Course Notes on SQL

SQL: Summary of Presentation

- Bases of SQL
- Discussion of SQL features through examples
- Criticism of SQL
- Standardization

1

SQL: Introduction

- SQL =
  - ◇ another syntax for expressions of DRC, TRC, and the relational algebra
  - ◇ + a data definition language
  - ◇ + some extensions
- SQL was defined in the mid 70's, after the relational algebra and calculi (early 70's): SQL was meant to be simpler (this is not obvious)
- SQL uses a vocabulary different from that of the usual relational model
  - ◇ "relation" → "table"
  - ◇ "tuple" → "row"
  - ◇ "attribute" → "column"
- A major difference with the relational model: tables can have identical rows in SQL

2

SQL Features: Summary

- Data definition language
- General form of queries
- Join
- Variables
- Redundancy, variants
- Duplicate tuples
- Functions
- Group by ... having
- Evaluation, claimed advantages, criticisms of SQL
- Standardization

3

Data Definition: Relation Schema Declaration

```
CREATE TABLE
  relation name
  list of attributes + data types
  each attribute can be declared "not null"
  PRIMARY KEY attribute(s)
  UNIQUE attribute(s)
  FOREIGN KEY attribute REFERENCES
      relation name (attribute)
```

4

Relation Schema Declaration: Example

```
CREATE TABLE Employee (
  SSN     CHAR(9)       NOT NULL,
  FName   VARCHAR(15)   NOT NULL,
  MInit   CHAR,
  LName   VARCHAR(15)   NOT NULL,
  BDate   DATE,
  Address VARCHAR(30),
  Sex     CHAR,
  SALARY  DECIMAL(10,2),
  SuperSSN CHAR(9),
  DNo     INT           NOT NULL,
  PRIMARY KEY (SSN)
  FOREIGN KEY (SUPERSSN) REFERENCES Employee(SSN)
  FOREIGN KEY (DNo) REFERENCES Department(DNumber) );
```

5

Other Data Definition Commands

- Relation suppression: `DROP TABLE relation name`

- Relation structure modification

  ```
  ALTER TABLE relation name
        ADD attribute + data type
  ALTER TABLE relation name
        DROP attribute
  ```

- View definition

  ```
  CREATE VIEW relation name + list of attributes
      AS SQL query
  ```

6

- Evaluation of SQL DDL:

  ◇ Any operational DBMS needs a DDL language

  ◇ Compared with relational query languages, the operations and structure of a relational DDL are simple and leave little room for variations

  ◇ The "SQL DDL" borrows little from the query part of SQL

General Form of SQL Queries

```
SELECT attributes
FROM relations
WHERE condition
```

- This is just a skeleton, with many variants
- Give the birthdate and address of employees named John Smith

  ```
  SELECT BDate, Address
  FROM Employee
  WHERE FName = 'John' and LName = 'Smith'
  ```

- Semantics
  (1) evaluate the Cartesian product of relations mentioned in FROM clause
  (2) apply the condition (algebra selection)
  (3) retain attributes mentioned in SELECT clause (projection **with** duplicates)

- Compare with the structure of the same query in TRC:

  ◇ $\{t.\text{BDate}, t.\text{Address} \mid \text{Employee}(t) \wedge t.\text{FName} = \text{John} \wedge t.\text{LName} = \text{'Smith'}\}$
  ◇ What is the major difference between SQL and TRC for this query?

A Simple Syntactic Shortcut

Give the data about employees named John Smith

```
SELECT *
FROM Employee
WHERE FName = 'John' and LName = 'Smith'
```

- Not a good idea in a multi-user database

  ◇ The number of attributes of the tuples in the answer will change if the relation schema of Employee changes (if attributes are added or deleted), without the query having changed: this is a violation of logical data independence
  ◇ The text of the query does not make explicit the structure of the result (lack of self-documentation)

Join

(1) Give name and address of employees who work in the Research department

```
SELECT LName, Address
FROM Employee, Department
WHERE DName = 'Research' and DNumber = DNo
```

(2) For each project located in Brussels, list project number, number of controling department, and name and address of department manager

```
SELECT PNumber, DNum, LName, Address
FROM Project, Department, Employee
WHERE DNum = DNumber and MgrSSN = SSN
        AND PLocation = 'Brussels'
```

