Overview of the Relational data model

Goals for

Today & Next week

(SQL, SQL, SQL)

Phase I: Intuition for SQL (1st half of today)

- Basic Relational model (aka tables)
- SQL concepts we'll study (similar to Python map-reduce)
- Example SQL (exploring real datasets)

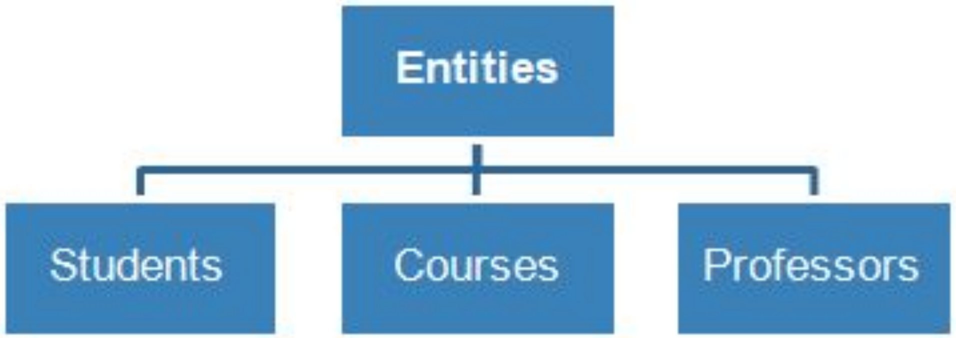
Phase II: Formal description

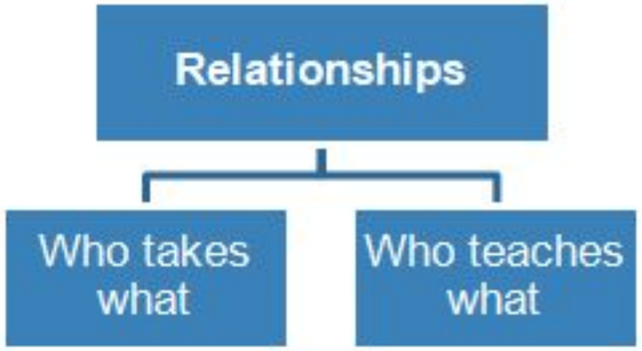
- Schemas, Query structure of SELECT-FROM-WHERE, JOINs, etc

A Motivating Example

A basic Course Management System (CMS):

Entities (e.g., Students, Courses)
Relationships (e.g., Alice is enrolled in 145)





Example Spreadsheet Tables

Students

File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive

Students

SID	Name	GPA
40001	Mickey	3.2
40002	Daffy	3.6
50003	Donald	3.3
50004	Minnie	3.9
10008	Pluto	4

Courses

CID	C-Name	Room
1012	CS145 - Toon systems	Nvidia
1017	CS161 - Animation art	Gates 300
1019	CS245 - Painting town red	Packard 45

Enrolled

SID	CID	Grade
40001	1012	A
50004	1012	A-
40001	1017	B+

Logical Schema

Student(sid: *string*, name: *string*, gpa

Courses(cid: *string*, c-name: *string*, r *string*)

Enrolled(sid: *string*, cid: *string*, grade

Keys [connect tables]

sid: Connects Enrolled to Stude

cid: Connects Enrolled to Cours

Queries ["compute" over tables]

- Minnie's GPA?
- AVG student GPA?
- Mickey's classes?
- AVG student GPA in CS145?

Example Spreadsheet Tables

Students

File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive

Students

SID	Name	GPA
40001	Mickey	3.2
40002	Daffy	3.6
50003	Donald	3.3
50004	Minnie	3.9
10008	Pluto	4

Enrolled

SID	CID	Grade
40001	1012	A
50004	1012	A-
40001	1017	B+

Courses

CID	C-Name	Room
1012	CS145 - Toon systems	Nvidia
1017	CS161 - Animation art	Gates 300
1019	CS245 - Painting town red	Packard 45

Queries ["compute" over tables]

- Minnie's GPA?
- AVG student GPA?
- Mickey's classes?
- AVG student GPA in CS145?

