http://dsg.csail.mit.edu/6.5830/

6.5830
Lecture 3

Tim Kraska

Lab 0 Due
Lab 1 Out

Key ideas:
Advanced SQL
Schema Design

Recap: Zoo Tables

Primary key

Animals

aid	name	age	species	acageno
1	Sam	3	Salamander	1
2	Mike	12	Giraffe	1
3	Sally	1	Student	2

Cages

no	feedtime	bldg
1	12:30	1
2	1:30	2

Keepers

id	name
1	Jane
2	Joe

Keeps

kid	cageno
1	1
1	2
2	1

Schema:

Animals: (aid, name, age, species, a*cageno*)

Cages: (no, feedtime, bldg)

Keepers: (id, name)

Keeps: (*kid, cageno)*

Foreign Key

Recap: Relational Principles

- Simple representation
- Set-oriented programming model that doesn't require "navigation"
- No physical data model description required(!)

Recap: Relational Data Model

- All data is represented as tables of records (*tuples)*
- Tables are unordered sets (no duplicates)
- Database is one or more tables
- Each relation has a *schema* that describes the types of the columns/fields
- Each field is a primitive type -- not a set or relation
- Physical representation/layout of data is not specified (no index types, nestings, etc)

Recap: Basic SQL structure

[informal grammar]

SELECT [DISTINCT] selectExpression

FROM tableExpression

WHERE expression

GROUP BY expression

HAVING expression

ORDER BY order

LIMIT number

Note: You learn SQL by writing SQL and not through this lecture. The lecture only covers the high-level concept. Please use the PSETs and the thousands of online tutorial to learn it. For the quiz we care less about that the syntax is 100% correct but that you understand the concept of working with relations.

Recap: Relational Algebra

- "Algebra" – Closed under its own operations
  - Every expression over relations produces a relation

- **Projection** (π(T,c1, …, cn))
  - select a subset of columns c1 .. cn
- **Selection** (σ(T, pred))
  - select a subset of rows that satisfy pred
- **Cross Product** (T1 x T2)
  - combine two tables
- **Join** ($\bowtie$(T1, T2, pred)) = σ(T1 x T2, pred)
  - combine two tables with a predicate
- Set operations (UNION, DIFFERENCE, etc)

Recap: Relational Algebra

- "Algebra" – Closed under its own operations
  - Every expression over relations produces a relation

- **Projection** (π(T,c1, …, cn))
  - select a subset of columns c1 .. cn
- **Selection** (**σ**(T, pred))
  - select a subset of rows that satisfy pred
- **Cross Product** (T1 x T2)
  - combine two tables
- **Join** (⋈(T1, T2, pred)) = **σ**(T1 x T2, pred)
  - combine two tables with a predicate
- Set operations (UNION, DIFFERENCE, etc)
- Aggregate operation

$$dept\_name\ \mathcal{G}\ \textbf{avg}(salary)\ \textbf{as}\ avg\_sal\ (instructor)$$

IMS v CODASYL v Relational

	IMS	CODASYL	Relational
Many to many relationships without redundancy	✗	✓	✓
Declarative, non “navigational” programming	✗	✗	✓

IMS v CODASYL v Relational

	IMS	CODASYL	Relational
Many to many relationships without redundancy	✗	✓	✓
Declarative, non “navigational” programming	✗	✗	✓
Physical data independence	✗	✗	✓

Physical Independence

Can change representation of data without needing to change code

Example:

```
SELECT a.name FROM animals AS a, cages AS c WHERE a.cageno =
c.no AND c.bldg = 32
```

- Nothing about how animals or cages tables are represented is evident
  - Could be sorted, stored in a hash table / tree, etc
  - Changing physical representation will not change SQL
- No specification of implementation
- Both CODASYL and IMS expose representation-dependent operations in their query API

IMS v CODASYL v Relational

	IMS	CODASYL	Relational
Many to many relationships without redundancy	✗	✓	✓
Declarative, non “navigational” programming	✗	✗	✓
Physical data independence	✗	✗	✓
Logical data independence	✗	✗	✓

Logical Data Independence

- What if I want to change the schema without changing the code?
- No problem if just adding a column or table
- *Views* allow us to map old schema to new schema, so old programs work
  - *Even when changing existing fields*

Key Idea: View

- View is a logical definition of a table in terms of other tables

- E.g., a view computing animals per cage

```
CREATE VIEW cage_count as (
    SELECT cageno, count(*)
    FROM animals JOIN cages ON cageno=no
    GROUP by cageno
)
```

This view can be used just like a table in other queries

Views Example

~~Animals~~ Animals2

id	Name	Feedtime	…
1	Sam	1:30	
2	Jenny	2:30	

- Suppose I want to add multiple feedtimes?
- How to support old programs?
  - Rename existing animals table to animals2
  - Create feedtimes table
  - Copy feedtime data from animals2
  - Remove feedtime column from animals2
  - Create a view called animals that is a query over animals2 and feedtimes

Feedtime

animalid	Feedtime	…
1	1:30	
2	2:30	

```
CREATE VIEW animals as (
SELECT id, name, age, species, cageno,
    (SELECT feedtime FROM feedtimes WHERE animalid = id  LIMIT 1) as feedtime
    FROM animals2
)
```

Note: in this example feedtimes are associated with animals, but they are associated with cages in the earlier DB

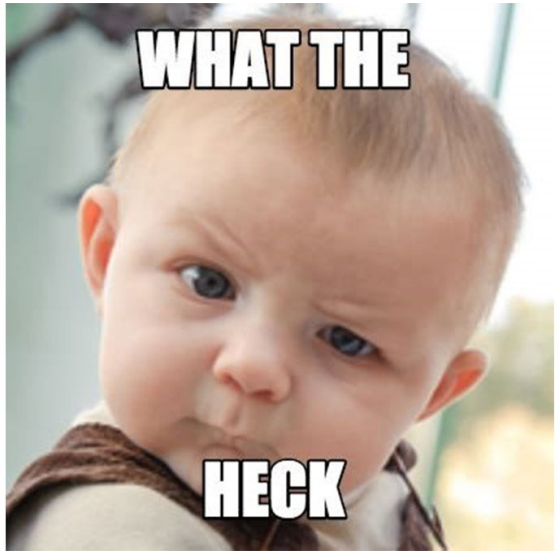
Correlated Subquery

```
SELECT id, name, age, species, cageno,
    (SELECT feedtime FROM feedtimes
     WHERE animalid = id LIMIT 1) as feedtime
FROM animals2
```

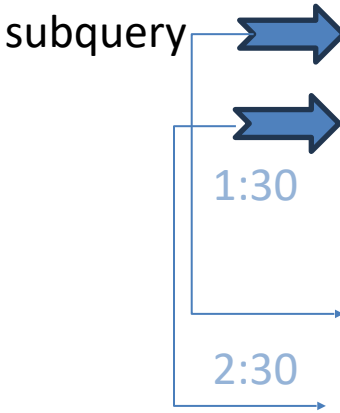
Evaluated once for each animal in animals2 table

Doesn't exist in feedtime table!

Return at most 1 feedtime



id	name	…	feedtime



id	Name	Feedtime	…
1	Sam	1:30	
2	Jenny	2:30	

animalid	Feedtime	…
1	1:30	
2	2:30	

Summary: IMS v CODASYL v Relational

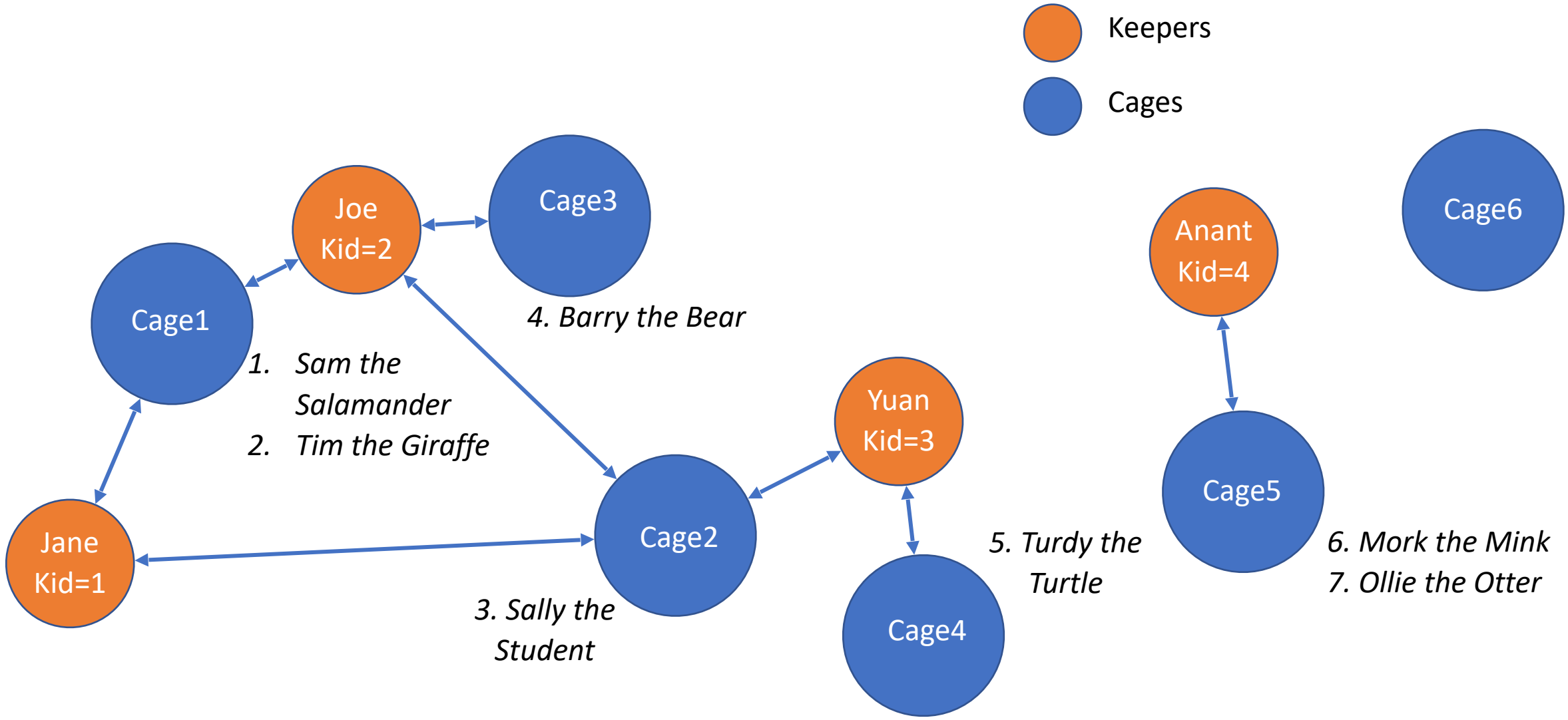
	IMS	CODASYL	Relational
Many to many relationships without redundancy	✗	✓	✓
Declarative, non “navigational” programming	✗	✗	✓
Physical data independence	✗	✗	✓
Logical data independence	✗	✗	✓