- DNumber = DNo (join condition) in (1) signals a join in SQL, between a relation with attribute DNumber (Department) and a relation with attribute DNo (Employee)

- Example (2) has two joins (and thus two join conditions)

Relation Names Used as Variables

Name and address of all employees who work in the Research department

(1)
```
SELECT LName, Address
FROM Employee, Department
WHERE DName = 'Research' and DNumber = DNo
```

(2)
```
SELECT LName, Address
FROM Employee, Department
WHERE DName = 'Research'
   and Department.DNumber = Employee.DNumber
```

(3)
```
SELECT Employee.LName, Employee.Address
FROM Employee, Department
WHERE Department.DName = 'Research'
   and Department.DNumber = Employee.DNumber
```

- Join condition DNumber = DNo in (1) is clearly decoded : DNumber and DNo are unique in relations Employee and Department

- If the attribute name DNumber is chosen in both relations to reference the domain of departments, then SQL must formulate the query as (2)

- (3) is a more general and systematic formulation

- Relation names can thus be used as variables that are quite similar to the tuple variables of TRC

- Strategies for choosing attribute names during database design

  ◇ use the same name (e.g., DNumber) for the same objects appearing in different relations

  ◇ leave freedom for attribute names (e.g., if the database schema is large and not designed by a single person)

Variables in SQL

Give the name of employees and the name of their supervisor

```
SELECT E.LName,S.LName
FROM Employee E, Employee S
WHERE E.SuperSSN = S.SSN
```

Give the name and address of employees who work in the Research department

(1)
```
   SELECT E.LName, E.Address
   FROM Employee E, Department D
   WHERE D.DName = 'Research'
     and D.DNumber = E.DNo
```

(2) $\{e.\text{LName}, e.\text{Address} \mid \text{Employee}(e) \land$
$\exists d \, (\text{Department}(d) \land d.\text{DName} = \text{`Research'} \land d.\text{DNumber} = e.\text{DNo})\}$

11

- The first SQL query above is the general form concerning the use of variables, of which the previous examples are special cases
- When the same relation is invoked more than once in the same query, then SQL uses variables exactly like TRC

  ◇ (1) is very similar to TRC, except that D is explicitly existentially quantified in TRC
  ◇ (2) is the TRC formulation

- SQL designers tried to avoid variables as much as possible

  ◇ the resulting language offers special simpler formulations for simple queries
  ◇ but the language definitions has become bigger (because of the need to describe the special cases and the general case)
  ◇ the question is for which kind of users that decision represents a real simplication (a difficult issue for language design in general)

Duplicate Control

- List salaries of employees

```
SELECT Salary    SELECT DISTINCT Salary
FROM Employee    FROM Employee
```

| 30000 |
| 40000 |
| 25000 |
| 43000 |
| 38000 |
| 25000 |
| 25000 |
| 55000 |

| 30000 |
| 40000 |
| 25000 |
| 43000 |
| 38000 |
| 55000 |

- Unorthodox, unclear semantics in general

12

EXISTS

Name and address of employees who work in the Research department

```
SELECT E.FName, E.LName
FROM Employee E
WHERE EXISTS (SELECT *
                FROM Department D
                WHERE D.DNumber = E.DNo
                  AND D.DName = 'Research')
```

13

- Another way to express a join

- "Exists (...)" is closer to a test "$\{...\} \neq \emptyset$" (i.e., the test that the result of Exists is not the empty set) than to the existential quantification of logic

---

EXISTS and NOT EXISTS

(1) List the name of employees with at least one dependent

```
SELECT FName, LName
FROM Employee
WHERE EXISTS (SELECT *
              FROM Dependent
              WHERE SSN = ESSN)
```

(2) List the name of employees with no dependent

```
SELECT FName, LName
FROM Employee
WHERE NOT EXISTS (SELECT *
                  FROM Dependent
                  WHERE SSN = ESSN)
```

- Here, the intuition of (1) extends well to the negated query (2)

Union

List project names of projects for which an employee whose last name is Smith is a worker or a manager of the department that controls the project

```
(SELECT PName
 FROM Employee, Project, Department
 WHERE DNum = DNumber AND MgrSSN = SSN AND LName = 'Smith')

   UNION

(SELECT PName
 FROM Employee, Project, WorksOn
 WHERE PNumber = PNo AND ESSN = SSN AND LName = 'Smith')
```

