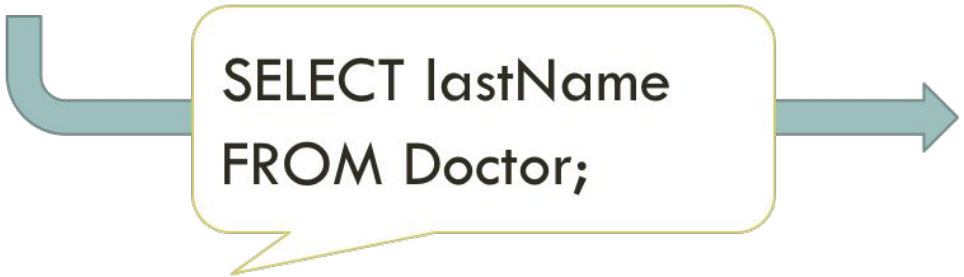


Simple query



Closure Property

Doctor		
ssn	firstName	lastName
943-23-9874	Rachel	Wang
862-74-3611	Chris	Patel
345-89-0122	Jignesh	Patel



"Result"
lastName
Wang
Patel
Patel

Another simple query

```
SELECT DISTINCT lastName
FROM Doctor;
```

"Result"
lastName
Wang
Patel

Distinct results

```
SELECT
   lastName, salary * 1.1
FROM Doctor;
```

Doctor

	firstName	lastName	salary
874	Rachel	Wang	$175K
611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

"Result"

lastName	"salary * 1.1"
Wang	$192.5K
Patel	$137.5K
Patel	$169.4M

Arithmetic

```
SELECT
   lastName, salary * 1.1 AS projected
FROM Doctor;
```

Doctor

lastName	salary
Wang	$175K
Patel	$125K
Patel	$154K

"Result"

lastName	projected
Wang	$192.5K
Patel	$137.5K
Patel	$169.4K

Renaming

```
SELECT *
FROM Doctor;
```

Doctor

ssn	firstName	lastName	salary
943-23-9874	Rachel	Wang	$175K
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

"Result"

ssn	firstName	lastName	salary
943-23-9874	Rachel	Wang	$175K
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

Star

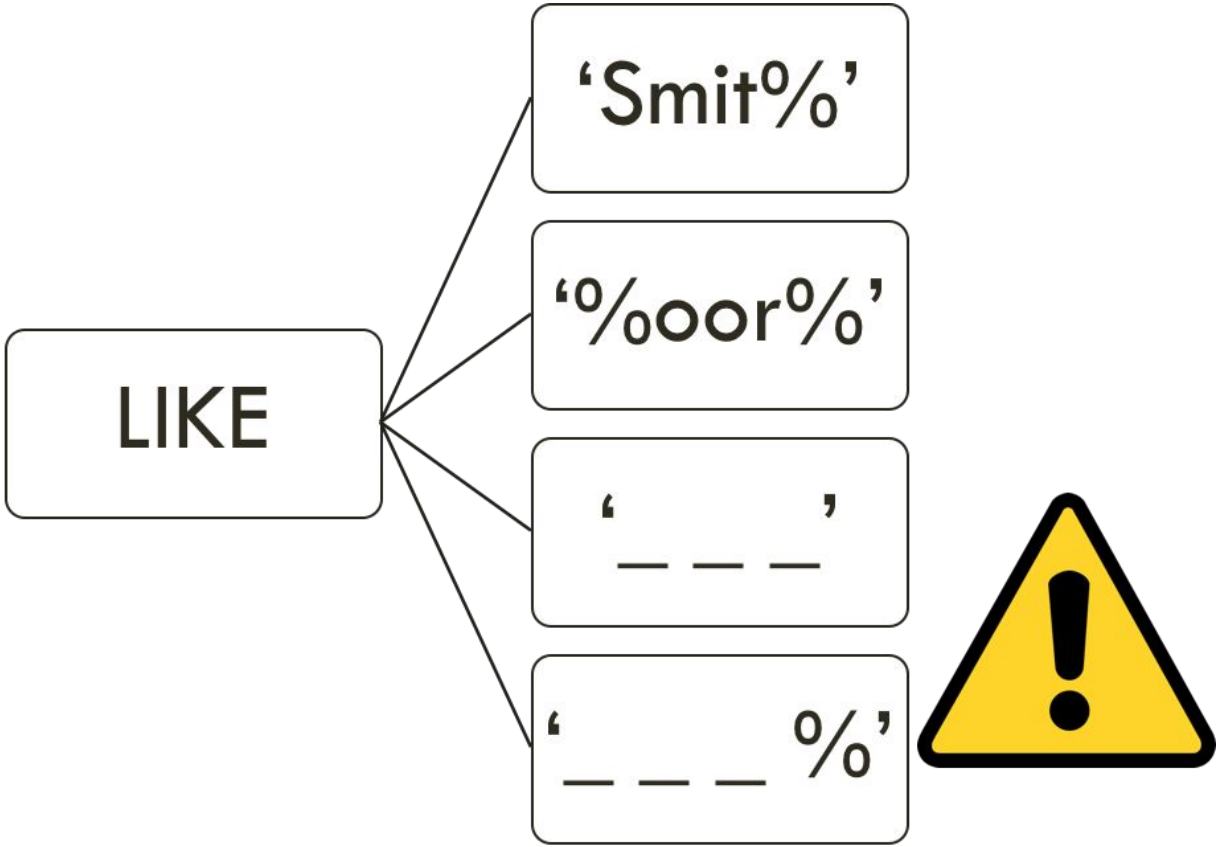
```
SELECT ssn, firstName
FROM Doctor WHERE
    (salary < $130K AND lastName = 'Patel')
        OR salary > $170K;
```

salary
$175K
$125K
$154K

"Result"

ssn	firstName
943-23-9874	Rachel
862-74-3611	Chris

Conditions



String Operations

```
SELECT ssn
FROM Doctor
WHERE lastName LIKE 'Smit%';
```

No indexes!

```
"foo" || "bar"          => "foobar"
CHAR_LENGTH('Hello')  => 5
LOWER('SQL')          => 'sql'
UPPER('yay')          => 'YAY'
...
```

