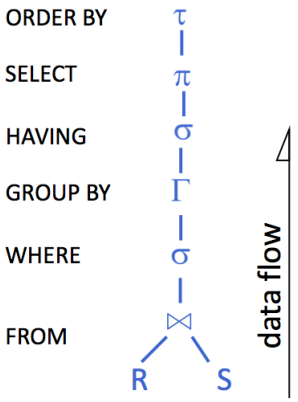
STRUCTURED QUERY LANGUAGE (SQL)

- SQL
  - Thin wrapper arroudn relational algebra
  - Declarative → say "what to do" rather than "how to do it"d
    - Avoid data-manipulation details needed by procedural languages
    - Database egine figures out "best" way to execute query
      - Called query optimization
      - Crucial for performance: "best" vs. worst can be 1000000x
  - Data independent
    - Decoupled from underlying data organization
      - Views = precomputed queries, increase decoupling even further
      - Correctness always assured, performance not so much
    - SQL is standard, identical among vendors
      - Differences = shallow, syntactical
  - Query syntax
    - **SELECT** ⟨desired attributes⟩
      - **Duplicate Elimination** δ; **Project** π; **Rename** ρ
    - **FROM** ⟨one or more tables ⟩
      - **Rename** ρ; **Cartesian Product** $\times$; **Natural Join** $\bowtie$
    - **WHERE** ⟨predicate holds for selected tuple⟩
      - **Select** σ
    - **GROUP BY** ⟨key columns, aggregations⟩
      - **Grouping** Γ
    - **HAVING** ⟨predicate holds for selected group⟩
      - **Select** σ
    - **ORDER BY** ⟨columns to sort⟩
      - **Sort** τ
  - Data flow



  - 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

    - Query: find the customers "Hansen" or "Jensen" have a total order of more than 1500

```
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
```

    - Result:

Customer	Total
Jensen	2000
Hansen	2000

SELECT-FROM-WHERE queries

- **SELECT** clause identifies which attribute(s) query returns
  - Attributes listed as comma-seperated list
  - Determines schema of query result
  - Optional
    - Extended projection → compute arbitrary expressions
      - Usually based on selected attributes, but not always
    - Rename attributes → set column names for output
      - Disambiguate (E1.name vs. E2.name)
    - Specify groupings
    - Duplicate elimination: **SELECT** DISTINCT ...
  - SELECT clause examples
    - Explicit attribute
      SELECT E. Name ...
    - Implicit attribute → causes an error if attribute Name does not exist
      SELECT Name ...
    - Prettified for output, AS usually not required
      SELECT E. Name **AS** "Employee Name" ...
    - Grouping → compute sum
      SELECT sum(S. Value) ...
    - Scalar expression based on aggregate
      SELECT sum(S. Value) ∗ 0.13 "HST" ...
    - Select all attributes (no projection)
      SELECT ∗ ...
    - Select all attributes from E (no projection)
      SELECT E.∗ ...

- **FROM** clause identifies the table (relations) to query
  - Attributes listed as comma-seperated list
  - Optional
    - Specify joins, but often use WHERE clause instead
    - Rename table
      - Using the same table twice (else they're ambiguous)
      - Nested queries (else they're unnamed)
  - FROM clause examples
    - Explicit relation
      ... **FROM** Employees
    - Table alias, most system don't require AS keyword
      ... **FROM** Employees **AS** E
    - Cartesian product
      ... **FROM** Employees, Sales
    - still Cartesian product b/c no join condition given
      ... **FROM** Employees E **JOIN** Sales S
    - equijoin
      ... **FROM** Employees E **JOIN** Sales S **ON** E. EID = S. EID
    - Natural join
      ... **FROM** Employees **NATURAL JOIN** Sales
    - Left join
      ... **FROM** Employees E **LEFT JOIN** Sales S **ON** E. EID = S. EID
    - Theta join on self
      ... **FROM** Employees E1 **JOIN** Employees E2
      ON E1. EID < E2. EID
  - Explicit Join Ordering
    - User parentheses to group joins
      - Ex. (A JOIN B) JOIN (C JOIN D)
    - Special-purpose feature
      - Helps some (inferior) systems optimize better
      - Helps align schemas for natural join
  - Note: Natural Join in practice
    - Uses "all" same-named attributes → may be too many or too few
    - Implicity nature reduces readability
      - Better to list explicitly all join conditions
    - Fragile under schema changes

- **WHERE** clause specifies conditions which all returned tuples must meet
  - Arbitrary boolean expression
  - Attention to data of interest → specific people, dates, places, quantities
  - Often used instead of JOIN
    - FROM tables specify join condition in WHERE clause
  - Scalar expressions in SQL
    - Literals, attributes, single-valued relations
    - Boolean expressions
      - Boolean T/F → 1/0 in arithmetic expressions
      - 0/non-zero → F/T in boolean expressions
    - Logical connectors: AND OR NOT
    - Conditionals: = ! = < > <= >= <>
      BETWEEN [NOT] LIKE IS [NOT] NULL
    - Operators: + − * / % & | ^
  - WHERE clause examples
    - Simple tuple-litral condition
      … **WHERE** S. Date > "01 − Jan − 2010"
    - Simple tuple-tuple condition
      - Equivalent to an equijoin
      … **WHERE** E. EID = S. EID
    - Conjunctive tuple-tuple condition
      - Equivalent to three-way equijoin
      … **WHERE** E. EID = S. EID **AND** S. PID = P. PID
    - Disjunctive tuple-literal condition
      … **WHERE** S. Value < 10 **OR** S. Value > 10000
  - 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 _ → replace X with character of choice
      - Escape the character following X
      - Matches strings containing "_" (the underscore character)
      LIKE "%X_%" **ESCAPE** "X"
  - WHERE clause example w/ pattern matching
    - Phone numbers leading w/ 268
      … **WHERE** phone **LIKE** "%268 − _ _ _ _"
    - Match names leading with "Jo" –ex. Jobs, Jones, Johnson, etc.
      … **WHERE** last_name **LIKE** "Jo%"
    - Match names w/o ending of "est" –ex. biggest, tallest, fastest
      … **WHERE** Dictionary. entry **NOT LIKE** "%Jo"
    - Escape % sign, match percentage –ex. 30%
      … **WHERE** sales **LIKE** "%30!%%" ESCAPE "! "
- Consider tables R, S, T w/ T is empty
  - Query:
    SELECT R. x
    FROM R, S, T
    WHERE R. x = S **OR** R. x = T. x
  - Result contains no tables b/c R × S × T = R × S × ∅ = ∅
    - WHERE operates on pre-joined tuples
    - No tuples for WHERE → WHERE cannot create tuples

SQL EXAMPLES

- SELECT-FROM-WHERE examples
  - Database Schemas
    `Employee(FirstName, Surname, Dept, Office, Salary, City)`
    `Department(DeptName, Address, 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

