COMP 430
Intro. to Database Systems

SQL 2

Aggregation & Grouping

One form of aggregation – Column totals

```
SELECT Sum(price)
FROM   Candy;
```

Candy

candy_name	price
Reese's Cup	0.50
5th Avenue	1.20
Almond Joy	0.75
Jelly Babies	2.39

4.84

Other aggregations

- Count
- Avg
- Max
- Min
- …

What else depends on SQL version.

Aggregation, DISTINCT, & NULLs

```
SELECT Count(price), Count(DISTINCT price)
FROM   Candy;
```

Candy

candy_name	price
Reese's Cup	0.50
5th Avenue	NULL
Almond Joy	0.75
Jelly Babies	2.39
Chocolate Orange	2.39
Jelly Nougats	NULL

→

4	3

No price since no longer made in original form.

Counting rows

```
SELECT Count(*)
FROM   Candy;
```

The only aggregate function that uses this *.

Candy

candy_name	price
Reese's Cup	0.50
5th Avenue	NULL
Almond Joy	0.75
Jelly Babies	2.39
Chocolate Orange	2.39
Jelly Nougats	NULL

6

Aggregation + other previous features

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

Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.50	10
banana	10/10	1	10
bagel	10/25	1.50	20

→

50

Grouping + Aggregation – Column subtotals

```
SELECT product, Sum(price * quantity) AS subtotal
FROM   Purchase
GROUP BY product;
```

Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.50	10
banana	10/10	1	10
bagel	10/25	1.50	20

→

product	date	price	quantity
bagel	10/21	1	20
	10/25	1.50	20
banana	10/03	0.50	10
	10/10	1	10

→

product	subtotal
bagel	50
banana	15

Grouping + Aggregation

```
SELECT product,
           Count(*) AS purchases,
           Sum(quantity) AS items,
           Sum(price * quantity) AS subtotal
FROM   Purchase
GROUP BY product;
```

Can only SELECT fields that you GROUP BY.

Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.50	10
banana	10/10	1	10
bagel	10/25	1.50	20

product	purchases	items	subtotal
bagel	2	40	50
banana	2	20	15

Conditions on aggregates

```
SELECT  product, Sum(price * quantity) AS subtotal
FROM    Purchase
WHERE  date > '10/05'
GROUP BY product
HAVING  SUM(quantity) >= 10;
```

Condition on individual rows.

Condition on aggregates.

Purchase

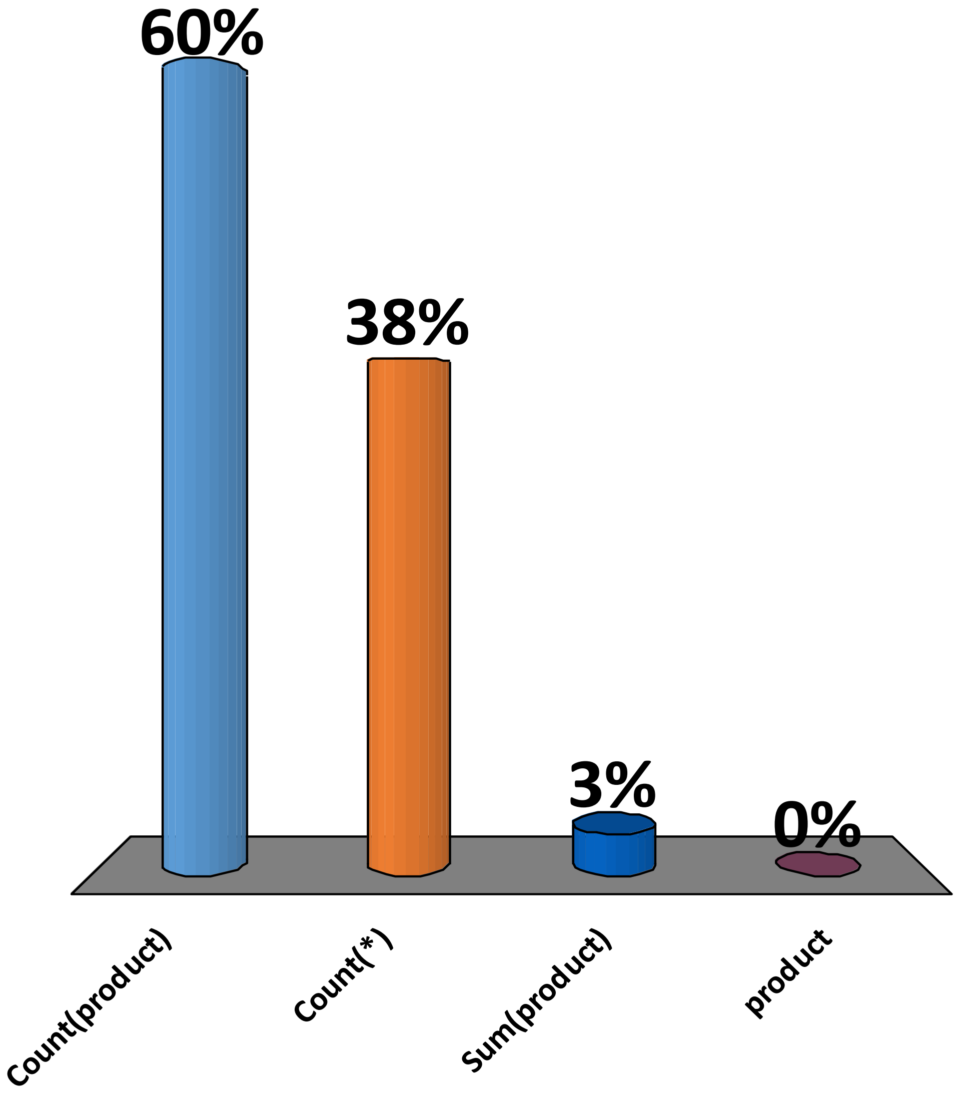
product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.50	10
banana	10/10	1	10
bagel	10/25	1.50	20
apple	10/10	1	5

product	date	price	quantity
bagel	10/21	1	20
	10/25	1.50	20
banana	10/10	1	10
apple	10/10	1	5

product	subtotal
bagel	50
banana	10

Query: Products with at least two purchases.

