Queries: Interlude on Extensions

- We just saw that some natural queries are not expressible in RC or RA

  Such an example was the Transitive Closure

- Sometimes SQL allows to express queries that are not expressible in RC or RA

  This is the case of the TC (as an SQL view, coming ...)

- We will analyze some of those query extensions in the light of RC and RA

  Coming back to SQL after that ...

- We will (re)visit:

  1. Compositionality
  2. Safe queries
  3. Duplicates
  4. Aggregate queries

- Compositionality:
- Query evaluation (QE) is compositional
- The truth value of a query on a DB depends on the truth values of its components (sub-queries)

  This is used to provide a semantics to QE, and to design/use compositional algorithms for QE

- Example: Names and manufacturer of beers Fred likes (see page 58)

  `Beers(name, manf)`   `Likes(drinker, beer)`

- In RC: $\mathcal{Q}(x,y)\colon\ (\underbrace{\mathit{Beers}(x,y)}_{\mathcal{Q}_1(x,y)} \wedge \underbrace{\mathit{Likes}(\text{fred},x)}_{\mathcal{Q}_2(x)})$
- For an instance D, $\langle a, b\rangle$ is an answer to $\mathcal{Q}$ in D iff
  - $\langle a, b\rangle$ is an answer to $\mathcal{Q}_1$ in D, and
  - $\langle a\rangle$ is an answer to $\mathcal{Q}_2$ in D
- Similarly for the whole of RC and SQL (based on Predicate Logic)

- Safe Queries:
- Me mentioned that in RA there is only Difference ($\smallsetminus$)
  As a relative complement; relative to another given relation
  Or relative to a virtual relation already defined by a (sub)query
- There is no absolute complement

  The complement of a relation is considered to be meaningless
  One would have to peek inside the infinite and unspecified data domain
- Difference takes the form of `EXCEPT` in SQL
- For RC to be in line with RA, consider only safe queries
  Safe with respect to the use of negation ($\neg$)

  It is with negation that we capture RA's Difference:

  $\mathcal{Q}(x, y):\ \mathit{Beers1}(x, y) \wedge \neg \mathit{Beers2}(x, y)$

  This is perfectly fine ...

- However, this is not fine

  $\mathcal{Q}_1(x, y)$: $\neg Beers(x, y)$

  "beeers (or anything) that are not in table Beers"

- No way to know without looking outside the table ...

  Not a safe query, not part of RC

  (but O.K. as a formula of Predicate Logic)

  Not expressible or acceptable in SQL either

- Another example: $\mathcal{Q}_2(y)$: $\forall x (Beers(x, y))$

  "manufactures that make all beers (or anything)"

  We would have to go outside the table to know what are "all"

  Not safe, not part of RC, not acceptable in SQL

- Notice the hidden negation: $\forall x Beers(x, y) \equiv \neg\exists\neg Beers(x, y)$

- There is a syntactic characterization of safe queries

  Just be alert and careful ...

- Duplicates:
- Example:

WINE	W#	GRAPE	VINTAGE	PERCENTAGE	QUALITY
	100	Volnay	1979	12.7	Good
	110	Chablis	1980	11.8	Average
	120	Tokay	1981	12.1	Excellent
	130	Chenas	1979	12.0	Good
	140	Volnay	1980	11.9	Average

Projection query: $\Pi_{\text{VINTAGE,QUALITY}}$

YEAR	VINTAGE	QUALITY
	1979	Good
	1980	Average
	1981	Excellent

- Some tuples in the query answer have different origins, i.e. same projection of different tuples

  For example, $\langle$1979,Good$\rangle$ (two different "provenances")

- The set-theoretic semantics of RA and RC do not capture the "duplicates"

  The answer is the set represented as table YEAR

- We could think of representing and collecting duplicates

  In bags or multisets: (order does not matter)

  YEAR = {{ $\langle$1979,Good$\rangle$, $\langle$1979,Good$\rangle$, $\langle$1980,Average$\rangle$, $\langle$1980,Average$\rangle$, $\langle$1981,Excellent$\rangle$ }}

- Exercise: Find the multiset answer to the join of the tables through `GRAPE` plus some extra projection

WINE	GRAPE	VINTAGE	QUALITY
	Chenas	1977	Good
	Chenas	1980	Excellent
	Chablis	1977	Good
	Chablis	1978	Bad
	Volnay	1980	Average

LOCATION	GRAPE	AREA	AVG-QUALITY
	Chenas	Beaujolais	Good
	Chablis	Bourgogne	Average
	Chablis	California	Bad

