Extended algebra

- Why extended algebra?
- Original algebra from Codd's paper is set-based
- Sets don't exist on a computer... We only have lists
- So we shall also consider an *extended algebra* in which we have *bags*, also called *multisets*. These are "sets" with repetitions
- We shall ignore the fact that relations are really ordered
- Set: $\{a, b, c\}$. Bag: $\{\{a, b, b, c, c, c\}\}$.
- We shall mostly use the notation $\{(a : 1), (b : 2), (c : 3)\}$
- Note that $\{(a : 1), (b : 0)\} = \{(a : 1)\}$

Extended algebra: Operations

- Bag-based union, intersection, difference
- Bag-based selection, projection, cartesian product,join
- New operators: δ (duplicate elimnation), τ (order)
- γ (grouping)
- Extended projection and outerjoin (could have been defined on sets as well)

Bag-based union

- $R \cup_B S$: Same rules on schemas as for set-based union
- If $(t : n) \in R$ and $(t : m) \in S$, then $(t : n + m) \in R \cup_B S$
- Rule also applies when n or m is 0
- $\{(t : 2), (s : 1)\} \cup_B \{(t : 3), (u : 2)\} = \{(t : 5), (s : 1), (u : 2)\}$
- Equivalently $\{\{t, t, s\}\} \cup_B \{\{t, t, t, u, u\}\} = \{\{t, t, t, t, t, s, u, u\}\}$.
- Note that bag union is much more efficient to implement than set-based union

Bag-based intersection and difference

Intersection:

- If $(t : n) \in R$ and $(t : m) \in S$, then $(t : \min(n, m)) \in R \cap_B S$
- $\{(t : 2), (s : 1)\} \cap_B \{(t : 3), (u : 2)\} = \{(t : 2)\}$
- Equivalently $\{\{t, t, s\}\} \cup_B \{\{t, t, t, u, u\}\} = \{\{t, t\}\}$.

Difference:

- If $(t : n) \in R$ and $(t : m) \in S$, then $(t : \max(0, (n - m)) \in R -_B S$
- $\{(t : 2), (s : 1)\} -_B \{(t : 3), (u : 2)\} = \{(s : 1)\}$
- Equivalently $\{\{t, t, s\}\} -_B \{\{t, t, t, u, u\}\} = \{\{s\}\}$.
- $\{(t : 3), (u : 2)\} -_B \{(t : 2), (s : 1)\} = \{(t : 1), (u : 2))\}$
- Equivalently $\{\{t, t, t, u, u\}\} -_B \{\{t, t, s\}\} = \{\{t, u, u\}\}$.

If $(t, n) \in R$ and $(s, m) \in S$ then $(t \times s, nm) \in R \times S$

$R(A, B)$:

A	B
1	1
1	1
1	4
2	3

$S(C, D)$:

C	D
1	5
1	5
4	7

$R \times S$:

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

Selection and projection

Selection and projection preserve the multiplicity of the original relations

$R(A, B)$:

A	B
1	1
1	1
1	4
2	3

$\sigma_{A=1}(R)$:

A	B
1	1
1	1
1	4

$\pi_A(R)$:

A
1
1
1
2

Natural join

Natural join is defined using bag-based $\times$, π, and σ

$R(A, B)$:

A	B
1	1
1	1
1	4
2	3

$S(B, C)$:

B	C
1	5
1	5
3	7

$R \bowtie S$:

A	B	C
1	1	5
1	1	5
1	1	5
1	1	5
2	3	7

New operators: δ

- δ: Remove duplicates
- Note that result is a set, not a bag

$R(A, B)$:

A	B
1	1
1	1
1	4
2	3

$\delta(R)$:

A	B
1	1
1	4
2	3

τ: ordering

τ: orders the relation. Result is a list, so that technically new definitions are needed

$R(A, B, C)$:

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

$\tau_A(R)$:

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

$\tau_{A,B}(R)$:

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

Extended projection

$\pi_L(R)$: L can now include computation as well as choosing components of tupes

L can include:

- Attributes of R
- $x \to y$ where x and y are attributes, i.e., renaming
- $E \to x$, where x is an expression using permitted operations over the domains (for example, arithmetic for integers)

Examples of extended projection

$R(A, B, C)$:

A	B	C
0	1	2
0	1	2
3	4	5

$\pi_{A,B+C\to X}(R)$:

A	X
0	3
0	3
3	9

$\pi_{B-A\to X,C-B\to Y}(R)$:

X	Y
1	1
1	1
1	1

Outerjoin

Motivation: In a join of two relations, we can only partly recover the original relation.

Some tuples are "dangling" – don't match any tuple in the other relation.