Key concept

Data model

Relational model (aka tables)

Simple and most popular

Elegant algebra (E.F. Codd et al)

Data model:

Organizing principle of data + operations

Every relation has a schema

Logical Schema: describes types, names

Physical Schema: describes data layout

Virtual Schema (Views): derived tables

Schema:

Describes blueprint of table (s)

Key concept

Data independence

Data independence

1. Can we add a new column or attribute without rewriting the application?

   <u>Logical Data Independence</u>
   Protection from changes in the Logical Structure
   of the data

2. Do you need to care which disks/machines are the data stored on?

   <u>Physical Data Independence</u>
   Protection from Physical Layout Changes

Python operating on Lists [reminder]

BASIC TYPES

Int, long int
string …

MAP + FILTER

map(function, list of inputs)

filter(function, list of inputs)

- Map applies function to input list
- Filter returns sub-list that satisfies a filter condition

REDUCE/AGGREGATE

reduce (...)

- Reduce runs a computation on a list and returns a result. E.g., SUM, MAX, MIN

For review, check out your favorite python tutorial (e.g, https://book.pythontips.com/en/latest/map_filter.html

SQL Queries on Tables (Lists of rows)

BASIC TYPES

Int32, int64
Char[n] …
Float32, float64

MAP + FILTER

Single table query

```
SELECT  c1, c2
FROM    T
WHERE   condition;
```

Multi table JOIN

```
SELECT  c1, c2
FROM    T1, T2
WHERE   condition;
```

REDUCE/AGGREGATE

```
SELECT  SUM(c1*c2)
FROM    T
WHERE   condition
GROUP BY  c3;
```

Map-Filter-Reduce pattern: Same simple/powerful idea in MapReduce, Hadoop, Spark, etc.

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

Relational Algebra

Operations in Relational Algebra

- C1- The usual set operations - union, intersection, and difference – applied to relations.
- C2 - Operations that remove parts of a relation: "selection" eliminates some rows (tuples), and "projection" eliminates some columns.
- C3 - Operations that combine the tuples of two relations, including "Cartesian product," which pairs the tuples of two relations in all possible ways, and various kinds of "join" operations, which selectively pair tuples from two relations.
- C4 - An operation called "renaming" that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

Single - table queries
Relational Algebra(C1)

Selection

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- Examples
  - $\sigma_{\text{Salary > 40000}}$ (Employee)
  - $\sigma_{\text{name = “Smith”}}$ (Employee)
- The condition c can be =, <, ≤, >, ≥, <>

SSN	Name	Salary
1234545	John	200000
5423341	Smith	600000
4352342	Fred	500000

$\sigma_{\text{Salary > 40000}}$ (Employee)

SSN	Name	Salary
5423341	Smith	600000
4352342	Fred	500000

Projection

- Eliminates columns, then removes duplicates
- Notation: $\Pi_{A1,\ldots,An}(R)$
- Example: project social-security number and names:
  - $\Pi_{SSN,\ Name}$ (Employee)
  - Output schema: Answer(SSN, Name)

SSN	Name	Salary
1234545	John	200000
5423341	John	600000
4352342	John	200000

$\Pi_{\text{Name,Salary}}$ (Employee)

Name	Salary
John	20000
John	60000

What you will learn about in this section

1. The SFW query

2. Other useful operators: LIKE, DISTINCT, ORDER BY

SQL Query

Basic form (there are many many more bells and whistles)

```
SELECT <attributes>

FROM   <one or more relations>

WHERE  <conditions>
```

Call this a **SFW** query.

Simple SQL Query: Selection

<u>**Selection**</u> is the operation of filtering a relation's tuples on some condition

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT *
FROM   Product
WHERE  Category = 'Gadgets'
```

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

SELECT Pname, Price, Manufacturer

FROM Product

WHERE Category = ‘Gadgets’

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

PName	Price	Manuf
Gizmo	$19.99	GWorks
Powergizmo	$29.99	GWorks

Notation

Input Schema

Product(PName, Price, Category, Manufacturer)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

Output Schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL **commands** are case insensitive:

  Same: SELECT, Select, select

  Same: Product, product

- **Values** are **not:**

  <u>Different:</u> 'Seattle', 'seattle'

- Use single quotes for constants:

  'abc' - yes

  "abc" - no

LIKE: Simple String Pattern Matching

```
SELECT *
FROM   Products
WHERE  PName LIKE '%gizmo%'
```

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
    - % = any sequence of characters
    - _ = any single character

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category
FROM   Product
```