- One could extend the semantics of RA and RC to deal with- and represent duplicates

- SQL does support duplicates and multiset semantics (coming)

- To "deduplicate" one could introduce tuple identifiers, as a "surrogate key"

WINE	GRAPE	VINTAGE	QUALITY
	Chenas	1977	Good
	Chenas	1980	Excellent
	Chenas	1977	Good
	Chablis	1977	Good
	Chablis	1978	Bad
	Volnay	1980	Average
	Chablis	1978	Bad

WINE	TID	GRAPE	VINTAGE	QUALITY
	1	Chenas	1977	Good
	2	Chenas	1980	Excellent
	3	Chenas	1977	Good
	4	Chablis	1977	Good
	5	Chablis	1978	Bad
	6	Volnay	1980	Average
	7	Chablis	1978	Bad

- Aggregations:

- Example: `Sells(bar,beer,price)`

bar	beer	price
Joe's	Bud	5
Sue's	Miller	6
Leo's	Duvel	8
Sue's	Duvel	6
Roe's	Miller	7

- *"Average price of a beer at Sue's Bar?"*
  "How many different beers does Leo's Bar sells?"
  "What's the maximum price for Miller?"
  "How much tasting every beer at Joe's Bar?

- Some of these queries cannot be expressed in RC or RA

- There is not counting in RC; nor arithmetic operations

- SQL does offer support for this kind of queries
  Going beyond RC and RA (coming)

Back to SQL: Extensions

- Multi-Set Semantics:
- A relation constructed with SQL is not really a set, but a *bag*
- A bag or multiset may contain a tuple more than once
  The tuples in it have no specific order (not a list)
- Example: $\{1; 2; 1; 3\}$ (or $\{\{1, 2, 1, 3\}\}$) is a bag, not a set
- BAG UNION: The number of times an element appears has to be considered
- Example: $\{1; 2; 1\} \cup \{1; 2; 3\} = \{1; 1; 1; 2; 2; 3\}$
- BAG INTERSECTION: Considers the smallest number of times the element appears in each bag
- Example: $\{1; 2; 1\} \cap \{1; 2; 3\} = \{1; 2\}$

- BAG DIFFERENCE: Subtract the number of times the element appears in each bag
- Example: $\{1;2;1\} \setminus \{1;2;3\} = \{1\}$
  $\{1;2;3\} \setminus \{1;2;1\} = \{3\}$
- Algebraic laws for bags differ from those for sets
  Some are shared: Commutative and Associativity
- Distributivity is not:

  Example: $R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$ true for sets

  Not for bags: $R = S = T = \{1\}$

  $S \cup T = \{1;1\}$ $\quad R \cap (S \cup T) = \{1\}$

  $R \cap S = R \cap T = \{1\}$

  $(R \cap S) \cup (R \cap T) = \{1;1\} \neq \{1\}$

- SQL Enforcing Set/Bag Semantics:
- Default is bag for `select-from-where`

  It takes time to find repeated answers (tuples)
- Set semantics enforced with: `DISTINCT` after `SELECT`

  If it is worth doing ...
- Example: Find the different prices charged for beers

  Sells(bar, beer, price)

```
SELECT DISTINCT price
FROM   Sells;
```

- Default is set for Union, Intersection, and Difference

  Bag semantics enforced with: `ALL`

  After `UNION`, `INTERSECT` or `EXCEPT`

- Example: (bank example, see page 62)

  The same queries using set operators

1. Find all customers who have both an account and a loan at the bank

```
(SELECT customer-name FROM Depositor)
        INTERSECT
(SELECT customer-name FROM Borrower);
```

   No duplicates

2. Find all customers who have a loan but not an account at the bank

```
(SELECT customer-name FROM Borrower)
        EXCEPT
(SELECT customer-name FROM Depositor) ;
```

   No duplicates

   The relational difference operator provides safe negation

- SQL Aggregation:

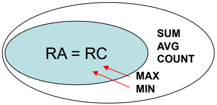
- Aggregation operators do not belong to RC or RA

  They extend them

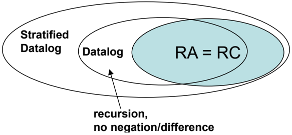
- We find: SUM, AVG, MIN, MAX, COUNT

  They apply to attributes (columns)

- However, MIN, MAX can be defined using RA/RC (cf. page 61)



Coming:



- COUNT(⋆) also applies to tuples, including duplicates

- Use them in lists following SELECT

- Example: Find the average price of Bud

Sells(bar, beer, price)

```
SELECT AVG(price)
FROM   Sells
WHERE  beer = 'Bud';
```