Next time: Fancy SQL

This Lecture

- Fancy SQL
- Database Design and Normalization

Expanded Animal DB, as a Graph



Animals: (aid, name, age, species, *acageno*)
Cages: (no, feedtime, bldg)
Keepers: (id, name)
Keeps: (*kid, cageno*)

Cages in Building 32

NESTED LOOPS

- Imperative

```
for each row a in animals
    for each row c in cages
        if a.acageno = c.no and c.bldg = 32
            output a
```

- Declarative

JOIN

```
SELECT name
FROM animals, cages
WHERE acageno = no AND bldg = 32
```

Alternate Syntax

```
SELECT name
FROM animals JOIN cages on acageno = no
WHERE bldg = 32
```

Aliases and Ambiguity

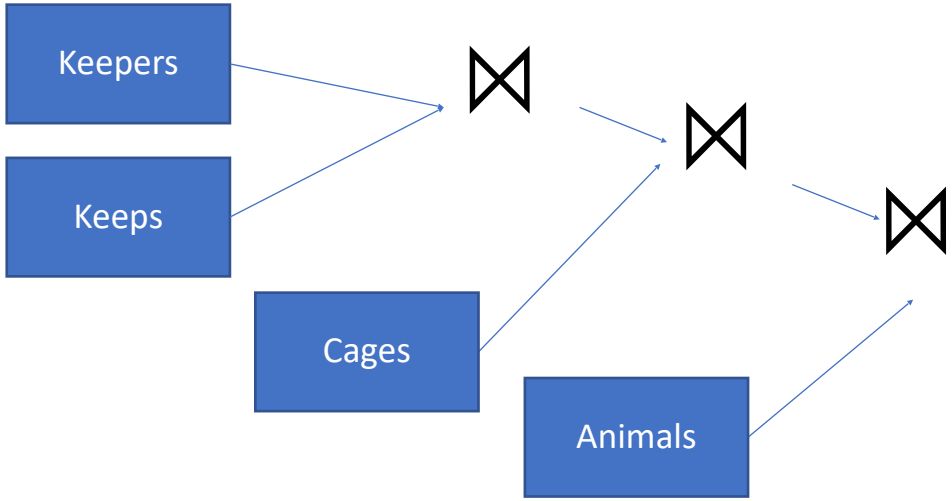
Animals: (aid, name, age, species, *acageno*)
Cages: (no, feedtime, bldg)
Keepers: (id, name)
Keeps: (*kid, cageno*)

- Keepers who keep bears

```
SELECT name
FROM keepers JOIN keeps ON id = kid
JOIN cages on cageno = no
JOIN animals on acageno =no
WHERE species = 'bear'
```

Unclear which "name" we are referring to

This doesn't work. Why?



4 table join
((keepers ⋈ *keeps)* ⋈ *cages)* ⋈ animals

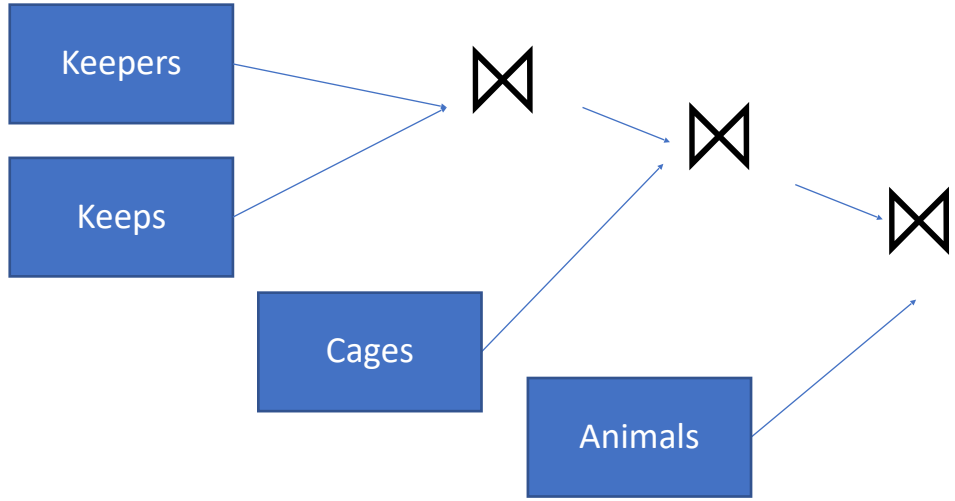
Aliases and Ambiguity

Animals: (aid, name, age, species, *acageno*)
Cages: (no, feedtime, bldg)
Keepers: (id, name)
Keeps: (*kid, cageno*)

- Keepers who keep bears

```
SELECT animals.name
FROM keepers JOIN keeps ON id = kid
JOIN cages on cageno = no
JOIN animals on acageno =no
WHERE species = 'bear'
```

This doesn't work. Why?



4 table join
((keepers ⋈ *keeps)* ⋈ *cages)* ⋈ animals

https://clicker.mit.edu/6.8530/

Fill in the blank to complete this query to “find cages kept by Jane”?
SELECT no FROM _____ WHERE name = 'jane'

A. keepers, cages
B. keepers JOIN cages ON keepers.id = cages.no
C. keepers JOIN keeps ON id = kid JOIN cages ON cageno = no
D. cages JOIN keepers on keepers.id = cages.no JOIN keeps ON cageno = no

Animals: (aid, name, age, species, *acageno*)
Cages: (no, feedtime, bldg)
Keepers: (id, name)
Keeps: (*kid, cageno*)

Aggregation

- Find the number of keepers of each cage

```
SELECT no, count(*)
FROM cages JOIN keeps ON no=cageno
GROUP BY no
```

- What about cages with 0 keepers?

Animals: (aid, name, age, species, *acageno*)
Cages: (no, feedtime, bldg)
Keepers: (id, name)
Keeps: (*kid, cageno*)

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't join with any row in T2
  - For those rows, fields of T2 are NULL

Example:

SELECT no, MAX(kid)

FROM cages LEFT JOIN keeps

ON no=cageno

GROUP BY no

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

cages

no	…
1	
2	
3	
4	
5	
6	

no	MAX
1	2
2	3
3	2
4	3
5	4
6	NULL

Can also use "RIGHT JOIN" and "FULL OUTER JOIN" to get all rows of T2 or all rows of both T1 and T2

In relational algebra

$_{no}G_{no,max(kid)}$ (cages $\overrightarrow{\bowtie}_{no=cageno}$ keeps)

$_{no}G_{no,max(kid)}$ (cages $\overleftrightarrow{\bowtie}_{no=cageno}$ keeps)

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't join with any row in T2
  - For those rows, fields of T2 are NULL

Example:

```
SELECT no, MAX(kid)
FROM cages LEFT JOIN keeps
ON no=cageno
GROUP BY no
```

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

cages

no	…
1	
2	
3	
4	
5	
6	

Can also use "RIGHT JOIN" and "OUTER JOIN" to get all rows of T2 or all rows of both T1 and T2

What about COUNT?

no	MAX
1	2
2	3
3	2
4	3
5	4
6	NULL

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't join with any row in T2
  - For those rows, fields of T2 are NULL

Example:

```
SELECT no, COUNT(*)
FROM cages LEFT JOIN keeps
ON no=cageno
GROUP BY no
```

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

cages

no	…
1	
2	
3	
4	
5	
6	

no	COUNT
1	2
2	2
3	1
4	1
5	1
6	1

Not what we wanted!

Left Join?