Same Example with Disjunction

```
SELECT DISTINCT PName
FROM Project
WHERE PNumber IN (SELECT PNumber
          FROM Project, Department, Employee
          WHERE DNum = DNumber AND MgrSSN = SSN
              AND LName = 'Smith')
      OR
      PNumber IN (SELECT PNo
          FROM WorksOn, Employee
          WHERE ESSN = SSN AND LName = 'Smith')
```

- The equivalence is based on a classical rule from logic and set theory:

  $\{x \mid P(x) \lor Q(x)\} \equiv \{x \mid P(x)\} \cup \{x \mid Q(x)\}$

- The fact that two different formulations exist for the same query (with union and with disjunction) illustrates the mixed origins of SQL, that borrows from both the relational calculi and the relational algebra

- These examples llustrate "nested queries" in SQL

- SQL also has UNION ALL to retain duplicates (What does that really mean?)

- Union is an ancient mathematical operation defined for SETS, and not tables with duplicate rows

---

Variants

List employees with a dependent of sex = F

```
(1) SELECT SSN
    FROM  Employee, Dependent
    WHERE ESSN = SSN and Dependent.Sex = F
(2) SELECT SSN
    FROM Employee
    WHERE EXISTS (SELECT *
                  FROM Dependent
                  WHERE ESSN = SSN AND Sex = F)
(3) SELECT SSN
    FROM Employee
    WHERE SSN IN  ( SELECT ESSN
                    FROM Dependent
                    WHERE Sex = F )
```

17

- TRC formulation of the same query:

  $\{e.SSN \mid Employee(e) \land \exists d \in Dependent(e.SSN = d.ESSN \land d.Sex = F)\}$

- Very different perceptions often exist in SQL for the same query

- This may not be a good idea, as learning the language gets more complicated

- (1) is a flat TRC-like formulation

- (2) evokes the TRC structure, but, like (3), is really a mixture of tuple-oriented and set-oriented formulations

---

More Examples with Variants

Give the name of each employee with a dependent of same first name and same sex as the employee

```
(1) SELECT E.FName, E.LName
    FROM  Employee E, Dependent D
    WHERE D.ESSN = E.SSN AND E.FName = D.DependentName AND D.Sex = E.Sex
(2) SELECT E.FName, E.LName
    FROM Employee E
    WHERE EXISTS (SELECT *
          FROM Dependent
          WHERE E.SSN = ESSN AND E.FName = DependentName AND E.Sex = Sex)
(3) SELECT E.FName, E.LName
    FROM Employee E
    WHERE E.SSN IN ( SELECT ESSN
                     FROM Dependent
                     WHERE E.FName = DependentName AND E.Sex = Sex)
```

18

- Example of join on 3 attributes

Some Additional Features of SQL

- **Test for null value**

  List the name of employees without a supervisor

  ```
  SELECT FName, LName
  FROM Employee
  WHERE SuperSSN IS NULL
  ```

- **Explicit set of values**

  List the name of employees who work in departments of number 1, 2, or 3

  ```
  SELECT FName, LName
  FROM Employee
  WHERE DNo IN {1,2,3}
  ```

  This requires a triple disjunction in TRC: $DNo = 1 \lor DNo = 2 \lor DNo = 3$

19

More Additional Features

(1) **Pattern matching**

   List the name of employees who live in Brussels

   ```
   SELECT FName, LName
   FROM Employee
   WHERE Address LIKE '%BRUSSELS%'
   ```

   List the name of employees born in the 1950's

   ```
   SELECT FName, LName
   FROM Employee
   WHERE BDate LIKE '_____5_'
   ```

(2) **Output ordering**

   ```
   SELECT FName, LName, Salary
   FROM Employee
   WHERE ...
   ORDER BY Salary
   ```

20

- Pattern matching is useful, but including it in SQL complicates the language

- Ordering the result of a query is obviously useful, but the way it is done in SQL jeopardizes closure:

  ◇ ordering should not be done in a query, since the result is not a relation (true relations are unordered)

  ◇ ordering the tuples in the result of a query should be requested as a function external to the query part of the language

Universal Quantifier: Element-to-Set Comparison

- List names of employees who make more than all employees in the Research department

  ```
  SELECT LName
  FROM Employee
  WHERE ( Salary > all
          ( SELECT Salary
            FROM Employee, Department
            WHERE DNo = Dnumber and DName = 'Research') )
  ```