String Operations

```
SELECT ssn
FROM Patient
WHERE
   middleName IS NOT NULL;
```

Null values

Doctor			
		lame	salary
		g	$175K
			$125K
345-89-0122	Jignesh	Patel	$154K

```
SELECT ssn, lastName FROM Doctor
ORDER BY lastName ASC, salary DESC;
```

"Result"	
ssn	lastName
345-89-0122	Patel
862-74-3611	Patel
943-23-9874	Wang

Sorting

Roadmap

1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing

```
Connection con = null;
PreparedStatement st = null;
ResultSet rs = null;

String url = "jdbc:postgresql://reddwarf/?currentSchema=foodmart";
String user = "USERNAME";
String pwd = "PASSWORD";
```

Programmatic Access

```
try {
    con = DriverManager.getConnection(url, user, pwd);
    st = con.prepareStatement("SELECT ssn FROM Patient");
    rs = st.executeQuery();

    if (rs.next())
        System.out.println(rs.getString("ssn"));
}
```

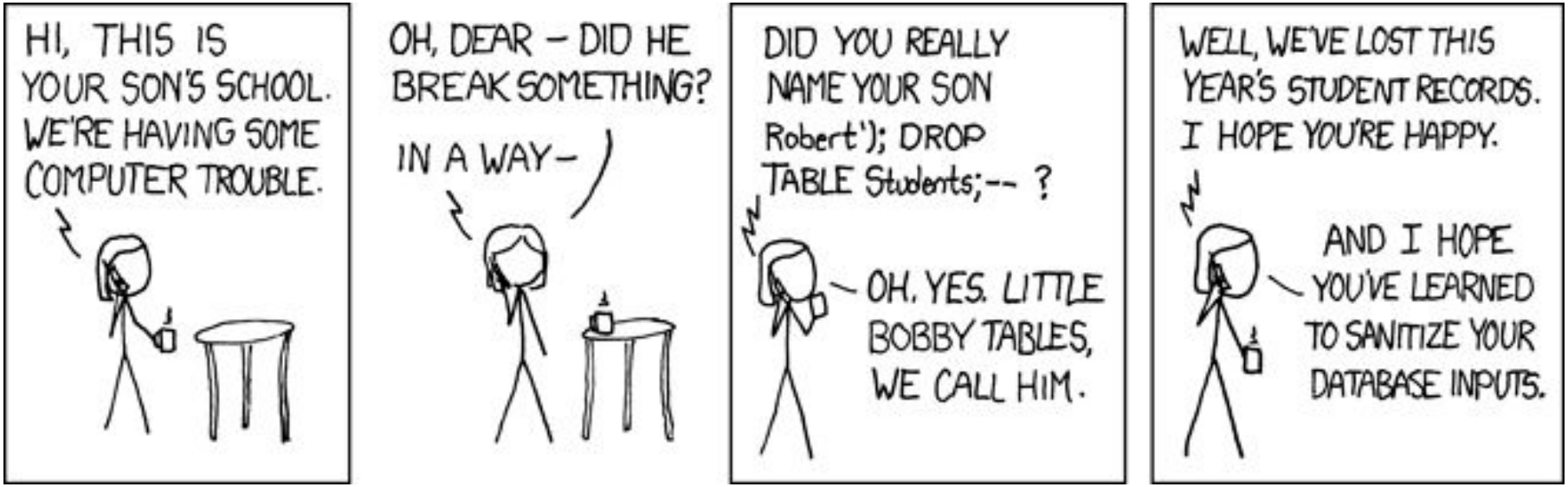
Programmatic Access

```
catch (SQLException oops) {
   System.out.println(“Something went wrong.”);
} finally {
   try {
    if (rs != null)   rs.close();
    if (st != null)   st.close();
    if (con != null)   con.close();
   catch (SQLException oops) {
    System.out.println(“Something went REALLY wrong.”);
   }
}
```

Programmatic Access

```
ps = con.prepareStatement("SELECT * FROM users " +
     "WHERE name = ?");
…
ps.setString(1, username);
…
```

Use prepared statements



Source: xkcd.com

SQL Injection

Researchers find SQL injection to bypass airport TSA security checks

By Sergiu Gatlan

August 30, 2024 03:02 PM 2

Security researchers have found a vulnerability in a key air transport security system that allowed unauthorized individuals to potentially bypass airport security screenings and gain access to aircraft cockpits.

SQL Injection

Source: BleepingComputer

Roadmap

1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing

INSERT INTO …
SELECT …;

DELETE FROM …
WHERE …;

UPDATE … SET …
WHERE …;

Insert, Delete, Update

```
DELETE FROM Doctor WHERE ssn IN
(SELECT primaryDoctor FROM Patient
WHERE ssn=3);
```

Delete

```
UPDATE Doctor SET salary = salary * 1.1 WHERE
ssn IN (SELECT supervisor FROM SupervisedBy);
```

Update

```
INSERT INTO SupervisedBy (SELECT 9, ssn FROM
Doctor WHERE NOT EXISTS (SELECT * FROM
SupervisedBy WHERE supervisee=ssn));
```

Insert

```
try {
    con.setAutoCommit(false);
    st1 = con.prepareStatement("INSERT INTO " +
        "Payment(id, paymentDate, amount, method) " +
        "VALUES (743, '09/02/16', 536, 'Check')");
    st2 = con.prepareStatement("INSERT INTO " +
        "IsPaidBy(bill, payment) " +
        "VALUES (123, 743)");
```

Transactions

```
    st1.executeUpdate();
    st2.executeUpdate();
    con.commit();
} catch (SQLException oops) {
    …
    con.rollback();
    …
}
```

Transactions

```
DROP TABLE Patient;
ALTER TABLE Patient DROP middleName;
ALTER TABLE Patient ADD id INT;
ALTER TABLE Patient RENAME
  lastName TO surname;
ALTER TABLE Patient ADD FOREIGN KEY …;
```