Home city

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

  - **Query 1**: Find the salaries of employees named Brown
    - Note: Simple SQL Query

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

Remuneration
45
80

  - **Query 2**: Find all the information relating to employees named Brown
    - Note: * in the Target List

SELECT *
FROM Employee
WHERE Surname = "Brown"

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

  - **Query 3**: Find the month salary of employees named White
    - Note: Attribute expressions

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

MonthlySalary
3.00

  - **Query 4**: Find the names of employees and their cities of work
    - Note: simple equijoin query

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

    - Alternative (more correct b/c Cartesian product expensive)

SELECT Employee. FirstName, Employee. Surnam, Department. City
FROM Employee E **JOIN** Department D **ON** E. Dept = D. DeptName

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

- **Query 5**: Find the names of employees and their cities of work
  - Note: Table Aliases

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

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

- **Query 6**: Find the first names and surnames of employees who work in office number 20 of the Administration department
  - Note: Predicate conjunction

```
SELECT FirstName, Surname
FROM Employee
WHERE Office = "Brown" AND Dept = "Administration"
```

FirstName	Surname
Gus	Green

- **Query 7**: Find the first names and surnames of employees who work in either the Administration or the Production department
  - Note: Predicate disjunction

```
SELECT FirstName, Surname
FROM Employee
WHERE Office = "Administration" OR Dept = "Production"
```

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

- **Query 8**: Find the first name of employees named Brown who work in the Administration department or the Production department
  - Note: Complex logical expression

```
SELECT FirstName
FROM Employee
WHERE Surname = "Brown" AND
        (Dept = "Administration" OR Dept = "Production")
```

FirstName
Mary

- **Query 9**: Find employees with surnames that have "r" as the second letter and end in "n"
  - Note: String matching operator LIKE

```
SELECT *
FROM Employee
WHERE Surname LIKE "_r%n"
```

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

- **Query 10**
  - Note: Aggregate queries (count operator)
  - **Query 10a**: Find the number of employees

```
SELECT count(*)
FROM Employee
```

  - **Query 10b**: Find the number of different values on attribute Salary for all tuples in employees

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

  - **Query 10c**: Find the number of tuples in Employee having non-null values on the attribute Salary
    - Yields same result if ALL is omitted

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


- **Query 11**:
  - Note: sum, avg, max, min operators
  - **Query 11a**: Find the sum of all salaries for the Administration department

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

SumSalary
125

  - **Query 11b**: Find the maximum and minimum salaries among all employees

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

MaxSal	MinSal
80	36

- **Query 12**: Find the maximum salary among the employees who work in a department based in London
  - Note: Aggregate operators with Join

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

MaxLondonSal
80

GROUP BY, HAVING, ORDERBY queries

- **GROUP BY** clause specifies grouping key of relational operator Γ
  - Comma-separated attributes list (names/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)
  - Warning:
    - WHERE clause cannot reference aggregated values
      - Aggregates don't "exist yet" when WHERE runs
    - GROUP BY must list all non-aggregate attributes used in SELECT clause (similar to projection)
    - Grouping often sorts on grouping key
  - GROUP BY clause examples
    - Show total sales for each employee ID

      SELECT EID, sum(Value)
      FROM Sales
      GROUP BY EID

    - Show total sales and largest sale for each employee ID
      - GROUP BY 1 refers attribute by position in SELECT

      SELECT EID, sum(Value), max(Value)
      FROM Sales
      GROUP BY 1

    - Show how many complaints each salesperson triggered

      SELECT EID, count(Value)
      FROM Complaints
      GROUP BY EID

    - Error: non-aggregate attribute (EID) missing from GROUP BY

      SELECT EID, sum(Value)
      FROM Sales

    - Not an error – eliminates duplicates

      SELECT EID, Value
      FROM Sales
      GROUP BY 1,2

    - Not an error, but useless
      - Report per-employee sales anonymously

      SELECT sum(Value)
      FROM Sales
      GROUP BY EID

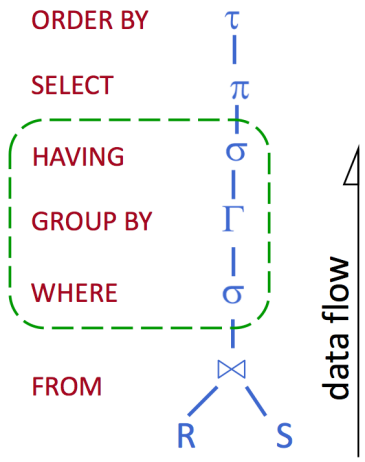
    - No GROUP BY → no grouping key → all tuples in same group

      SELECT sum(Value)
      FROM Sales

  - DISTINCT inside an aggregation → eliminate duplicates in aggregation
    - Ex. Number of customers who complained about the employee

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

  - GROUP BY in practice
    - Useful for retrieving information about a group of data
      - If there's only 1 product of each type then its not as useful
    - Useful for when there's similar things
    - Remember to GROUP BY on the column you want information about, not the on the one with the aggregate function applied



- **HAVING** clause allows predicates on aggregate values
  - Groups which do not match the predicate are eliminated
    - HAVING to groups is similar to WHERE 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
  - HAVING clause examples
    - Highlight employees with above 10 000 sales

      SELECT EID, sum(value)
      FROM Sales
      GROUP BY EID
      HAVING sum(Sales. Value) > 10 000

    - Highlight employees w/ below-average sales
      - Subqeury to find the B (the value of average employee sales)

      SELECT EID, avg(Value)
      FROM Sales
      GROUP BY EID
      HAVING avg(Value) < (
      SELECT avg(GroupAVG)
      FROM (
      SELECT EID, avg(Value) **AS** GroupAvg
      FROM Sales
      GROUP BY EID
)
)

  - HAVING in practice
    - If a condition refers to an aggregate function, put that condition in the HAVING clause
      - Otherwise, use the WHERE clause
    - HAVING has to be used in conjunction with GROUP BY
- **ORDER BY** clause sorts query by one or more attributes, equivalent to τ
  - Refer to attributes by name or position in SELECT
  - Ascending (default) or descending (reverse) order
  - Sorting depends on data type
    - Numbers use natural order
    - Date/time uses earlier-first ordering
    - NULL values are not comparable → cluster at end or beginning
    - Strings are complicated
      - Intuitively sort in alphabetical order → problem w/ choosen alphabet, and case sensitivity
      - Solution" user-specified collation order → by default uses case-sensitive latin (ASCII) alphabet
  - ORDER BY clause examples
    - Defaults to ascending order

      ... **ORDER BY** E. name

    - Explicitly ascending order

      ... **ORDER BY** E. name **ASC**

    - Explicitly descending order

      ... **ORDER BY** E. name **DESC**

    - SQL form of Sorting example

      ... **ORDER BY** CarCount **DESC**, CarName **ASC**

    - Specify attribute's position instead of its name

      SELECT E. name ...
      ORDER BY 1

COMPLEX SQL QUERIES

- Example 1
  - Database Schemas

    `Employee(FirstName, Surname, Dept, Office, Salary, City)`

    `Department(DeptName, Address, 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

