SQL-99

SQL = Structured Query Language (pronounced “sequel”).

An ANSI/ISO standard language for querying and manipulating relational DBMSs.

Developed at IBM (San Jose Lab) during the 1970’s, and standardised during the 1980’s.

Appears that SQL will database systems.

Designed to be a “human readable” language

- relational algebra operations
- aggregation operations

Sample Database

To illustrate the features of SQL, we use a small example database below:

Beers(*name*, manf), Bars(*name*, addr, license)

Drinkers(*name*, ad r)

Sells(*bar, beer*, price), Frequents(*dri*

keys are in *italic* font and highlighted by underscore.

Sample Database(cont)

Bars:

Name	Addr	License
Australia Hotel	The Rocks	123456
Coogee Bay Hotel	Coogee	966500
Lord Nelson	The Rocks	123888
Marble Bar	Sydney	122123
Regent H		4
Royal Ho		0

Drinkers:

Name	Addr	
Adam	Randwick	9385-4444
Gernot	Newtown	9415-3378
John	Clovelly	9665-1234
Justin	Mosman	9845-4321

Sample Database(cont)

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melb	
New	
Old	Tooh
Old Admiral	Lord
Pale Ale	Sierra
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

Sample Database(cont)

Frequents:

Drinker	Bar
Adam	Coogee Bay Hotel
Gernot	Lord Nelson
John	Co
John	Lor
John	Australia Hotel
Justin	Regent Hotel
Justin	Marble Bar

Likes:

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
Adam	New
	Premium Lager
	Sparkling Ale
Jan	[illegible]
John	Bigfoot Barley Wine
John	Pale Ale
John	Three Sheets
Justin	Sparkling Ale
Justin	Victoria Bitter

Sample Database(cont)

Sells:

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	2.5
Coogee Bay Hotel	Sparkling Ale	2.8
Coogee Bay Hotel	Victoria Bitter	2.3
Lord Nel		
Lord Nel		
Marble Bar	New	
Marble Bar	Old	
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Example:

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	Toohey's
Old	Toohey's
Old A	
Pale	
Prem	
Red	Tooh
Sheaf Stout	Tooh
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

SQL Queries: What beers are made by Toohey's?"

SELECT Name FROM Beers WHERE Manf = 'Toohey"s';

SQL Queries

To answer the question “What beers are made by Toohey’s?”, we could ask:

SELECT Name FROM Beers WHERE Manf = ‘Toohey”s’;

This gives a subset of the Beers relation, displayed as:


```
Name
--------------
New
Old
Red
Sheaf Stout
```

Quotes are escaped by doubling them (‘ ‘)

SQL Queries(cont)

Query syntax is:

SELECT attributes

FROM relations

WHERE condition

The result of this s pically displayed on output.

The SELECT statement contains the f *select*, *project* and *join* from the relational algebra.

SQL Identifiers

Names are used to identify objects such as tables, attributes, views, ...

Identifiers in SQL use similar conventions to common programming languages:

- a sequence alphabetic,
- not case-sensitive,
- reserve word disallowed, ...

SQL Keywords

Some of the frequently-used ones:

- ALTER AND CREATE
- FROM INSERT NOT OR
- SELECT TABLE WHERE

SQL Data Types

All attributes in SQL relations have domain specified.

SQL supports a small set of useful built-in data types: strings, numbers, dates, bit-strings.

Self defined data t

Various type conversions are availabl

- date to string, string to date, integer to
- applied automatically “where they make sense”

SQL Data Types(cont.)

Basic domain (type) checking is performed automatically.

Constraints can be used to “enforce” more complex domain membership conditions.

The NULL value is

SQL Data Types(cont.)

Comparison operators are defined on all types.

< > <= >= = !=

Boolean operators AND, OR, NOT are available within WHERE expressions to combine results of comparisons.

Comparison against NU

Can explicitly test for NULL using:

- ***attr* IS NULL** ***attr* IS NOT NU**

Most data types also have type-specific operations available (e.g. arithmetic for numbers).

Which operations are actually applied depends on the implementation.

SQL Strings

Two kinds of string are available:

- CHAR(*n*) ... uses *n* bytes, left-justified, blank-padded
- VARCHAR(*n*) ... uses *0..n* bytes, no padding

String types can be uncation.