A. Count(product)
B. Count(*)
C. Sum(product)
D. product



Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.50	10
banana	10/10	1	10
bagel	10/25	1.50	20
apple	10/10	1	5

```
SELECT  product
FROM    Purchase
GROUP BY product
HAVING  ??? >= 2 ;
```

Sorting on aggregates

SELECT product, Sum(price * quantity)
FROM   Purchase
GROUP BY product
~~ORDER BY Sum(price * quantity);~~

```
SELECT product, Sum(price * quantity) AS subtotal
FROM   Purchase
GROUP BY product
ORDER BY  subtotal;
```

Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.50	10
banana	10/10	1	10
bagel	10/25	1.50	20

product	date	price	quantity
bagel	10/21	1	20
	10/25	1.50	20
banana	10/03	0.50	10
	10/10	1	10

product	subtotal
banana	15
bagel	50

Syntax summary

```
SELECT [DISTINCT]  Computations
FROM        Table_0
INNER JOIN Table_1 ON JoinCond_1
…
WHERE       FilterCond
GROUP BY  attributes
HAVING      AggregateFilterCond
ORDER BY  attributes
LIMIT        n;
```

Uses attributes a_1, …, a_k or aggregates.

Can also use computation results.

Semantics summary

SELECT [DISTINCT] Computations
FROM Table$_0$
INNER JOIN Table$_1$ ON JoinCond$_1$
…
WHERE FilterCond
GROUP BY attributes
HAVING AggregateFilterCond
ORDER BY attributes
LIMIT n;

5. Computations, [eliminate duplicates]
8. Projections
1. Joins

2. Filter rows
3. Group by attributes
4. Filter groups
6. Sort
7. Use first n rows

What we're they thinking?!?

(Potentially confusing) details

Response Counter

DISTINCT Sum() vs. Sum(DISTINCT)

Data

item	tag	value
1	a	2
2	a	2
3	b	1
4	b	4
5	c	4

```
SELECT DISTINCT Sum(DISTINCT value)
FROM Data
GROUP BY tag;
```

A.

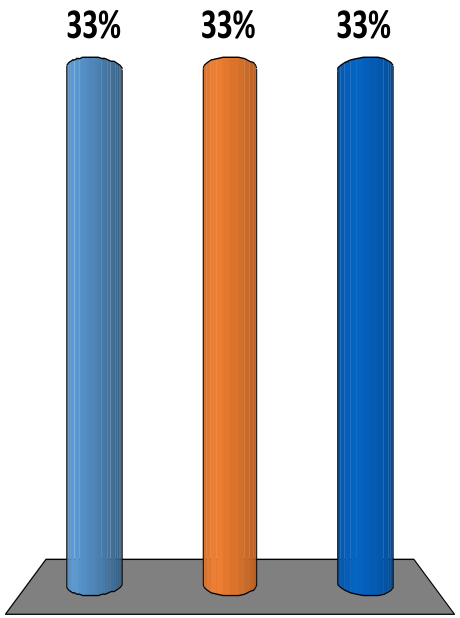
SUM
5
4

B.

SUM
2
5
4

C.

SUM
2
1
4



Response Counter

HAVING without grouping

Data

item	tag	value
1	a	2
2	a	2
3	b	1
4	b	4
5	c	4

```
SELECT value, COUNT(*) AS count
FROM Data
HAVING count > 1;
```

Without GROUP BY, the entire results form one group.

A.

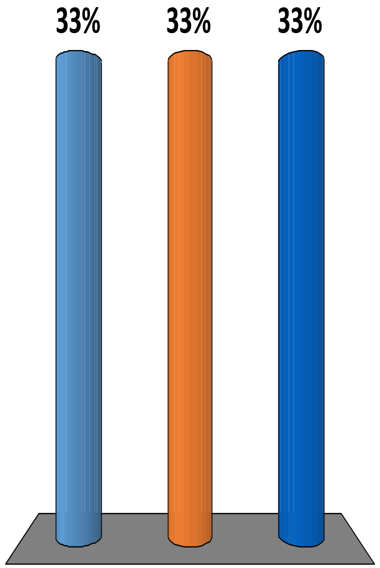
value	count
2	2
1	1
4	2

B.

value	count
2	5
1	5
4	5

✓ C.

value	count
2	5
2	5
1	5
4	5
4	5



Grouping without aggregation

```
SELECT product
FROM   Purchase
GROUP BY  product;
```

Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

product
bagel
banana

GROUP BY vs. DISTINCT

```
SELECT product
FROM   Purchase
GROUP BY  product;
```

=

```
SELECT DISTINCT  product
FROM   Purchase;
```

But only GROUP BY works well with aggregation.