- T1 LEFT JOIN T2 ON pred produces all rows in T1 x T2 that satisfy pred, plus all rows in T1 that don't join with any row in T2
  - For those rows, fields of T2 are NULL

Example:

```
SELECT no, COUNT(cageno)
FROM cages LEFT JOIN keeps
ON no=cageno
GROUP BY no
```

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

cages

no	…
1	
2	
3	
4	
5	
6	

no	COUNT
1	2
2	2
3	1
4	1
5	1
6	0

COUNT(*) counts all rows including NULLs, COUNT(col) only counts rows with non-null values in col

https://clicker.mit.edu/6.8530/

Return all keeper names who keep bears and giraffes

OPTION A

```
SELECT keepers.name
FROM keepers JOIN keeps ON id = kid
JOIN cages ON cageno = no
JOIN animals ON acageno = cageno
WHERE species = 'Bear’ AND species  = 'Giraffe'
```

OPTION B

```
SELECT keepers.name
FROM keepers JOIN keeps ON id = kid
JOIN cages ON cageno = no
JOIN animals ON acageno = cageno
WHERE species = 'Bear’ OR species  = 'Giraffe'
```

OPTION C

```
SELECT keepers.name
FROM keepers JOIN keeps ON id = kid
JOIN cages ON cageno = no
JOIN animals ON acageno = cageno
GROUP BY species
HAVING species = 'Bear’ AND species  = 'Giraffe'
```

OPTION D

None of the options work

Animals: (aid, name, age, species, *acageno*)
Cages: (no, feedtime, bldg)
Keepers: (id, name)
Keeps: (*kid, cageno)*

Self Joins

- Keepers who keep bears and giraffes

```
SELECT keepers.name
FROM keepers JOIN keeps ON id = kid
JOIN cages ON cageno = no
JOIN animals ON acageno = cageno
WHERE species = 'Bear' AND species  = 'Giraffe'
```

Doesn't work!

OR species = 'Giraffe'?

Also doesn't work!

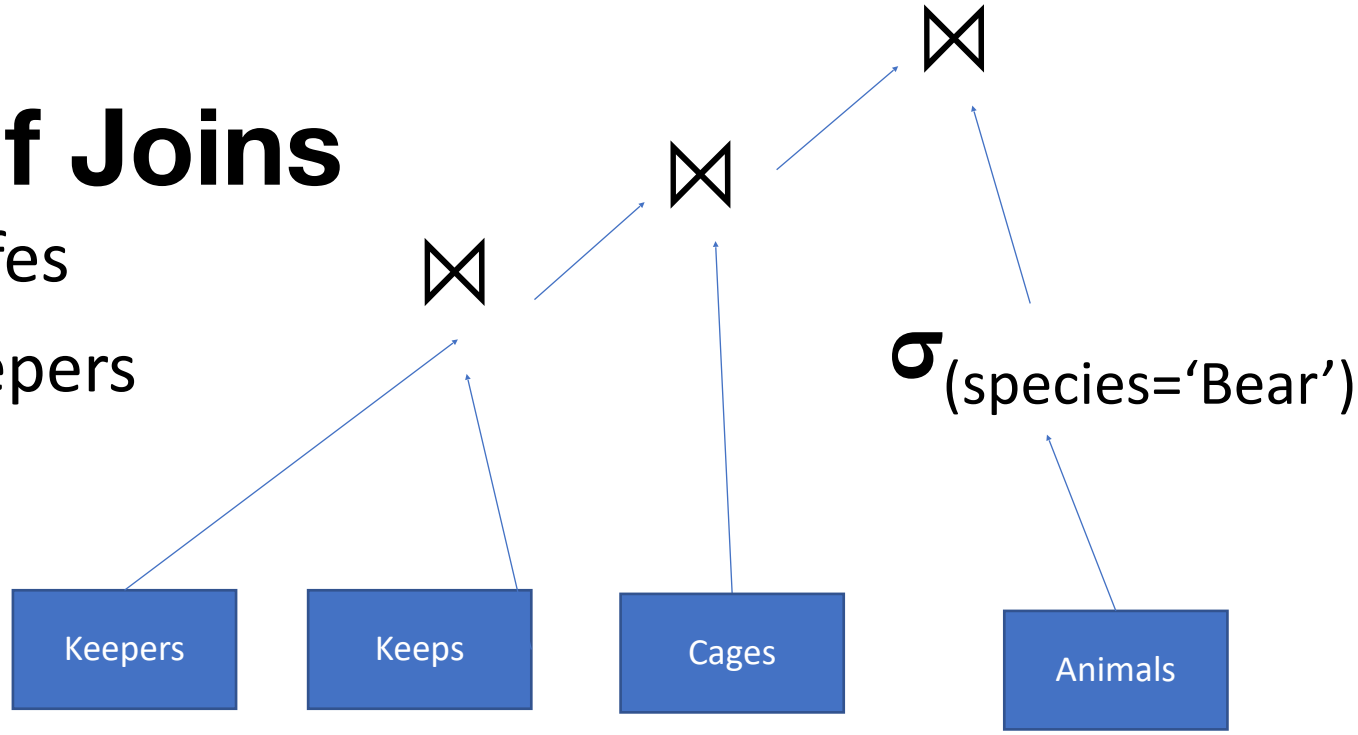
Self Joins

- Keepers who keep bears and giraffes
- Need to build two tables, Bear keepers and Giraffe keepers, and intersect them

```
SELECT bear_keepers.name
FROM keepers AS bear_keepers
JOIN keeps AS bear_keeps ON bear_keepers.id = bear_keeps.kid
JOIN cages AS bear_cages ON bear_keeps.cageno = bear_cages.no
JOIN animals AS bear_animals ON bear_animals.acageno = bear_cages.no
JOIN keepers AS giraffe_keepers
JOIN keeps AS giraffe_keeps ON giraffe_keepers.id = giraffe_keeps.kid
JOIN cages AS giraffe_cages ON giraffe_keeps.cageno = giraffe_cages.no
JOIN animals AS giraffe_animals ON giraffe_animals.acageno = giraffe_cages.no
WHERE bear_animals.species = 'Bear'
AND giraffe_animals.species = 'Giraffe'
AND giraffe_keepers.id = bear_keepers.id
```

Self Joins

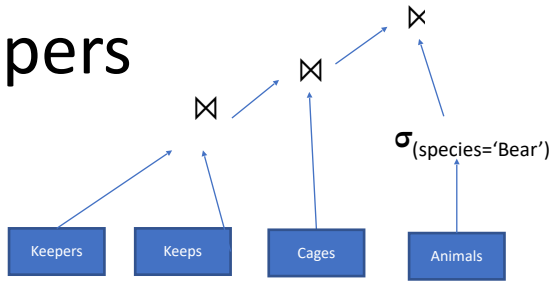
- Keepers who keep bears and giraffes
- Need to build two tables, Bear keepers and Giraffe keepers, and intersect them




```
SELECT bear_keepers.name
FROM keepers AS bear_keepers
JOIN keeps AS bear_keeps ON bear_keepers.id = bear_keeps.kid
JOIN cages AS bear_cages ON bear_keeps.cageno = bear_cages.no
JOIN animals AS bear_animals ON bear_animals.acageno = bear_cages.no
JOIN keepers AS giraffe_keepers
JOIN keeps AS giraffe_keeps ON giraffe_keepers.id = giraffe_keeps.kid
JOIN cages AS giraffe_cages ON giraffe_keeps.cageno = giraffe_cages.no
JOIN animals AS giraffe_animals ON giraffe_animals.acageno = giraffe_cages.no
WHERE bear_animals.species = 'Bear'
AND giraffe_animals.species = 'Giraffe'
AND giraffe_keepers.id = bear_keepers.id
```

Self Joins

- Keepers who keep bears and giraffes
- Need to build two tables, Bear keepers and Giraffe keepers, and intersect them

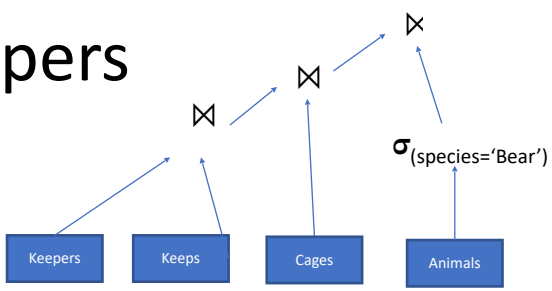


Bears

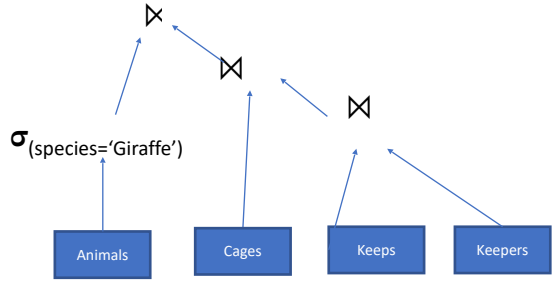
```
SELECT bear_keepers.name
FROM keepers AS bear_keepers
JOIN keeps AS bear_keeps ON bear_keepers.id = bear_keeps.kid
JOIN cages AS bear_cages ON bear_keeps.cageno = bear_cages.no
JOIN animals AS bear_animals ON bear_animals.acageno = bear_cages.no
JOIN keepers AS giraffe_keepers
JOIN keeps AS giraffe_keeps ON giraffe_keepers.id = giraffe_keeps.kid
JOIN cages AS giraffe_cages ON giraffe_keeps.cageno = giraffe_cages.no
JOIN animals AS giraffe_animals ON giraffe_animals.acageno = giraffe_cages.no
WHERE bear_animals.species = 'Bear'
AND giraffe_animals.species = 'Giraffe'
AND giraffe_keepers.id = bear_keepers.id
```

Self Joins

- Keepers who keep bears and giraffes
- Need to build two tables, Bear keepers and Giraffe keepers, and intersect them