Category
Gadgets
Photography
Household

Versus

```
SELECT Category
FROM   Product
```

Category
Gadgets
Gadgets
Photography
Household

ORDER BY: Sorting the Results

```
SELECT      PName, Price, Manufacturer
FROM        Product
WHERE       Category='gizmo' AND Price > 50
ORDER BY    Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

SQL Part II
JOIN, Aggregate Functions, Group By and HAVING

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

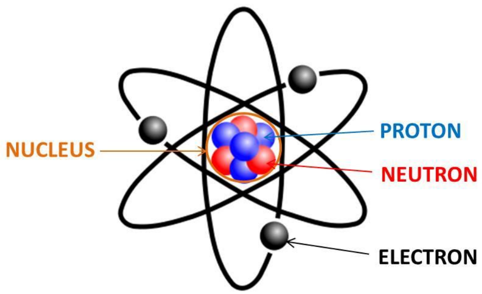
```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

Reminder on schemas



Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Students(<u>sid</u>*: string,* name: *string*, gpa: *float*)

Enrolled(<u>student_id</u>: *string,* <u>cid</u>: *string*, grade: *string*)

We'll use different Tables/tuples, for examples to build ideas

Data about local areas (for real-world examples)

SolarPanel(<u>region_name</u>*: string,* kw_total: *float*, carbon_offset_ton_metrics: *float, … *)

Census(<u>zipcode</u>: *string, population*: *int*, ...)

Pollution(<u>zipcode</u>: string, Particle_count: int…)

BikeShare(<u>zipcode</u>: string, trip_origin: float, trip_end: float, …)

...

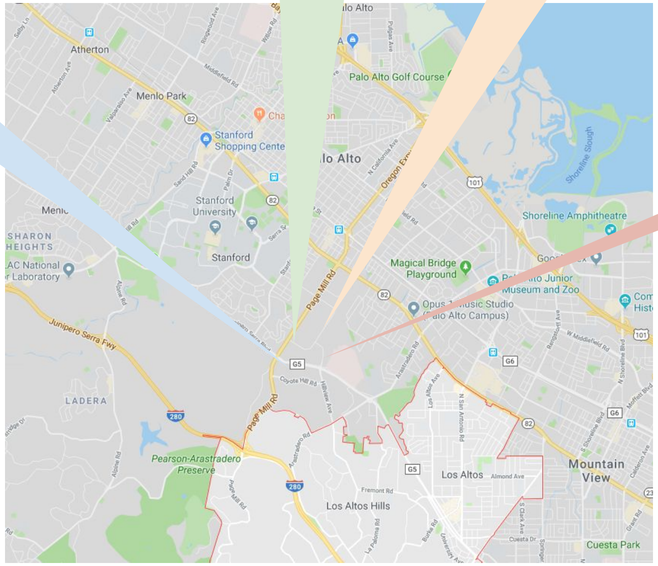
Option 1: ‘Good’ tables, with 10s-100s of columns

Census	
Zipcode	Population Census
94305	14301
94040	20301
94041	189

Bike share locations		
Zipcode	Lat	Lng
94305	35.1	122.12
94305	35.2	122.13
94041	35.1	121.27
94041		
94041		

SolarPanel			
Zipcode	KW-Total	Carbon offset	...
94305	14.4	29	
94040	32.1	42	
94041	29.1	37.38	

Pollution	
Zipcode	Particle count
94305	40
94040	22
94041	57



Option 2: ‘FrankenTable’ (with 1000s of columns)

Omnidata							
	SolarPanel			Census	Pollution	Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng
94305	14.4	29		14301	40	35.1	122.12
94305						35.2	122.13
94305						35.2	122.1
94305						35.1	122.12
94305						...	...

????

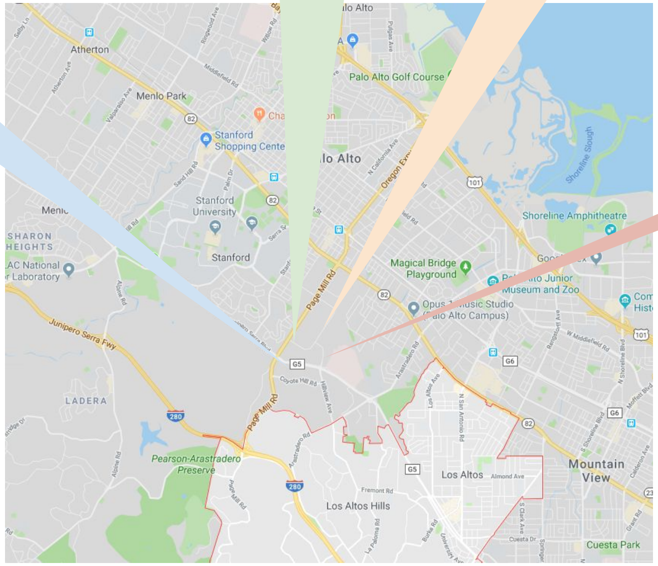
Option 1: A few "good" tables (with 10s of columns)

SolarPanel			
Zipcode	KW-Total	Carbon offset	...
94305	14.4	29	
94040	32.1	42	
94041	29.1	37.38	

Census	
Zipcode	Population Census
94305	14301
94040	20301
94041	189

Bike share locations		
Zipcode	Lat	Lng
94305	35.1	122.12
94305	35.2	122.13
94041	35.1	121.27
94041		
94041		

Pollution	
Zipcode	Particle count
94305	40
94040	22
94041	57



Trade offs?