Purchase

product	date	price	quantity
bagel	10/21	1	20
banana	10/03	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

→

product
bagel
banana

More complicated examples

Activity – will break activity into segments

03a-aggregation.ipynb

- Students (id and name) taking more than 5 courses

- Courses and their average ratings
- Highest average rating of a course
- Course with highest average rating

Students with >5 courses – Two solutions

```
SELECT s.s_id, s.first_name, s.last_name
FROM Student s
INNER JOIN Enrollment e ON s.s_id = e.s_id
GROUP BY Student.s_id
HAVING COUNT(crn) > 5;
```

```
SELECT s_id, first_name, last_name
FROM   Student
WHERE (SELECT Count(*)
           FROM  Enrollment
           WHERE Student.s_id = Enrollment.s_id
          ) > 5;
```

Execution & efficiency comparison

```
SELECT s.s_id, s.first_name, s.last_name
FROM Student s
INNER JOIN Enrollment e ON s.s_id = e.s_id
GROUP BY s.s_id
HAVING COUNT(crn) > 5;
```

Assuming no major optimizations:

1. How many times do we SELECT … FROM Enrollment?
2. How many joins?

```
SELECT s_id, first_name, last_name
FROM   Student s
WHERE (SELECT Count(*)
           FROM  Enrollment e
           WHERE s.s_id = e.s_id
          ) > 5;
```

Courses and their average ratings

Enrollment (s_id, crn, grade, rating)

```
SELECT  crn, Avg(rating)
FROM   Enrollment
GROUP BY  crn;
```

Course with highest average rating – Attempts

~~SELECT crn, Max(Avg(rating))~~
~~FROM Enrollment~~
~~GROUP BY crn;~~

Nesting aggregate functions is not syntactically allowed.

```
SELECT  Max(avg_rating)
FROM    (SELECT  Avg(rating) as avg_rating
         FROM   Enrollment
         GROUP BY  crn
        );
```

Gets the rating, but not the course.

Course with highest avg rating – Solution 1

```
SELECT  crn
FROM   (SELECT  crn, Avg(rating) AS avg_rating1
        FROM   Enrollment
        GROUP BY  crn
       )
WHERE  avg_rating1 = (SELECT  Max(avg_rating2)
                      FROM    (SELECT  Avg(rating) as avg_rating2
                               FROM   Enrollment
                               GROUP BY  crn
                              )
                     );
```

1. Calculate all average ratings.
2. Calculate the maximum.
3. Find which courses have this value as their average.

What about repeated subquery?

```
SELECT  crn
FROM   (SELECT  crn, Avg(rating) AS avg_rating1
        FROM   Enrollment
        GROUP BY  crn
       )
WHERE  avg_rating1 = (SELECT  Max(avg_rating2)
                      FROM    (SELECT  Avg(rating) AS avg_rating2
                               FROM   Enrollment
                               GROUP BY  crn
                              )
                     );
```

```
CREATE VIEW   Average  AS
SELECT  crn, Avg(rating) AS avg_rating
FROM   Enrollment
GROUP BY  crn;

SELECT  crn
FROM   Average
WHERE  avg_rating = (SELECT  Max(avg_rating)
                     FROM    Average
                    );
```

Course with highest avg rating – Solution 2

```
CREATE VIEW   Average  AS
SELECT  crn, Avg(rating) AS avg_rating
FROM   Enrollment
GROUP BY  crn;

SELECT  crn
FROM   Average
ORDER BY  avg_rating DESC
LIMIT 1;
```

Doesn’t work for ties.

Execution & efficiency comparison

```
CREATE VIEW   Averages  AS
SELECT  crn, Avg(rating) AS avg_rating
FROM   Enrollment
GROUP BY  crn;
```

```
SELECT  crn
FROM   Averages
WHERE  avg_rating = (SELECT  Max(avg_rating)
                     FROM    Averages
                    );
```

Assuming no major optimizations:

Cost estimate of each?

```
SELECT  crn
FROM   Averages
ORDER BY  avg_rating
LIMIT 1;
```

Three Views of VIEW

```
CREATE VIEW   Averages  AS
SELECT  crn, Avg(rating) AS avg_rating
FROM   Enrollment
GROUP BY  crn;
```

- Convenience: Macro; shorthand for repeated query
- Efficiency: Hint to cache query results (“materialized views”)
- Semantics: Just another relation, but one having values dependent on other tables

What's the difference?

```
CREATE VIEW StudentName AS
SELECT first_name, last_name
FROM Student;
```

Dynamic: StudentName changes whenever Student changes.

```
SELECT first_name, last_name
INTO StudentName
FROM Student;
```

Static: StudentName has data from one point in time.

Add, modify, delete views

```
CREATE VIEW … AS
SELECT …;
```

```
CREATE OR REPLACE VIEW … AS
SELECT …;
```

```
DROP VIEW …;
```

Set & multiset operations

Two ways to think about multisets

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)

=

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

Multiset union

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cup$

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

$$\lambda(\boldsymbol{Z}) = \lambda(\boldsymbol{X}) + \lambda(\boldsymbol{Y})$$

Multiset intersection

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cap$

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Tuple	$\lambda(Z)$
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0

$$\boldsymbol{\lambda}(\boldsymbol{Z}) = min\big(\boldsymbol{\lambda}(\boldsymbol{X}), \boldsymbol{\lambda}(\boldsymbol{Y})\big)$$

Multiset difference

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$-$

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Tuple	$\lambda(Z)$
(1, a)	0
(1, b)	0
(2, c)	1
(1, d)	0

$$\lambda(\boldsymbol{Z}) = \lambda(\boldsymbol{X}) - \lambda(\boldsymbol{Y})$$

Set vs. multiset operations – SQL syntax

Sets:

```
SELECT  a
FROM    R
UNION
SELECT  a
FROM    S;
```

```
…
INTERSECTION
…
```

```
…
EXCEPT
…
```

Multisets:

```
SELECT  a
FROM    R
UNION ALL
SELECT  a
FROM    S;
```

```
…
INTERSECTION ALL
…
```

```
…
EXCEPT ALL
…
```

Activity – Sets & multisets

03b-sets-multisets.ipynb

Exercise

Product

p_name	manufacturer	factory_loc
Gizmo	GizmoWorks	U.S.
WhatNot	What	China

Company

c_name	hq_loc
GizmoWorks	U.S.
What	U.S.