String literals are written using single

- ‘John’ = "John" = "John " != "JOHN"

String comparison

$str_1 < str_2$ ... compare using dictionary order

str LIKE *pattern* ... matches string to pattern

Two kinds of pattern matching:

- % matches anythin
- _ matches any singl

Examples:

◦ Name LIKE ‘Ja%’	Name begins with ‘Ja’
◦ Name LIKE ‘_i%’	Name has ‘i’ as 2nd letter
◦ Name LIKE ‘%o%o%’	Name contains two ‘o’s

String manipulation

string || *string* … concatenate two strings

- ‘Post’|| ‘greSQL’ -> PostgreSQL

LENGTH(*str*) … return length of string

SUBSTR(*str,start,l* … trim string

- substring('Thomas' from 2 for 3) -> hom

SQL Dates

Dates are simply specially-formatted strings, with a range of operations to implement date semantics.

Format is typically DD-Mon-YYYY, e.g. ’18-Aug-1998’

Accepts other formats

Comparison operators

(start1, end1) OVERLAPS (start2, end2)

- This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap.
- SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS (DATE '2001-10-30', DATE '2002-10-30'); -> *Result:* true

SQL Numbers

Various kinds of numbers are available:

smallint, int, bigint ... 2-bytes, 4-bytes and 8-bytes integers

real, double precision... 4-bytes and 8-bytes floating point

numeric(precision,

- The *scale* of a numeric is the count of the fractional part, to the right of the decimal point.
- The *precision* of a numeric is the total count of significant digits in the whole number

SQL Numbers(cont.)

Arithmetic operations:

- + - * / abs ceil floor power sqrt sin …

Some operations apply to a column of numbers in a relation:

- AVG(*attr*) ... mean
- COUNT(*attr*) ... nu
- MIN/MAX(*attr*) ... min/max of values for
- SUM(*attr*) ... sum of values for *attr*

Note: NULL value produces NULL result for arithmetic operation, but NULL is ignored in column operations.

Tuple and Set Literals

Tuple and set constants are both written as:

- (val1, val2, val3, ...)

The correct interpretation is worked out from the context.

Examples:

```
Student(stude#, name, course)
( 2177364, 'Jack Smith', 'BSc')      -- t

SELECT name
FROM Employees
WHERE job IN ('Lecturer', 'Tutor', 'Professor');        -- set literal
```

Querying a Single Relation

Formal semantics (relational algebra):

- start with relation *R* in FROM clause
- apply σ using Condition in WHERE clause
- apply π using Attributes in SELECT clause

SELECT *Attrib*
FROM *R*
WHERE *Conditions*

Querying a Single Relation(cont.)

Operationally, we think in terms of a *tuple variable* ranging over all tuples of the relation.

Operational semantics:


```
FOR EACH tuple
    check wheth                                  HERE clause
    IF it does THEN
            print the attributes of T
            specified in the SELECT clause
    END
END
```

Projection by SQL

Assume a relation R and attributes $X \subseteq R$.

$\pi_X (R)$ is implemented in SQL as:

- SELECT X FROM R

Example:

Names of drinkers:

- SELECT Name FROM Drinkers;


```
Name
--------------------
Adam
Gernot
John
Justin
```

Drinkers:

Name	Addr	Phone
Adam	Randwick	9385-4444
Gernot	Newtown	9415-3378
John	Clovelly	9665-1234
Justin	Mosman	9845-4321

Projection by SQL(cont.)

Example:

Names and addresses of drinkers = $\pi_{Name,Addr}(Drinkers)$

- SELECT Name, Addr FROM Drinkers;

NAME	A
Adam	Randwick
Gernot	Newtown
John	Clovelly
Justin	Mosman

Projection by SQL(cont.)

The symbol * denotes a list of all attributes.

Example:

All informati

- SELECT * F

NAME	ADDR	PH
Adam	Randwick	9385-4444
Gernot	Newtown	9415-3378
John	Clovelly	9665-1234
Justin	Mosman	9845-4321

Selection by SQL

$\sigma_{Cond}(Rel)$ is implemented in SQL as:

SELECT * FROM Rel WHERE Cond

Example: Find the price that **Regent Hotel** charges for **New**

```
SELECT price
FROM Sells
WHERE bar = 'Regent Hotel'
```

```
PRICE
----------
2.2
```

The condition can be an arbitrarily complex boolean-valued expression using the operators mentioned previously.