- Reads? Writes?
- 100s - thousands of applications reading/writing data

⇒ Hybrids: 1 column ⟶ all columns

(Week 7: What's a good schema design?)

Option 2: 'FrankenTable' (with 1000s of columns)

Omnidata							
	SolarPanel			Census	Pollution	Bike share locations	
Zipcode	KW-Total	Carbon offset	...	Population	Particle count	Lat	Lng
94305	14.4	29		14301	40	35.1	122.12
94305	14.4	29		14301	40	35.2	122.13
94305	14.4	29		14301	40	35.2	122.1
94305	14.4	29		14301	40	35.1	122.12

Assume
(for now)

- Assume we have a set of “good” tables
  - How do we “connect” (join) those tables?

- Related important question
  - How to break up a “Franken” Table into “good” Tables? (i.e, design “good” schema)
  - Study in Week 5,6,7

Basic Example of Join

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Note: we will often omit attribute types in schema definitions for brevity, but assume attributes are always atomic types

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

PName	Price	Category	Manufacturer
Gizmo	$19	Gadgets	GizmoWorks
Powergizmo	$29	Gadgets	GizmoWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

A **join** between tables returns all unique combinations of their tuples **which meet some specified join condition**

CName	Stock Price	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Joins

Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Several equivalent ways to write a basic join in SQL:

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer =
CName
         AND Country='Japan'
        AND Price <= 200
```

```
SELECT PName, Price
FROM   Product
JOIN   Company
ON     Manufacturer =
Cname
WHERE  Price <= 200
       AND Country='Japan'
```

A few more later on

Joins

Product

PName	Price	Category	Manufacturer
Gizmo	$19	Gadgets	GizmoWorks
Powergizmo	$29	Gadgets	GizmoWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

CName	Stock Price	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
         AND Country='Japan'
        AND Price <= 200
```

PName	Price
SingleTouch	$149

Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)

Company(name, address)

1. SELECT DISTINCT name, address
2. FROM Person, Company
3. WHERE worksfor = name

Which "address" does this refer to?

Which name"s??

Tuple Variable Ambiguity in Multi-Table

Person(<u>name</u>, address, worksfor)

Company(<u>name</u>, address)

Both equivalent ways to resolve variable ambiguity

```
SELECT DISTINCT Person.name,
Person.address
FROM      Person, Company
WHERE         Person.worksfor =
Company.name
```

```
SELECT DISTINCT p.name, p.address
FROM      Person p, Company c
WHERE         p.worksfor = c.name
```

Semantics of JOINs

SELECT $x_1.a_1, x_1.a_2, \ldots, x_n.a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions($x_1, \ldots, x_n$)

Answer = {}
for x_1 **in** R_1 **do**
    for x_2 **in** R_2 **do**
        …..
        for x_n **in** R_n **do**
            if Conditions($x_1, \ldots, x_n$)
                then Answer = Answer $\cup$ {($x_1.a_1, x_1.a_2, \ldots, x_n.a_k$)}
return Answer

Note:
This is a *multiset* union

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

- Take **cross product**

  V = R x S

- Apply **selections/conditions**

  Y = {(r, s) in V | r.A == s.B}

- Apply **projections** to get final output

  Z = (y.A) for y in Y

Recall: Cross product (R X S) is the set of all unique tuples in R,S

Ex: {a,b,c} X {1,2}

= {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

= Filtering!

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on…)

An example of SQL semantics

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

R

A
1
3

S

B	C
2	3
3	4
3	5

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

Apply Selections / Conditions

A	B	C
3	3	4
3	3	5

Apply Projection

Output

A
3
3

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not actually how the DBMS executes it under the covers

A Subtlety about Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Find all countries that manufacture some product in the 'Gadgets' category.