21

Universal Quantifier: Set-to-Set Comparison

- List names of employees who work on all projects controlled by department 5

```
SELECT LName
FROM Employee
WHERE ( ( SELECT PNo
           FROM WorksOn
           WHERE SSN = ESSN )
         CONTAINS
         ( SELECT PNumber
           FROM Project
           WHERE DNum = 5) )
```

- CONTAINS expresses set inclusion, related to universal quantification in logic:

$$\{y \mid \forall x\ P(x) \rightarrow Q(x,y)\} \equiv \{y \mid \{z \mid\ P(z)\} \subseteq \{x \mid\ Q(x,y)\}\}$$

- CONTAINS mirrors the division operator of the relational algebra

- CONTAINS was included in early versions of SQL but was suppressed later

- The general version of the universal quantifier of logic and of the relational calculi is not available in SQL

- There were some attemps to include a more general universal quantifier in earlier versions of SQL

Universal Quantifier: Expressing Division in SQL

R

A	B
a1	b1
a1	b2
a1	b3
a1	b4
a2	b1
a2	b3
a3	b2
a3	b3
a3	b4
a4	b1
a4	b2
a4	b3

S

B
b1
b2
b3

$T = R \div S$

A
a1
a4

```
SELECT DISTINCT A
FROM R R1
WHERE ( SELECT B
          FROM R R2
          WHERE R1.A = R2.A )
        CONTAINS
        ( SELECT *
          FROM S )
```

Universal Quantifier as Negated Existential

```
SELECT FName, LName
FROM Employee
WHERE NOT EXISTS
  ( SELECT *
    FROM Project P
    WHERE DNum = 5
    AND NOT EXISTS
      ( SELECT *
        FROM WorksOn W
        WHERE W.ESSN = SSN AND W.PNo = P.PNumber) )
```

Correspondence between Universal Quantifier and Negated Existential Quantifier in TRC

Basic rule:

$\{y \mid P(x) \rightarrow Q(x,y)\} \equiv \{y \mid \neg \exists x \, (P(x) \land \neg Q(x,y))\}$

$\{e.\text{FName}, e.\text{LName} \mid \text{Employee}(e) \land$
$\quad \forall p \, (\text{Project}(p) \land p.\text{DNum} = 5 \rightarrow$
$\quad\quad \exists w(\text{WorksOn}(w) \land w.\text{ESSN} = e.\text{SSN} \land w.\text{PNo} = p.\text{PNo}) \,) \}$

$\{e.\text{FName}, e.\text{LName} \mid \text{Employee}(e) \land$
$\quad \forall p \, (\neg\text{Project}(p) \lor p.\text{DNum} \neq 5 \ \lor$
$\quad\quad \exists w(\text{WorksOn}(w) \land w.\text{ESSN} = e.\text{SSN} \land w.\text{PNo} = p.\text{PNo}) \,) \}$

$\{e.\text{FName}, e.\text{LName} \mid \text{Employee}(e) \land$
$\quad \neg\neg\forall p \, (\neg\text{Project}(p) \lor p.\text{DNum} \neq 5 \ \lor$
$\quad\quad \exists w(\text{WorksOn}(w) \land w.\text{ESSN} = e.\text{SSN} \land w.\text{PNo} = p.\text{PNo}) \,) \}$

$\{e.\text{FName}, e.\text{LName} \mid \text{Employee}(e) \land$
$\quad \neg\exists p \, (\text{Project}(p) \land p.\text{DNum} = 5 \ \land$
$\quad\quad \neg\exists w(\text{WorksOn}(w) \land w.\text{ESSN} = e.\text{SSN} \land w.\text{PNo} = p.\text{PNo}) \,) \}$

Universal Quantifier as Difference

```
SELECT FName, LName
FROM Employee
WHERE NOT EXISTS (
        ( SELECT *
          FROM Project P
          WHERE DNum = 5)
        EXCEPT
        ( SELECT *
          FROM WorksOn
          WHERE ESSN = SSN) )
```

25

- This formulation is based on the logical equivalence:

  $\{y \mid \forall x \, P(x) \rightarrow Q(x,y)\} \equiv \{ \, \{y \mid \{z \mid P(z)\} - \{x \mid Q(x,y)\} \, \} = \emptyset$

