Introduction to SQL
(Structured Query Language)

Introduction to Databases

Sina Meraji

Thanks to Ryan Johnson, John Mylopoulos, Arnold Rosenbloom
and Renee Miller for material in these slides

Announcement

- TA forgot to show up for the first hour of class on Wed May 27th!!!
- A1 deadline is extended to Friday June 5th
- Midterm will on Wednesday July 8, class time

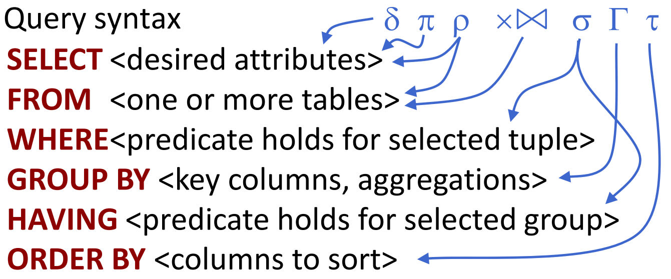
What is SQL?

- Declarative
  - Say "what to do" rather than "how to do it"
    - Avoid data-manipulation details needed by procedural languages
  - Database engine figures out "best" way to execute query
    - Called "query optimization"
    - Crucial for performance: "best" can be a million times faster than "worst"
- Data independent
  - Decoupled from underlying data organization
    - Views (= precomputed queries) increase decoupling even further
    - Correctness always assured… performance not so much
  - SQL is standard and (nearly) identical among vendors
    - Differences often shallow, syntactical

Fairly thin wrapper around relational algebra

What does SQL look like?

- Query syntax



SELECT <desired attributes>
FROM <one or more tables>
WHERE<predicate holds for selected tuple>
GROUP BY <key columns, aggregations>
HAVING <predicate holds for selected group>
ORDER BY <columns to sort>

Example

Orders

OID	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

Find if the customers "Hansen" or "Jensen“ have a total order of more than 1500

Query:

```
SELECT     Customer, SUM(OrderPrice) AS Total
FROM       Orders
WHERE      Customer = 'Hansen' OR Customer = 'Jensen'
GROUP BY   Customer
HAVING     SUM(OrderPrice) > 1500
ORDER BY   Customer DESC
```

Query Result:

Customer	Total
Jensen	2000
Hansen	2000

What does SQL *really* look like?

ORDER BY

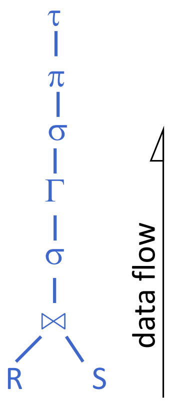
SELECT

HAVING

GROUP BY

WHERE

FROM



That’s not so bad, is it?

Other aspects of SQL

- Data Definition Language (“DDL”)
  - Manipulate database schema
  - Specify, alter physical data layout
- Data Manipulation Language (“DML”)
  - Manipulate data in databases
  - Insert, delete, update rows
- “Active” Logic
  - Triggers and constraints
  - User-defined functions, stored procedures
  - Transaction management/ Consistency levels

We’ll come back to these later in the course

SELECT-FROM-WHERE QUERIES

‘SELECT’ clause

- Identifies which attribute(s) query returns
  - Comma-separated list

  => Determines schema of query result
- Optional: extended projection
  - Compute arbitrary expressions
  - Usually based on selected attributes, but not always
- Optional: rename attributes
  - “Prettify” column names for output
  - Disambiguate (E1.name vs. E2.name)
- Optional: specify groupings
  - More on this later
- Optional: duplicate elimination
  - SELECT DISTINCT …

‘SELECT’ clause – examples

- **SELECT** E.name …

  => Explicit attribute
- **SELECT** name …

  => Implicit attribute (error if R.name and S.name exist)
- **SELECT** E.name **AS** ‘Employee name’ …

  => Prettified for output (like table renaming, ‘AS’ usually not required)
- **SELECT** sum(S.value) …

  => Grouping (compute sum)
- **SELECT** sum(S.value)*0.13 ‘HST’ …

  => Scalar expression based on aggregate
- **SELECT** * …

  => Select all attributes (no projection)
- **SELECT** E.* …

  => Select all attributes from E (no projection)

‘FROM’ clause

- Identifies the tables (relations) to query
  - Comma-separated list
- Optional: specify joins
  - … but often use WHERE clause instead
- Optional: rename table
  - Using the same table twice (else they’re ambiguous)
  - Nested queries (else they’re unnamed)