Bears

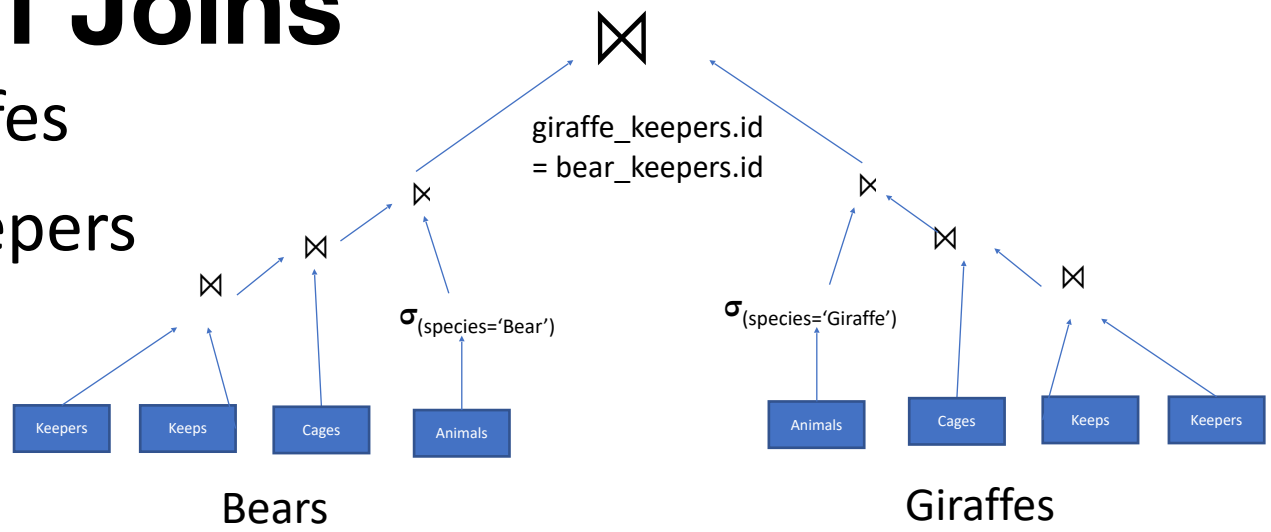


Giraffes

```
SELECT bear_keepers.name
FROM keepers AS bear_keepers
JOIN keeps AS bear_keeps ON bear_keepers.id = bear_keeps.kid
JOIN cages AS bear_cages ON bear_keeps.cageno = bear_cages.no
JOIN animals AS bear_animals ON bear_animals.acageno = bear_cages.no
JOIN keepers AS giraffe_keepers
JOIN keeps AS giraffe_keeps ON giraffe_keepers.id = giraffe_keeps.kid
JOIN cages AS giraffe_cages ON giraffe_keeps.cageno = giraffe_cages.no
JOIN animals AS giraffe_animals ON giraffe_animals.acageno = giraffe_cages.no
WHERE bear_animals.species = 'Bear'
AND giraffe_animals.species = 'Giraffe'
AND giraffe_keepers.id = bear_keepers.id
```

Self Joins

- Keepers who keep bears and giraffes
- Need to build two tables, Bear keepers and Giraffe keepers, and intersect them




```
SELECT bear_keepers.name
FROM keepers AS bear_keepers
JOIN keeps AS bear_keeps ON bear_keepers.id = bear_keeps.kid
JOIN cages AS bear_cages ON bear_keeps.cageno = bear_cages.no
JOIN animals AS bear_animals ON bear_animals.acageno = bear_cages.no
JOIN keepers AS giraffe_keepers
JOIN keeps AS giraffe_keeps ON giraffe_keepers.id = giraffe_keeps.kid
JOIN cages AS giraffe_cages ON giraffe_keeps.cageno = giraffe_cages.no
JOIN animals AS giraffe_animals ON giraffe_animals.acageno = giraffe_cages.no
WHERE bear_animals.species = 'Bear'
AND giraffe_animals.species = 'Giraffe'
AND giraffe_keepers.id = bear_keepers.id
```

7-way join, for a pretty simple query!

Nested Queries

```
SELECT bear_keepers.name
FROM (
    SELECT id, keepers.name FROM
    keepers JOIN keeps ON id = kid
    JOIN cages ON cageno = no
    JOIN animals ON acageno = no
    WHERE species = 'Bear'
) AS bear_keepers
JOIN (
    SELECT id, keepers.name FROM
    keepers JOIN keeps ON id = kid
    JOIN cages ON cageno = no
    JOIN animals ON acageno = no
    WHERE species = 'Giraffe'
) AS giraffe_keepers
ON giraffe_keepers.id = bear_keepers.id
```

Every query is a relation (table)

Anywhere you can use a table, you can use a query!

Simplify with Common Table Expressions (CTEs)

```
WITH bear_keepers AS (
   SELECT id, keepers.name FROM
   keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Bear'
),
giraffe_keepers AS (
   SELECT id, keepers.name FROM
   keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
)
SELECT bear_keepers.name
FROM bear_keepers JOIN giraffe_keepers
ON giraffe_keepers.id = bear_keepers.id
```

CTEs work better than nested expressions when the CTE needs to be referenced in multiple places

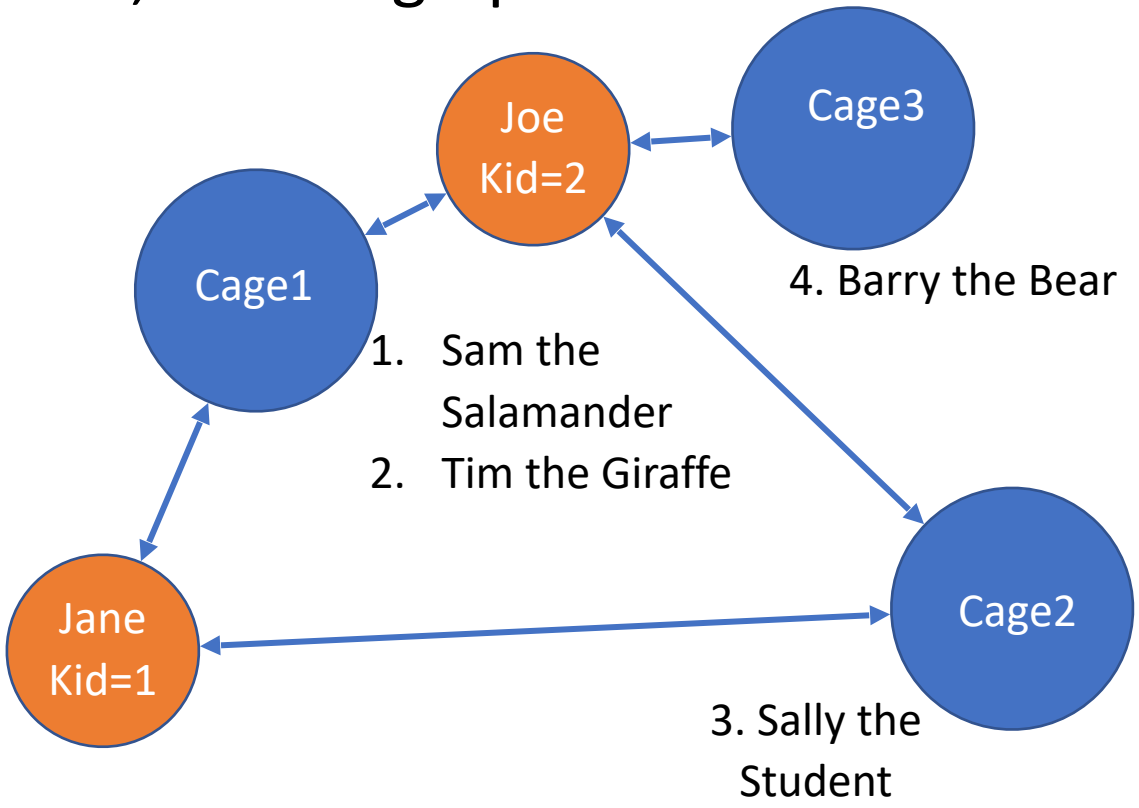
SQL can get complex

```
with one_phone_tags as (
   select tag_mac_address
   from mapmatch_history
   where uploadtime > '9/1/2021'::date and uploadtime < '9/10/2021'::date
   and json_extract_path_text(device_config,'manufacturer') = 'Apple’
   group by 1
   having count(distinct device_config_hint) = 1
),
ios15_tags as (
select  json_extract_path_text(device_config,'version_release') os_version,
          json_extract_path_text(device_config,'model') model_number,
          tag_mac_address
   from mapmatch_history
   where uploadtime >= '10/11/2021'::date
   and json_extract_path_text(device_config,'manufacturer') = 'Apple'
   and tag_mac_address in (select tag_mac_address from one_phone_tags)
  and substring(os_version, 1, 2) = '15'
   group by 1,2,3
),
ios14_tags as (
select  json_extract_path_text(device_config,'version_release') os_version,
          json_extract_path_text(device_config,'model') model_number,
          tag_mac_address
   from mapmatch_history
   where uploadtime >= '9/15/2021'::date and uploadtime <= '9/20/2021'::date
   and json_extract_path_text(device_config,'manufacturer') = 'Apple'
   and tag_mac_address in (select tag_mac_address from one_phone_tags)
```

```
ios15_trip_stats as (
   select tag_mac_address, count(*) ios15_num_trips,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end)
ios15_num_trips_no_phone,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end) /
count(*)::float ios15_frac_none,
   from triplog_trips join ios15_tags using(tag_mac_address)
   where created_date >= '10/11/2021'::date
   and trip_start_ts >= '10/09/2021'::date
   and substring(model_number, 1, 8) = 'iPhone13'
   group by tag_mac_address
   having count(*) > 0
),
ios14_trip_stats as (
   select tag_mac_address, count(*) ios14_num_trips,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end)
ios14_num_trips_no_phone,
   sum(case when mmh_display_distance_km isnull then 1 else 0 end) /
count(*)::float ios14_frac_none,
   from triplog_trips join ios14_tags using(tag_mac_address)
   where created_date >= '9/15/2021'::date and created_date <= '9/20/2021'::date
   and trip_start_ts >= '9/13/2021'::date and trip_start_ts <= '9/20/2021'::date
   and substring(model_number, 1, 8) = 'iPhone13'
   group by tag_mac_address
   having count(*) > 0
)
select
tag_mac_address,ios14_num_trips,ios14_num_trips_no_phone,ios14_frac_none,
      ios15_num_trips,ios15_num_trips_no_phone,ios15_frac_none
```