Home city

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

  - **Query 1**: Find the sum of salaries of all the employees of each department (use GROUP BY)

```
SELECT Dept, sum(Salary) AS Total
FROM Employee
GROUP BY Dept
```

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

    - Query breakdown:
      - 1. Query executed w/o GROUP BY and w/o aggregate operators
      - 2. Results divided into subsets characterized by the same values for the GROUP BY attributes:

Dept	Salary
Administration	45
Production	36
Administration	40
Distribution	45
Planning	80
Planning	73
Administration	40
Production	46

→

Dept	Salary
Administration	45
Administration	40
Administration	40
Distribution	45
Planning	80
Planning	73
Production	36
Production	46

      - 3. Aggregate operator sum is applied separately to each group:

Dept	TotSal
Administration	125
Distribution	45
Planning	153
Production	82

    - Incorrect queries
      - Column that GROUP BY is call upon doesn't appear in SELECT

```
SELECT Office
FROM Employee
GROUP BY Dept
```

      - Not all non-aggregate columns in SELECT listed in GROUP BY

```
SELECT DeptName, D. City, count(*)
FROM Employee E JOIN Department D
                ON (E. Dept = D. Dept)
GROUP BY DeptName
```

      - Solution:

```
SELECT DeptName, D. City, count(*)
FROM Employee E JOIN Department D
                ON (E. Dept = D. Dept)
GROUP BY DeptName, D. City
```

  - **Query 2**: Find which departments spend more than 100 on salaries

```
SELECT Dept
FROM Employee
GROUP BY Dept
HAVING sum(Salary) > 100
```

Dept
Administration
Planning

  - **Query 3**: Find the departments where the average salary of employees working in office number 20 is higher than 20

```
SELECT Dept
FROM Employee
WHERE office = "20"
GROUP BY Dept
HAVING avg(Salary) > 25
```

- Example 2
  - Database Schemas
    - Semester are: YYYY(F|S|W) –ex. "2006F", "2007W"

    `Professor(ID, Name, DeptId)`

    `Course(CrsCode, DeptId, CrsName, Description)`

    `Teaching(ProfID, CrsCode, Semester)`

  - **Query 1**: Find the names of all professors who taught in Fall 2004

```
SELECT P. Name
FROM Professor P JOIN Teaching T ON P. ID = T. ProfID
WHERE P. Semester = "2004F"
GROUP BY P. Name
```

  - **Query 2**: Find the names of all courses taught in Fall 2005, together with the names of professors who taught them

```
SELECT DISTINCT P. Name, C. CrsName
FROM Professor P, Teaching T, Course C
WHERE P. ID = T. ProfId AND T. CrsCode = C. CrsCode
                       AND T. Semester = "2005F"
```

  - **Query 3**: Find the average number of courses taught by professors in Computer Science (CS)

```
SELECT count(C. CrsCode)/count(DISTINCT T. ProfID) AS Result
FROM Teaching T, Course C
WHERE T. CrsCode = C. CrsCode AND C. DeptId = "CS"
```

  - **Query 4**: Find the number of courses taught by each professor in Computer Science (CS)

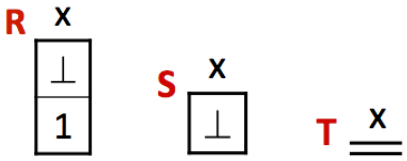
```
SELECT T. ProfID, count(*)
FROM Teaching T, Course C
WHERE T. CrsCode = C. CrsCode AND C. DeptID = "CS"
GROUP BY T. ProfID
```

  - **Query 5**: Find the number of courses taught by each professor in Computer Science (CS) in 2008

```
SELECT T. ProfID, count(*)
FROM Teaching T, Course C
WHERE T. CrsCode = C. CrsCode AND C. DeptID = "CS"
                       AND T. Semester LIKE "2008_"
GROUP BY T. ProfID
```

OTHER SQL CONCEPTS

- NULL values in SQL
  - Values allowed to be NULL
    - Explicitly stored in relations
    - Results of outer joins
  - Possible meanings
    - No present –ex. homeless man's address
    - Unknown –ex. Julian Assange's address
  - Effect: poison
    - Arithemtic: unknown value takes over expression
    - Conditional: ternary logic → TRUE, FALSE, UNKNOWN
    - Grouping: "not present"
  - NULL in expressions
    - Arithmetic: NaN
      - NULL*0 → NULL; NULL – NULL → NULL
    - Logic: TRUE, FALSE, NULL
      - NULL **OR** FALSE → NULL
      - NULL **OR** TRUE → TRUE
      - NULL **AND** TRUE → NULL
      - NULL **AND** FALSE → FALSE
      - **NOT** NULL → NULL
    - Ternary logic trick:
      - TRUE = 1; FALSE = 0; NULL = 0.5
      - AND = min(…) OR = max(…) NOT = 1 – X
  - NULL in groupings
    - count
      - count(R.*) = 2 count(R.x) = 1
      - count(S.*) = 1 count(S.x) = 0
      - count(T.*) = 0 count(T.x) = 0
    - min/max
      - min(R.x) = 1 max(R.x) = 1
      - min(S.x) = NULL max(S.x) = NULL
      - mn(T.x) = NULL max(T.x) = NULL



- Set queries
  - Operations on pairs of subqueries
    - Operations are by default set-based
  - Syntax:
    - ALL keyword forces bag semantics (duplicates allowed)

(subquery) **UNION** [**ALL**] (subquery)
(subquery) **INTERSECT** [**ALL**] (subquery)
(subquery) **EXCEPT** [**ALL**] (subquery)

  - Ex. Find first and surnames of employees
    - Solution 1: UNION w/ duplicates are removed

SELECT FirstName **AS** Name **FROM** Employee
UNION
SELECT Surname **AS** Name **FROM** Employee

    - Solution 2: UNION w/ duplicates are kept

SELECT FirstName **AS** Name **FROM** Employee
UNION ALL
SELECT Surname **AS** Name **FROM** Employee

  - Ex. Find surnames of employees that are also first names
    - Solution 1: INTERSECT

SELECT FirstName **AS** Name **FROM** Employee
INTERSECT
SELECT Surname **AS** Name **FROM** Employee

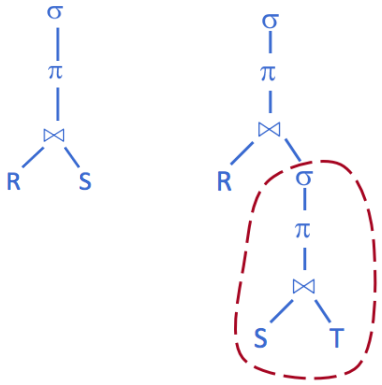
    - Solution 2: WHERE

SELECT E1.FirstName **AS** Name
FROM Employee E1, Employee E2
WHERE E1.FirstName = E2.Surname

