CS411
Database Systems

06a: SQL-1
The Basics– Select-From-Where

Why Do We Learn This?

~ Why we need to sp English

SQL → data access

~ Learn to ask questions.

~ on the M.T.

Reminder

- Tutorial #3 = TODAY 4:30~5:30
  at 1302 SC

Announcement

- Midterm.
  Mar 9 (Wed) in class.
  75 minutes closed books/notes/wifi.
  Prob 1 = short answers
  2.3.4 = longer questions.
  ID

- Review Session = Mon.

SQL Introduction

Standard language for querying and manipulating data

Structured **Q**uery **L**anguage

Many standards out there: SQL92, SQL2, SQL3, SQL99
Vendors support various subsets of these, but all of what we'll be talking about.

SQL 2003
2006
2008

Why SQL?

Don Chamberlin
Ray Boyce.

1

- SQL is a very-high-level language, in which the programmer is able to avoid specifying a lot of data-manipulation details that would be necessary in languages like C++.

2

- What makes SQL viable is that its queries are "optimized" quite well, yielding efficient query executions.

- easier to write prog.
- more efficient querying.

Select-From-Where Statements

- The principal form of a query is:

choose

```
SELECT     desired attributes
FROM       one or more tables
WHERE      condition about tuples of
           the tables
```

Single-Relation Queries

Our Running Example

- Most of our SQL queries will be based on the following database schema.
  - Underline indicates key attributes.

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

✓ Sells(bar, beer, price)

Frequents(drinker, bar)

Example

- Using Beers(name, manf), what beers are made by Anheuser-Busch?

```
SELECT name
FROM Beers
WHERE manf = '~~Anheuser~~-Busch';
```

$$\pi_{name}\,\sigma_{manf='Busch'}(\text{Beers})$$

Result of Query

name
‘Bud’
‘Bud Lite’
‘Michelob’

The answer is a relation with a single attribute, name, and tuples with the name of each beer by Anheuser-Busch, such as Bud.

Meaning of Single-Relation Query

- Begin with the relation in the FROM clause.
- Apply the selection indicated by the WHERE clause.

bag

set

- Apply the extended projection indicated by the SELECT clause.

duplicate-allowed

SQL

distinct.

$$\Pi_A \sigma_c(R)$$

Operational Semantics

- To implement this algorithm think of a *tuple variable* ranging over each tuple of the relation mentioned in FROM.

t

- Check if the "current" tuple satisfies the WHERE clause.
- If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

```
For each tuple t ∈ R.
    if( check C ):
        output A. of t
```

* In SELECT clauses

- When there is one relation in the FROM clause, * in the SELECT clause stands for “all attributes of this relation.”
- Example using Beers(name, manf):

```
SELECT *
FROM Beers
WHERE manf = ‘Anheuser-Busch’;
```

Result of Query:

name	manf
‘Bud’	‘Anheuser-Busch’
‘Bud Lite’	‘Anheuser-Busch’
‘Michelob’	‘Anheuser-Busch’

Now, the result has each of the attributes of Beers.

Renaming Attributes

- If you want the result to have different attribute names, use “AS <new name>” to rename an attribute.
- Example based on Beers(name, manf):

```
SELECT name AS beer, manf
FROM Beers
WHERE manf = ‘Anheuser-Busch’
```

Result of Query:

beer	manf
‘Bud’	‘Anheuser-Busch’
‘Bud Lite’	‘Anheuser-Busch’
‘Michelob’	‘Anheuser-Busch’

Expressions in SELECT Clauses

- Any expression that makes sense can appear as an element of a SELECT clause.
- Example: from Sells(bar, beer, price):

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

Result of Query

bar	beer	priceInYen
Joe’s	Bud	300
Sue’s	Miller	360
…	…	…

Complex Conditions in WHERE Clause

- From Sells(bar, beer, price), find the price Joe's Bar charges for Bud:

```
SELECT price
FROM Sells
WHERE bar = ‘joe bar’ AND
       price < 5.0;
```

Selections

What you can use in WHERE:

attribute names of the relation(s) used in the FROM.

comparison operators: =, <>, <, >, <=, >=

apply arithmetic operations: stockprice*2

operations on strings (e.g., “||” for concatenation).

Lexicographic order on strings.

Pattern matching: s LIKE p

Special stuff for comparing dates and times.

Important Points

- Two single quotes inside a string represent the single-quote (apostrophe).
- Conditions in the WHERE clause can use AND, OR, NOT, and parentheses in the usual way boolean conditions are built.
- SQL is *case-insensitive*. In general, upper and lower case characters are the same, except inside quoted strings.

Patterns

- WHERE clauses can have conditions in which a string is compared with a pattern, to see if it matches.
- General form: <Attribute> LIKE <pattern>
  or <Attribute> NOT LIKE <pattern>

- Pattern is a quoted string with % = “any string”; _ = “any character.”