Study Break

- Write a SQL query to find animals kept by a keeper who keeps Giraffes
- I.e., for our graph:



keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

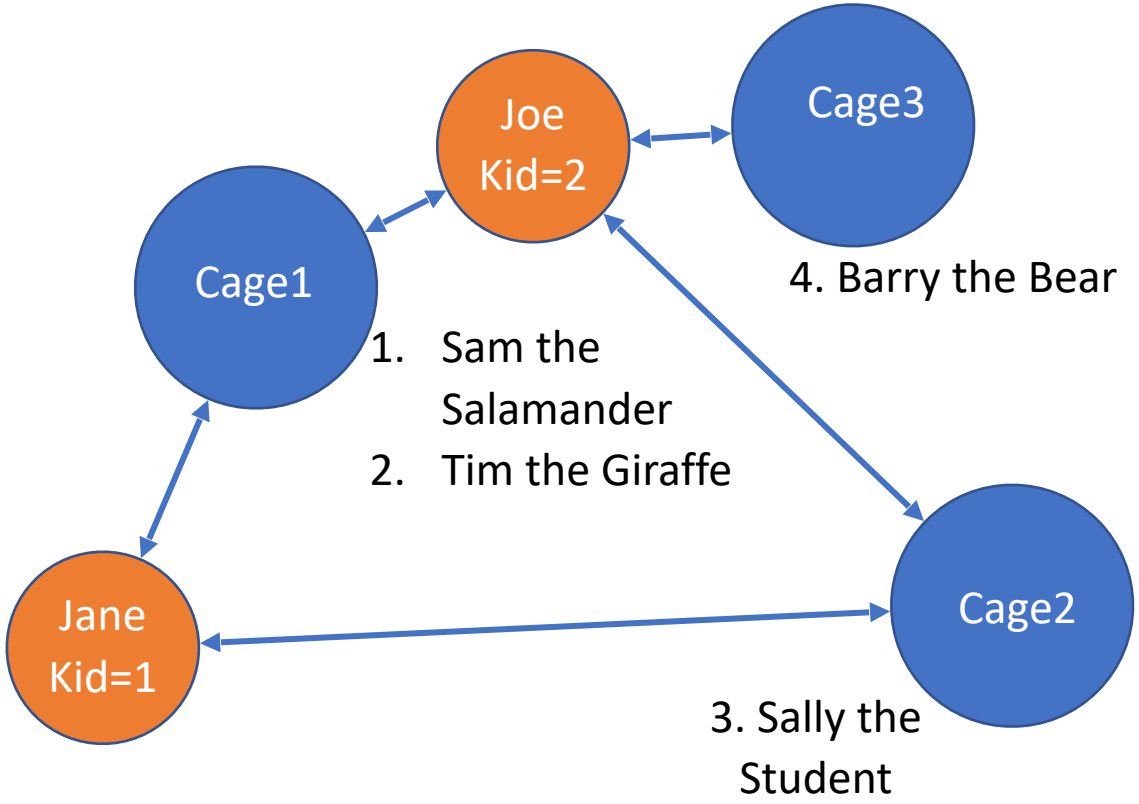
The keepers who keep Giraffes and the animals they keep are:

Joe, who keeps Sam, Barry, and Tim
Jane, who keeps Sally, Sam, and Tim

Sam, Barry, Sally

Solution

- Write a SQL query to find animals kept by a keeper who keeps Giraffes

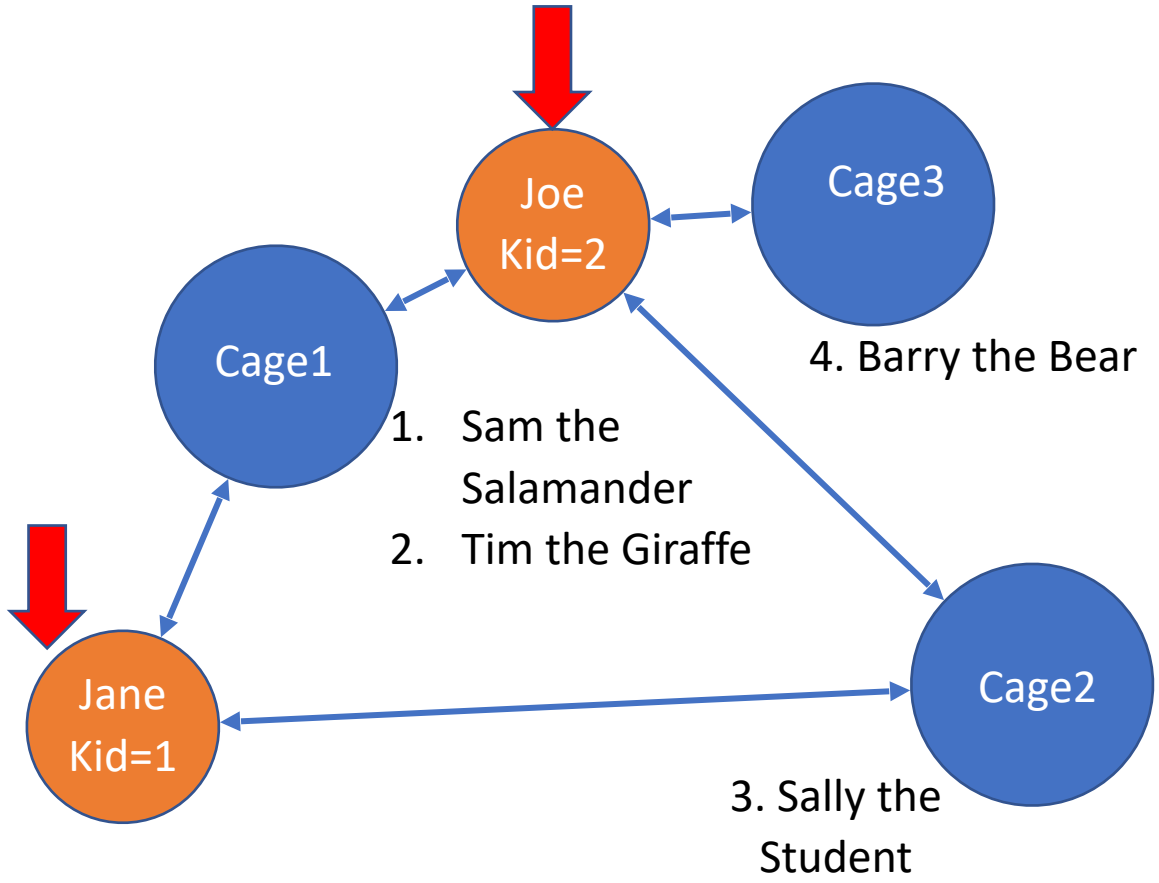



```
WITH giraffe_keepers AS (
   SELECT id
   FROM keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
), giraffe_keeper_cages AS (
   SELECT cageno FROM
   giraffe_keepers JOIN keeps ON kid = id
)
SELECT name,species
FROM animals JOIN giraffe_keeper_cages
ON cageno = acageno
WHERE species != 'Giraffe'
```

keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

Solution

- Write a SQL query to find animals kept by a keeper who keeps Giraffes

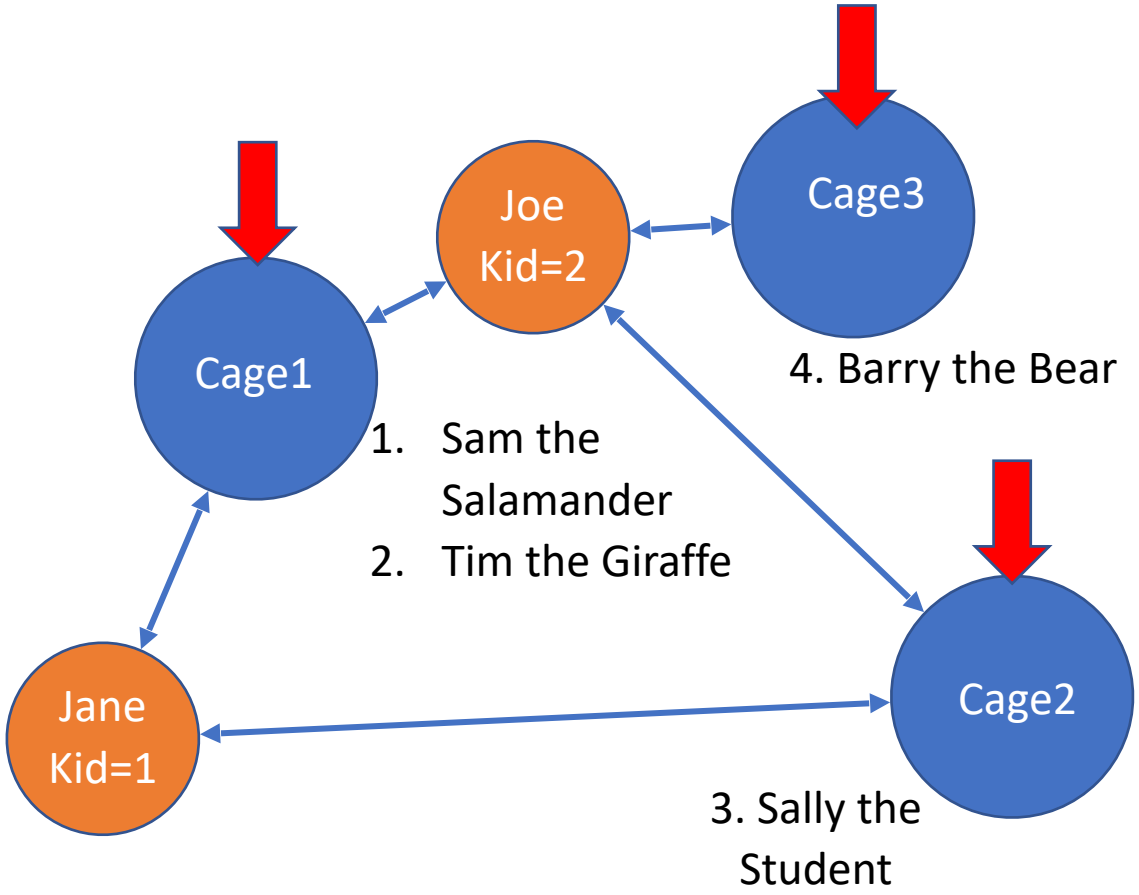



```
WITH giraffe_keepers AS (
   SELECT id
   FROM keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
), giraffe_keeper_cages AS (
   SELECT cageno FROM
   giraffe_keepers JOIN keeps ON kid = id
)
SELECT name,species
FROM animals JOIN giraffe_keeper_cages
ON cageno = acageno
WHERE species != 'Giraffe'
```

keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

Solution

- Write a SQL query to find animals kept by a keeper who keeps Giraffes

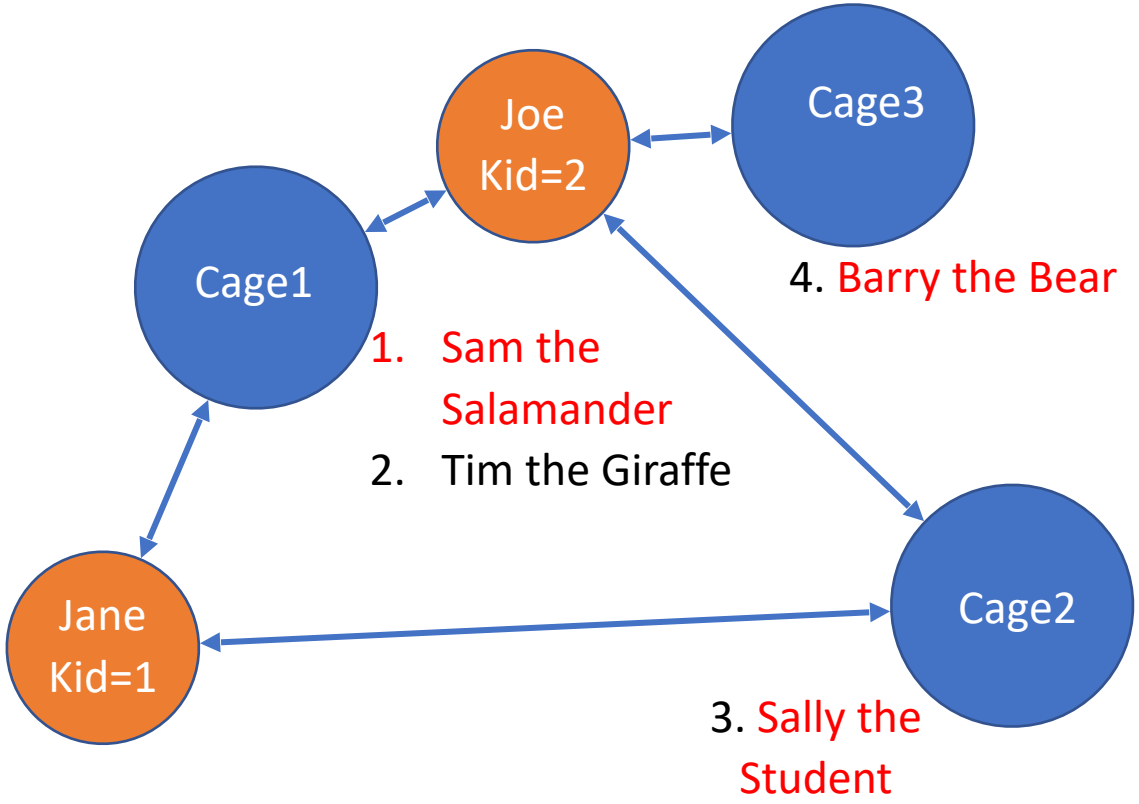



```
WITH giraffe_keepers AS (
   SELECT id
   FROM keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
), giraffe_keeper_cages AS (
   SELECT cageno FROM
   giraffe_keepers JOIN keeps ON kid = id
)
SELECT name,species
FROM animals JOIN giraffe_keeper_cages
ON cageno = acageno
WHERE species != 'Giraffe'
```

keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

Solution

- Write a SQL query to find animals kept by a keeper who keeps Giraffes




```
WITH giraffe_keepers AS (
   SELECT id
   FROM keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
), giraffe_keeper_cages AS (
   SELECT cageno FROM
   giraffe_keepers JOIN keeps ON kid = id
)
SELECT name,species
FROM animals JOIN giraffe_keeper_cages
ON cageno = acageno
WHERE species != 'Giraffe'
```

Solution

keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

Write a SQL query to find animals kept by a keeper who keeps Giraffes

```
WITH giraffe_keepers AS (
   SELECT id
   FROM keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
), giraffe_keeper_cages AS (
   SELECT cageno FROM
   giraffe_keepers JOIN keeps ON kid = id
)
SELECT name,species
FROM animals JOIN giraffe_keeper_cages
ON cageno = acageno
WHERE species != 'Giraffe'
```

1,2

1
2
1
3

Run it:

```
Sally|Student
Sam|Salamander
Sally|Student
Barry|Bear
```

Problem: Duplicates!

Solution

keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

- Write a SQL query to find animals kept by a keeper who keeps Giraffes

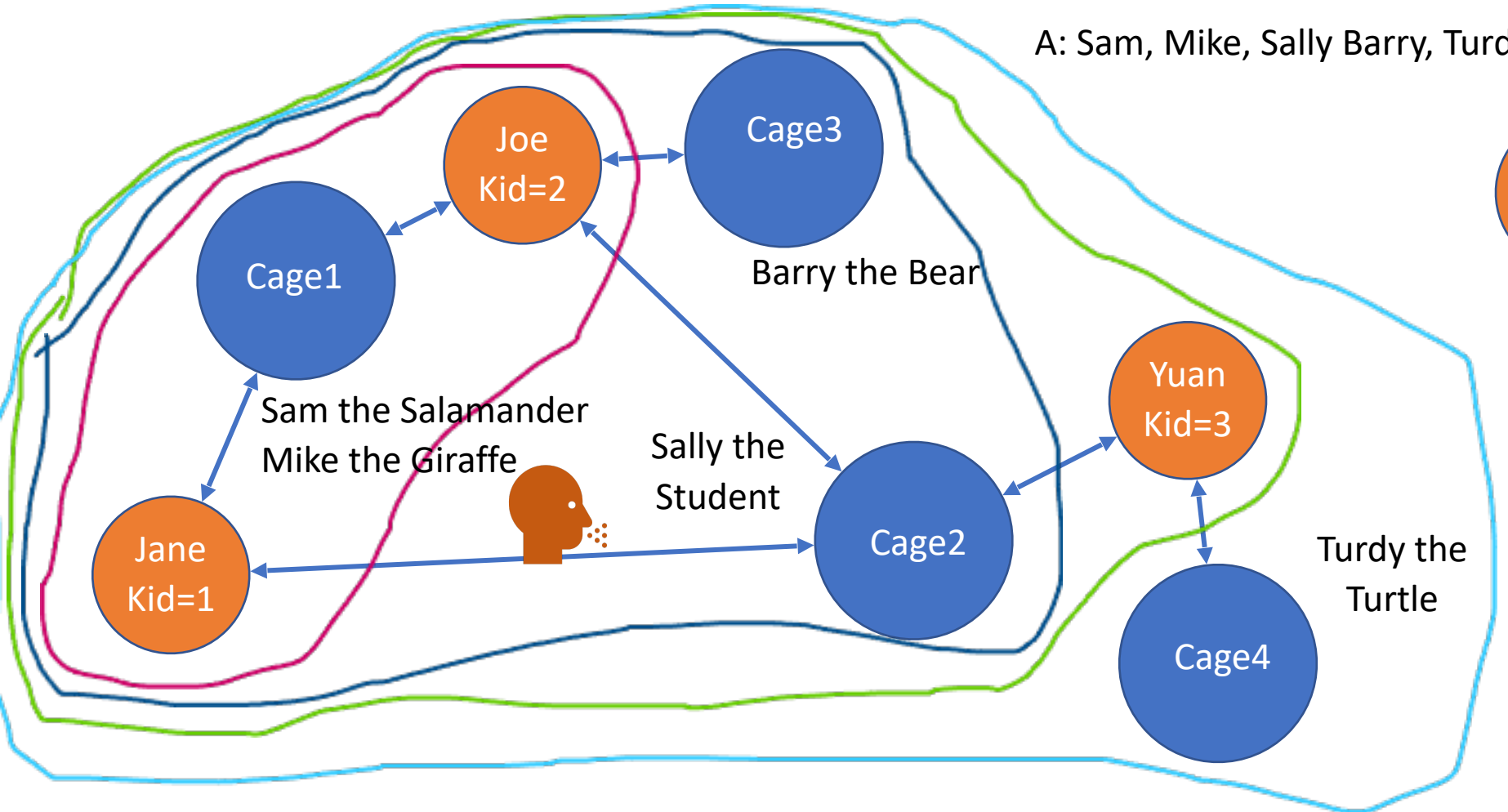
```
WITH giraffe_keepers AS (
   SELECT id
   FROM keepers JOIN keeps ON id = kid
   JOIN cages ON cageno = no
   JOIN animals ON acageno = no
   WHERE species = 'Giraffe'
), giraffe_keeper_cages AS (
   SELECT cageno FROM
   giraffe_keepers JOIN keeps ON kid = id
)
SELECT DISTINCT name,species
FROM animals JOIN giraffe_keeper_cages
ON cageno = acageno
WHERE species != 'Giraffe'
```