  - Ex. Find the surname of employees that are not first names
    - Solution: EXCEPT (set difference)

SELECT SurName **AS** Name **FROM** Employee
EXCEPT
SELECT FirstName **AS** Name **FROM** Employee

- Nested queries
  - Concept
    - Treat one query's output as input to another query
    - Inner schema determined by inner SELECT clause



  - Nested queries uses
    - Explicity join ordering

FROM (A **JOIN** B)

    - Input relation for a set operation: UNION, INTERSECT, DIFFERENCE
    - Input relation for a larger query → appears in FROM clause
      - Usually joined with other tables

FROM A, (**SELECT** …) B
WHERE …

    - Conditional relation expression
      - Dynamic list for [NOT] IN operator

WHERE (**E.id**, **S.Name**) **IN** (**SELECT** id, name **FROM** …)

WHERE NOT EXISTS (**SELECT** ∗ **FROM** …)

    - Scalar expression
      - Must return single tuple, usually containing a single attribute

0.13 ∗ (
SELECT sum(value)
FROM Sales
WHERE Taxable
)

S.value > (
SELECT avg(S.value)
FROM Sales S
)

  - List comparisons compares a value against many others
    - List of literals, result of nested query
    - Let OP be any comparator (>, <=, !=, etc.)
      - ANY → there exists; ALL → for each
      - X **OP ANY**(A, B, C) = X **OP** A **OR** X **OP** B **OR** X **OP** C
      - X **OP ALL**(A, B, C) = X **OP** A **AND** X **OP** B **AND** X **OP** C
    - IN is shorthand for = **ANY**
      - X **IN** (A, B, C) ⇔ X = **ANY**(A, B, C)
    - NOT IN is a shorthand for <> **ALL**
      - X **NOT IN** (A, B, C) ⇔ X <> **ALL**(A, B, C)
  - EXISTS Operator checks whether a subquery returns result
  - Tuple Constructors
    - The comparison w/i a nested query may involved several attributes bundled into a tuple
    - A tuple constructor is represented in terms of a pair of angle brackets

- Nested Query examples
  - **Query 1**: Find the names of employees who work in departments in London
    - Solution 1 w/ nested query:
      ```
      SELECT FirstName, Surname
      FROM Employee
      WHERE Dept = ANY (
              SELECT DeptName
              FROM Department
              WHERE City = "London")
      ```
    - Solution 2 w/ IN operator
      ```
      SELECT FirstName, Surname
      FROM Employee
      WHERE Dept IN (
              SELECT DeptName
              FROM Department
              WHERE City = "London")
      ```
    - Solution 3 w/o nested query:
      ```
      SELECT FirstName, Surname
      FROM Employee, Department D
      WHERE Dept = DeptName AND D.City = "London"
      ```
  - **Query 2**: Find employees of the Planning department, having the same first name as member of the Product department
    - Solution 1 w/ nested query:
      ```
      SELECT FirstName, Surname
      FROM Employee
      WHERE Dept = "PLAN" AND FirstName = ANY (
              SELECT FirstName
              FROM Employee
              WHERE Dept = "Prod")
      ```
    - Solution 2 w/ nested query:
      ```
      SELECT FirstName, Surname
      FROM Employee
      WHERE Dept = "PLAN" AND FirstName IN (
              SELECT FirstName
              FROM Employee
              WHERE Dept = "Prod")
      ```
    - Solution 3 w/o nested query:
      ```
      SELECT FirstName, Surname
      FROM Employee, Department D
      WHERE Dept = DeptName AND D.City = "London"
      ```
  - **Query 3**: Find departments where there is no employee named Brown
    - Solution 1 w/ nested query:
      ```
      SELECT DeptName
      FROM Department
      WHERE DeptName <> ALL (
              SELECT Dept
              FROM Employee
              WHERE Surname = "Brown")
      ```
    - Solution 2 w/ NOT IN operator
      ```
      SELECT DeptName
      FROM Department
      WHERE DeptName NOT IN (
              SELECT Dept
              FROM Employee
              WHERE Surname = "Brown")
      ```
    - Solution 3 w/o nested query:
      ```
      (SELECT DeptName
      FROM Department)
      EXCEPT
      (SELECT Dept
      FROM Employee
      WHERE Surname = "Brown")
      ```
  - **Query 4**: Find the department of the employee earning the highest salary
    - Solution 1 w/ max
      ```
      SELECT Dept
      FROM Employee
      WHERE Salary IN (
              SELECT max(Salary)
              FROM Employee)
      ```
    - Solution 2 w/o max
      ```
      SELECT Dept
      FROM Employee
      WHERE Salary >= ALL (
              SELECT Salary
              FROM Employee)
      ```
  - **Query 5**: Find all persons who have the same first name and surname with someone else (synonymous folds) but different tax codes"
    - Solution w/ **EXISTS** operator
      ```
      SELECT *
      FROM Person P
      WHERE EXISTS (
              SELECT *
              FROM Person P1
              WHERE P1.FirstName = P.FirstName
                      AND P1.Surname = P1.Surname
                      AND P1.TaxCode <> P.TaxCode)
      ```
  - **Query 6**: Find all persons who have no synonymous persons
    - Solution 1 w/ **NOT EXISTS** operator
      ```
      SELECT *
      FROM Person P
      WHERE NOT EXISTS (
              SELECT *
              FROM Person P1
              WHERE P1.FirstName = P.FirstName
                      AND P1.Surname = P1.Surname
                      AND P1.TaxCode <> P.TaxCode)
      ```
    - Solution 2 w/ **tuple constructors**
      ```
      SELECT *
      FROM Person P
      WHERE < FirstName, Surname > NOT IN (
              SELECT FirstName, Surname
              FROM Person P1
              WHERE P1.TaxCode <> P.TaxCode)
      ```
- Comments on Nested Queries
  - Use of nesting
    - Produce less declarative queries
    - Often results in imporved readability
  - Complex queries can be difficult to understand
  - The use of variables must respect scoping conventions
    - A variable can be used only within the query where it is defined, OR within a query that is recursively nested within the query where it is defined

DATA MANIPULATION LANGUAGE (DML)

- Modification Statements → operate on set of tuples (no duplicates)
- Example Database Schemas
  `Employee`(`FirstName`, `Surname`, `Dept`, `Office`, `Salary`, `City`)
  `Department`(`DeptName`, `Address`, `City`)
  `Product`(`Code`, `Name`, `Description`, `ProdArea`)
  `LondonProduct`(`Code`, `Name`, `Description`)