Goal: Find HQs of companies that manufacture in both U.S. and China.

Proposed solution:

```
SELECT hq_loc
FROM Product
INNER JOIN Company ON manufacturer = c_name
WHERE factory_loc = ‘U.S.’
INTERSECT
SELECT hq_loc
FROM Product
INNER JOIN Company ON manufacturer = c_name
WHERE factory_loc = ‘China’;
```

What does this query result in?

One solution: set membership + subqueries

Product

p_name	manufacturer	factory_loc
Gizmo	GizmoWorks	U.S.
WhatNot	What	China

Company

c_name	hq_loc
GizmoWorks	U.S.
What	U.S.

Goal: Find HQs of companies that manufacture in both U.S. and China.

```
SELECT DISTINCT hq_loc
FROM Product
INNER JOIN Company ON manufacturer = c_name
WHERE c_name IN (
                    SELECT manufacturer
                    FROM   Product
                    WHERE  factory_loc = 'U.S.')
          AND
          c_name IN (
                    SELECT manufacturer
                    FROM   Product
                    WHERE  factory_loc = 'China');
```

Set membership on multiple fields

Product

name	version	other_info
Gizmo	1	…
Gizmo	2	…
Widget	1	…
Widget	2	…
Widget	3	…

Goal: For each product name, we want to find the latest/largest version and its associated information.

```
SELECT *
FROM Product
WHERE (name, version) IN
          (SELECT name, MAX(version)
           FROM Product
           GROUP BY name);
```

Not allowed in all SQL versions.

Quantification

Review of SQL conditions

- Compare elements of same record
  - Combine with join to work with multiple records
- Compare aggregations
- Check membership – IN

Existential quantification

$\{c \mid Customer(c) \wedge$
$(\exists\ o.\ Order(o) \wedge$
$c.id = o.customer\_id)\}$

```
SELECT *
FROM Customer
WHERE EXISTS (SELECT *
              FROM Order
              WHERE Customer.id = customer_id);
```

EXISTS (subquery) returns a Boolean.

Subquery's SELECT irrelevant.

Customer

id	name
1	Joe
4	Mary
3	Scott
6	Elizabeth

Order

id	customer_id	Item
105	4	Shoes
107	4	Pants
108	1	Pants
109	3	Tie

id	name
1	Joe
4	Mary
3	Scott

Existential quantification negated

$\{c \mid Customer(c) \wedge$
$(\neg \exists\, o.\ Order(o) \wedge$
$c.id = o.customer\_id)\}$

```
SELECT *
FROM Customer
WHERE NOT EXISTS (SELECT *
                  FROM Order
                  WHERE Customer.id = customer_id);
```

Customer

id	name
1	Joe
4	Mary
3	Scott
6	Elizabeth

Order

id	customer_id	Item
105	4	Shoes
107	4	Pants
108	1	Pants
109	3	Tie

id	name
6	Elizabeth

Existential quantification – efficiency

{c | Customer(c) $\wedge$
(∃ o. Order(o) $\wedge$
c.id = o.customer_id)}

```
SELECT *
FROM Customer
WHERE EXISTS (SELECT *
              FROM Order
              WHERE Customer.id = customer_id);
```

Executes subquery for each Customer record.

However, can stop subquery at first satisfying Order.

Existential quantification negated

{c | Customer(c) $\wedge$
(¬ ∃ o. Order(o) $\wedge$
c.id = o.customer_id)}

```
SELECT *
FROM Customer
WHERE NOT EXISTS (SELECT *
                  FROM Order
                  WHERE Customer.id = customer_id);
```

找到相等的就停止

Executes subquery for each Customer record.

However, can stop subquery at first **non**-satisfying Order.

Logical equivalence of quantifiers

$$\neg \exists x . R(x) = \forall x. \neg R(x)$$

$$\neg \forall x . R(x) = \exists x. \neg R(x)$$

并不是所有的

Universal quantification

- No directly comparable FORALL!
- Generally less useful idea in SQL.
  - E.g., find Customer that has placed all of the Orders.
  - However, see today's activity.

Don't confuse with slightly different ALL. (Coming soon.)

- $\forall x \,.\, R(x) \;=\; \neg\, \exists x.\, \neg\, R(x)$

EXISTS vs. other approaches

```
SELECT *
FROM Customer
WHERE EXISTS (SELECT *
                    FROM Order
                    WHERE Customer.id = customer_id);
```

```
SELECT *
FROM Customer
WHERE id IN (SELECT customer_id
                   FROM Order);
```

```
SELECT *
FROM Customer
WHERE (SELECT Count(*)
          FROM   Order
          WHERE Customer.id = customer_id);
```

```
SELECT DISTINCT Customer.id, name
FROM Customer
INNER JOIN Order ON Customer.id = customer_id;
```

EXISTS vs. INTERSECT

```
SELECT id
FROM Customer
WHERE EXISTS (SELECT *
              FROM Order
              WHERE Customer.id = customer_id);
```

```
SELECT id
FROM Customer
INTERSECT
SELECT customer_id
FROM Order;
```

Inconvenient if you want more attributes.

Activity – Use EXISTS in queries

Given:

Store (id, name, type)

City (id, name)

CityStore (store_id, city_id)

Warm up:

- The stores that are in at least one city each are of what types?
- The stores that are in no city each are of what types?
- The stores that are present in all cities are of what types?