Change tables

Integrity Constraints

- PRIMARY KEY
- FOREIGN KEY
- NOT NULL
- UNIQUE
- CHECK

Referential Integrity

- Ensures that a value appearing in one relation appears in another
- This is enforced by foreign keys

```
CREATE TABLE Account(id INTEGER
PRIMARY KEY, balance FLOAT,
CHECK (balance >= 0));
```

CHECK Constraints

OLTP vs OLAP

- **OLTP**

  Online Transaction Processing
  - High volume of transactions
  - Reads, writes, updates, deletes

- **OLAP**

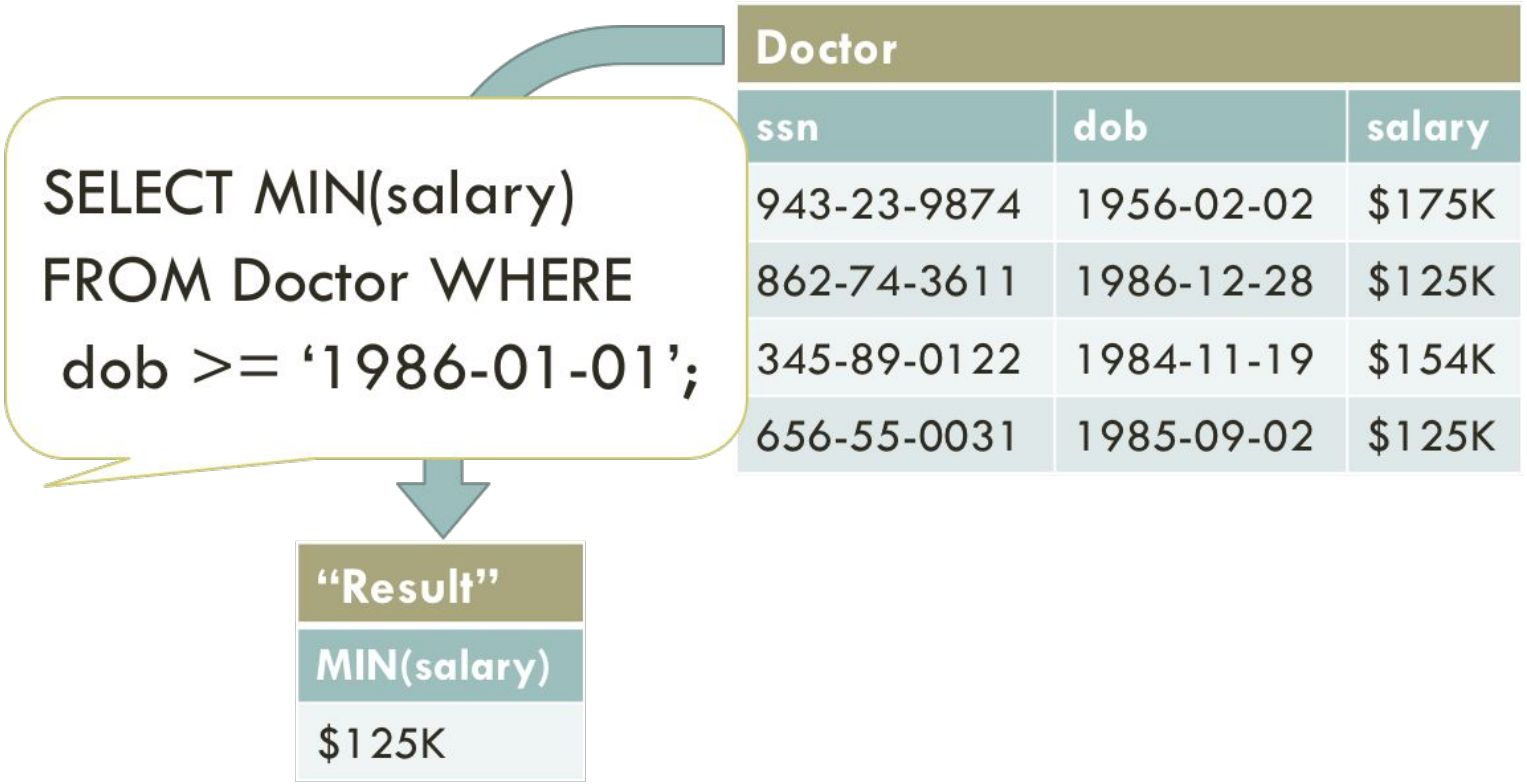
  Online Analytical Processing
  - Smaller volume of complex queries
  - Usually read-only