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=CName AND Category='Gadgets'
```

A Subtlety about Joins

Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

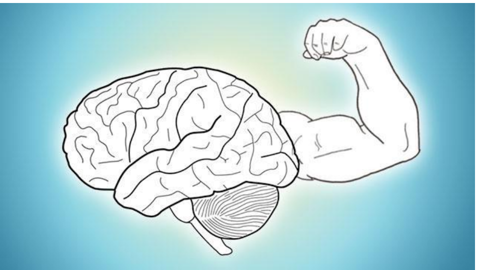
Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
  AND Category='Gadgets'
```

Country
? USA
? USA

Ans? DISTINCT



What is the Problem? What is the Solution?

JOIN TYPES and NULL OPERATION

Inner, Full, Left and Right Outer Join

NULLS in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs?

Null Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL

- *For boolean operations,* in SQL there are three values:

FALSE = 0
UNKNOWN = 0.5
TRUE = 1

  - If x= NULL then x == “Joe” is UNKNOWN (Is x equal to ‘Joe’?)

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
  AND (height > 6 AND weight > 190)
```

Won't return e.g.
(age=20
height=NULL
weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

Null Values

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
```

Some Persons are not included !

Null Values

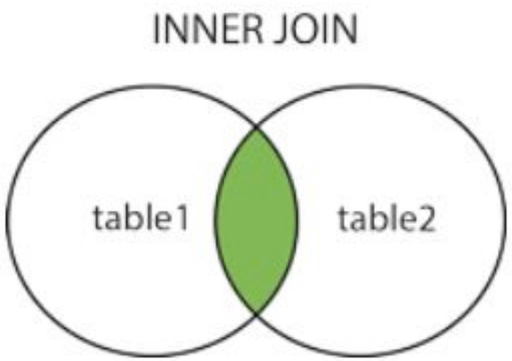
Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
   OR age IS NULL
```

Now it includes all Persons!

RECAP: Joins



By default, joins in SQL are **“inner joins”**:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product
 JOIN Purchase ON Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

[Like Below]

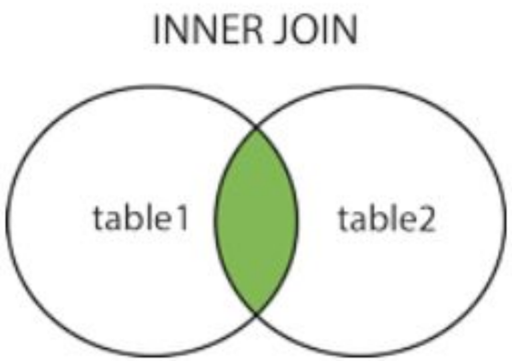
```
SELECT Product.name, Purchase.store
FROM   Product
 INNER JOIN Purchase
       ON Product.name = Purchase.prodName
```

Inner Joins + NULLS = Lost data?

By default, joins in SQL are **“inner joins”**:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```



Product	
Name	**Category**
Iphone	Media
Roomba	Cleaner
Ford Pinto	Car
Tesla	Car

Purchase	
ProdName	**Store**
Iphone	Apple Store
Tesla	Tesla Store

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
  - A LEFT OUTER JOIN will return a tuple (a, NULL)!

- Left outer joins in SQL:

```sql
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase ON
       Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

LEFT OUTER JOIN

Product

name	category
iphone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	car
iPhone	Apple store