Another form of existential quantification

```
SELECT *
FROM Customer
WHERE EXISTS (SELECT *
              FROM Order
              WHERE Customer.id = customer_id);
```

```
SELECT *
FROM Customer
WHERE id = ANY (SELECT customer_id
                FROM Order);
```

Syntactically strange, but think of ANY as part of the comparison operator.

Can use any comparison operator.

Or SOME.

Another form of universal quantification

```
SELECT *
FROM Product p1
WHERE p1.price >= ALL (SELECT p2.price
                       FROM  Product p2);
```

```
SELECT *
FROM Product p1
WHERE NOT (p1.price < ANY (SELECT p2.price
                           FROM  Product p2));
```

Joins

Review – inner joins

Product

p_name	price	manufacturer
Gizmo	19.99	GizmoWorks
Powergizmo	39.99	GizmoWorks
Widget	19.99	WidgetsRUs
HyperWidget	203.99	Hyper

Company

c_name	address	city	state
GizmoWorks	123 Gizmo St.	Houston	TX
WidgetsRUs	20 Main St.	New York	NY
Hyper	1 Mission Dr.	San Francisco	CA

By far, the most common kind of join.

p_name	price	manufacturer	c_name	address	city	state
Gizmo	19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Widget	19.99	WidgetsRUs	WidgetsRUs	20 Main St.	New York	NY
HyperWidget	203.99	Hyper	Hyper	1 Mission Dr.	San Francisco	CA

Inner join – a special case

Product (p_name, price, c_name)

Company (c_name, address, city, state)

Both tables use the same attribute name.

```
SELECT  *
FROM    Product
INNER JOIN Company ON Product.c_name = Company.c_name;
```

Join condition is simple equality. *Equi-join*

Result (p_name, price, Product.c_name, Company.c_name, address, city, state)

```
SELECT  *
FROM    Product
INNER JOIN Company USING (c_name);
```

Can list multiple attributes.

Result (p_name, price, c_name, address, city, state)

Inner join – another special case

Product (<u>p_name</u>, price, c_name)
Company (<u>c_name</u>, address, city, state)

```
SELECT  *
FROM    Product
INNER JOIN Company USING (c_name);
```

Result (p_name, price, c_name, address, city, stat

Considered dangerous, since it constraint is implicit, and attribute name sets can change.

<u>Don't use.</u>

```
SELECT  *
FROM    Product
NATURAL INNER JOIN Company;
```

Assumes equality of same-named attributes.
Equi-join

Result (p_name, price, c_name, address, city, state)

What about *dangling tuples*?

Company

c_name	state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

p_name	manufacturer
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

c_name	state	p_name	manufacturer
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper

NewCo is forgotten in join, since it has no products.

```
SELECT  *
FROM    Company
INNER JOIN Product ON manufacturer = c_name;
```

Left outer join

Company

c_name	state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

p_name	manufacturer
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

c_name	state	p_name	manufacturer
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper
NewCo	TX	NULL	NULL

"LEFT" because result includes dangling tuples from the left table..

Original tables have no data.

```
SELECT  *
FROM    Company
LEFT OUTER JOIN Product ON manufacturer = c_name;
```

Right outer join

Company

c_name	state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA

Product

p_name	manufacturer
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper

c_name	state	p_name	manufacturer
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper

Useless in this example. Referential integrity guarantees Product has no dangling tuples.

```
SELECT  *
FROM    Company
RIGHT OUTER JOIN Product ON manufacturer = c_name;
```

Right outer join

Company

c_name	state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA

Product

p_name	manufacturer
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper
NewThing	NULL

Assume no referential integrity now.

c_name	state	p_name	manufacturer
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper
NULL	NULL	NewThing	NULL

```
SELECT  *
FROM    Company
RIGHT OUTER JOIN Product ON manufacturer = c_name;
```

Full outer join

Company

c_name	state
GizmoWorks	TX
WidgetsRUs	NY
Hyper	CA
NewCo	TX

Product

p_name	manufacturer
Gizmo	GizmoWorks
Powergizmo	GizmoWorks
Widget	WidgetsRUs
HyperWidget	Hyper
NewThing	NULL

c_name	state	p_name	manufacturer
GizmoWorks	TX	Gizmo	GizmoWorks
GizmoWorks	TX	Powergizmo	GizmoWorks
WidgetsRUs	NY	Widget	WidgetsRUs
Hyper	CA	HyperWidget	Hyper
NewCo	TX	NULL	NULL
NULL	NULL	NewThing	NULL

Includes dangling tuples from both sides.

```
SELECT  *
FROM    Company
FULL OUTER JOIN Product ON manufacturer = c_name;
```

Cross join

```
SELECT  *
FROM    Product
CROSS JOIN  Company;
```

CROSS JOIN doesn't have join condition.

Old, deprecated style:

```
SELECT  *
FROM    Product, Company;
```

p_name	price	manufacturer	c_name	address	city	state
Gizmo	19.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Gizmo	19.99	GizmoWorks	WidgetsRUs	20 Main St.	New York	NY
Gizmo	19.99	GizmoWorks	Hyper	1 Mission Dr.	San Francisco	CA
Powergizmo	39.99	GizmoWorks	GizmoWorks	123 Gizmo St.	Houston	TX
Powergizmo	39.99	GizmoWorks	WidgetsRUs	20 Main St.	New York	NY
Powergizmo	39.99	GizmoWorks	Hyper	1 Mission Dr.	San Francisco	CA
Widget	19.99	WidgetsRUs	GizmoWorks	123 Gizmo St.	Houston	TX
…	…	…	…	…	…	…

All combinations of records! – Cross-product of tables.