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	2.5
Coogee Bay Hotel	Sparkling Ale	2.8
ay	Victoria Bitter	2.3
on	Three Sheets	3.75
	Old Admiral	3.75
	New	2.8
Marble Bar	Old	2.8
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Selection by SQL$_{(cont.)}$

The "typical" SELECT query:

```
SELECT a1, a2, a3
FROM Rel
WHERE Cond
```


This correspo by project:

$\pi_{\{a1,a2,a3\}}(\sigma_{Cond}(Rel))$.

Renaming via as

Ullman/Widom define a renaming operator ρ to avoid name clashes.

For example, *Address* field in *Academic* and *Student*.

Example: $\rho_{Beers(Br}$

Gives a new relation, with same da ut with attribute names changed.

SQL provides *AS* to achieve this; it is used in the SELECT part.

Renaming via as(cont.)

Example:

- Beers(name, manf)

SELECT name AS Brand, manf AS Brewer FROM Beers;

BRAND	BREWER
80/-	
Bigfoot Barley W	
Burragorang Bock	Geo
Crown Lager	Carl
Fosters Lager	Carlton
Invalid Stout	Carlton
…	

Expressions as Values in Columns

AS can also be used to introduce computed values

Example:

- Sells(bar, beer, price)

```
SELECT bar, beer, price*120 AS PriceInYen
FROM Sells;
```


BAR	BEER	PRICEINYEN
Australia Hotel	Burragorang	420
Coogee Bay Hotel	New	270
Coogee Bay Hotel	Old	300
Coogee Bay Hotel	Sparkling Ale	336
Coogee Bay Hotel	Victoria Bitter	276
…		

Just Display but no change to the database

Inserting Text in Result Table

Trick: to put text in output columns, use constant expression with *AS*.

Example:

Likes(drinker, beer)

```
SELECT drinker, 'lik
FROM Likes
WHERE beer = 'Sparkling Ale';
```

DRINKER	WHOLIKES
Gernot	likes Cooper's
Justin	likes Cooper's

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
Adam	New
t	Premium Lager
	Sparkling Ale
	80/-
John	Bigfoot Barley Wine
John	Pale Ale
John	Three Sheets
Justin	Sparkling Ale
Justin	Victoria Bitter

Find the brewers whose beers John likes.

```
SELECT Manf
FROM Likes, Beers
WHERE drinker = 'John' AND beer = name;
```

Likes:

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
Adam	New
Gernot	Pre
Gernot	Sp
John	80/-
John	Bigfoot Barley Wine
John	Pale Ale
John	Three Sheets
Justin	Sparkling Ale
Justin	Victoria Bitter

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
t	Carlton
itter	Carlton
	Toohey's
	Toohey's
	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

Querying Multi-relations

Example: Find the brewers whose beers John likes.

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

```
SELECT Manf
FROM Likes, Beers
WHERE drinker = 'John' AND beer = name;
```



```
MANF
-------------------
Caledonian
Sierra Nevada
Sierra Nevada
Lord Nelson
```

Note: could eliminate the duplicates by using *DISTINCT*.

Relational algebra: $\pi_{manf}(\sigma_{drinker='John'} Likes \bowtie Beers)$.

Querying Multi-relations(cont.)

Syntax:

SELECT *Attributes*

FROM *R1*, *R2*, ...

WHERE *Condition*

FROM clause contains a list of relations.

Querying Multi-relations(cont.)

For SQL *SELECT* statement on several relations:

SELECT *Attributes*

FROM *R1*, *R2*, ...

WHERE *Condition*

Formal semantics (

- start with product *R1* × *R2* × ... in FRO

- apply σ using Condition in WHERE clause
- apply π using Attributes in SELECT clause

Querying Multi-relations(cont.)

Operational semantics of *SELECT*:

```
FOR EACH tuple T1 in R1 DO
    FOR EACH tuple T2 in R2 DO
        ...
            check WHERE condition for current
            assign
            IF hol
                print attributes of T1,
                specified in SELECT
            END
        ...

END
```


For efficiency reasons, it is not implemented in this way!

Attribute Name Clashes

If a selection condition

- refers to two relations
- the relations have attributes with the same name

use the relation na

Example: Which h r?

Beers(name, manf)
Bars(name, addr, license)

```
SELECT Bars.name
FROM Bars, Beers
WHERE Bars.name = Beers.name;
```

None of them do, so the result is empty.

Attribute Name Clashes(cont.)

Can use such qualified names, even if there is no ambiguity:

```
SELECT Sells.beer
FROM Sells
WHERE Sells.price > 3.00;
```


Advice:

- qualify attribute names only when absol
- SQL's AS operator cannot be used to resolve name clashes.