Example

- From Drinkers(name, addr, phone) find the drinkers with exchange 555:

```
SELECT name
FROM Drinkers
WHERE phone LIKE ‘%555-_ _ _ _’;
```

Motivating Example for Next Few Slides

- From the following Sells relation:

bar	beer	price
….	….	…

```
SELECT bar
FROM Sells
WHERE price < 2.00  OR  price >= 2.00;
```

Null Values

NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components.
- Meaning depends on context. Two common cases:
  - *Missing value* : e.g., we know Joe's Bar has some address, but we don't know what it is.
  - *Inapplicable* : e.g., the value of attribute *spouse* for an unmarried person.

Comparing NULL's to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- When any value is compared with NULL, the truth value is UNKNOWN.
- But a query only produces a tuple in the answer if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN).

price < 4.0

✓ 6.5 → False ×

✓ 3.0 → True

✗ Null → unknown ×

OR

price ≥ 4.0

6.5 → T

3.0 → F

Null → Unk

Truth Table

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR ?

Three-Valued Logic

- To understand how AND, OR, and NOT work in 3-valued logic, think of TRUE = 1, FALSE = 0, and UNKNOWN = ½.
- AND = MIN; OR = MAX, NOT(*x*) = 1-*x*.
- Example:

TRUE AND (FALSE OR NOT(UNKNOWN)) = MIN(1, MAX(0, (1 - ½))) =

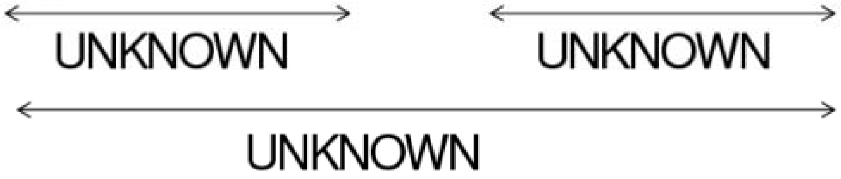
MIN(1, MAX(0, ½) = MIN(1, ½) = ½.

Surprising Example

- From the following Sells relation:

bar	beer	price
Joe’s Bar	Bud	NULL

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```



UNKNOWN UNKNOWN

UNKNOWN

Reason: 2-Valued Laws != 3-Valued Laws

- Some common laws, like the commutativity of AND, hold in 3-valued logic.
- But others do not; example: the "law of excluded middle," *p* OR NOT *p* = TRUE.
  - When *p* = UNKNOWN, the left side is MAX(½, (1 – ½)) = ½ != 1.

Testing for Null

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM    Person
WHERE  age < 25  OR  age >= 25 OR age IS NULL
```

Now it includes all Persons

Multi-Relation Queries

Multi-relation Queries $\Leftrightarrow$ R.A.

$\times$

$\bowtie$

$\bowtie_\theta$

- Interesting queries often combine data from more than one relation.
- We can address several relations in one query by listing them all in the FROM clause.
- Distinguish attributes of the same name by "<relation>.<attribute>"

beers.name

Example

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents Joe’s Bar.
- Tip: Always prefix with relation name to make it clear/easier to read.

```
SELECT Likes.beer
FROM Likes, Frequents
WHERE Frequents.bar = ‘Joe Bar’ AND
   Frequents.drinker = Likes.drinker;
```

Another Example

Product (pname, price, category, maker)
Purchase (buyer, seller, store, product)
Company (cname, stockPrice, country)
Person(pname, phoneNumber, city)

Find names of people living in Champaign that bought gizmo products, and the names of the stores they bought from

```
SELECT   pname, store
FROM     Person, Purchase
WHERE    pname=buyer AND city=“Champaign”
         AND product=“gizmo”
```

Disambiguating Attributes

Find names of people buying telephony products:

Product (name, price, category, maker)
Purchase (buyer, seller, store, product)
Person(name, phoneNumber, city)

```
SELECT   Person.name
FROM     Person, Purchase, Product
WHERE    Person.name=Purchase.buyer
     AND Purchase.product=Product.name
     AND Product.category="telephony"
```

Semantics

multi-rel q.

- Almost the same as for single-relation queries:
  1. Start with the product of all the relations in the FROM clause.
  2. Apply the selection condition from the WHERE clause.
  3. Project onto the list of attributes and expressions in the SELECT clause.

Select A

From $R_1, R_2, \ldots R_n$

Where C

$$\underline{\pi_A \sigma_C (\underline{R_1 \times \cdots \times R_n})}$$

Explicit Tuple-Variables

- Sometimes, a query needs to use two copies of the same relation.
- Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- It's always an option to rename relations this way, even when not essential.

Example