Self-joins – Joining table with itself

Employee

id	name	boss_id
1	Joe	3
2	Mary	3
3	Charles	4
4	Lisa	NULL

name	boss_name
Joe	Charles
Mary	Charles
Charles	Lisa

```
SELECT  emp.name, boss.name AS boss_name
FROM    Employee emp
INNER JOIN Employee boss ON emp.boss_id = boss.id;
```

Useful when one column (boss_id) refers to another column's (id) data.

Visual summary

Of inner & outer joins

Examples adapted from Coding Horror.

A

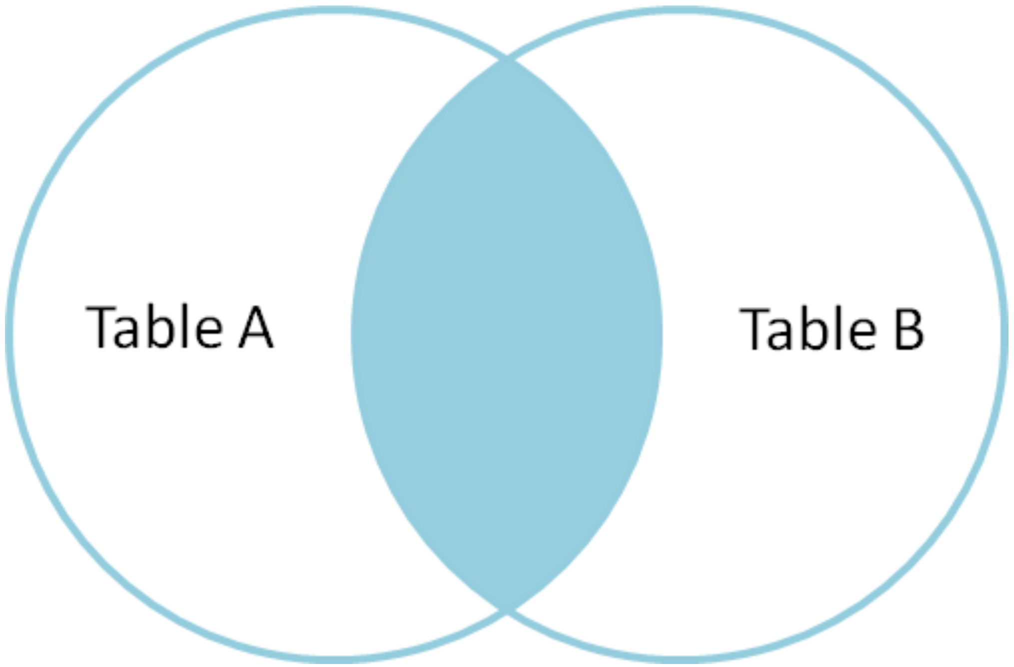
<u>id</u>	data
1	A
2	B
3	C
4	D

B

<u>id</u>	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
2	B	W
4	D	X

```
SELECT  *
FROM    A
INNER JOIN  B ON A.id = B.id;
```



Records matching both A and B.

A

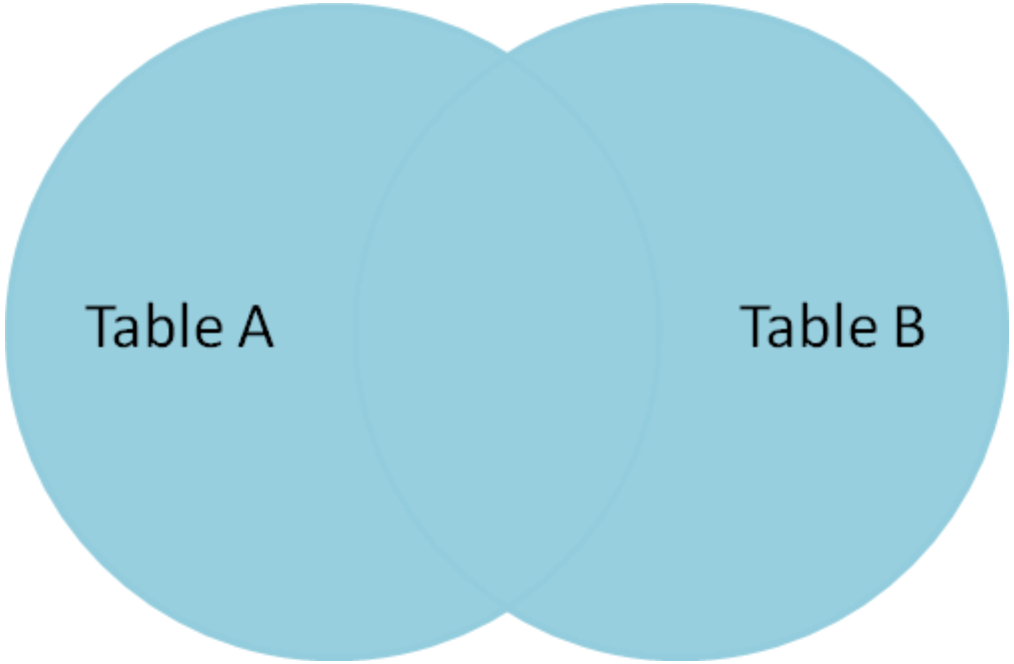
id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
1	A	
2	B	W
3	C	
4	D	X
6		Y
8		Z

```
SELECT  *
FROM    A
FULL OUTER JOIN  B ON A.id = B.id;
```