- Every tuple that sells Bud is considered
- If there are duplicates, the same bar will be considered more than once
- Duplicates can be eliminated before aggregation

Example: Number of different prices at which Bud is sold

Sells(bar, beer, price)

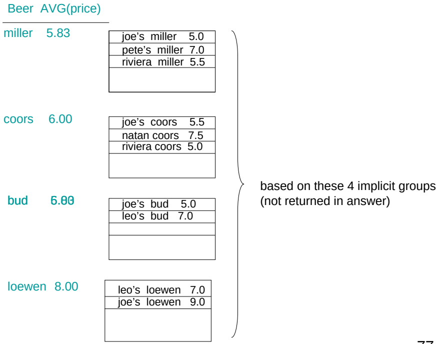
```
SELECT COUNT(DISTINCT price)
FROM   Sells
WHERE  beer = 'Bud';
```

- DISTINCT can be used with any aggregation, but it is generally used with COUNT

- Grouping:
- GROUP BY follows "select-from-where" with a list of attributes
- The relation resulting with the FROM and WHERE clauses is grouped according to the values of those attributes
- Aggregations take place only within each group
- Example: Find the average sales price for each beer

```
Sells(bar, beer, price)

SELECT beer, AVG(price)
FROM   Sells
GROUP BY beer;
```



- Example: Find, for each drinker, the average price of Bud at the bars he/she frequents

```
Sells(bar, beer, price)    Frequents(drinker, bar)
```

```
SELECT   drinker, AVG(price)
FROM     Frequents, Sells
WHERE    beer = 'Bud' AND Frequents.bar = Sells.bar
GROUP BY drinker;
```

- Notice the grouping occurring after applying the join (`Frequents.bar = Sells.bar`) and selecting (`beer = 'Bud'`)

- When rows (tuples) are grouped, one line of output is produced for each group

- Query above without the `GROUP BY` would have unclear semantics

  Average price per drinker or overall?

- Restriction on SELECT Lists With Aggregation:

  When aggregation is used, each element of a SELECT clause must either be aggregated or appear in a group-by clause (if a WHERE clause is present)

- The previous example is fine:

```
SELECT   drinker, AVG(price)
FROM     Frequents, Sells
WHERE    beer = 'Bud' AND Frequents.bar = Sells.bar
GROUP BY drinker;
```

  SELECT clause has two attributes: drinker and price

  The former is in the GROUP BY, and the latter in AVG

- What about finding the bar that sells Bud the cheapest?

  Using the MIN aggregate function

- Example: Find the bar that sells Bud the cheapest

  `Sells(bar, beer, price)`

- What about this?

```
SELECT bar, MIN(price)
FROM   Sells
WHERE  beer = 'Bud';
```

- It is illegal in most SQL implementations
- Compare with aggregations on page 76
- Alternative:

```
SELECT bar
FROM   Sells
WHERE  beer = 'Bud' AND price =(
       SELECT MIN(price)
       FROM   SELLS
       WHERE  beer = 'Bud')
```

- Exercise: Express this query without using `MIN` or `MAX`

- HAVING Clauses:
- HAVING clauses are selections on groups
  Just as WHERE clauses are selections on tuples
- The HAVING condition deletes groups
  Those for which the condition after the HAVING is false
- Condition in HAVING can use the tuple variables (alias for a relation) or relations in the FROM and their attributes
  Just like in the WHERE clause
  But the tuple variables range only over the group
- The same applies to aggregations
  They can be used in the condition
  But they apply to the group
- Condition in HAVING applies locally to each group

- Example: Find the average price of those beers that are either served in at least 3 bars or manufactured by Anheuser-Busch

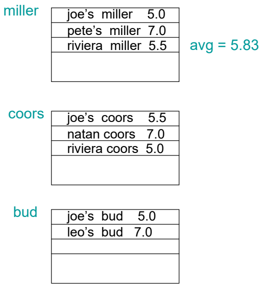
```
Beers(name, manf)    Sells(bar, beer, price)
```

```
SELECT          beer, AVG(price)
FROM            Sells
GROUP BY        beer
HAVING COUNT(*) >= 3 OR beer IN (
                SELECT name
                FROM   Beers
                WHERE  manf = 'Anheuser-Busch' );
```

- `COUNT(*)` counts the whole tuple (within a group)

  The `AVG` aggregation applies to a single attribute (column)

  (Bud is made by AB)



- With condition "`HAVING COUNT(*) >= 3`" only, last group would be ignored