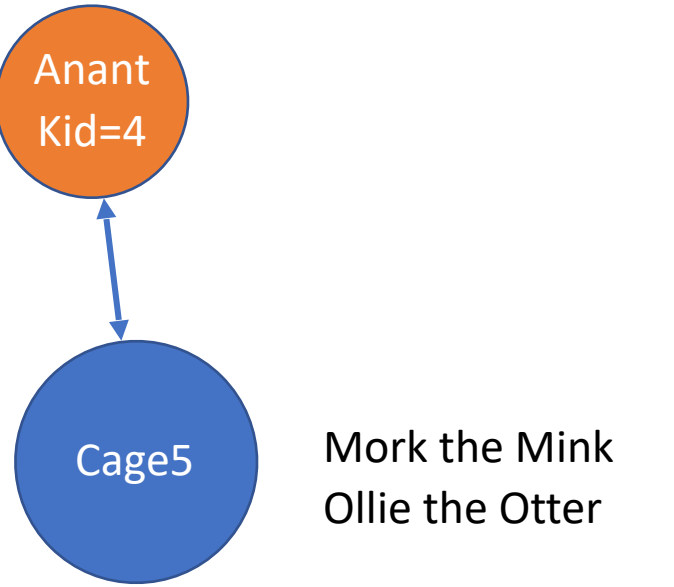
Run it:

```
Sally|Student
Sam|Salamander
Barry|Bear
```

Recursive Queries

- Suppose there is a breakout of a dangerous disease that spreads through humans and animals, and we need to find all animals that have been in contact with a keeper or animal who might be sick





Challenge: each successive join follows one set of edges. Size of graph is unbounded!

Recursive Queries

- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

```
WITH RECURSIVE t(n) AS
(SELECT 1  as n
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Recursive Queries

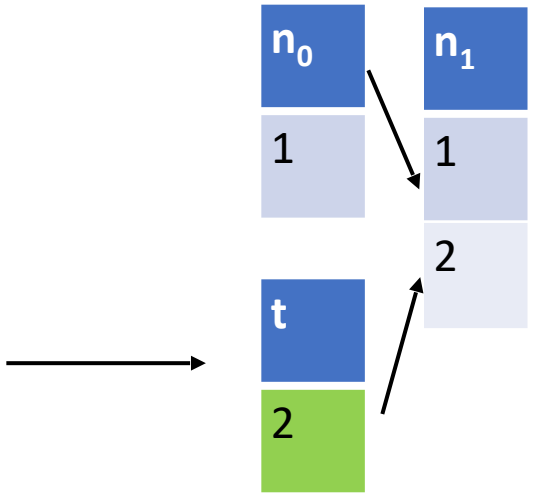
- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

```
WITH RECURSIVE t(n) AS
(SELECT 1  as n
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Recursive Queries

- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

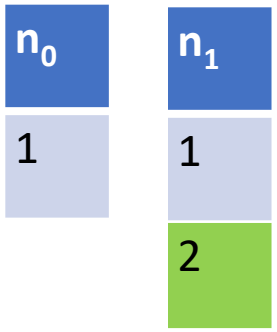
```
WITH RECURSIVE t(n) AS
(SELECT 1  as n
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```



Recursive Queries

- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

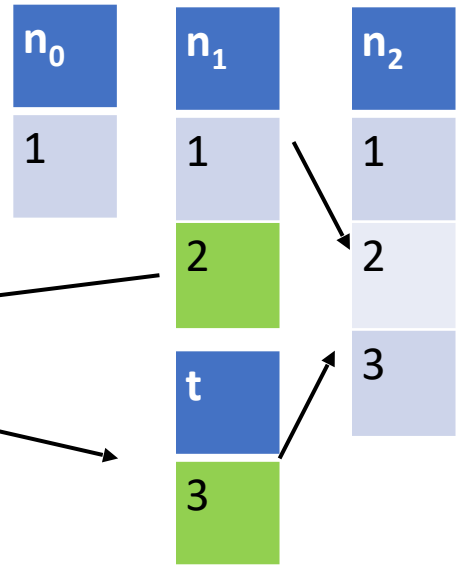
```
WITH RECURSIVE t(n) AS
(SELECT 1  as n
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```



Recursive Queries

- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

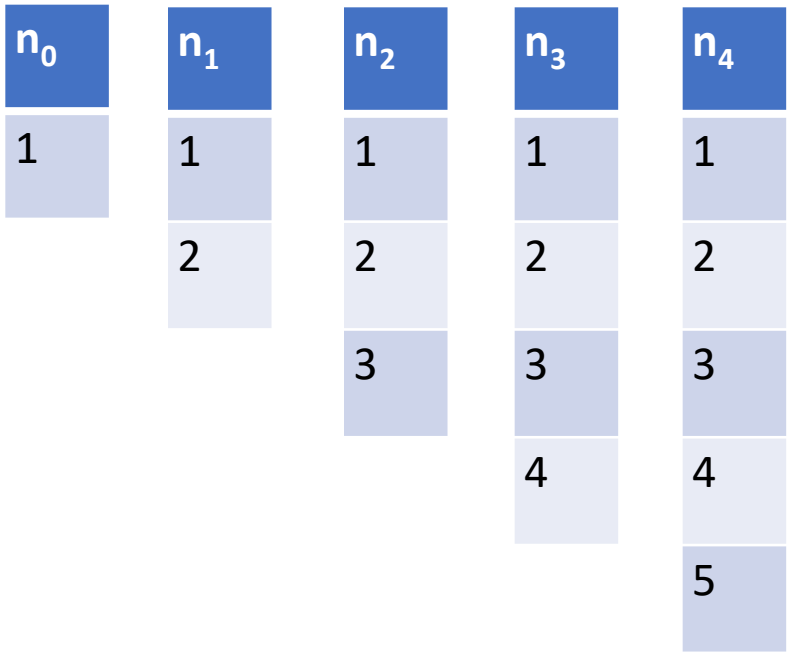
```
WITH RECURSIVE t(n) AS
(SELECT 1  as n
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```



Recursive Queries

- Recursive WITH clause can join with itself
- Example: define a table t with one column n, iteratively join with with itself

```
WITH RECURSIVE t(n) AS
(SELECT 1  as n
UNION
SELECT n+1
FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```



The Power of Recursion

- Recursion makes SQL Turing complete
- Some logical are surprisingly easy to express, e.g., Sudoku solver:

```
WITH RECURSIVE
 input(sud) AS (VALUES('53..7....6..195....98....6.8...6...34..8.3..17...2...6.6....28....419..5....8..79')),
 digits(z, lp) AS (
  VALUES('1', 1)
  UNION ALL SELECT
  CAST(lp+1 AS TEXT), lp+1 FROM digits WHERE lp<9
 ),
 x(s, ind) AS (
  SELECT sud, instr(sud, '.') FROM input
  UNION ALL
  SELECT substr(s, 1, ind-1) || z || substr(s, ind+1),
   instr( substr(s, 1, ind-1) || z || substr(s, ind+1), '.' )
   FROM x, digits AS z WHERE ind>0
   AND NOT EXISTS (
       SELECT 1
        FROM digits AS lp
        WHERE z.z = substr(s, ((ind-1)/9)*9 + lp, 1)
          OR z.z = substr(s, ((ind-1)%9) + (lp-1)*9 + 1, 1)
          OR z.z = substr(s, (((ind-1)/3) % 3) * 3
                + ((ind-1)/27) * 27 + lp
                + ((lp-1) / 3) * 6, 1))
)
```

Table of digits, 1-9

Solution, given "." at position ind

Find an assignment to a "." that satisfies constraints of Sudoku

Expression of constraints

Puzzle encoding ("." = blank)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Recursive Queries

- Suppose we need to find all animals that have been in contact with a keeper or animal who might be sick

```
WITH recursive sick_keepers as (
   SELECT kid as sick_id  -- keepers who keep an animal who is sick
   FROM keeps
   JOIN animals on acageno = cageno
   WHERE animals.name = 'Mike'
UNION
   SELECT k1.kid  -- keepers who keep the same cage as another
                  --     keeper who might be sick
   FROM keeps k1
   JOIN keeps k2 on k2.cageno = k1.cageno
   JOIN sick_keepers on k2.kid = sick_id
)
SELECT distinct(name) FROM animals -- animals in cages with keepers who might be sick
JOIN keeps on cageno = acageno
JOIN sick_keepers ON sick_id = kid
```

Base case: keepers of Mike (note: no need to look at cages table)

Each successive iteration: keepers who keep the same cage as a keeper who might be sick

Animals kept in the cages that possibly sicky keepers keep

keepers (id, name)
cages (no, feedtime, bldg)
animals (aid, age, species, acageno, name)
keeps (kid, cageno)

Recursion Example

- Mike is in cageno 1, kept by keepers 1 & 2

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

animals

Name	cageno
Mike	1
Sam	2
Sally	1
Barry	3
Turdy	4
Mork	5
Ollie	5

```
WITH recursive sick_keepers(kid) as (
   SELECT kid as sick_id
   FROM keeps k
   JOIN animals a on a.cageno = k.cageno
   WHERE animals.name = 'Mike'
UNION
   SELECT k2.kid as sick_id
   FROM sick_keepers
   JOIN keeps k1 on k1.kid = sick_id
   JOIN keeps k2 on k2.cageno = k1.cageno
)
```