$R(A, B, C)$:

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

$S(B, C, D)$:

B	C	D
2	3	10
2	3	11
6	7	12

$R \bowtie S$:

A	B	C	D
1	2	3	10
1	2	3	11

Tuples $(4, 5, 6)$ and $(7, 8, 9)$ are "dangling", so missing in the join. Similarly $(6, 7, 12)$ in S

Outerjoin

Solution: Add the missing tuples with "nulls" for the missing values

We shall study "null values" in detail in SQL

$R \bowtie S$:

A	B	C	D
1	2	3	10
1	2	3	11

$R \overset{\text{O}}{\bowtie} S$:

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	NULL
7	8	9	NULL
NULL	6	7	12

Very useful for some queries

Left and right outerjoin

Simliar to outerjoin, but only preserve tuples from S (relation on the right) or R (relation on the left)

$R \overset{\text{O}}{\bowtie}_R S$:

A	B	C	D
1	2	3	10
1	2	3	11
NULL	6	7	12

$R \overset{\text{O}}{\bowtie}_L S$:

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	NULL
7	8	9	NULL

Grouping

- Most important new operator, and the most complicated
- The same complications are needed for the analogous part of SQL
- Used for *aggregation*: Compute average and sum, for example
- Important to know when to remove duplicates and when not. Sum of $\{1, 2, 3\}$ is different from sum of $\{\{1, 1, 2, 3\}\}$.

Why grouping

- Consider the relation $\mathrm{Student}(\mathrm{name}, \mathrm{course}, \mathrm{grade})$ ($R(n, c, g)$)
- we want to compute a relation $\mathrm{AvgGrade}(\mathrm{name}, \mathrm{avgGrade})$ ($S(n, a)$)

$R(n, c, g)$:

n	c	g
a	10	18
a	11	30
a	12	30
b	10	22
b	11	22

Result should be

$S(n, a)$:

n	a
a	26
b	22

To evaluate: Take each value of n (or "group-by" n), find the *bag* of corresponding values of g, ignoring c, and take the average of this bag

The group by operator, γ

$\gamma_L(R)$. L contains

- Attributes of R, to be grouped by (n above)
- Aggregation operators to be defined to other attributes
- A name for the column for each aggregate

Example: $\gamma_{n,\texttt{AVG}(g)\rightarrow a}(R)$

Another example

`Movies(title,year,length,genre,studioName,producerC#)`

For each studio, find the average length of all movies produced by this studio

$\gamma_{\texttt{studioName},\texttt{AVG}(\texttt{length})\rightarrow\texttt{avgLength}}(\texttt{Movies})$

Aggregate operators

- `SUM`: Sum of values
- `AVG`
- `MAX` and `MIN`: Smallest and largest values in a column
- `COUNT`: number of values, including duplicates

Another example

`StarsIn(title,year,starName)`

For each star, who has appeared in at least three movies, find the earliest year they appeared.

First, construct a relation with attributes: Starname, first year they appeared, and number of movies

$R := \gamma_{\text{starName},\texttt{MIN}(\text{year})\rightarrow\text{minYear},\texttt{Count}(\text{title})\rightarrow\text{ctTitle}}(\text{StarsIn})$

Then, select those with at least three movies and project onto year

$S := \pi_{\text{year}}\sigma_{\text{ctTitle}>=3}(R)$

SQL

- SQL: Declarative language: Say what we want, not how to compute it
- Standards: ANSI SQL, updated as SQL-92/SQL2
- SQL-99: Adds object-relational features
- Other extensions later

Basic form

Schema `R(A,B,C)`

Query `Q`

```
SELECT A,B
FROM R
WHERE C=5;
```

SELECT

- SELECT says which attributes we are interested in
- In the relation model, everything is a relation
- Data are in relations, and so are results of queries
- Every relation has a schema, and the attributes of the SELECT clause are the schema of the resuly
- So Q creates a relation with schema (A, B)

`FROM` and `WHERE`

- `FROM` clause: Which relation(s) the attributes come from
- `WHERE`: Select clause

Q corresponds to

$$\pi_{A,B}\sigma_{C=5}(R)$$

Another example

`Movies(title,year,length,genre,studioName,producerC#)`

Query: Find the movies produced by Disney Studios in 1990

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

New features

- Quotes for string values
- `AND` for conjunction
- `SELECT *`: short for all attributes in `FROM` clause

Output:

title	year	length	genre	studioName	producerC#
Pretty Woman	1990	119	drama	Disney	999

Relational algebra in SQL

Note on capitalization: SQL allow lower or upper case. For clarity we use uppercase for keywords, names starting with upper case for relations, and names starting with lower case for attributes

To illustrate key features of SQL, we show how the basic algebraic operators can be expressed in SQL

Projection is done in the `SELECT` clause

We repeat the previous query, but project onto movie title and length

```
SELECT title, length
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Result:

title	length
Pretty Woman	119

We can also rename the attributes in the result

```
SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Result:

name	duration
Pretty Woman	119

We can also apply operations to attribute values

```
SELECT title AS name, length*.016667 AS lengthInHours
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Result:

name	lengthInHours
Pretty Woman	1.98334

Note that the following is not allowed, as the second attribute has no name

```
SELECT title AS name, length*.016667
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Comment: This will work in PostgresQL, but is not guaranteed by the standard to work in any implementation (and therefore will not work in the exam...)