Roadmap

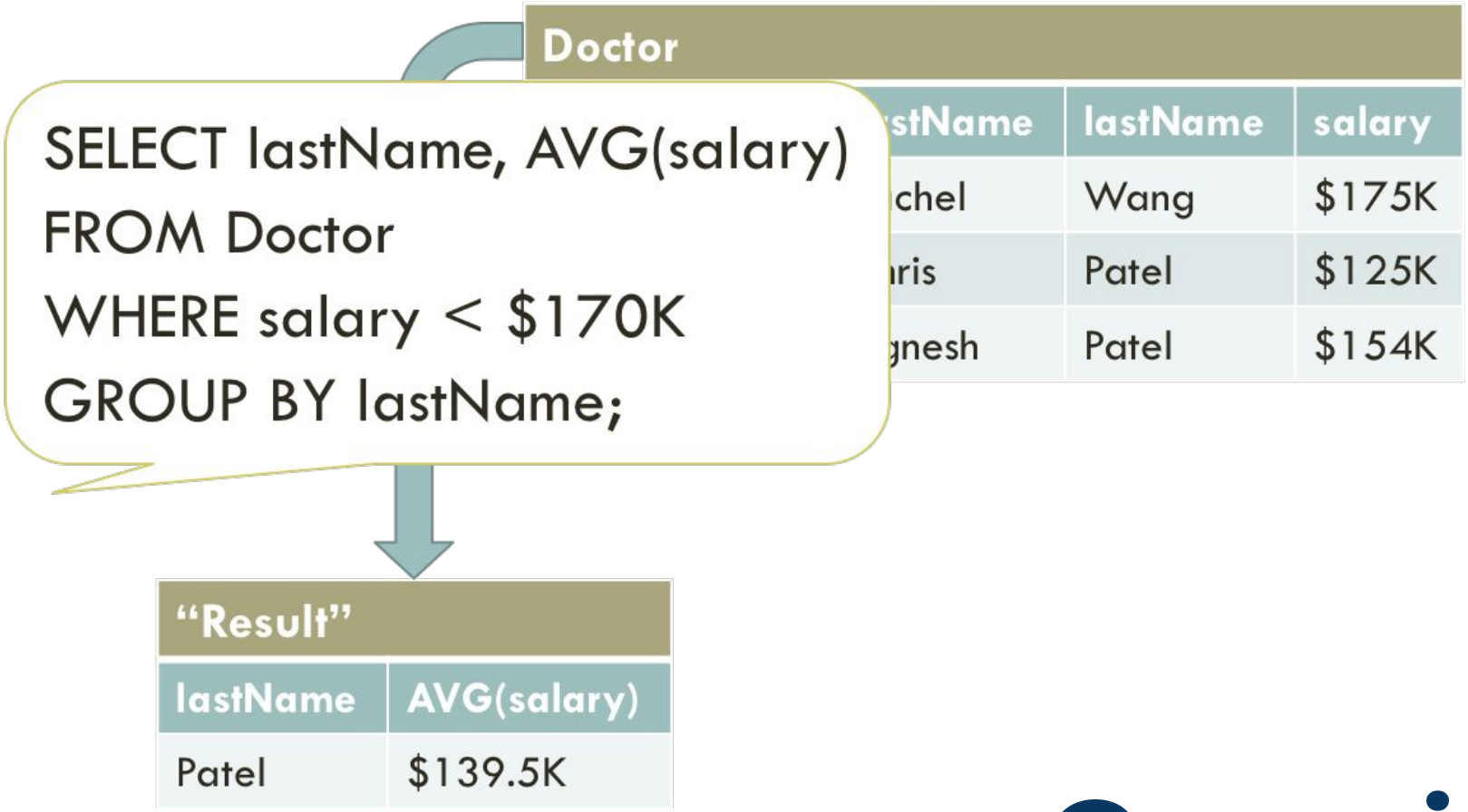
1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing

Aggregation

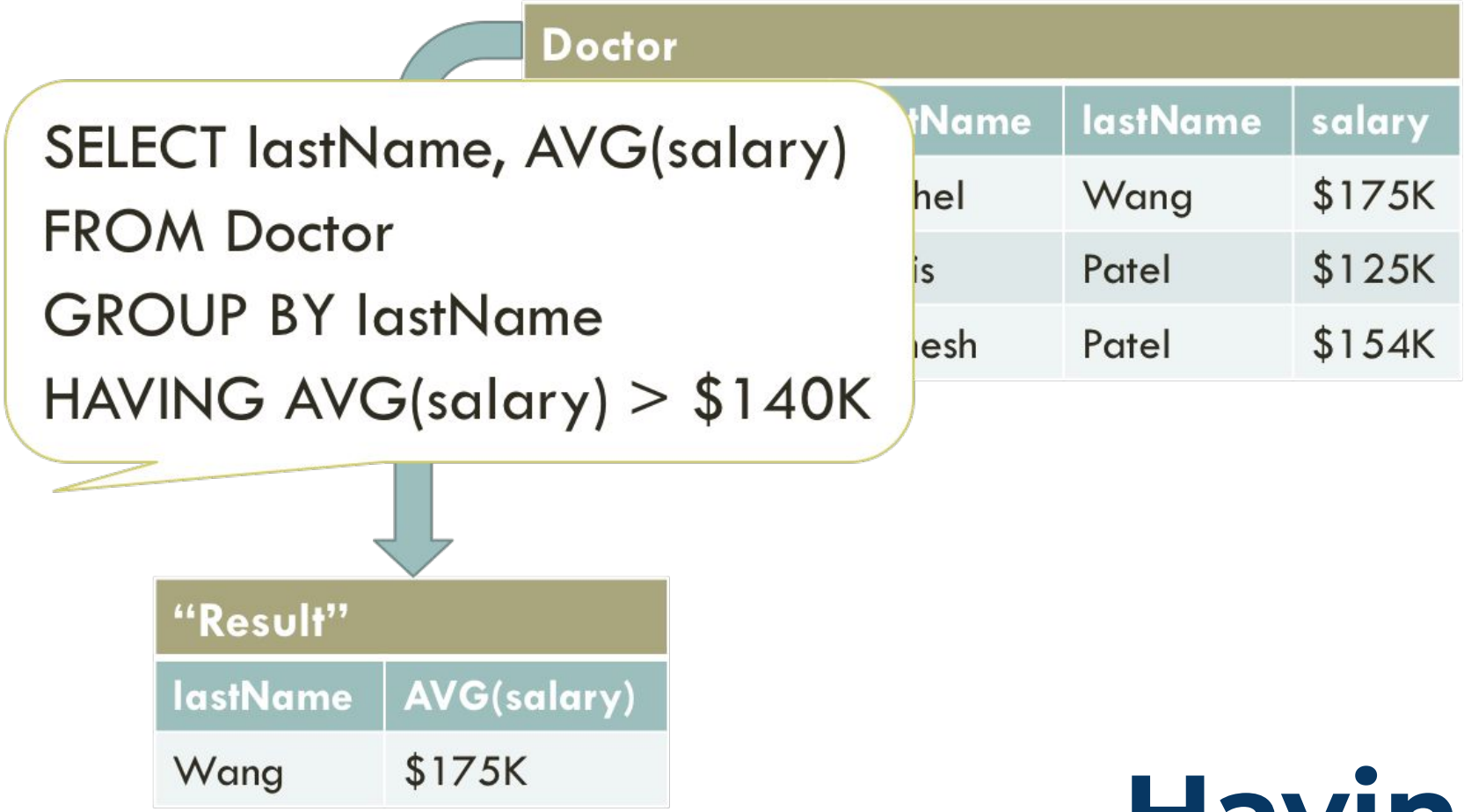
- SQL has five aggregation functions:
  - Average (AVG)
  - Count (COUNT)
  - Minimum (MIN)
  - Maximum (MAX)
  - Sum (SUM)