- **INSERT** inserts a tuple into a table
  - Syntax variations
    - 1. Using only **Values** (insert only 1 tuple)
      - # of Values must match the # of Attributes
      - Ordering of Values matches the ordering of Attributes

      INSERT INTO ⟨$TableName$⟩
      VALUES (⟨val_1⟩, ⟨val_2⟩, … , ⟨val_n⟩)

    - 2. Using both **Attribute Names** and **Values** (insert only 1 tuple)
      - Inserted tuple has default (if defined) or NULL value for unfilled attributes

      INSERT INTO ⟨$TableName$⟩(⟨$attr_1$⟩, ⟨$attr_2$⟩, … , ⟨$attr_n$⟩)
      VALUES (⟨val_1⟩, ⟨val_3⟩ … , ⟨val_n⟩)

    - 3. Using a **Subquery**
      - Subquery values must match all Attributes

      INSERT INTO ⟨TableName⟩
      (**SELECT** …
      FROM …
      WHERE …)

  - INSERT examples
    - Insert a new Department

      INSERT INTO Department
      VALUES ("Production", "Rue du Louvre 23", "Toulouse")

    - Insert a new Department without an Address value

      INSERT INTO Department(DeptName, City)
      VALUES ("Production", "Toulouse")

    - Insert Tuples from Product that are located in London

      INSERT INTO LondonProducts
      (**SELECT** Code, Name, Description
      FROM Product
      WHERE ProdArea = "London")

- **DELETE** deletes tuples from a table
  - Synatx variations
    - 1. Delete all tuples from the table, schema is kept

      DELETE FROM ⟨$TableName$⟩

    - 2. Delete tuples from a table that satisfy a condition specified by the WHERE clause

      DELETE FROM ⟨$TableName$⟩
      WHERE ⟨$condition$⟩

  - DELETE Examples
    - Remove the Production department

      DELETE FROM Department
      WHERE DeptName = "Production"

    - Remove departments with no employees

      DELETE FROM Department
      WHERE DeptName **NOT IN** (
      SELECT Dept
      FROM Employees)

  - Note: removal may produce deletions for other tables
  - To remove a table completed:

    DROP TABLE ⟨$TableName$⟩ **CASCADE**

- UPDATE update a tuple in a table
  - Syntax variations
    - 1. Update the values of tuples under a specified attribute from a table that satisfy a condition specified by the WHERE clause
      - ⟨$expression$⟩ must evaluate to an accepted value in the attribute, or can be NULL or DEFAULT

      UPDATE ⟨$TableName$⟩
      SET ⟨$attr_1$⟩ = ⟨$expression$⟩
      WHERE ⟨$condition$⟩

    - 2. Multiple attributes

      UPDATE ⟨$TableName$⟩
      SET ⟨$attr_1$⟩ = ⟨$expression$⟩,
      ⟨$attr_2$⟩ = ⟨$expression$⟩,
      …
      ⟨$attr_3$⟩ = ⟨$expression$⟩
      WHERE ⟨$condition$⟩

  - Note: order of updates is important b/c the updates to the table happen right after SQL statement is evaluated

DATA DEFINITION LANGUAGE (DDL)

- Creating (Declaring) a Schema
  - Schema contains tables, data types, functions, etc.
  - Syntax:
    - AUTHORIZATION specifies the owner of the schema

    CREATE SCHEMA ⟨$SchemaName$⟩
    AUTHORIZATION ⟨$UserName$⟩

- Declaring Relation/Table
  - Create a Table

    CREATE TABLE ⟨$TableName$⟩ (
    ⟨$list\ of\ elements$⟩
)

  - Delete a Table

    DROP TABLE ⟨$TableName$⟩ **CASCADE**

  - Alter a table by adding/removing elements (columns)

    ALTER TABLE ⟨$TableName$⟩ **ADD** ⟨$ElementName$⟩
    ALTER TABLE ⟨$TableName$⟩ **ADD** ⟨$ElementName$⟩

  - Declaring Elements of a Table
    - Syntax: ⟨$ElementName$⟩ ⟨$ElementType$⟩ ⟨$constraints$⟩
      - See later for details on constraints
    - Example

      CREATE TABLE ⟨$TableName$⟩ (
      id INTEGER,
      first_name CHAR(20),
      last_name VARCHAR(100)
      );

  - Common types:
    - **INT** or **INTEGER** = an integer value
    - **REAL** or **FLOAT** = a floating point value
    - **CHAR(n)** = fixed-length string of **n** chars
    - **VARCHAR(n)** = variable-length string of up to **n** chars
    - **DATE 'yyyy-mm-dd'** = date value of specified format
      - ex. For 19 Oct 2011 → DATE '2011-10-19'
    - **TIME 'hh:mm:ss'** = time value, accept demical for seconds
      - ex. For 6:40PM → TIME 18:40:02.5'

- Key Constraints
  - **Keys**, for referential-integrity –i.e. Primary Key
    - Enforce no two tuples of the relation have the same value
    - Syntax for one key: use PRIMARY KEY or UNIQUE

      $\langle EleName \rangle \langle EleType \rangle$ **PRIMARY KEY**

    - Syntax for multiple keys

      $\langle EleName_1 \rangle \langle EleType \rangle$,
      $\langle EleName_2 \rangle \langle EleType \rangle$,
      PRIMARY KEY$(EleName_1, EleName_2)$

    - Note: PRIMARY KEY and UNIQUE are different constraints, no attribute of PRIMARY KEY can be NULL; UNIQUE attributes can be NULL
    - Examples.

```
CREATE TABLE Beers (
    name    CHAR(20)    PRIMARY KEY,
    manf    CHAR(20));

CREATE TABLE Sells (
    bar     CHAR(20),
    beer    VARCHAR(20),
    price   REAL,
    PRIMARY KEY(bar, beer));
```

  - **Foreign-key**, enforces that values that appear in attributes of one relation must appear in certain attributes of another relation
    - Referenced attributes must be PRIMARY KEY or UNIQUE
    - Syntax for one-attribute keys:

      $\langle EleName \rangle \langle EleType \rangle$ **REFERENCES** $\langle relation \rangle (\langle attribute \rangle)$,

    - Syntax for multiple attributes

      $\langle EleName_1 \rangle \langle EleType \rangle$,
      $\langle EleName_2 \rangle \langle EleType \rangle$,
      FOREIGN KEY $(\langle EleName_1 \rangle, \langle EleName_2 \rangle)$
      REFERENCES $\langle relation \rangle (\langle attr_1 \rangle, \langle attr_2 \rangle)$

    - Example: suppose a value in Sells.beer must appear as value in Beers.name, then (both versions of Sells is valid)

```
CREATE TABLE Beers (
    name    CHAR(20)    PRIMARY KEY,
    manf    CHAR(20));

CREATE TABLE Sells (
    bar CHAR(20),
    beer VARCHAR(20) REFERENCES Beers(name),
    price REAL);
                    -- OR --
CREATE TABLE Sells (
    bar CHAR(20),
    beer VARCHAR(20),
    price REAL,
    FOREIGN KEY (beer ) REFERENCES Beers(name));
```