Using constants

```
SELECT title AS name, length*.016667 AS length,
                                    'hrs.' AS inHours
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

Result includes:

name	length	InHours
Pretty Woman	1.98334	hrs.

Selection

- Expressions can include `=`, `<`, `<=`, `>=`, `<>` (not equals)
- Numeric values \,*+. etc.
- Strings || (concatenation)
- Comparison results in a `BOOLEAN`, `TRUE` or `FALSE` (but see `NULL` values later)
- Can be combined with `NOT`, `AND` and `OR`
- `AND` has precedence over `OR`, `NOT` has precedence over both (can use parentheses if not sure)

```
SELECT title
FROM Movies
WHERE (year > 1970 OR length<90) AND studioName = 'MGM';
```

Finds the names of all MGM movies that are

- After 1970, or
- shorter than 90 minutes

< can be used for strings, with lexicographic order

- `'a'<'c'`
- `'fodder'<'foo'`
- `'bar'<'bargain'`

Pattern Matching

`s LIKE p`

- `p` is a pattern, a string with special characters
- `%` matches any sequence of 0 or more characters
- `_` matches any single character

```
SELECT title
FROM Movies
WHERE title LIKE 'Star ____';
```

Result includes:

title
Star Wars
Star Trek

```
SELECT title
FROM Movies
WHERE title LIKE '%''s%';
```

Result includes:

title
Alice's Restaurant
Logan's Run

Dates and times

- `DATE '2012-06-12'`
- `TIME '15:11:02.5'`
- < can be used to compare dates and times

Ordering the output

`ORDER BY` with syntax like the τ operator

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
ORDER BY length,title;
```

Order first by length. If two movies have the same length, order alphabetically by title.

Attributes for `ORDER BY` don't have to be in the `SELECT` clause:

```
SELECT producerC#
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length,title;
```

Or even (what does this produce?)

```
SELECT studioName
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
ORDER BY length,title;
```

Even expressions are allowed

```
SELECT *
FROM R
ORDER BY A+B DESC;
```

`DESC` means in descending order (largest first). `ASC` is allowed, but is redundant, as it is the default

Queries over several relations

`Movies(title,year,length,genre,studioName,producerC#)`
`MovieExec(name,address,cert#,netWorth)`

The attributes `producerC#` and `cert#` identify executives. We can use them to combine the two tuples. For example, to find the name (not the number) of the producer of *Star Wars*:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

How do we know what relations the attributes `name`, `title`, `producerC#`, and `cert#` belong to?

In this example, there is no ambiguity

```
MovieStar(name,address,gender,birthDate)
MovieExec(name,address,cert#,netWorth)
```

Find pairs of Star and Exec (possibly same person) with the same address

```
SELECT name,name
FROM MovieStar,MovieExec
WHERE address = address;
```

Doesn't work: Which name and address are we referring to?

Solution is to use prefixes (as we already saw with the definition of join)

```
SELECT MovieStar.name,MovieExec.name
FROM MovieStar,MovieExec
WHERE MovieStar.address = MovieExec.address;
```

How to we make sure they are not the same person?

Permissible even when there is no ambiguity.

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

can also be written as

```
SELECT MovieExec.name
FROM Movies, MovieExec
WHERE Movies.title = 'Star Wars'
       AND Movies.producerC# = MovieExec.cert#;
```

Tuple Variables

```
MovieStar(name,address,gender,birthDate)
MovieExec(name,address,cert#,netWorth)
```

Now, lets try to find pairs of (different) Stars at the same address

```
SELECT MovieStar.name, MovieStar.name
FROM MovieStar, MovieStar
WHERE MovieStar.address = MovieStar.address
     AND MovieStar.name <> MovieStar.name;
```

Even with the relation name, we still have ambiguity. Solution is to use *tuple variables* `Star1` and `Star2`

Tuple variables are variables that range over all the tuples in a relation

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
     AND Star1.name <> Star2.name;
```

Result includes pairs like

Star1.name	Star2.name
Paul Newman	Joanne Woodward
Joanne Woodward	Paul Newman

How to get only one of these?

We could require that the first precede the second in lexicographic order

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
     AND Star1.name < Star2.name;
```

Interpretation of multirelation queries: Nested loops

```
FOR each tuple Star1 in MovieStar
  FOR each tuple Star2 in MovieStar
    Check if the condition is satisfied
    If so, return the attributes in the FROM clause
```

Intersection

```
MovieStar(name,address,gender,birthDate)
MovieExec(name,address,cert#,netWorth)
```

Query: Names and addresses of all female movie stars who are also movie executives with netWorth over 10.000.000

First, find female movie stars

```
SELECT name, address
FROM MovieStar
WHERE gender = 'F';
```

Next, find movie executives with netWorth over 10.000.000

```
SELECT name, address
FROM MovieExec
WHERE netWorth > 10000000;
```

Then take the intersection (note that the schemas are the same)

```
(SELECT name, address
FROM MovieStar
WHERE gender = 'F')
    INTERSECT
(SELECT name, address
FROM MovieExec
WHERE netWorth > 10000000);
```