Table Name Clashes

The relation-dot-attribute convention doesn't help if we use the same relation twice in SELECT.

To handle this, we need to define new names for each "instance" of the relation in the FRO

Example: Find pairs of beers by the sa rer.

Note: we should avoid:

- pairing a beer with itself e.g. (New,New)
- same pairs with different order e.g. (New,Old) (Old,New)

```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND b1.name < b2.name;
```

NAME	NAME
Crown Lager	Fosters Lager
Crown Lager	Invalid Stout
Fosters Lager	Inval
Fosters Lager	Melbourne Bitter
….	

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
t	Carlton
Bitter	Carlton
	Toohey’s
	Toohey’s
	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lager	Cascade
Red	Toohey’s
Sheaf Stout	Toohey’s
Sparkling Ale	Cooper’s
Stout	Cooper’s
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

Subqueries

The result of a SELECT-FROM-WHERE query can be used in the WHERE clause of another query.

Simplest Case: Subquery returns one tuple.

- Can treat the resul

Example: Find bars that sell New at the price same as the Coogee Bay Hotel charges for VB.

Sells:

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	2.5
Coogee Bay Hotel	Sparkling Ale	2.8
Coogee Bay Hotel	Victoria Bitter	2.3
L		3.75
L		3.75
Marble Bar	New	
Marble Bar	Old	
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Subqueries(cont.)

Example: Find bars that sell New at the price same as the Coogee Bay Hotel charges for VB.

```
SELECT bar
FROM Sells
WHERE beer = 'New'
        AND price =
                (S
                 F
                 WHERE bar = 'Coogee
                 AND beer = 'Victoria B
```



```
BAR
--------------------
Royal Hotel
```

Parentheses around the subquery are required.

NOT use subqueries

Example: Find bars that sell New at the price same as the Coogee Bay Hotel charges for VB.

```
SELECT b2.bar
FROM Sells b1, Sells b2
WHERE b1.beer = 'Victoria Bitter' and b1.bar = 'Coogee Bay Hotel' and
b1.price = b2.price

BAR
--------------------
Royal Hotel
```

Subqueries(cont.)

Complex Case: Subquery returns multiple tuples/a relation.

- Treat it as a list of values, and use the various operators on lists/sets (e.g. IN).

IN Operator

Tests whether a sp lation.

tuple IN relation: is true iff the tuple is he relation.

Conversely for *tuple* NOT IN relation.

Example: Find the name and brewers of beers that John likes.

Likes:

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
Adam	New
Gernot	Pre
Gernot	Sp
John	80/-
John	Bigfoot Barley Wine
John	Pale Ale
John	Three Sheets
Justin	Sparkling Ale
Justin	Victoria Bitter

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
t	Carlton
Bitter	Carlton
	Toohey's
	Toohey's
	Lord Nelson
Pale Ale	Sierra Nevada
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

Subqueries(cont.)

Example: Find the name and brewers of beers that John likes.

```
SELECT *
FROM Beers
WHERE name IN
        (SELECT beer
         FROM Lik
         WHERE d
         );
```

NAME	MAKE
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Pale Ale	Sierra Nevada
Three Sheets	Lord Nelson

- The subquery answers the question ”What are the names of the beers that John likes?”

this query can be answered ll witho version is potentially (but not always) less efficient.

Subqueries(cont.)

Example: Find the name and brewers of beers that John likes.

```
SELECT *
FROM Beers
WHERE name IN
        (SELECT beer
         FROM
         WHERE
         );
```

```
SELECT Beers.*
FROM Beers, Likes
Where Beers.name = Likes.beer and
rinker = 'John';
```


NAME	MANF
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Pale Ale	Sierra Nevada
Three Sheets	Lord Nelson

Example: Find the beers uniquely made by their manufacturer.

Beers:

Name	Manf
80/-	Caledonian
Bigfoot Barley Wine	Sierra Nevada
Burragorang Bock	George IV Inn
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
New	
Old	
Old Ad	
Pale Ale	Sierra N
Premium Lager	Cascade
Red	Toohey's
Sheaf Stout	Toohey's
Sparkling Ale	Cooper's
Stout	Cooper's
Three Sheets	Lord Nelson
Victoria Bitter	Carlton

EXISTS Function

EXISTS(relation) is true iff the relation is non-empty.