  - Enforcing Foreign-Key Constraints
    - Possible for INSERT, UPDATE, DELETE to violate foreign-key constraints, given Sells and Beers were Sells.beer references Beers.name
      - Ex. INSERT or UPDATE into Sells that introduce a value for Sells.beer that's non-existent in Beers.name
      - Ex. DELETE or UPDATE to Beers that removes a tuple with Beers.name being referenced by tuples in Sells.
    - **DEFAULT**: Reject the Modification
      - DELETE/UPDATE referenced tupled in Beer → reject modifications in Sells tuples
    - **CASCADE**: Make the same changes in Sells
      - DELETE referenced beer in Beer → delete Sells tuples
      - UPDATE referenced beer in Beer → change values in Sells
    - **SET NULL**: Change the beer to NULL
      - DELETE/UPDATE referenced tupled in Beer → set NULL values in Sells tuples
    - Ex. Delete "Bud" tuple from Beers
      - DEFAULT: Sells unaffected
      - CASCADE: Delete tuples of Sells with beer = "Bud"
      - SET NULL: Change tuples of Sells with beer = "Bud" to NULL
    - Syntax for one-attribute keys:

```
CREATE TABLE ⟨TableName⟩ (
    ele1 CHAR(20) REFERENCES ⟨relation⟩(⟨attribute⟩)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
    ele2 CHAR(20)
);
```

    - Syntax for multiple attributes

```
CREATE TABLE ⟨TableName⟩ (
    ele1        CHAR(20),
    ele2        VARCHAR(20),
    FOREIGN KEY (⟨ele1⟩, ⟨ele2⟩)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
);
```

  - Other Constraints
    - **Attributes-based constraints**
      - The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery
      - Checks are performed only when a value for that attribute is INSERTED or UPDATED (unlike REFERENCES)
      - Syntax: $\langle EleName \rangle \langle EleType \rangle$ **CHECK**$(\langle Condition \rangle)$
      - Ex.

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20)    CHECK (beer IN(SELECT name
                                       FROM Beers),
    price   REAL        CHECK (price ≤ 5.00)
);
```

      - Special Case: **Not Null**, syntax:
        $\langle EleName \rangle \langle EleType \rangle$ **NOT NULL**

    - **Tuple-Base constraints**
      - CHECK may be added as a relation-schema element
      - The condition may refer to any attribute of the relation, but other attributes or relation require a subquery
      - Checks are preformed on INSERT or UPDATE
      - Ex. Only Joe's Bar can sell beer for more than $5

```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    CHAR(20),
    beer    REAL,
    CHECK (bar = "Joe's Bar" OR price ≤ 5.00));
```

    - **Assertions**
      - Permit definition of constraints over whole tables, beyond individual tuples
      - Declared when the schema is declared
      - Syntax: **CREATE ASSERTION** ⟨name⟩ **CHECK**(⟨cond⟩)
      - Ex. enforce at least one tuple in table Employee

```
CREATE ASSERTION AlwaysOneEmployee
CHECK (1 ≤ (
    SELECT count(*)
    FROM Employee ))
```

    - Integrity checks may occur immediately when a change takes place, or at the end of a transcation
      - Example: commit or roll back, respectively

- Declaring Views
  - **View** = relation defined in terms of stored tables (base tables) and views
    - **Virtual View** = not stored in database, just a query for constructing the relation, syntax:
      CREATE VIEW ⟨name⟩ **AS** ⟨query⟩
    - **Materialized** View = actually constructed and stored, syntax:
      CREATE MATERIALIZED VIEW ⟨name⟩ **AS** ⟨query⟩
    - Allows Data Independence = hide schema from apps
      - Ex. split CustomerInfo into Customer and Address base tables
    - Encourage Data Hiding = access data on need-to-know basis
    - Code Reuse, Readability and Correctness
      - Similar subqueries are used multiple times in a query
  - Example Database Schemas
    `Beers(Name, Manf)`
    `Bars(Name, Addr, License)`
    `Drinkers(Name, Addr, Phone)`
    `Likes(drinker, Beer)`
    `Sells(Bar, Beer, Price)`
    `Frequents(drinker, bar)`
    - Ex. **View** "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS (
    SELECT drinker, beer
    FROM Frequents, Sells
    WHERE Frequents.bar = Sells.bar
);
```

    - Ex. Accessing a View (as if it were a base table)
      - Ex. find all the beers drank by Sally

```
SELECT beer
FROM CanDrink
WHERE drinker = "Sally"
```

  - Example Database Schema
    `Employee(ReNo, FirstName, Surname, Dept, Office, Salary, City)`
    `Department(DeptName, Address, City)`
    - Ex. Find the departments w/ highest salaray expenditures (w/o using views)

```
SELECT Dept
FROM Employee
GROUP BY Dept
HAVING sum(Salary) ≥ ALL (
    SELECT sum(Salary)
    FROM Employee
    GROUP BY Dept
)
```

    - Ex. Find the departments w/ highest salaray expenditures (using views)

```
CREATE VIEW SalBudget(Dept, SalTotal) AS (
    SELECT Dept, sum(Salary)
    FROM Employee
    GROUP BY Dept);

SELECT Dept
FROM SaleBudget
WHERE SalTotal = (
    SELECT max(SalTotal)
    FROM SaleBudget)
```

  - Updating on Views
    - Can't modify a virtual view b/c doesn't exist
    - Each time a base table changes, the materailzied view may change
      Need: periodic reconstruction of materialized view

- **Index** = auxiliary data base structures which provides random access to data
  - Problem: WHERE clauses need to find needle in haystack
    - Ex. All phone names w/ first name "Mary"
  - INDEX indexes a set of attributes, allows quick random access to any indexed attributes
    - Similar to hash table, stored as balanced serach tree (B-tree)
  - Syntax (Non-standard):
    CREATE INDEX ⟨name⟩ **ON** ⟨table⟩(⟨$attr_1$⟩, ⟨$attr_2$⟩, … , ⟨$attr_n$⟩)
  - Ex. using INDEX on single attribute of a table

```
SELECT fname
FROM people
WHERE lname = "Papgelis"
```

    - Without an index → DBMS look at lname column on every row in table (full table scan)
    - With an index → DBMS follows B-tree data structure until desired rows are found (less computationally expensive)
  - Ex. Create Indexes on multiple tables and attributes;
    - Query: find the prices of beers manufactured by Pete's and solf by Joe's bar

```
CREATE INDEX BeerIndex ON Beers(manf);
CREATE INDEX SellIndex ON Sells(bar, beer);

SELECT prices
FROM Beers, Sells
WHERE manf = "Pete's"
    AND Beers.name = Sells.beer
    AND bar = "Joe's Bar"
```