‘FROM’ clause – examples

- … **FROM** Employees

  => Explicit relation
- … **FROM** Employees **AS** E

  => Table alias (most systems don’t require “AS” keyword)
- … **FROM** Employees, Sales

  => Cartesian product
- … **FROM** Employees E **JOIN** Sales S

  => Cartesian product (*no join condition given!*)
- … **FROM** Employees E **JOIN** Sales S **ON** E.EID=S.EID

  => Equi-join

‘FROM’ clause – examples (cont)

- … **FROM** Employees **NATURAL JOIN** Sales

  => Natural join (bug-prone, use equijoin instead)

- … **FROM** Employees E
  LEFT JOIN Sales S **ON** E.EID=S.EID

  => Left join

- … **FROM** Employees E1
  JOIN Employees E2 **ON** E1.EID < E2.EID

  => Theta self-join (*what does it return?*)

Gotcha: natural join in practice

- Uses *all* same-named attributes
  - May be too many or too few
- Implicit nature reduces readability
  - Better to list explicitly all join conditions
- Fragile under schema changes
  - Nasty interaction of above two cases..

Moral of the story: Avoid using Natural Join

Gotcha: join selectivity

- Consider tables **R**, **S**, **T** with **T=Ø** and this query:

```
SELECT R.x
FROM R,S,T
WHERE R.x=S.x OR R.x=T.x
```

(*what does it return?*)

- Result contains no rows!
  - Selection (WHERE) operates on pre-joined tuples
  - R × S × T = R × S × Ø = Ø

  => No tuples for WHERE clause to work with!

- Workaround?
  - Two coming up later

Moral of the story: WHERE cannot create tuples

Explicit join ordering

- Use parentheses to group joins
  - e.g. (A join B) join (C join D)
- Special-purpose feature
  - Helps some (inferior) systems optimize better
  - Helps align schemas for natural join
- Recommendation: avoid
  - People are notoriously bad at optimizing things
  - Optimizer usually does what it wants anyway

    … but sometimes treats explicit ordering as a constraint

‘WHERE’ clause

- Conditions which all returned tuples must meet
  - Arbitrary boolean expression
  - Combine multiple expressions with AND/OR/NOT
- Attention to data of interest
  - Specific people, dates, places, quantities
  - Things which do (or do not) correlate with other data
- Often used instead of JOIN
  - FROM tables (Cartesian product, e.g. A, B)
  - Specify join condition in WHERE clause (e.g. A.ID=B.ID)
  - Optimizers (usually) understand and do the right thing

Scalar expressions in SQL

- Literals, attributes, single-valued relations
- Boolean expressions
  - Boolean T/F coerce to 1/0 in arithmetic expressions
  - Zero/non-zero coerce to F/T in boolean expressions
- Logical connectors: AND, OR, NOT
- Conditionals

  = != < > <= >= <>

  BETWEEN, [NOT] LIKE, IS [NOT] NULL, …
- Operators: + - * / % & | ^
- Functions: math, string, date/time, etc. (more later)

Similar to expressions in C, python, etc.

‘WHERE’ clause – examples

- … **WHERE** S.date > ‘01-Jan-2010’

  => Simple tuple-literal condition
- … **WHERE** E.EID = S.EID

  => Simple tuple-tuple condition (equi-join)
- … **WHERE** E.EID = S.EID **AND** S.PID = P.PID

  => Conjunctive tuple-tuple condition (three-way equijoin)
- … **WHERE** S.value < 10 **OR** S.value > 10000

  => Disjunctive tuple-literal condition

Pattern matching

- Compare a string to a pattern
  - <attribute> **LIKE** <pattern>
  - <attribute> **NOT LIKE** <pattern>
- Pattern is a quoted string

  % => “any string”

  _ => “any character”
- To escape ‘%’ or ‘_’:
  - LIKE ‘%x_%’ ESCAPE ‘x’ (replace ‘x’ with character of choice)

  ⇒ matches strings containing ‘_’ (the underscore character)

DBMS increasingly allow regular expressions

Pattern matching – examples

- … **WHERE** phone **LIKE** ‘%268-_ _ _ _’
  - phone numbers with exchange 268
  - WARNING: spaces are wrong, only shown for clarity
- … **WHERE** last_name **LIKE** ‘Jo%’
  - Jobs, Jones, Johnson, Jorgensen, etc.
- … **WHERE** Dictionary.entry **NOT LIKE** ‘%est’
  - Ignore ‘biggest’, ‘tallest’, ‘fastest’, ‘rest’, …
- … **WHERE** sales **LIKE** '%30!%%' **ESCAPE** ‘!’
  - Sales of 30%