Example: Find the beers uniquely made by their manufacturer.

```
SELECT name
FROM Beers b1
WHERE NOT EXISTS
                (
                 F

                 AND name <> b1.name
                );
```

```
NAME
--------------------
80/-
Burragorang Bock
Premium Lager
```


A subquery that refers to values from a surrounding query is called a *correlated subquery.*

Quantifiers

ANY and ALL behave as existential and universal quantifiers respectively.

Example: Find the beers sold for the highest price.

```
SELECT beer
FROM Sells
WHERE price >=
        ALL(
            SELE
            FROM
             );
BEER
--------------------
Three Sheets
Old Admiral
```


Beware: in common use, ”any” and ”all” are often synonyms.

E.g. ”I’m better than any of you” vs. ”I’m better than all of you”.

Find the drinkers and beers such that the drinker likes the beer and frequents a bar that sells it.

Sells

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	2.5
Coogee Bay Hotel	Sparkling A	
Coogee Bay Hotel	Victoria Bit	
Lord Nelson	Three Sheets	3.75
Lord Nelson	Old Admiral	3.75
Marble Bar	New	2.8
Marble Bar	Old	2.8
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Likes

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
Adam	New
Gernot	Premium Lager
Gernot	Sparkling Ale
John	80/-
John	Bigfoot Barley Wine
	Pale Ale
	Three Sheets
	Sparkling Ale
	Victoria Bitter

Frequents

Drinker	Bar
Adam	Coogee Bay Hotel
Gernot	Lord Nelson
John	Coogee Bay Hotel
John	Lord Nelson
John	Australia Hotel
Justin	Regent Hotel
Justin	Marble Bar

Union, Intersection, Difference

R1 UNION R2: produces the union of the two relations R1 and R2.

Similarly for R1 INTERSECT R2 and R1 Except R2.

Example: Find the drinkers and beers such that the drinker likes the beer and frequents a bar that sells it.

```
(SELECT *
  FROM Likes
)
 INTERSECT
(SELECT drinker,beer
  FROM Sells, Frequents
  WHERE Frequents.bar = Sells.bar
);
```

DRINKER	BEER
Adam	New
John	Three Sheets
Justin	Victoria Bitter

Divide Operation

Find bars each of which sell all beers Justin likes.

Relational Algebra: $\pi_{bar,beer} Sells \div (\pi_{beer}(\sigma_{drinker='Justin'} Likes))$

Bar	Beer	Price
Australia Hotel	Burragorang Bock	3.5
Coogee Bay Hotel	New	2.25
Coogee Bay Hotel	Old	
Coogee Bay Hotel	Sparkling Ale	
Coogee Bay Hotel	Victoria Bitter	2.3
Lord Nelson	Three Sheets	3.75
Lord Nelson	Old Admiral	3.75
Marble Bar	New	2.8
Marble Bar	Old	2.8
Marble Bar	Victoria Bitter	2.8
Regent Hotel	New	2.2
Regent Hotel	Victoria Bitter	2.2
Royal Hotel	New	2.3
Royal Hotel	Old	2.3
Royal Hotel	Victoria Bitter	2.3

Drinker	Beer
Adam	Crown Lager
Adam	Fosters Lager
	New
	Premium Lager
	Sparkling Ale
	80/-
	Bigfoot Barley Wine
John	Pale Ale
John	Three Sheets
Justin	Sparkling Ale
Justin	Victoria Bitter

Divide Operation

Find bars each of which sell all beers Justin likes.

Relational Algebra: ~~$Sells \div (\pi_{beer}(\sigma_{drinker='Justin'} Likes))$~~

$$\pi_{bar,beer} Sells \div (\pi_{beer}(\sigma_{drinker='Justin'} Likes))$$

```
select distinct a.bar
from sells a
where not exists
      ( (select b.be
         where b.d
         except
        (select c.beer from sells c
         where c.bar = a.bar )
        );
```



```
BAR
----------------------------
Coogee Bay Hotel
```

Aggregation

Selection clauses can contain aggregation operations.

Example: What is the average price of New?

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

AVG (DISTINCT price)


```
AVG(PRICE)
----------
2.3875
```


All prices for 'New' will be included, even if two hotels sell it at the same price.

If set semantics used, the result would be wrong.

Aggregation(cont.)

If we want set semantics, we can force using DISTINCT.

Example: How many different bars sell beer?

```
SELECT COUNT(DISTINCT bar)
FROM Sells;
```



```
COUNT(DISTIN
----------
6
```


Without DISTINCT, the result is 15 .. f entries in the Sells table.

Aggregation(cont.)