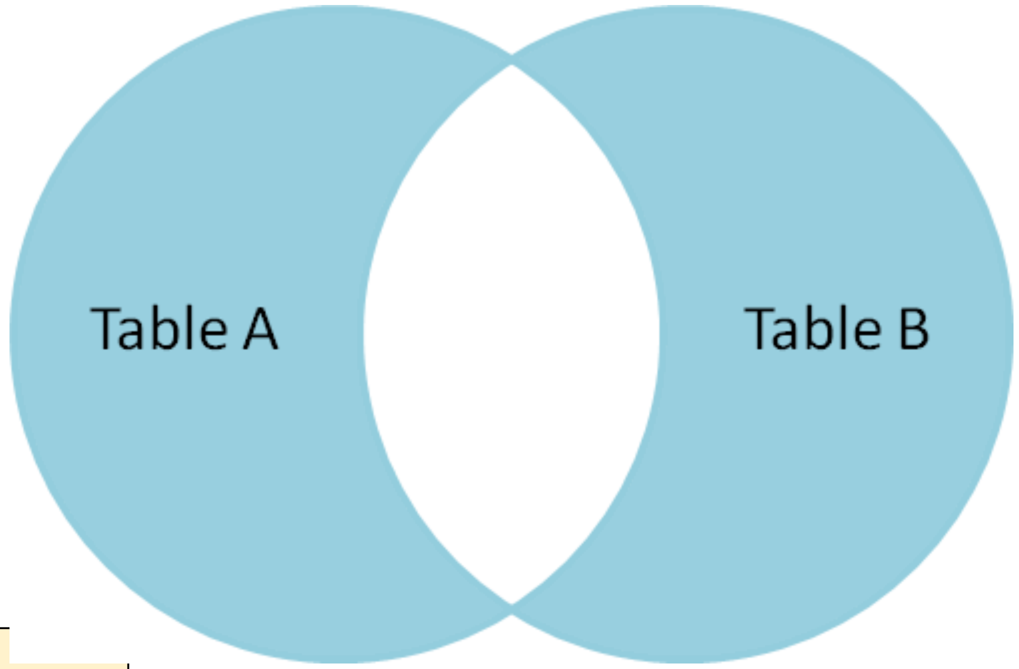


Records matching A or B.

A

id	data
1	A
2	B
3	C
4	D

id	A.data	B.data
1	A	
3	C	
6		Y
8		Z



B

id	data
2	W
4	X
6	Y
8	Z

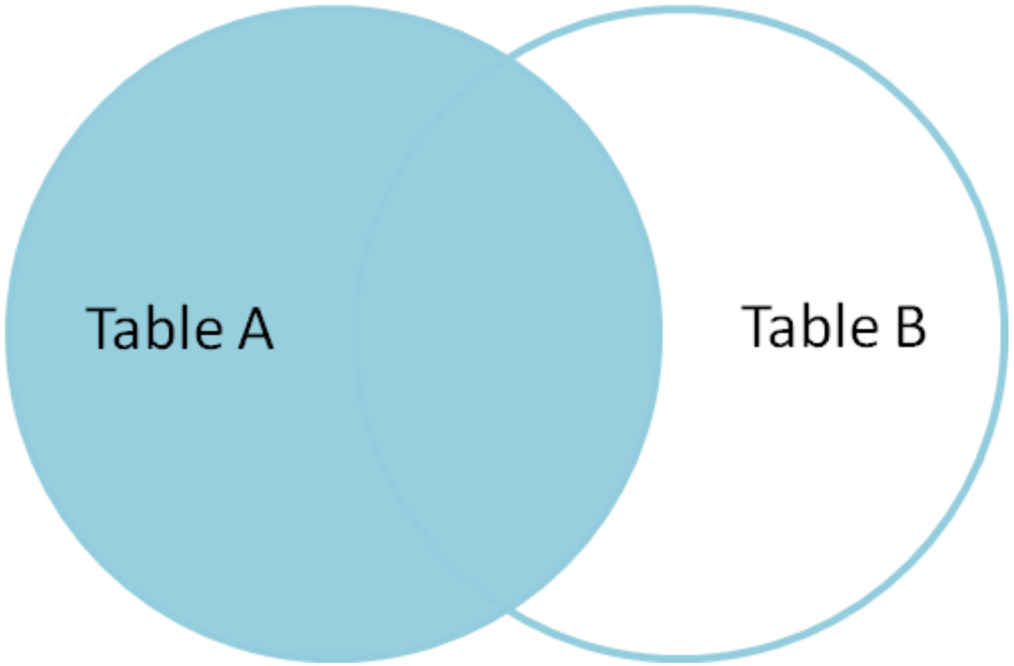
```
SELECT  *
FROM    A
FULL OUTER JOIN  B ON A.id = B.id
WHERE A.data IS NULL OR B.data IS NULL;
```

Records matching either A or B, but not both.

A

id	data
1	A
2	B
3	C
4	D

id	A.data	B.data
1	A	
2	B	W
3	C	
4	D	X



B

id	data
2	W
4	X
6	Y
8	Z

```
SELECT  *
FROM    A
LEFT OUTER JOIN  B ON A.id = B.id;
```

Records matching A.