```
SELECT s1.bar
FROM Sells s1, Sells s2
WHERE s1.beer = s2.beer AND
   s1.price < s2.price;
```

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, …, ak
FROM   R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

1. Nested loops: Freq

```
Answer = { }
for x1 in R1 do                Likes
    for x2 in R2 do
        …..
            for xn in Rn do
                if Conditions
                    then Answer = Answer U {(a1,…,ak)}
return Answer
```

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, …, ak
FROM    R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

3. Translation to Relational algebra:

$$\Pi_{a1,\ldots,ak} (\sigma_{\text{Conditions}} (R1 \text{ x } R2 \text{ x } \ldots \text{ x } Rn))$$

Select-From-Where queries are precisely Select-Project-Join

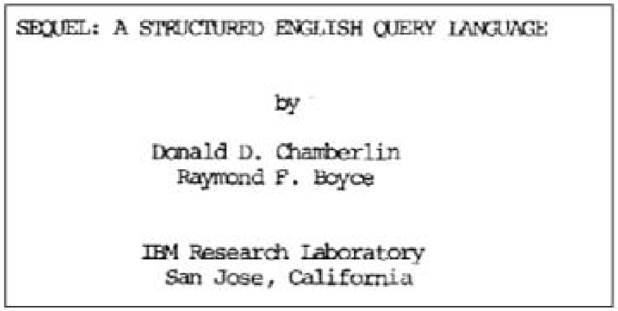
Exercises

Product (pname, price, category, maker)
Purchase (buyer, seller, store, product)
Company (cname, stock price, country)
Person(per-name, phone number, city)

Ex #1: Find people who bought telephony products.
Ex #2: Find names of people who bought American products
Ex #3: Find names of people who bought American products and did not buy French products
Ex #4: Find names of people who bought American products and they live in Champaign.
Ex #5: Find people who bought stuff from Joe or bought products from a company whose stock prices is more than $50.

Behind the Scene: The Birth of SQL

- Chamberlin, D. D. and Boyce, R. F. 1974. SEQUEL: A structured English query language. In Proceedings of the 1974 ACM SIGFIDET (Now Sigmod) Workshop on Data Description, Access and Control (Ann Arbor, Michigan, May 01 - 03, 1974). FIDET '74. ACM, New York, NY, 249-264.