MORE COMPLEX QUERIES (GROUP BY-HAVING-ORDER BY)

‘GROUP BY’ clause

- Specifies **grouping key** of relational operator Γ
  - Comma-separated list of attributes (names or positions) which identify groups
  - Tuples agreeing in their grouping key are in same “group”
  - SELECT gives attributes to aggregate (and functions to use)
- SQL specifies several **aggregation functions**
  - COUNT, MIN, MAX, SUM, AVG, STD (standard deviation)
  - Some systems allow user-defined aggregates

‘GROUP BY’ clause – gotchas

- WHERE clause cannot reference aggregated values (sum, count, etc.)
  - Aggregates don’t “exist yet” when WHERE runs
  - => Use **HAVING** clause instead (coming next)
- GROUP BY **must** list all non-aggregate attributes used in SELECT clause
  - Think projection
  - => Some systems do this implicitly, others throw error
- Grouping often (but not always!) sorts on grouping key
  - Depends on system and/or optimizer decisions
  - => Use **ORDER BY** to be sure (coming next)

‘GROUP BY’ clause – examples

- SELECT EID, SUM(value)
  FROM SALES **GROUP BY** EID
  - Show total sales for each employee ID
- SELECT EID, SUM(value), MAX(value)
  FROM Sales **GROUP BY** 1
  - Show total sales and largest sale for each employee ID
- SELECT EID, COUNT(EID)
  FROM Complaints **GROUP BY** EID
  - Show how many complaints each salesperson triggered

‘GROUP BY’ clause – examples (cont)

- SELECT EID, SUM(value) FROM Sales
  - Error: non-aggregate attribute (EID) missing from GROUP BY
- SELECT EID, value FROM Sales **GROUP BY** 1,2
  - Not an error – eliminates duplicates
- SELECT SUM(value) FROM Sales **GROUP BY** EID
  - Not an error, but rather useless: report per-employee sales anonymously
- SELECT SUM(value) FROM Sales
  - No GROUP BY => no grouping key => all tuples in same group

Eliminating duplicates in aggregation

- Use **DISTINCT** inside an aggregation

  SELECT EmpID, COUNT(**DISTINCT** CustID)
  FROM CustomerComplaints
  GROUP BY 1

=> Number of customers who complained about the employee

‘HAVING’ clause

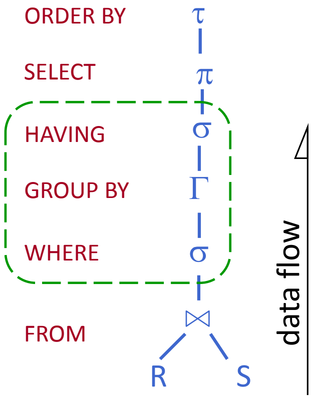
- Allows predicates on aggregate values
  - Groups which do not match the predicate are eliminated

  => **HAVING** is to groups what **WHERE** is to tuples
- Order of execution
  - WHERE is before GROUP BY

  => Aggregates not yet available when WHERE clause runs
  - GROUP BY is before HAVING

  => Scalar attributes still available

- In tree form:



ORDER BY τ

SELECT π

HAVING σ

GROUP BY Γ

WHERE σ

FROM ⋈

‘HAVING’ clause – examples

- SELECT EID, SUM(value)
  FROM Sales GROUP BY EID
  HAVING SUM(Sales.value) > 10000
  - Highlight employees with “impressive” sales
- SELECT EID, SUM(value)
  FROM Sales GROUP BY EID
  HAVING AVG(value) < (SELECT AVG(GroupAVG)
  FROM (SELECT EID, AVG(value) AS GroupAVG
  FROM Sales GROUP BY EID) AS B);
  - Highlight employees with below-average sales
  - Subquery to find the avg value of average employee sales

‘ORDER BY’ clause

- Each query can sort by one or more attributes
  - Refer to attributes by name or position in SELECT
  - Ascending (default) or descending (reverse) order
  - Equivalent to relational operator τ
- Definition of ‘sorted’ depends on data type
  - Numbers use natural ordering
  - Date/time uses earlier-first ordering
  - NULL values are not comparable, cluster at end or beginning
- Strings are more complicated
  - Intuitively, sort in “alphabetical order”
  - Problem: which alphabet? case sensitive?
  - Answer: user-specified “collation order”
  - Default collation: case-sensitive latin (ASCII) alphabet

String collation not covered in this class

‘ORDER BY’ clause – examples

- **… ORDER BY** E.name
  => Defaults to ascending order
- **… ORDER BY** E.name **ASC**
  => Explicitly ascending order
- **… ORDER BY** E.name **DESC**
  => Explicitly descending order
- **… ORDER BY** CarCount **DESC**, CarName **ASC**
  => Matches our car example from previous lecture
- SELECT E.name … **ORDER BY** 1
  => Specify attribute’s position instead of its name

What’s next?