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase
        ON Product.name = Purchase.prodName
```

name	store
iPhone	Apple store
iPhone	Apple store
Tesla	car
Ford Pinto	NULL

Other Outer Joins

- Left outer join:
  - Include the left tuple even if there’s no match
- Right outer join:
  - Include the right tuple even if there’s no match
- Full outer join:
  - Include the both left and right tuples even if there’s no match

Natural Join

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
9	10	11

(b) Relation S

A	$R.B$	$S.B$	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

(c) Result $R \times S$

A	B	C	D
1	2	5	6
3	4	7	8

Difference between INNER and NATURAL JOIN

SR.NO.	NATURAL JOIN	INNER JOIN
1.	Natural Join joins two tables based on same attribute name and datatypes.	Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause.
2.	In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column	In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also
3.	In Natural Join, If there is no condition specifies then it returns the rows based on the common column	In Inner Join, only those records will return which exists in both the tables
4.	SYNTAX: SELECT * FROM table1 NATURAL JOIN table2;	SYNTAX: SELECT * FROM table1 INNER JOIN table2 ON table1.Column_Name = table2.Column_Name;

Geekforgeeks

Theta Join

A	B	C
1	2	3
6	7	8
9	7	8

(a) Relation U

B	C	D
2	3	4
2	3	5
7	8	10

(b) Relation V

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

(c) Result $U \bowtie V$

$U \bowtie_{A<D} V$

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

$U \bowtie_{A<D \text{ AND } U.B \neq V.B} V$

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	7	8	10

Self Join

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

w3schools

CustomerName1	CustomerName2	City
Cactus Comidas para llevar	Océano Atlántico Ltda.	Buenos Aires
Cactus Comidas para llevar	Rancho grande	Buenos Aires
Océano Atlántico Ltda.	Cactus Comidas para llevar	Buenos Aires
Océano Atlántico Ltda.	Rancho grande	Buenos Aires
Rancho grande	Cactus Comidas para llevar	Buenos Aires

EquiJoin and Natural Join

- Equi Join is a join using one common column (referred to in the "on" clause)
- Natural Join is an implicit join clause based on the common columns in the two tables being joined. Common columns are columns that have the same name in both tables.

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

SELECT c1, c2 FROM t;
Query data in columns c1, c2 from a table

SELECT * FROM t;
Query all rows and columns from a table

SELECT c1, c2 FROM t
WHERE condition;
Query data and filter rows with a condition

SELECT DISTINCT c1 FROM t
WHERE condition;
Query distinct rows from a table

SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
Sort the result set in ascending or descending order

SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
Skip *offset* of rows and return the next n rows

SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
Group rows using an aggregate function

SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
Inner join t1 and t2

SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
Left join t1 and t1

SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
Right join t1 and t2

SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
Perform full outer join

SELECT c1, c2
FROM t1
CROSS JOIN t2;
Produce a Cartesian product of rows in tables

SELECT c1, c2
FROM t1, t2;
Another way to perform cross join

SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
Combine rows from two queries

SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
Return the intersection of two queries

SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
Subtract a result set from another result set

SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
Query rows using pattern matching %, _

SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
Query rows in a list

SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
Query rows between two values

SELECT c1, c2 FROM t
WHERE c1 IS [NOT] NULL;
Check if values in a table is NULL or not

Aggregation

```
SELECT AVG(price)
FROM   Product
WHERE  maker = “Toyota”
```

```
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

We probably want:

```
SELECT COUNT(DISTINCT category)
FROM   Product
WHERE  year > 1995
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

Example 1

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

⇨ 50
(= 1*20 + 1.50*20)

Example 2

```
SELECT SUM(price * quantity)
FROM   Purchase
```

⇨ 65
(= 1*20 + 1.50*20 + 0.5*10 + 1*10)

Grouping and Aggregation

What GROUPings are possible?

- Type, Size, Color
- Number of holes
- mbination?

What GROUPings are possible?

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

Possible Groups

- Product? (e.g. SUM(quantity) by product) # product units sold
- Date? (e.g., SUM(price*quantity) by date) # daily sales
- Price?
- Product, Date?

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
         SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > ‘10/1/2005’
GROUP BY product
```

Find total sales after 10/1/2005 per product.

Let’s see what this means…

Grouping and Aggregation

```
SELECT   product,
         SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

<u>Semantics of the query:</u>

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > ‘10/1/2005’
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples…***

General form of Grouping and Aggregation

SELECT	S
FROM	$R_1,\ldots,R_n$
WHERE	C_1
GROUP BY	$a_1,\ldots,a_k$
HAVING	C_2

Why?

- S = Can ONLY contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- C_1 = is any condition on the attributes in $R_1,\ldots,R_n$
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