Basic Aggregation


Doctor
stName
lastName
salary
chel
Wang
$175K
ris
Patel
$125K
gnesh
Patel
$154K
SELECT lastName, AVG(salary)
FROM Doctor
WHERE salary < $170K
GROUP BY lastName;
"Result"
lastName
AVG(salary)
Patel
$139.5K

Grouping


Doctor
tName
lastName
salary
hel
Wang
$175K
is
Patel
$125K
esh
Patel
$154K
SELECT lastName, AVG(salary)
FROM Doctor
GROUP BY lastName
HAVING AVG(salary) > $140K
"Result"
lastName
AVG(salary)
Wang
$175K

Having

Roadmap

1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing

Doctor		
ssn	last	salary
1	Smith	80K
2	James	90K

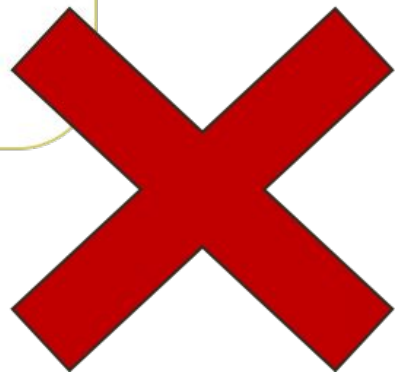
Patient		
ssn	last	prim
7	Bailey	1
8	Jolie	2

```
SELECT * FROM
  Doctor AS d,
  Patient AS p;
```

"Result"					
d.ssn	p.ssn	d.last	p.last	salary	prim
1	7	Smith	Bailey	80K	1
1	8	Smith	Jolie	80K	2
2	7	James	Bailey	90K	1
2	8	James	Jolie	90K	2

Cartesian Products

```
SELECT d.lastName, p.lastName
FROM Doctor AS d, Patient AS p
WHERE
    p.primaryDoctor = d.ssn;
```



“Handmade” joins

```
SELECT d.lastName, p.lastName
FROM Doctor AS d JOIN Patient AS p
    ON p.primaryDoctor = d.ssn;
```

Join using ON

Doctor

ssn	last	salary
1	Smith	80K
2	James	90K

Patient

ssn	last	prim
7	Bailey	1
8	Jolie	2

```
SELECT d.lastName, p.lastName
FROM Doctor AS d JOIN Patient AS p
    ON p.primaryDoctor = d.ssn;
```

"Result"

d.ssn	p.ssn	d.last	p.last	salary	prim
1	7	Smith	Bailey	80K	1
~~1~~	~~8~~	~~Smith~~	~~Jolie~~	~~80K~~	~~2~~
~~2~~	~~7~~	~~James~~	~~Bailey~~	~~90K~~	~~1~~
2	8	James	Jolie	90K	2

Join = Product + Filter

```
SELECT DISTINCT sr.lastName
FROM Doctor AS sr JOIN SupervisedBy
                        ON sr.ssn = supervisor
        JOIN Doctor AS se
                        ON se.ssn = supervisee
WHERE    se.salary > $100K;
```

Self joins

Doctor		
ssn	lastName	salary
1	Smith	$99K
2	James	$95K
3	Perez	$98K
4	Patel	$155K
5	Johnson	$105K

SupervisedBy	
supervisor	supervisee
1	2
1	3
4	5

Data

"Result"			
supervisor	sr.lastName	sr.salary	supervisee
1	Smith	$99K	2
1	Smith	$99K	3
4	Patel	$155K	5

First join

"Result"					
supervisor	sr.lastName	sr.salary	supervisee	se.lastName	se.salary
1	Smith	$99K	2	James	$95K
1	Smith	$99K	3	Perez	$98K
4	Patel	$155K	5	Johnson	$105K

Second join and filtering

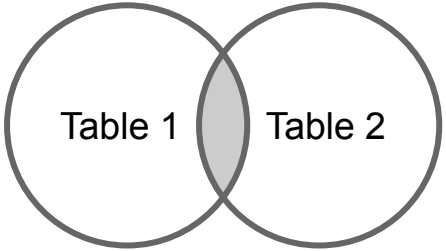
```
SELECT d.lastName, p.lastName
FROM Doctor AS d JOIN Patient AS p
    ON p.primaryDoctor = d.ssn;
```

1. For each tuple **d** in Doctor
    1.1 For each tuple **p** in Patient
        1.1.1 Combine **d** and **p** in new tuple **t**
        1.1.2 If **p**.primaryDoctor = **d**.ssn
            1.1.2.1 Add **t** to result
1.2 Return result

Interpretation of joins

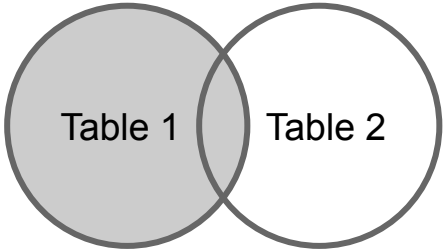
Types of joins

Inner Join

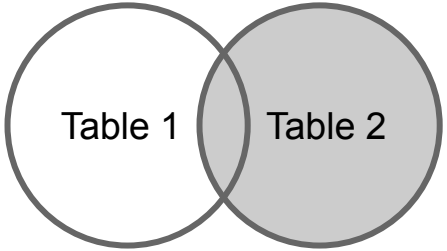


Outer Join

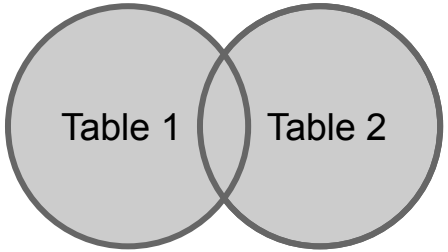
Left



Right



Full



```
SELECT ssn, lastName, supervisor
FROM Doctor AS se LEFT JOIN SupervisedBy
                    ON se.ssn = supervisee;
```

Left outer join

Doctor		
ssn	lastName	salary
1	Smith	$99K
2	James	$95K
3	Perez	$98K
4	Patel	$155K
5	Johnson	$105K