- Examples

WORKING EXAMPLES

Example Database

Employee(FirstName,Surname,Dept,Office,Salary,City)
Department(DeptName,Address,City)

Home city

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

City of work

Example: Simple SQL Query

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the salaries of employees named Brown"

SELECT Salary **AS** Remuneration
FROM Employee
WHERE Surname = 'Brown'

Result:

Remuneration
45
80

Example: * in the Target List

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find all the information relating to employees named Brown" :

SELECT *
FROM Employee
WHERE Surname = 'Brown'

Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	Brown	Planning	14	80	London

Example: Attribute Expressions

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the monthly salary of employees named White" :

SELECT Salary / 12 **AS** MonthlySalary
FROM Employee
WHERE Surname = 'White'

Result:

MonthlySalary
3.00

Example: Simple (Equi-)Join Query

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the names of employees and their cities of work“

SELECT Employee.FirstName, Employee.Surname, Department.City
FROM Employee, Department
WHERE Employee.Dept = Department.DeptName

Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

(alternative?)

Alternative (and more correct):
SELECT Employee.FirstName, Employee.Surname, Department.City
FROM Employee E **JOIN** Department D **ON** E.Dept = D.DeptName

Example: Table Aliases

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the names of employees and their cities of work" (using an alias):

SELECT FirstName, Surname, D.City
FROM Employee, Department D
WHERE Dept = DeptName

Result:

FirstName	Surname	City
Mary	Brown	London
Charles	White	Toulouse
Gus	Green	London
Jackson	Neri	Brighton
Charles	Brown	London
Laurence	Chen	London
Pauline	Bradshaw	London
Alice	Jackson	Toulouse

Example: Predicate Conjunction

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the first names and surnames of employees who work in office number 20 of the Administration department":

SELECT FirstName, Surname
FROM Employee
WHERE Office = '20' **AND** Dept = 'Administration'

Result:

FirstName	Surname
Gus	Green

Example: Predicate Disjunction

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the first names and surnames of employees who work in either the Administration or the Production department":

SELECT FirstName, Surname
FROM Employee
WHERE Dept = 'Administration' **OR** Dept = 'Production'

Result:

FirstName	Surname
Mary	Brown
Charles	White
Gus	Green
Pauline	Bradshaw
Alice	Jackson

Example: Complex Logical Expressions

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the first names of employees named Brown who work in the Administration department or the Production department":

SELECT FirstName
FROM Employee
WHERE Surname = 'Brown' **AND** (Dept = 'Administration' **OR** Dept = 'Production')

Result:

FirstName
Mary

Example: String Matching Operator LIKE

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find employees with surnames that have 'r' as the second letter and end in 'n'":

SELECT *
FROM Employee
WHERE Surname LIKE '_r%n'

Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Gus	Green	Administration	20	40	Oxford
Charles	Brown	Planning	14	80	London

Example: Aggregate Queries: Operator **Count**

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the number of employees":

SELECT count(*) **FROM** Employee

"Find the number of different values on attribute Salary for **all** tuples in Employee":

SELECT count(**DISTINCT** Salary) **FROM** Employee

"Find the number of tuples in Employee having **non-null values** on the attribute Salary":

SELECT count(**ALL** Salary) **FROM** Employee

Example: Operators **Sum**, **Avg**, **Max** and **Min**

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the sum of all salaries for the Administration department":

```
SELECT sum(Salary) AS SumSalary
FROM Employee
WHERE Dept = 'Administration'
```

Result:

SumSalary
125

Example: Operators **Sum**, **Avg**, **Max** and **Min**

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the maximum and minimum salaries among all employees":

```
SELECT max(Salary) AS MaxSal, min(Salary) AS MinSal
FROM Employee
```

Result:

MaxSal	MinSal
80	36

Example: Aggregate Operators with Join

Employee(FirstName, Surname, Dept, Office, Salary, City)
Department(DeptName, Address, City)

"Find the maximum salary among the employees who work in a department based in London":

```
SELECT max(Salary) AS MaxLondonSal
FROM Employee, Department
WHERE Dept = DeptName AND Department.City = 'London'
```

Result:

MaxLondonSal
80