A

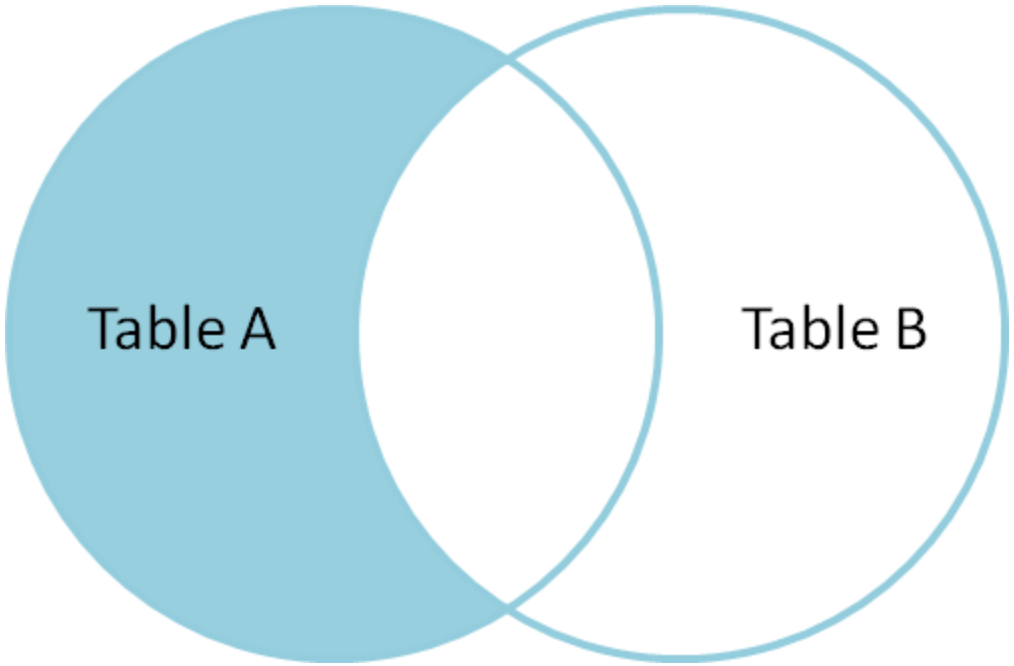
id	data
1	A
2	B
3	C
4	D

B

id	data
2	W
4	X
6	Y
8	Z

id	A.data	B.data
1	A	
3	C	

```
SELECT  *
FROM    A
LEFT OUTER JOIN  B ON A.id = B.id
WHERE B.data IS NULL;
```



Records matching only A.

A few notes

- Short-hands: JOIN = INNER JOIN, LEFT JOIN = LEFT OUTER JOIN, …
- Outer joins allow NATURAL, USING.
- Left outer join of A,B = Right outer join of B,A.

- These “joins” describe semantics. Later see various join implementations.

Constraints

Review

```
CREATE TABLE Product (
   id             INT,
   name           VARCHAR(50) NOT NULL,
   company_ID     VARCHAR(50) NOT NULL,
   units_sold     INT DEFAULT 0,
   PRIMARY KEY (id),
   FOREIGN KEY (company_id) REFERENCES Company (id)
);
```

Can name key constraints

name constraints

```
CREATE TABLE Product (
    id              INT,
    name            VARCHAR(50) NOT NULL,
    company_ID  VARCHAR(50) NOT NULL,
    units_sold      INT DEFAULT 0,
    CONSTRAINT pk_productid PRIMARY KEY (id),
    CONSTRAINT fk_companyid FOREIGN KEY (company_id) REFERENCES Company (id)
);
```

Other syntax variations possible, partly depending on SQL vendor.

Other key constraints

Companies won't give multiple of their own products the same name. (Or so we'd hope.)

```
CREATE TABLE Product (
    id              INT,
    name            VARCHAR(50) NOT NULL,
    company_ID      VARCHAR(50) NOT NULL,
    units_sold      INT DEFAULT 0,
    CONSTRAINT pk_productid PRIMARY KEY (id),
    CONSTRAINT un_nameco UNIQUE (name, company_ID),
    CONSTRAINT fk_companyid FOREIGN KEY (company_id) REFERENCES Company (id)
);
```

Combination of fields must be UNIQUE. But, unlike PRIMARY KEY, allows them to be NULL.

Why would we want UNIQUE?

As discussed later, tables/relations can have multiple keys, but only one key is primary.

One common usage: Some advocate that all tables use a synthetic key, even though this breaks some “normal forms”. The synthetic key would be PRIMARY KEY. The natural key(s) would be UNIQUE.

Adding constraints

```
ALTER TABLE Product
ADD PRIMARY KEY (id);
```

```
ALTER VIEW … AS
…;
```

```
ALTER TABLE Product
ADD CONSTRAINT pk_productid PRIMARY KEY (id);
```

```
ALTER TABLE Product
MODIFY name VARCHAR(50) NOT NULL;
```

Deleting constraints

Most:

```
ALTER TABLE Product
DROP CONSTRAINT pk_productid;
```

Some:

```
ALTER TABLE Product
DROP PRIMARY KEY;
```

Some: N/A

General-purpose constraint: CHECK

```
CREATE TABLE Product (
    …
    price MONEY CHECK (price > 0),
    …
);
```

```
CREATE TABLE Person (
    …
    gender    CHAR(1),
    CONSTRAINT chk_gender CHECK (gender IN (‘M’, ‘F’)),
    …
);
```

```
CREATE TABLE Student (
    …
    matriculation_date  DATE,
    graduation_date       DATE,
    CONSTRAINT chk_dates CHECK (matriculation_date < graduation_date),
    …
);
```

Lookup tables vs. CHECK constraints

	Lookup table	CHECK constraint
Constrain to a finite set	✓	✓
Constrain to an infinite set	✘	✓
Constrain to arbitrary relation	✘	✓
Constraint is DB-user maintainable	Potentially	✘
Can share constraint among fields	✓	✓
Fast	✓	Depends on constraint complexity
Useful for other purposes than constraints	✓	✘
Summary	**Use for finite sets**	**Use for infinite sets and complex conditions**

(Potential) abuse of CHECK constraints

- Constraint accesses other table's contents (via user-defined function)
  - Limitation: Constraint not checked when other table's contents change.
  - Might signal that attributes should be put in the same table.

- Constraint ensures consistency of redundant columns X and Y.
  - Better: Calculate Y from X in a query or view.

Later: Should complex constraints be in the DB or the application?

Activity – Add & verify constraints

13-constraints.ipynb