SupervisedBy	
supervisor	supervisee
1	2
1	3
4	5

Data

"Result"		
ssn	lastName	supervisor
1	Smith	<null>
2	James	1
3	Perez	1
4	Patel	<null>
5	Johnson	4

Results


Efficiency
Straight Ahead

Nested-loop

Sort-merge

Hash

Strategies for joining

1 For each tuple **d** in Doctor

1.1 For each tuple **p** in Patient

1.1.1 Combine **d** and **p** in new tuple **t**

1.1.2 If **p**.primaryDoctor = **d**.ssn

1.1.2.1 Add **t** to result

2 Return result

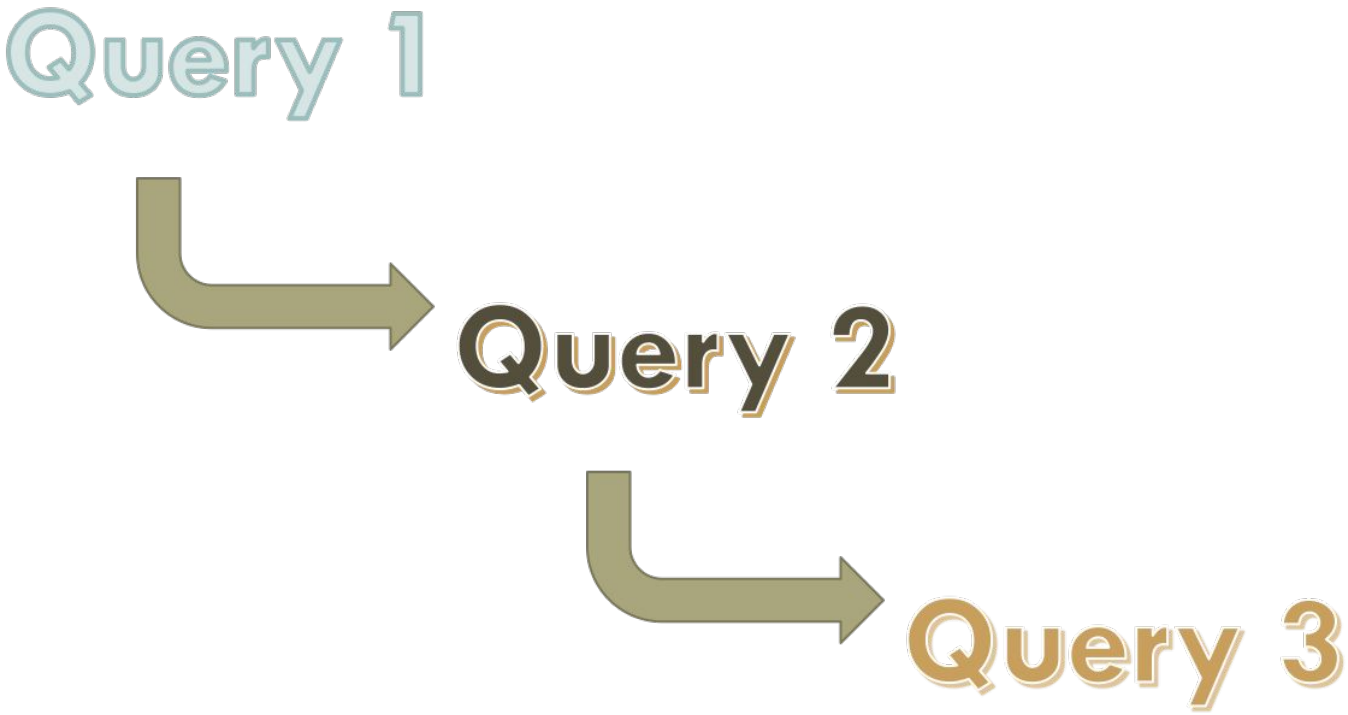
Nested loop $O(n^2)$

1 Sort both relations Doctor and Patient
2 While i < Size(Doctor) AND j < Size(Patient)
    2.1 If **Patient[j]**.primaryDoctor != **Doctor[i]**.ssn
        2.1.1 Advance either i or j
    2.2 Else
        2.2.1 Combine **Patient[j]** and **Doctor[i]** in **t**
        2.2.2 Add **t** to result
        2.2.3 Advance both i and j
3 Return result

Sort merge O(n log n)

1 Hash all values of primaryDoctor for Patient
2 For each tuple **y** in Doctor
    2.1 If Hash(**y**.ssn) is in the hash table of Patient
        2.2.1 Let **x** be the tuple in the hash table
        2.2.2 Combine **x** and **y** in **t**
        2.2.3 Add **t** to result
3 Return result

Hash O(n)



Closure property

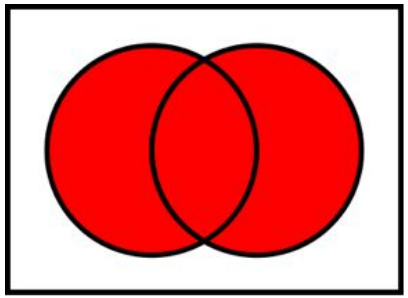
Doctor		
lastName	dob	…
Smith	10/30/1980	
James	01/08/1975	
Perez	09/02/1982	

Patient		
lastName	dob	…
Smith	05/17/1985	
Perez	11/30/1985	

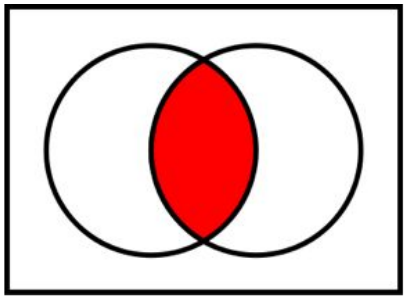
"Result"
lastName
Smith
James
Perez

```
(SELECT lastName FROM Patient
WHERE dob >= '1980-09-01')
UNION
(SELECT lastName FROM Doctor
WHERE dob >= '1980-09-01');
```