Recursion Example

- Mike is in cageno 1, kept by keepers 1 & 2

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

animals

Name	cageno
Mike	1
Sam	2
Sally	1
Barry	3
Turdy	4
Mork	5
Ollie	5

Base case

Sick_id
1
2

```
WITH recursive sick_keepers(kid) as (
   SELECT kid as sick_id
   FROM keeps k
   JOIN animals a on a.cageno = k.cageno
   WHERE animals.name = 'Mike'
UNION
   SELECT k2.kid as sick_id
   FROM sick_keepers
   JOIN keeps k1 on k1.kid = sick_id
   JOIN keeps k2 on k2.cageno = k1.cageno
)
```

Recursion Example

- Mike is in cageno 1, kept by keepers 1 & 2

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

animals

Name	cageno
Mike	1
Sam	2
Sally	1
Barry	3
Turdy	4
Mork	5
Ollie	5

```
WITH recursive sick_keepers(kid) as (
    SELECT kid as sick_id
    FROM keeps k
    JOIN animals a on a.cageno = k.cageno
    WHERE animals.name = 'Mike'
UNION
    SELECT k2.kid as sick_id
    FROM sick_keepers
    JOIN keeps k1 on k1.kid = sick_id
    JOIN keeps k2 on k2.cageno = k1.cageno
)
```

Base case

Sick_id
1
2

Recursion Example

- Mike is in cageno 1, kept by keepers 1 & 2

keeps

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

animals

Name	cageno
Mike	1
Sam	2
Sally	1
Barry	3
Turdy	4
Mork	5
Ollie	5

Base case

Sick_id
1
2

```
WITH recursive sick_keepers(kid) as (
   SELECT kid as sick_id
   FROM keeps k
   JOIN animals a on a.cageno = k.cageno
   WHERE animals.name = 'Mike'
UNION
   SELECT k2.kid as sick_id
   FROM sick_keepers
   JOIN keeps k1 on k1.kid = sick_id
   JOIN keeps k2 on k2.cageno = k1.cageno
)
```

Recursion Example

- Mike is in cageno 1, kept by keepers 1 & 2

```
WITH recursive sick_keepers(kid) as (
    SELECT kid as sick_id
    FROM keeps k
    JOIN animals a on a.cageno = k.cageno
    WHERE animals.name = 'Mike'
UNION
    SELECT k2.kid as sick_id
    FROM sick_keepers
    JOIN keeps k1 on k1.kid = sick_id
    JOIN keeps k2 on k2.cageno = k1.cageno
)
```

Keeps k1

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

Keeps k1

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

t0

Sick_id
1
2

t1

Sick_id
1
2
3

Recursion Example

- Mike is in cageno 1, kept by keepers 1 & 2

Keeps k1

kid	cageno
1	1
1	2
2	1
3	2
3	4
2	3
4	5

animals

Name	cageno
Mike	1
Sam	2
Sally	1
Barry	3
Turdy	4
Mork	5
Ollie	5

```
WITH recursive sick_keepers(kid) as (
   SELECT kid as sick_id
   FROM keeps k
   JOIN animals a on a.cageno = k.cageno
   WHERE animals.name = 'Mike'
UNION
   SELECT k2.kid as sick_id
   FROM sick_keepers
   JOIN keeps k1 on k1.kid = sick_id
   JOIN keeps k2 on k2.cageno = k1.cageno
)
```

t0

Sick_id
1
2

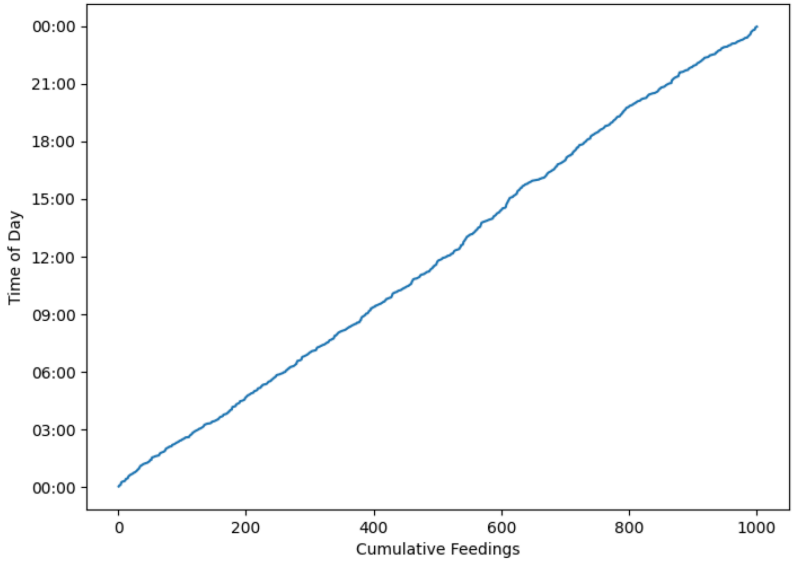
t1

Sick_id
1
2
3

t2

Sick_id
1
2
3

Window Functions



- Suppose I want to compute a CDF of animal feedtimes
- Consider a table like:

  times (hour int, minute int, animalid int)

- Tricky to do this in regular SQL; idea:
  - Sort by hour, minute
  - For each row X, select the number of rows with hour <= X.hour and minute <= X.minute

```
SELECT hour, minute,
  (SELECT count(*)
   FROM times t2
   WHERE (t.hour = t2.hour AND t. minute >= t2.minute)
     OR (t.hour > t2.hour))
FROM times t
ORDER BY hour, minute
```

Correlated subexpression: references outer table, evaluated once per outer table row!

  - What if we want to partition this and get a CDF for each animal separately?
  - What if we want the 7 day moving average of feedtimes?
- Generally a pain to work with ordered data in SQL….

Window Functions

- Suppose I want to create a table with a running sum over the number of animals per cage

CageID	Animal_Count	Running_Sum
1	2	2
2	1	3
3	1	4
4	1	5
5	2	7

Window Functions

times (hour int, minute int, animalid int)

hour	minute
4	30
1	15
2	00

Compute the value of window_func
for each row of each partition

```
SELECT x, y, ..., window_func(params)
    OVER (PARTITION BY alist1 ORDER BY alist2)
```

Split the rows into
partitions by alist1

Within each partition
order rows by alist2

Example:

```
SELECT hour, minute, RANK() OVER (ORDER BY hour, minute) FROM times
```

Compute the rank of each row

hour	minute	
1	15	1
2	00	2
4	30	3

Window Functions

times (hour int, minute int, animalid int)

hour	min	animalid
4	30	1
1	15	2
2	00	2
3	10	1
5	00	2
1	30	1

Example:

SELECT animalid hour, minute, RANK()

OVER (PARTITION BY animalid ORDER BY hour, minute) FROM times

animal	hour	minute
1	4	30
1	2	00
1	3	10

animal	hour	minute
2	1	15
2	2	00
2	5	00

animal	hour	minute	
1	2	00	1
1	3	10	2
1	4	30	3

animal	hour	minute	
2	1	15	1
2	2	00	2
2	5	00	3

Split by animal, compute the rank of each row

Other Window Functions

- `cume_dist()` : cumulative position of the row (between 0 and 1) in total ordering

- `lag(value, offset):` return the value for the record offset records before this one

- `sum() / count() / avg()` : sum / count / average of all rows in partition
  - For these expressions, OVER clause can include a *frame* that defines the subset of the partition to be included (Example on next slide)

Examples

Times with feed quantities

times

hour	min	qty
4	30	10
1	15	20
2	00	30
3	10	40

SELECT hour, min, cume_dist()
OVER (ORDER BY hour, min) as c FROM times

hour	min	c
1	15	0.25
2	0	0.5
3	10	0.75
4	30	1

SELECT hour, min, qty, lag(qty,1)
OVER (ORDER BY hour, min) as lag FROM times

hour	min	qty	lag
1	15	20	
2	0	30	20
3	10	40	30
4	30	10	40

SELECT hour, min, avg(qty)
OVER (ORDER BY hour, min
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
AS rolling_avg FROM times

"Frame"

hour	min	rolling_avg
1	15	20
2	0	25
3	10	30
4	30	26.67

Study Break

- Write a SQL query with window function to compute the difference between sales a week ago and today

Sales Table

	Date	Sales
	1/1/2022	5540
	…	
+1460	1/8/2022	7000
	…	
+2000	1/15/2022	9000

Assume 1 row per day

Functions

- `rank()`: rank of items in ordering
- `cume_dist()`: cumulative position of the row (between 0 and 1) in total ordering
- `lag(value, offset)`: return the value for the record offset records before this one
- `sum() / count() / avg()`: sum / count / average of all rows in partition

Queries

```
SELECT hour, min, cume_dist()
OVER (ORDER BY hour, min) as c FROM times
```

```
SELECT hour, min, avg(qty)
OVER (ORDER BY hour, min
ROWS BETWEEN 2 PRECEDING
AND CURRENT ROW) AS rolling_avg
FROM times
```

```
SELECT hour, min, qty, lag(qty,1)
OVER (ORDER BY hour, min) as lag FROM times
```

Soln

- Write a SQL query with window function to compute the difference between sales a week ago and today

	Date	Sales
	1/1/2022	5540
	…	
+1460	1/8/2022	7000
	…	
+2000	1/15/2022	9000

```
SELECT date, sales, sales - lag(sales,7)
OVER (ORDER BY date) difference FROM sales
```