    - Result DBMS uses
      - 1. BeerIndex to get all Pete's Beers fast
      - 2. SellIndex to get Prices of beers, and bars fast
  - Database Tuning
    - Index slows down all modifications on its relation as the index must be modified with INSERT/UPDATE/DELETE
      - Want: make common case fast
    - Suppose the database load was:
      - Insert new facts into a relation = 10% queries
      - Find the price of given beer at a given bar = 90% queries
      - Result: SellIndex would be beneficial (common case)
      - Result: BeerIndex would be harmful (edge case)
    - Use Advisors,
      - Gets a query load by random queries from history of queries ran, or from sample workload
      - Generates candidate indexes

Non-interactive SQL

Introduction to Databases

Manos Papagelis

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

Using a Database

- Interactive SQL: Statements typed in from terminal; DBMS outputs to screen. Interactive SQL is inadequate in many situations:
  - It may be necessary to process the data before output
  - Amount of data returned not known in advance
- Non-interactive SQL: Statements included in an application program written in a host language — such as C, Java, PHP, …

Non-interactive SQL

- Traditional applications often need to "embed" SQL statements inside the instructions of a program written in a procedural programming language (C, JAVA, etc.)
- There is a severe problem (impedance mismatch) between the computational model of a programming language (PL) and that of a DBMS:
  - The variables of a PL take as values single records, those of SQL whole tables
  - PL computations are generally on a main memory data structure, SQL ones on bulk data

The best of both worlds

- Host language
  - A conventional programming language (e.g., C, Java) that supplies control structures, computational capabilities, interaction with physical devices, …
- SQL
  - supplies ability to interact with database
- Non-interactive SQL
  - the application program can act as an intermediary between the user at a terminal and the DBMS

Elements of Non-interactive SQL

- Non-interactive SQL may use a pre-compiler to manage SQL statements
- Program variables may be used as parameters in the SQL statements (variable interchange)
- Results may be
  - a single row (easy to handle)
  - sets of rows (tricky to handle)
- Execution status
  - predefined variable sqlstate (="00000" if executed successfully).

SQL Statement Preparation

- Before any SQL statement is executed, it must be prepared by the DBMS:
  - What indices can be used?
  - In what order should tables be accessed?
  - What constraints should be checked?
- Decisions are based on schema, table size, etc.
  - Result is a query execution plan

Non-interactive SQL Approaches

- In the DBMS
  - Persistent Stored Modules (PSM):
    Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL)
- Out of the DBMS
  - Statement-level Interface (SLI):
    SQL statements are embedded in a host language (e.g., C)
  - Call-level Interface (CLI):
    Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB)

PERSISTENT STORED MODULES (PSM)

Persistent Stored Procedures

- Allow to store procedures as database schema
- A mixture of conventional statements (if, while, etc.) and SQL
- Allow do things we cannot do in SQL alone
- Most DBMSs offer SQL extensions that support persistent stored procedures:
  - PostgreSQL: PL/pgPSM
  - Oracle: PL/SQL
  - …

Basic PSM Form

CREATE PROCEDURE <name> (
    <parameter list>)
<optional local declarations>
<body>;

Function alternative:
CREATE FUNCTION <name> (
    <parameter list>) **RETURNS** <type>

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the mode can be:
  - IN = procedure uses value, does not change value
  - OUT = procedure changes value, does not use value
  - INOUT = both

Example

- Write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells(bar, beer, price) that has bar = 'Joe''s Bar', beer = *b*, and price = *p*
  - Used by Joe to add to his menu more easily.

```
CREATE PROCEDURE JoeMenu (
        IN    b    CHAR(20),
        IN    p    REAL
)
INSERT INTO Sells
VALUES ('Joe''s Bar', b, p);
```

Parameters are both read-only, not changed

The body is a single insertion

Invoking Procedures

- Use SQL/PSM statement **CALL**, with the name of the desired procedure and arguments.

```
CALL JoeMenu('Moosedrool', 5.00);
```

Advantages of Stored Procedures

- Intermediate data need not be communicated to application (time and cost savings)
- Procedure's SQL statements prepared in advance
- Authorization can be done at procedure level
- Added security since procedure resides in server
- Applications that call the procedure need not know the details of database schema

STATEMENT-LEVEL INTERFACE (SLI)

Statement Level Interface

- SQL statements and directives in the application have a special syntax that sets them off from host language constructs

  e.g., EXEC SQL SQL_statement

- Pre-compiler scans program and translates SQL statements into calls to host language library procedures that communicate with DBMS
- Host language compiler then compiles program

Static vs Dynamic Embedding

- SQL constructs in an application take two forms:
  - Standard SQL statements (static SQL): Useful when SQL portion of program is known at compile time
  - Directives (dynamic SQL): Useful when SQL portion of program not known at compile time; Application constructs SQL statements at run time as values of host language variables that are manipulated by directives
- Pre-compiler translates statements and directives into arguments of calls to library procedures

Example of Static SQL

```
EXEC SQL SELECT C.NumEnrolled
   INTO    :num_enrolled
   FROM    Course C
   WHERE C.CrsCode = :crs_code;
```

- Variables shared by host and SQL (num_enrolled, crs_code)
  - ":" used to set off host variables
  - Names of (host language) variables are contained in SQL statement and available to pre-compiler
- Routines for fetching and storing argument values can be generated
- Complete statement (with parameter values) sent to DBMS when statement is executed

19

Example of Dynamic SQL

```
strcpy (tmp, "SELECT  C.NumEnrolled FROM Course C
                WHERE C.CrsCode = ?" ) ;
EXEC SQL PREPARE st FROM :tmp;
EXEC SQL EXECUTE st INTO :num_enrolled USING :crs_code;
```

- **st** is an SQL variable; names the SQL statement
- **tmp**, **crs_code**, **num_enrolled** are host language variables (note colon notation)
- **crs_code** is an IN parameter; supplies value for placeholder (**?**)
- **num_enrolled** is an OUT parameter; receives value from C.NumEnrolled

20

CALL-LEVEL INTERFACE (CLI)

21

Call Level Interface

- Application program written entirely in host language (no precompiler) using library calls
  - Java + JDBC
  - PHP + PEAR/DB
- SQL statements are values of string variables constructed at run time using host language
  - similar to dynamic SQL
- Application uses string variables as arguments of library routines that communicate with DBMS

  e.g. executeQuery("SQL query statement")

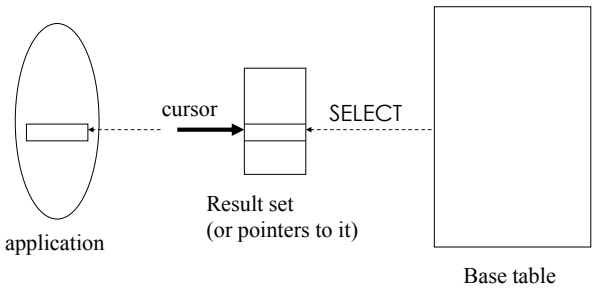
22

Cursors

- Fundamental problem with database technology: *impedance mismatch*
  - traditional programming languages process records one-at-a-time (tuple-oriented)
  - SQL processes tuple sets (set-oriented).
- *Cursors* solve this problem: A cursor returns tuples from a result set, to be processed one-by-one

23

How Cursors Work?