Aggregate Functions

- Idea: specify mathematical **aggregate functions** on collections of values from the database

- **SQL Functions**: count tuples, compute sum, average, maximum, minimum of numeric attribute values in relations
    - ◇ compute the average or total salary of all employees

- Repeated function application on groups of tuples:
    - ◇ for each department, list the name of the department and the average salary of employees in the department

- There is no agreed-upon notation for specifying aggregate functions in the algebra or the calculi

26

Functions in SQL: Summary

- SQL, the algebra, the calculi (and similar nonprocedural languages) are not well adapted for computing

- Some simple cases look OK in SQL, but, as complexity increases, SQL breaks down abruptly

- Moral: computing is best left to algorithmic languages

27

Some Simple Examples with Functions

- Find the average salary of employees and the maximum salary

  ```
  SELECT AVG(Salary), MAX(Salary)
  FROM Employee
  ```

- Find the average salary and the maximum salary for employees in the Research department

  ```
  SELECT AVG(Salary), MAX(Salary)
  FROM Employee, Department
  WHERE DNo = DNumber AND DName = 'Research'
  ```

28

Functions: the SQL Syntax is Inadequate

- Queries have to be read entirely to understand what is actually counted

- Functions should explicitly indicate their argument as follows (illegal in SQL)
  Find the average salary of employees and the maximum salary

  ```
  (AVG, MAX) (SELECT Salary FROM Employee)
  ```

- The following is not expressible simply in SQL
  Find the average salary of all employees (i.e., for the whole company) and the maximum salary of employees in the Research department

  ```
  AVG (SELECT Salary FROM Employee),
  MAX (SELECT Salary
       FROM Employee, Department
       WHERE DNo = DNumber AND DName = 'Research')
  ```

29

Functions: Counting is Tricky

Count the number of projects on which some employee is working

- Incorrect: counts the tuples of WorksOn

  ```
  SELECT COUNT(PNo)
  FROM WorksOn
  ```

- Incorrect: counts the tuples of WorksOn

  ```
  SELECT DISTINCT COUNT(PNo)
  FROM WorksOn
  ```

- Correct in SQL

  ```
  SELECT COUNT(DISTINCT PNo)
  FROM WorksOn
  ```

- The adequate intuition (i.e., `COUNT` after `SELECT DISTINCT PNo`) is illegal in SQL

  ```
  COUNT(SELECT DISTINCT PNo FROM WorksOn)
  ```

30

Duplicate Control is Tricky

Give the average salary of employees who work more than 10 hours on some project

```
(1) SELECT AVG(Salary)
    FROM Employee, WorksOn
    WHERE SSN = ESSN AND Hours > 10
(2) SELECT AVG(DISTINCT Salary)
    FROM Employee, WorksOn
    WHERE SSN = ESSN AND Hours > 10
(3) SELECT AVG(Salary)
    FROM Employee
    WHERE SSN IN
          (SELECT ESSN
           FROM WorksOn
           WHERE Hours > 10)
```

31

- Duplicates should be carefully controlled to produce the correct result

- (1) is incorrect: if an employee works for 10 hours on 2 projects, his/her salary will participate twice in the average

- (2) is incorrect: 2 employees with the same salary will participate only once in the average

- Thus, with the flat TRC-like formulations (1) or (2) of the join, it is impossible to correctly control duplicates

- (3) is correct

GROUP BY ... HAVING ...

```
SELECT ...
FROM ...
WHERE condition1
GROUP BY attribute(s)
HAVING condition2
```

- Partitions a relation into a set of relations

- Only available at one level, i.e., not to build a set of sets of relations

- Intuitive and adhoc

- Intuitive operational semantics
  ◇ evaluate SELECT ... FROM ... WHERE, without the SELECT part
  ◇ partition the result according to the different values of the attributes appearing in the GROUP BY clause
  ◇ if the HAVING clause is present, apply condition2 to each class of the partition
  ◇ execute the SELECT part

32

Example of GROUP BY ... HAVING ...