```
SELECT     S
FROM       R_1,…,R_n
WHERE      C_1
GROUP BY   a_1,…,a_k
HAVING     C_2
```

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may need to compute aggregates)
4. Compute aggregates in S and return the result

Example SQL

Part I

Example 1

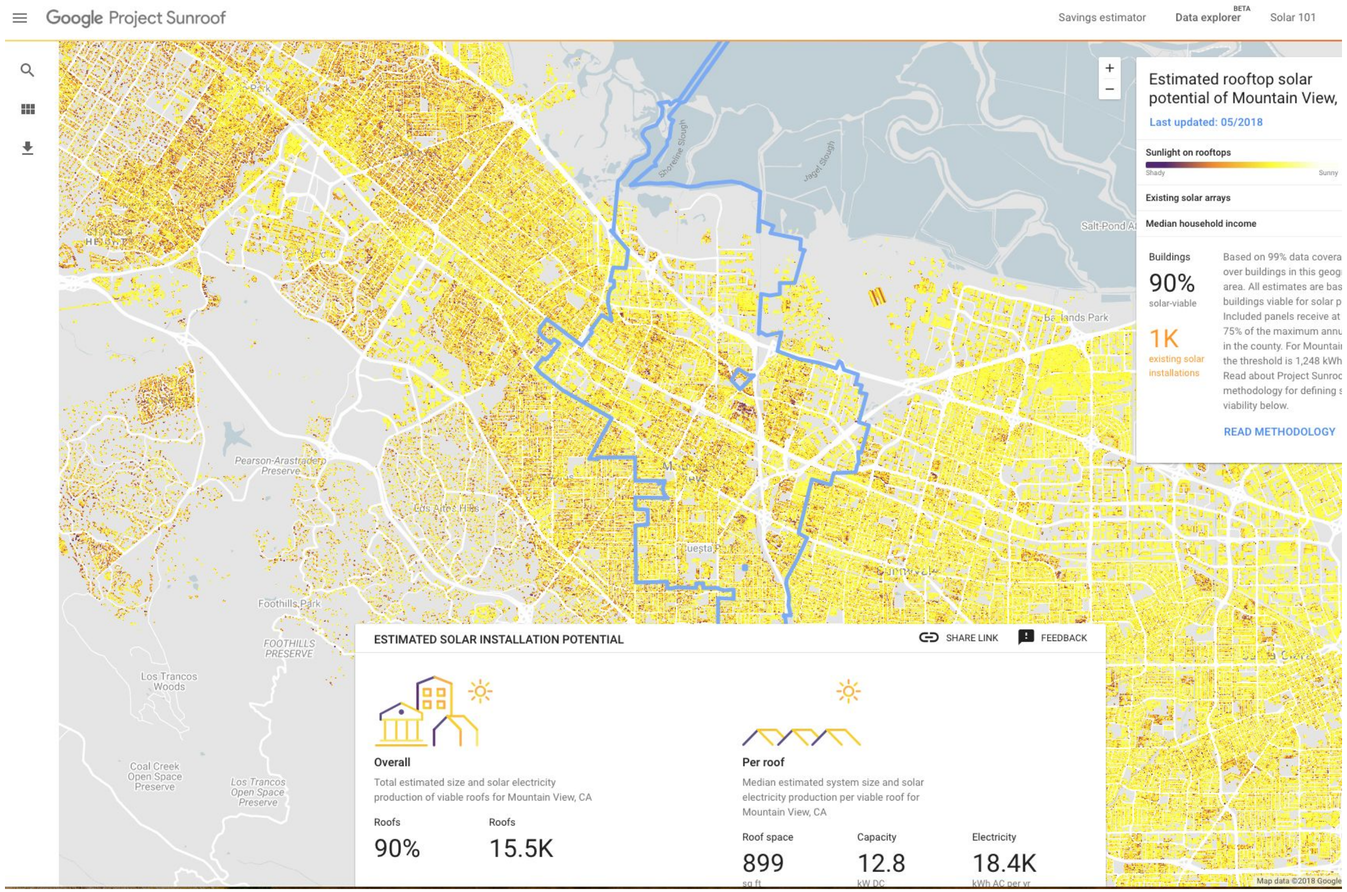
Sun Roof potential

from Satellite images

Example 1

SunRoof potential

SunRoof explorer



Example 1

Public dataset

Public Dataset: Solar potential by postal code

region_name	percent_covered	kw_total	carbon_offset_metric_tons
94043	97.79146031321109	215612.5	84929.0098507134 7
94041	99.05200433369447	56704.25	22189.34823862318

Public Dataset: USA.population by zip2010

zipcode	population
99776	124
38305	49808
37086	31513
41667	720
67001	1676

Example 1

SunRoof

On BigQuery Public dataset

What is the solar potential of Mountain View, CA? [ ]

Saved Query: MTV sunroof [edited] ?

```sql
#StandardSQL
SELECT
  region_name,
  percent_covered,
  kw_total,
  carbon_offset_metric_tons
FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code`
WHERE
  region_name = '94040'
  OR region_name = '94041'
  OR region_name = '94043'
```

Standard SQL Dialect ✕

Ctrl + Enter: ru

RUN QUERY | Save Query | Save View | Format Query | Schedule Query | Show Options

Query complete (1.6s elapsed, 346 KB processed)

Results | Details | Download as CSV | Download as JSON | Save

Row	region_name	percent_covered	kw_total	carbon_offset_metric_tons
1	94043	97.79146031321109	215612.5	84929.00985071347
2	94041	99.05200433369447	56704.25	22189.34823862318
3	94040	98.9440337909187	139745.5	55039.74974407879

Example 2

SunRoof

Public dataset On BigQuery

How many metric tons of carbon would we offset, if building in communities with 100% coverage all had solar roofs? [ ]

Saved Query: CO2 offset in 100percent zips