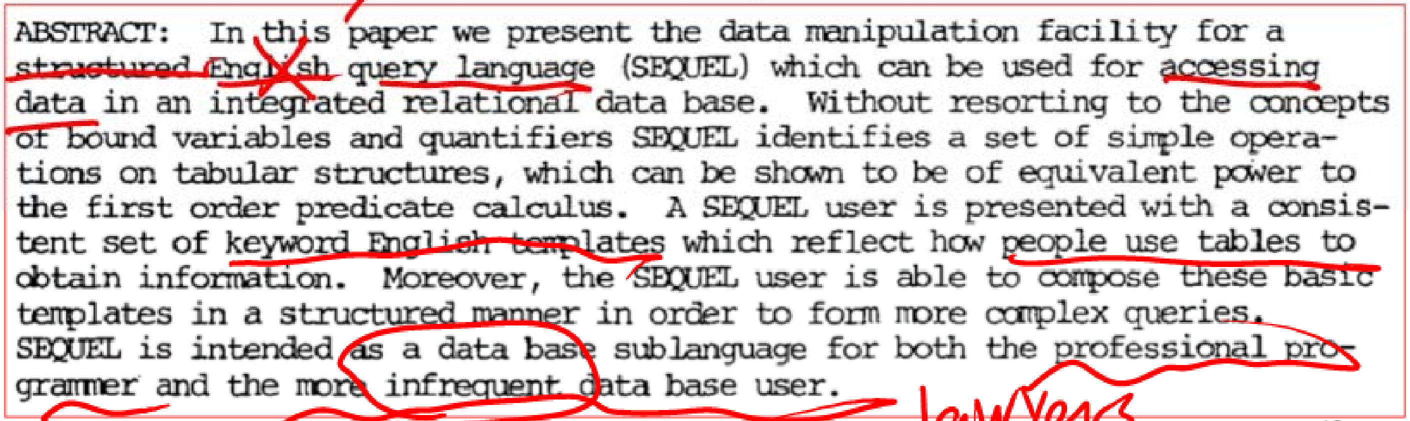
SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE

by

Donald D. Chamberlin
Raymond F. Boyce

IBM Research Laboratory
San Jose, California

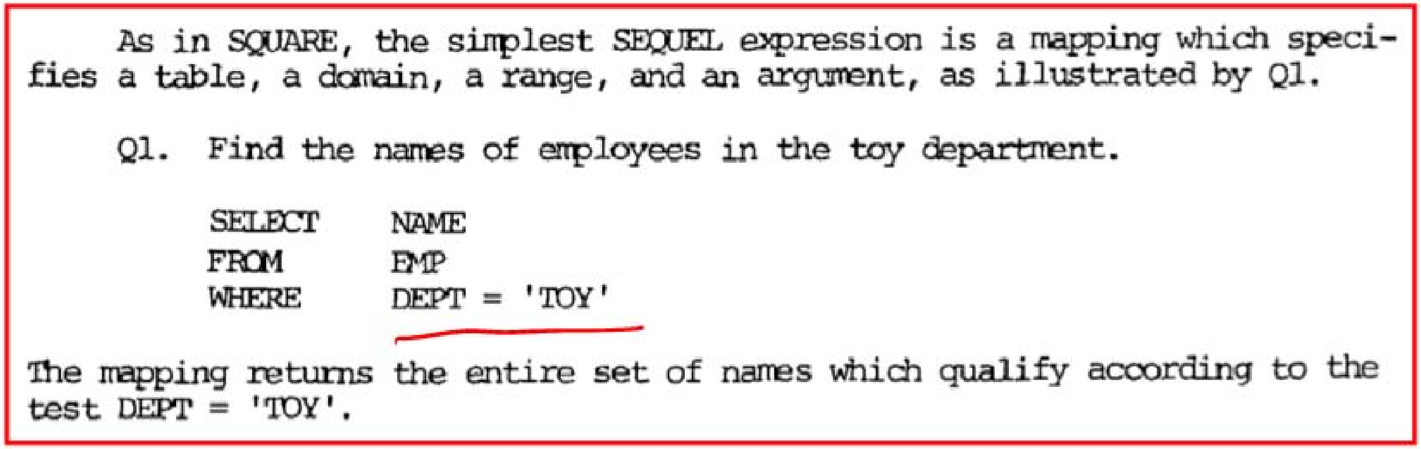
SQL



ABSTRACT: In this paper we present the data manipulation facility for a ~~structured~~ ~~English~~ query language (SEQUEL) which can be used for accessing data in an integrated relational data base. Without resorting to the concepts of bound variables and quantifiers SEQUEL identifies a set of simple operations on tabular structures, which can be shown to be of equivalent power to the first order predicate calculus. A SEQUEL user is presented with a consistent set of keyword English templates which reflect how people use tables to obtain information. Moreover, the SEQUEL user is able to compose these basic templates in a structured manner in order to form more complex queries. SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

lawyers.

The very FIRST SQL



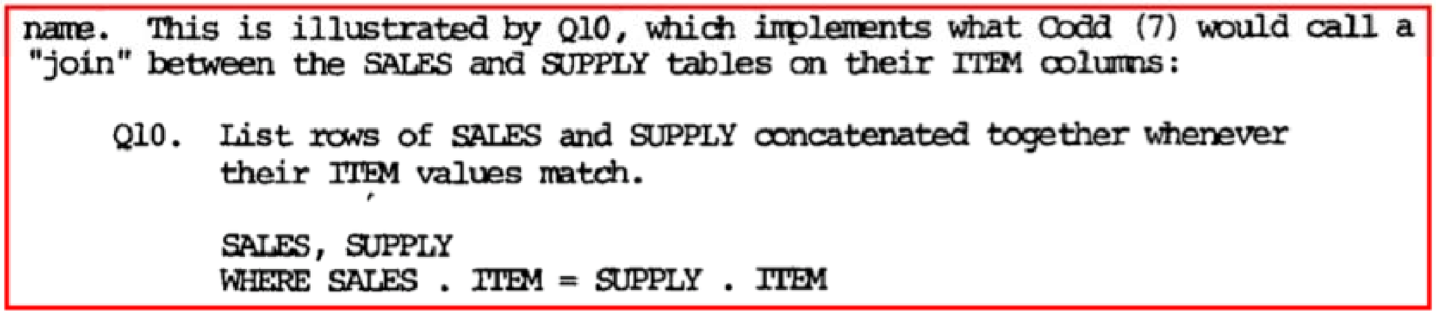
As in SQUARE, the simplest SEQUEL expression is a mapping which specifies a table, a domain, a range, and an argument, as illustrated by Q1.

Q1. Find the names of employees in the toy department.

```
SELECT    NAME
FROM      EMP
WHERE     DEPT = 'TOY'
```

The mapping returns the entire set of names which qualify according to the test DEPT = 'TOY'.

The first JOIN SQL



name. This is illustrated by Q10, which implements what Codd (7) would call a "join" between the SALES and SUPPLY tables on their ITEM columns:

Q10. List rows of SALES and SUPPLY concatenated together whenever their ITEM values match.

```
SALES, SUPPLY
WHERE SALES . ITEM = SUPPLY . ITEM
```