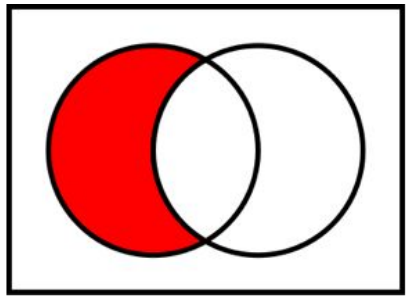
Union



Union



Intersection



Difference

Set operations

Doctor		
ssn	lastName	…
1	Smith	
2	James	
3	Perez	

Patient		
ssn	lastName	…
1	Smith	
2	James	
7	Smith	

```
(SELECT ssn FROM Doctor)
INTERSECT
(SELECT ssn FROM Patient);
```

"Result"
ssn
1
2

Intersection

Doctor	
ssn	…
1	
2	
3	

Patient		
ssn	primaryDoctor	…
7	1	
8	2	
9	2	

```
(SELECT ssn FROM Doctor)
EXCEPT
(SELECT primaryDoctor AS ssn
FROM Patient);
```

"Result"
ssn
3

Difference

```
SELECT DISTINCT lastName
FROM Doctor AS d JOIN Patient
    ON d.ssn = primaryDoctor
WHERE
  d.ssn IN (
        SELECT supervisee
        FROM SupervisedBy JOIN Doctor
            ON supervisor = ssn
        WHERE salary > $100K);
```

Set membership

Doctor		
ssn	lastName	salary
1	Smith	$99K
2	James	$95K
3	Perez	$98K
4	Patel	$155K
5	Johnson	$105K

SupervisedBy	
supervisor	supervisee
1	2
1	3
4	5

Patient	
ssn	primaryDoctor
10	2
11	5
12	5

Data

"Result"			
supervisor	supervisee	lastName	salary
1	2	James	$95K
1	3	Perez	$98K
4	5	Johnson	$105K

"Result"
supervisee
5

Inner query (join/filter)

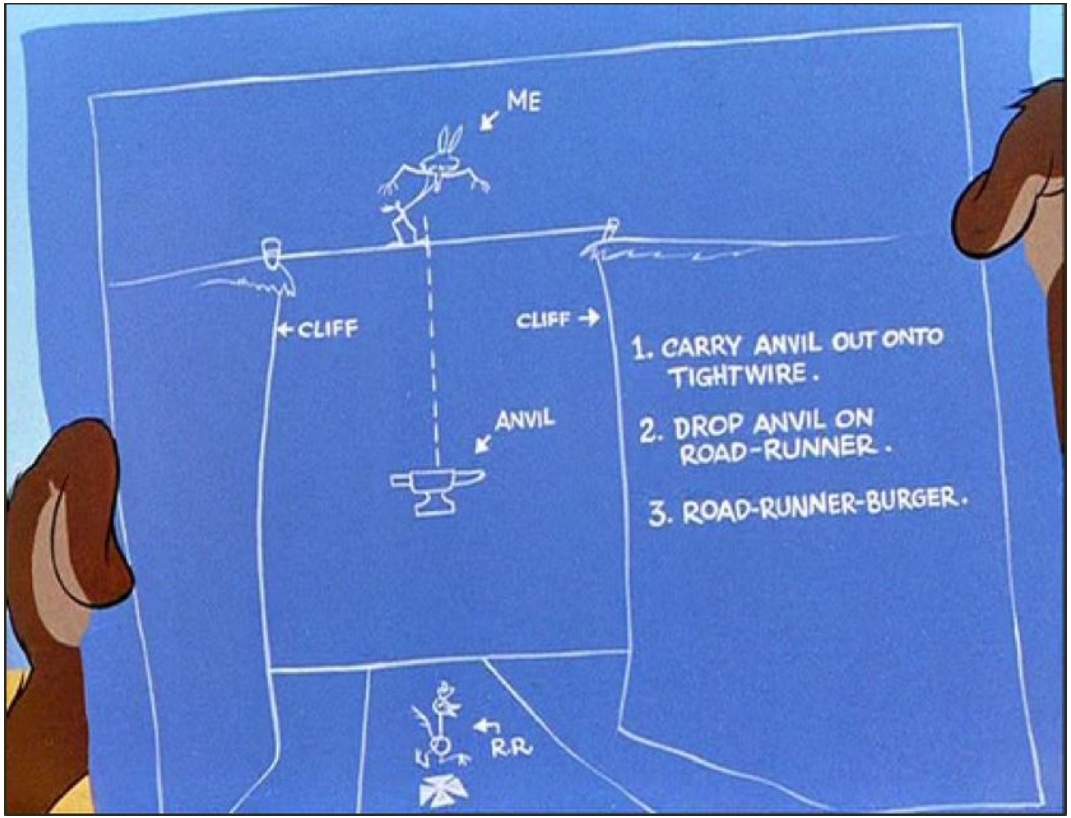
"Result"			
d.ssn	lastName	salary	ssn
2	James	$95K	10
5	Johnson	$105K	11
5	Johnson	$105K	12

"Result"
supervisee
5

Outer query (join/filter)

Roadmap

1. Relational Algebra
2. Basic SQL
3. Programmatic access
4. Updates
5. Aggregation
6. Joins, sets, and nested queries
7. Indexing



Execution plan

Query Execution

- Database engines can’t directly execute relational algebra operators
- For each operator, the database needs to decide on a *plan* for execution
- For example, choosing a particular algorithm to execute a join

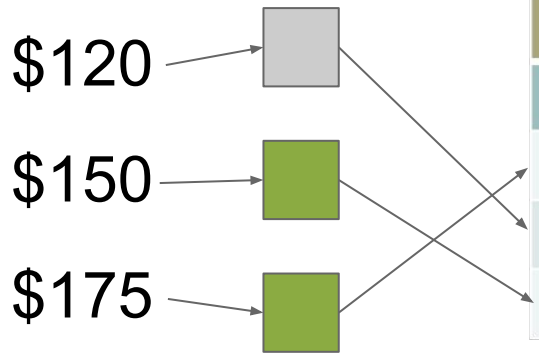
Execution using indexes

- Without any secondary indexes, the database must execute a *table scan* (read every record) unless queries use the primary key
- With indexes, the database can sometimes find values matching query conditions without scanning

Execution using indexes

- Indexes can be used in several cases
  - Filtering conditions in the WHERE clause when columns are indexed
  - Sorting when the index is sorted in the same order as the desired result
  - Sometimes whole queries can be answered using an index