```
#StandardSQL
SELECT
  ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metric_tons
FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
ON s.region_name = c.zipcode
WHERE
  percent_covered = 100.0
  AND c.population > 0
```

Standard SQL Dialect ✕

Ctrl + Enter: run qu

RUN QUERY | Save Query | Save View | Format Query | Schedule Query | Show Options | Query com

Results | Details | Download as CSV | Download as JSON | Save as

Row	total_carbon_offset_possible_metric_tons
1	3689508.33

Example 2

SunRoof

Public dataset
On BigQuery

How many metric tons of carbon would we offset, per zipcode?

Saved Query: CO2 offset in 100percent zips [edited] ?

```
#StandardSQL
SELECT
  zipcode, ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metri
FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
ON s.region_name = c.zipcode
WHERE
  percent_covered = 100.0
  AND c.population > 0
GROUP BY c.zipcode
```

Standard SQL Dialect ✕

Ctrl + Enter: run c

RUN QUERY | Save Query | Save View | Format Query | Schedule Query | Show Options

Query complete (2.5s elapsed, 23.5 MB processed)

Results | Details | Download as CSV | Download as JSON | Save a

Row	zipcode	total_carbon_offset_possible_metric_tons
1	35119	3417.26
2	10165	162.1
3	21810	6650.09
4	74078	61515.66
5	47876	5544.3
6	10170	831.22

Example 2

SunRoof

How many metric tons of carbon would we offset, per zipcode

sol

Saved Query: CO2 offset in 100percent zips [edited] ?

Query Editor | UDF

```
#StandardSQL
SELECT
  zipcode, ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metric_tons
FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
ON s.region_name = c.zipcode
WHERE
  percent_covered = 100.0
  AND c.population > 0
GROUP BY c.zipcode
ORDER BY total_carbon_offset_possible_metric_tons
DESC
```

Standard SQL Dialect ✕

Ctrl + Enter: run query, Tab or Ctrl + Space

RUN QUERY | Save Query | Save View | Format Query | Schedule Query | Show Options

Query complete (2.8s elapsed, 23.5 MB processed)

Results | Details | Download as CSV | Download as JSON | Save as Table | Save to

Row	zipcode	total_carbon_offset_possible_metric_tons
1	18503	715700.55
2	44243	271861.55
3	38677	266787.12
4	96860	225850.35
5	47809	141087.91

Reminder

Special Databases

Query with SQL, universally over ‘all’ DBs

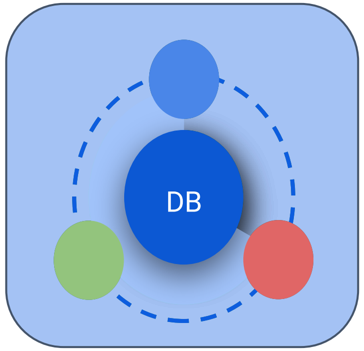
SQL

```
= QUERY(T,
   "SELECT c1, c2
    FROM    T
    WHERE condition;)
```

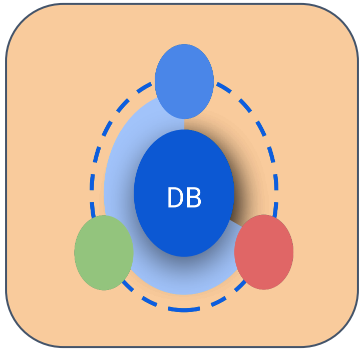
```
SELECT c1, c2
FROM    T
WHERE condition;
```

```
results =
 spark.SQL(
  "SELECT c1, c2
   FROM    T
   WHERE condition;")
```

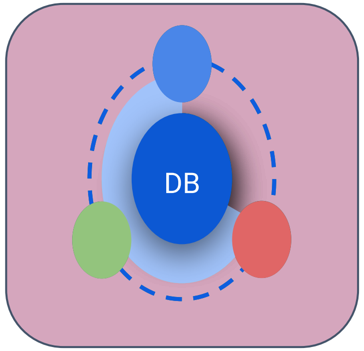
DB



‘Spreadsheets’



GCP BigQuery, AWS Redshift, MySQL, PostgresSQL, Oracle



Spark, Hadoop

Data

Data

100s of Scaling algorithms/systems? [Weeks 3..]

- Data layout? [Row vs columns…]
- Data structs? [Indexing…]