24

Operations on Cursors

- Result set: rows returned by a SELECT statement
- To execute the query associated with a cursor:

  open CursorName
- To extract one tuple from the query result:

  fetch [Position from] CursorName into FetchList
- To free the cursor, discarding the query result:

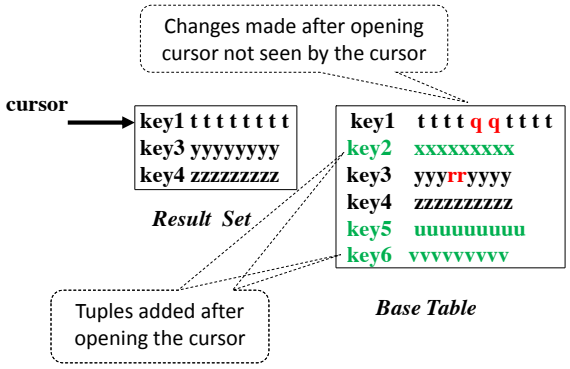
  close CursorName
- To access the current tuple (when a cursor reads a relation, in order to update it):

  current of CursorName (in a where clause)

Cursor Types

- Insensitive cursors: Result set computed and stored in separate table at OPEN time
  - Changes made to base table subsequent to OPEN (by any transaction) do not affect result set
  - Cursor is read-only
- Sensitive cursors: Specification not part of SQL standard
  - Changes made to base table subsequent to OPEN (by any transaction) can affect result set
  - Cursor is updatable

Insensitive Cursor



Cursor Scrolling

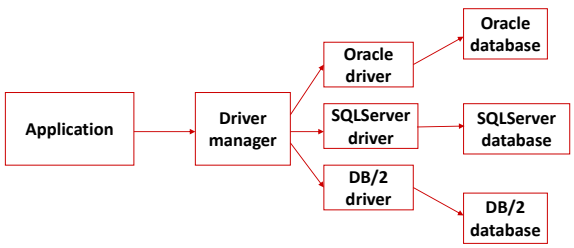
- If SCROLL option is not specified in cursor declaration, FETCH always moves cursor forward one position
- If SCROLL option is included in cursor declaration, cursor can be moved in arbitrary ways around result set (e.g., FIRST, LAST, ABSOLUTE n, RELATIVE n)

JAVA: JDBC

JDBC

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS
  - Can be interfaced to any DBMS that has a JDBC driver
- Part of SQL:2003 Standard

JDBC Run-Time Architecture



Making a Connection

```
// Importing JDBC
import java.sql.*
//load the driver for PostgreSQL
Class.forName("org.postgresql.Driver");
//connect to the db
Connection conn =
        DriverManager.getConnection(url, user, passwd);
//disconnect
conn.close();
```

Processing a Simple Query in JDBC

```
// Create a Statement
Statement st = conn.createStatement();
//Execute Statement and obtain ResultSet
ResultSet rs = st.executeQuery("SELECT * FROM mytable WHERE
   columnfoo = 500");
// Process the Results
while (rs.next()) {
   System.out.println(rs.getString(1));
}
// Close ResultSet and Statement
rs.close();
st.close();
```

Same, but using PreparedStatement

```
int foovalue = 500;
// Prepare Statement
PreparedStatement ps = conn.prepareStatement("SELECT * FROM
   mytable WHERE columnfoo = ?");
// Set value of in-parameter
ps.setInt(1, foovalue);
// Execute Statement and obtain ResultSet
ResultSet rs = ps.executeQuery();
// Process the Results
while (rs.next()) {System.out.println(rs.getString(1));}
// Close ResultSet and PreparedStatement
rs.close();ps.close();
```

placeholder

Advantages of PreparedStatements

- Performance:
  The overhead of compiling and optimizing the statement is incurred only once, although the statement is executed multiple times
- Security:
  Resilient against SQL injection (see next)

Result Sets and Cursors

- Three types of result sets in JDBC:
  - *Forward-only*: not scrollable
  - *Scroll-insensitive*: scrollable; changes made to underlying tables after the creation of the result set are not visible through that result set
  - *Scroll-sensitive*: scrollable; updates and deletes made to tuples in the underlying tables after the creation of the result set are visible through the result set

Result Set

```
Statement stat = con.createStatement (
   ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE
);
```

- Concurrency mode of ResultSet (read-only/updatable cursor):
  - CONCUR_READ_ONLY
  - CONCUR_UPDATABLE
- Type of ResultSet (cursor operations allowed):
  - TYPE_FORWARD_ONLY
  - TYPE_SCROLL_INSENSITIVE
  - TYPE_SCROLL_SENSITIVE

37

Handling Exceptions

```
try {
    ...Java/JDBC code...
} catch ( SQLException  ex ) {
    ...exception handling code...     }
```

- try/catch is the basic structure within which an SQL statement should be embedded
- If an exception is thrown, an exception object, *ex*, is created and the catch clause is executed
- The exception object has methods to print an error message, return SQLSTATE, etc.

38

Transactions in JDBC

- Default for a connection is autocommit
  - each SQL statement is a transaction
- Group several statements into a Transaction:
  - Set autocommit to false: conn.setAutoCommit (false);
  - Several SQL statements: …UPDATE, UPDATE, INSERT, etc.
  - Commit statements: conn.commit();
  - Set autocommit back to true: conn.setAutoCommit(true);

39

PHP: PEAR DB

40

PHP

- A language to be used for actions within HTML
  - Indicated by <? PHP code ?>
- Basic programming elements:
  - Variables: must begin with $
  - Two kinds of Arrays: numeric and associative
- DB library exists within PEAR (PHP Extension and Application Repository)
  - include with include(DB.php)

41

Making a Connection

- With the DB library imported and the array $myEnv available:

```
$conn = DB::connect($myEnv);
```

Function connect in the DB library

$conn is a Connection returned by DB::connect()

42

Executing SQL Statements

- Method query() applies to a Connection object
- It takes a string argument and returns a result
  - Could be an error code or the relation returned by a query

Ex. Query: "Find all the bars that sell a beer given by the variable $beer."

```
$beer = 'Bud';
$result = $conn->query("SELECT bar FROM Sells WHERE beer =
$beer ;");
```

Cursors in PHP

- The result of a query *is* the tuples returned
- Method fetchRow() applies to the result and returns the next tuple, or FALSE if there is none

```
while ($bar =$result->fetchRow()) {
    // do something with $bar
}
```