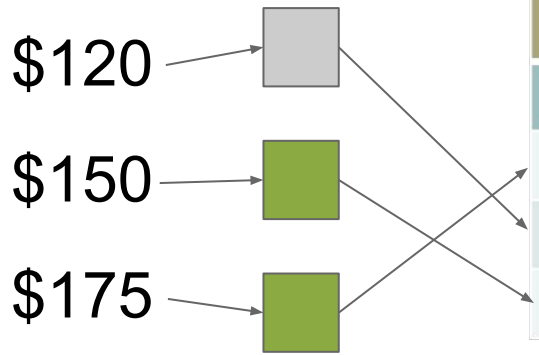
SELECT * FROM Doctor
WHERE salary > $150K



Doctor			
ssn	firstName	lastName	salary
943-23-9874	Rachel	Wang	$175K
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

Simple index usage

```
SELECT * FROM Doctor
WHERE salary > $150K
and lastName = 'Patel'
```



Doctor			
ssn	firstName	lastName	salary
~~943-23-9874~~	~~Rachel~~	~~Wang~~	~~$175K~~
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

Partial filtering

Creating indexes

- Indexes can only be created on a single table (but indexing columns used for joins will still help those queries)
- Postgres has many different types of index, but we will focus on the default B-tree index which is ordered

```
CREATE INDEX
doctor_salary_lastname ON
Doctor(salary, lastName);
```

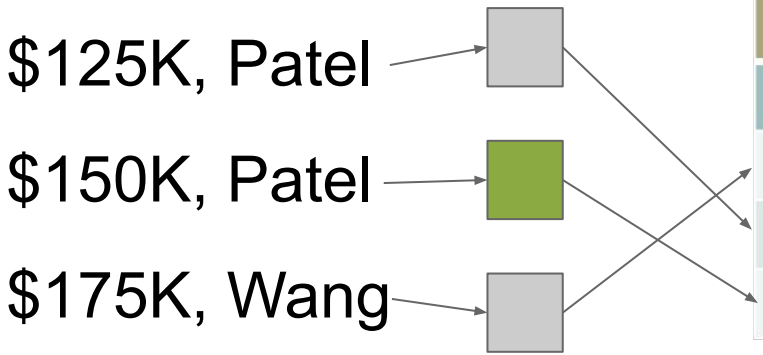
CREATE INDEX

Multiple column indexes

- Indexes can be created on more than one column in the same table
- Column values are combined to create the search key
- The database can efficiently use the index for queries using any prefix of the columns which are in the index

Note that in the examples that follow, the indexes are sorted on both salary and last name. This is an unfortunate coincidence and *not* true in the general case. (Sorting is on the first column and *then* the second column).

```
SELECT * FROM Doctor
WHERE salary > $150K
and lastName = 'Patel'
```

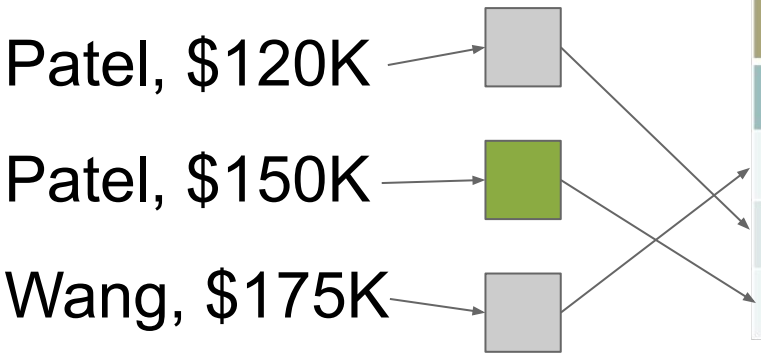


Doctor

ssn	firstName	lastName	salary
943-23-9874	Rachel	Wang	$175K
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

Multiple column indexes

SELECT * FROM Doctor
WHERE salary > $150K
and lastName = 'Patel'



Doctor

ssn	firstName	lastName	salary
943-23-9874	Rachel	Wang	$175K
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

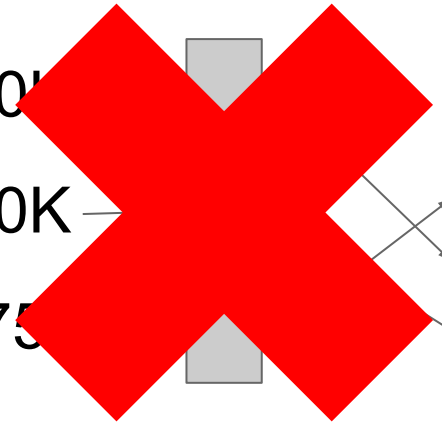
Multiple column indexes

SELECT * FROM Doctor
WHERE salary > $150K

Patel, $120

Patel, $150K

Wang, $175



Doctor			
ssn	firstName	lastName	salary
943-23-9874	Rachel	Wang	$175K
862-74-3611	Chris	Patel	$125K
345-89-0122	Jignesh	Patel	$154K

Multiple column indexes

EXPLAIN SELECT lastName FROM Doctor WHERE salary < $170K

No index

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on doctor  (cost=0.00..3.25 rows=75 width=7)
   Filter: (salary < '170000'::double precision)
```

Postgres must scan the whole table

EXPLAIN

EXPLAIN SELECT lastName FROM Doctor WHERE salary < $170K

Index on salary

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on doctor  (cost=8.72..11.66 rows=75 width=7)
   Recheck Cond: (salary < '170000'::double precision)
   -> Bitmap Index Scan on doctor_salary (cost=0.00..8.71 rows=75 width=0)
        Index Cond: (salary < '170000'::double precision)
```

Postgres can use the index to find matching records, but then must go to the table to get the lastName value

EXPLAIN SELECT lastName FROM Doctor WHERE salary < $170K

Index on salary, lastName

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using doctor_salary_lastname on doctor (cost=0.14..9.46
                                                         rows=75 width=7)
   Index Cond: (salary < '170000'::double precision)
```

Postgres can use the index to answer the entire query