The following operators apply to a list of numeric values in one column of a relation:

- SUM AVG MIN MAX COUNT

The notation COU es in a relation.

Example: How many different beers a


```
SELECT COUNT(*) FROM Beers;


COUNT(*)
----------
18
```

Grouping

SELECT-FROM-WHERE can be followed by *GROUP BY* to:

- partition result relation into groups (according to values of specified attribute)
- treat each group separately in computing aggregations

Example: How many beers does each brewer make?


```
SELECT manf
FROM Beers
GROUP BY manf;
```

	COUNT(beer)
Anheuser-Busch	1
Carlton	5
Cascade	1
Cooper's	2
George IV Inn	1
Lord Nelson	2
Sierra Nevada	2
Toohey's	4

Grouping(cont.)

GROUP BY is used as follows:

SELECT *attributes/aggregations*

FROM *relations*

WHERE *condition*

GROUP BY *attribute*

Semantics:

- partition result int [illegible] attribute
- apply any aggregation separately to eac

Grouping(cont.)

Grouping is typically used in queries involving the phrase “for each”.

Example: For each drinker, find the average price of New at the bars they frequently go to.


```
SELECT drinker,
FROM Frequents
WHERE beer = ’New’ AND Frequents
GROUP BY drinker;
```


DRINKER	AVG(PRICE)
Adam	2.25
John	2.25
Justin	2.5

Grouping(cont.)

When using grouping, every attribute in the SELECT list must:

- have an aggregation operator applied to it OR
- appear in a GROUP-BY clause

Incorrect Exampl in each bar.


```
SELECT bar, MIN(price)
```



```
FROM Sells;
```

ERROR: column "sells.bar" must appear in the GROUP BY clause or be used in an aggregate function

LINE 1: select bar, min(price) from sells;

Grouping(cont.)

How to answer the above query?

```
SELECT bar, MIN(price)
FROM Sells
GROUP BY bar;
```


bar	
Australia Hotel	3.5
Coogee Bay Hotel	2.25
Lord Nelson	3.75
Marble Bar	2.8
Regent Hotel	2.2
Royal Hotel	2.3

Eliminating Groups

In some queries, you can use the WHERE condition to eliminate groups.

Example: Average beer price by suburb excluding hotels in The Rocks.

```
SELECT Bars.addr, AVG(Sells.price)
FROM Sells, Bar
WHERE Bars.ad
AND Sells.bar =
GROUP BY Bars.addr;
```


ADDR	AVG(SELLS.PRICE)
Coogee	2.4625
Kingsford	2.2
Randwick	2.3
Sydney	2.8

Eliminating Groups(cont.)

For more complex conditions on groups, use the HAVING clause.

HAVING is used to qualify a GROUP-BY clause:

```
SELECT attributes/aggregations
FROM relations
WHERE conditio
GROUP BY attri
HAVING condition (on group);
```


Semantics of HAVING:

- generate the groups as for GROUP-BY
- eliminate any group not satisfying HAVING condition
- apply an aggregation to remaining groups

Eliminating Groups(cont.)

Example: Find the average price of popular beers (i.e. those that are served in more than one hotel).

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY be
HAVING COU
```


BEER	AVG(PRIC
New	2.3875
Old	2.53333333
Victoria Bitter	2.4

Defining a Database Schema

Relations (tables) are created using:

CREATE TABLE *RelName* (
    attribute$_1$ ˜ *domain*$_1$ ˜ *properties*
    attribute$_2$ ˜ *domain*$_2$ ˜ *properties*
    attribute$_3$ ˜ *d*
    ...
)

where properties can include details about primary keys,

foreign keys, default values, and constraints on attribute values.

Tables are removed via **DROP TABLE** *RelName*;

Defining a Database Schema(cont.)

Example:

```
CREATE TABLE Beers (
        name VARCHAR(20) PRIMARY KEY,
        manf VARCHAR(20),
);
CREATE TABL
        name VARCHAR(30) PRIMAR
        addr VARCHAR(30),
        license INTEGER
);
```

Declaring Keys

Primary keys:

- if a single attribute, declare with attribute
- if several attributes, declare at end of attribute list

For attributes which have distinct values for each tuple, can not

- *attribute dom*

Declaring Keys(cont.)

Declaring foreign keys assures referential integrity.

Foreign a key:

- specify Relation (Attribute) to which it refers.

For instance, if we rs, and there are tuples

in Sells that refer t

- **reject** the deletion
- **cascade** the deletion and remove Sells records
- **set-NULL** the foreign key attribute

Can force cascade via *ON DELETE CASCADE* after *REFERENCES*.

Other Attribute Properties

Can specify that an attribute is not allowed to be *NULL*.

This property applies automatically to *PRIMARY KEY* attributes.

Can specify a *DEFAULT* value which will be assigned if none is supplied during insert.

Example:

```
CREATE TABLE Likes (
        drinker VARCHAR(20) DEFAULT 'Joe',
        beer VARCHAR(30) DEFAULT 'New',
        PRIMARY KEY(drinker, beer)
);
```

Other Attribute Properties(cont.)

In fact, *NOT NULL* is a special case of a constraint on the value that an attribute is allowed to take.

SQL has a more general mechanism for specifying such constraints.

- *attr_name type* C

The Condition can be arbitrarily comp ven involve other attributes, relations and *SELECT* queri

Other Attribute Properties(cont.)

Example:

```
CREATE TABLE Example
(
        gender CHAR(1) CHECK (gender IN ('M','F')),
        Xvalue INT NOT NULL,
        Yvalue INT CHECK (Yvalue > Xvalue),
        Zvalue FLO                                    MAX(price)
                                                      ROM Sells))
);
```

Database Modification

Simple Insertion

Accomplished via the INSERT operation:

```
INSERT INTO Relation VALUES
    (val1, val2, val3, ...)
```


Example: Add the fact

```
INSERT INTO Likes VALUES('Justin', 101);
```

Can re-order attributes in tuple constant as long as order is specified in the INTO clause.

```
INSERT INTO Sells(price,bar,beer) VALUES
    (2.50, 'Coogee Bay Hotel', 'Pale Ale');
```

Simple Insertion

Example: insertion with insufficient values.

E.g. we specify that drinkers' phone numbers cannot be NULL.

```
ALTER TABLE Drinkers ALTER COLUMN phone SET NOT NULL;
```


And then try to ins number we don't know:


```
INSERT INTO Drinkers(name,addr)
```



```
VALUES ('Zoe', 'Manly');
```

ERROR: null value in column "phone" violates not-null constraint

DETAIL: Failing row contains (Zoe, Manly, null).

Insertion from Queries

Can use the result of a query to perform insertion of multiple tuples at once.

INSERT INTO Relation (Subquery);

Tuples of Subquery le format (i.e. matching the

tuple-type of Relati

Insertion from Queries(cont.)

Example: Create a relation of John's potential drinking buddies (i.e. people who go to the same bars as John).

```
CREATE TABLE DrinkingBuddies (
        name varchar(20)
);

INSERT INTO Dri
(
        SELECT DISTINCT f2.drinker
        FROM Frequents f1, Frequents f2
        WHERE f1.drinker = 'John'
                        AND f2.drinker != 'John'
                        AND f1.bar = f2.bar
);
```

Deletion

Accomplished via the DELETE operation:

DELETE FROM Relation

WHERE *Condition*

Removes all tuples from Relation that satisfy Condition.

Example: Justin no

DELETE FROM Likes

WHERE drinker = 'Justin'

AND beer = 'Sparkling Ale';

Special case: Make relation R empty.

DELETE FROM R;

Deletion(cont.)

Example: Delete all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS
    ( SELECT name
      FROM Beers
      WHERE ma
      AND name
```


Semantics here is subtle...

If there is a manufacturer that makes only two beers, how many of them will be deleted?

E.g. after first beer is deleted, second beer no longer satisfies condition.

In fact, condition is evaluated for each tuple before making any changes.

Deletion(cont.)

Semantics of the above Deletion:

Evaluation of DELETE FROM R WHERE Cond can be viewed as:

```
FOR EACH tuple T in R DO
    IF T satisfi
        make a
    END
END
FOR EACH noted tuple T DO
    remove T from relation R
END
```

Updates

An update allows you to modify values of specified attributes in specified tuples of a relation:

UPDATE *R*

SET *list of assignments*

WHERE *Condition*

Each tuple in relatio he assignments applied to it.

Example: John moves to Coogee.

UPDATE Drinkers

SET addr = ’Coogee’ ,

phone = ’9665-4321’

WHERE name = ’John’;

Updates(cont.)

Can update many tuples at once (all tuples that satisfy condition)

"Good" Example: Make $3 the maximum price for beer.

```
UPDATE Sells
SET price = 3.00
WHERE price > 3.00;
```


"Bad" Example: Increase beer prices by 10%.

```
UPDATE Sells
SET price = price * 1.10;
```

Changing Tables

Accomplished via the ALTER TABLE operation:

- ALTER TABLE *Relation Modifications*

Some possible modifications are:

- add a new column
- change the propert
- remove an attribute

Changing Tables(cont.)

Example: Add phone numbers for hotels.

```
ALTER TABLE Bars
ADD phone char(10) DEFAULT 'Unlisted';
```


This appends a new e for this attribute to 'Unlisted' in every t

Specific phone numbers can subsequentl

```
UPDATE Bars
SET phone = '9665-0000'
WHERE name = 'Coogee Bay Hotel';
```

If no default values is given, new column is set to all NULL.

Changing Tables(cont.)

Can make multiple changes to one relation with a single ALTER.

Example: Add opening and closing times to Bars

```
ALTER TABLE Bars
Add opens NUMERIC(4,2) DEFAULT 10.00 ,
Add closes NUMERIC(4,2) DEFAULT 23.00 ,
Add manager V
;
```


Note that manager will be initially NULL in all tuples.

Views

A **view** is like a ”virtual relation” defined in terms of other relations.

The other relations may be views (*intensional relations*) or stored relations (*extensional relations, base relations*).

View are defined v AS *Query*

The view is valid only as long as the u y is valid.

Views may be removed via: DROP VIEW *ViewName*

Removing a view has no effect on the relations used by the view.

Views(cont.)

Example: An avid CUB drinker might not be interested in any other kinds of beer.

```
CREATE VIEW MyBeers AS
    SELECT name, manf
    FROM Beers
    WHERE ma
    SELECT * F
```


NAME	MANF
Crown Lager	Carlton
Fosters Lager	Carlton
Invalid Stout	Carlton
Melbourne Bitter	Carlton
Victoria Bitter	Carlton

Views(cont.)

A view might not use all attributes of the base relations.

Example: We don't really need the address of inner-city hotels.

```
CREATE VIEW InnerCityHotels AS
    SELECT name, license
    FROM Bars
    WHERE address LIKE '%Chicago%';
    SELECT *
```


NAME	LIC
Australia Hotel	123456
Lord Nelson	123888
Marble Bar	122123

Renaming View Attributes

This can be achieved in two different ways:

```
CREATE VIEW InnerCityPubs AS
    SELECT name AS pub, license AS lic
    FROM Bars
    WHERE ad
CREATE VIEW InnerCityPubs(pub,lic
    SELECT name, license
    FROM Bars
    WHERE addr IN ('The Rocks', 'Sydney');
```

Querying Views

Views can be used in queries just as if they were stored relations.

Unlike stored relations, views can ”change” without explicit modification operations (i.e. by changing underlying relations).

Example: The Lord


```
UPDATE Bars SET license='111223'                    ord Nelson'
SELECT * FROM InnerCityHotels;
```


NAME	LICENSE
Australia Hotel	123456
Marble Bar	12212
Lord Nelson	111223

Querying Views(cont.)

We can treat views as ”macros” that will be re-written into queries on the base relation.

This is most easily seen by converting to relational algebra, and following transfor ator might make.

Example: Using the InnerCityHotels v

```
CREATE VIEW InnerCityHotels AS
    SELECT name, license
    FROM Bars
    WHERE addr IN (’The Rocks’, ’Sydney’);
SELECT pub FROM InnerCityHotels WHERE lic = ’123456’;
```

Updating Views

Under the following conditions, it makes sense to allow view updates:

- the view involves a single relation R
- the WHERE cla query
- there must be att the new tuple to be retrieved; unmentioned attributes ar

Updating Views(cont.)

Example: Our InnerCityHotel view is not updatable.

```
INSERT INTO InnerCityHotels
VALUES ('Jackson''s on George', '9876543');
```


creates a new tuple


```
('Jackson''s on George', NULL, '9876
```


when we SELECT from the view, this new tuple does not satisfy the view condition:

```
addr IN ('The Rocks', 'Sydney')
```

Updating Views(cont.)

If we had chosen to omit the license attribute instead, it would be updatable:

```
CREATE VIEW CityHotels AS
    SELECT name,addr FROM Bars
    WHERE addr IN ('The Rocks', 'Sydney');
INSERT INTO C
    VALUES ('J
    SELECT * FROM CityHotels;
```


NAME	ADDR
Australia Hotel	The Rocks
Marble Bar	Sydney
Jackson's on George	Sydney