- List each department number, the number of employees in the department and their average salary

```
SELECT DNo, COUNT(*), AVG(Salary)
FROM Employee
GROUP BY DNo
```

33

More Examples of GROUP BY

List the numbers of employees who work on more than 3 projects

```
SELECT  ESSN
FROM WorksOn
GROUP BY ESSN
HAVING COUNT (*) > 3

SELECT DISTINCT ESSN
FROM WorksOn w
WHERE (SELECT COUNT (*)
       FROM WorksOn w1
       WHERE w1.ESSN = w.ESSN)
     > 3
```

34

GROUP BY is Difficult

For departments with more than 5 employees, list department number and number of employees with a salary greater than 40k

```
SELECT DNo, COUNT(*) FROM Employee
WHERE Salary > 40k
  AND DNo IN (SELECT DNo
         FROM Employee
         GROUP BY DNo
         HAVING COUNT(*) > 5)
GROUP BY DNo

SELECT DNo, COUNT(*) FROM Employee
WHERE Salary > 40k
GROUP BY DNo
HAVING COUNT(*) > 5
```

35

The second formulation is incorrect

Universal Quantifier with GROUP BY et HAVING

List names of employees who work on all projects

```
SELECT FName, LName
FROM Employee E
WHERE SSN IN (
    SELECT W.ESSN
    FROM WorksOn W
    GROUP BY W.ESSN
    HAVING COUNT( DISTINCT W.PNO ) =
        ( SELECT COUNT(*) FROM Project ) )
```

36

Universal Quantifier with GROUP BY et HAVING

List names of employees who work on all projects controlled by department 5

```
SELECT FName, LName
FROM Employee E
WHERE SSN IN (
    SELECT W.ESSN
    FROM WorksOn W, Project P
    WHERE W.PNO = P.PNumber AND P.DNum = 5
    GROUP BY W.ESSN
    HAVING COUNT( DISTINCT W.PNO ) =
        ( SELECT COUNT(*) FROM Project WHERE DNum = 5 ) )
```

37

Updating in SQL

- Single-row insert: Enter a new project

```
INSERT
INTO Project(PName,PNumber,PLocation,DNum)
VALUES ('Toy', 36, 'Brussels', 4)
```
- Multi-row insert: Copy Brussels projects into R

```
INSERT
INTO R(PName,PNumber,PLocation,DNum)
      SELECT PName,PNumber,PLocation,DNum
      FROM Project
      WHERE PLocation = 'Brussels'
```
- Multi-row update: Raise the salary of employees in department 5 by 5%

```
UPDATE Employee
SET Salary = Salary * 1.05
WHERE DNo = 5
```

38

Updating in SQL (cont'd)

- Multi-row update: Make Smith the manager of all departments in Brussels

```
UPDATE Department
SET MgrSSN = (SELECT SSN
      FROM Employee
      WHERE LName = 'Smith')
WHERE DNumber IN (SELECT DNumber
          FROM DeptLocations
          WHERE DLocation = 'Brussels')
```

- Multi-row delete: Suppress the dependents of Smith

```
DELETE FROM Dependent
WHERE ESSN = (SELECT SSN
      FROM Employee
      WHERE LName = 'Smith')
```

39

SQL versus the Relational Model

- SQL supports a simple data structure, namely tables

- SQL supports project, select and join, and (more or less directly) the other operators of the relational algebra, and operates on entire relations to generate new relations

- SQL provides physical data independence to a large extent
  - ◇ Indexes may be added and deleted freely
  - ◇ Logical data independence is provided through the definitions of appropriate SQL view

- SQL combines table creation, querying and updating, and view definition into a uniform syntax

- SQL can be used as a stand-alone query language and embedded within a host general-purpose programming language

- SQL can be optimized and compiled, or may be interpreted and executed on line

40

- All of the statements above are true

- Yet, they are misleading because they describe the relational model and the relational approach to data management, rather than specific properties of SQL

- These claimed advantages of SQL really are properties of the relational style of data management

- They were included in an early edition of a popular textbook, and later withdrawn

Evaluation and Criticism of SQL

- Compared to what could have been done, much of SQL-specific design was bad

- Unlike calculi, SQL does not have a few powerful constructions that would account for its power of expression (orthogonality)

- The closure property of algebra and calculi is not realized in the definition of SQL

- SQL definition is (unnecessarily) large (the definition of calculi is much shorter)

- The limits of SQL are fuzzy: there is no simple way to characterize what can and what cannot be expressed (e.g., with functions)

- SQL is very redundant, many variants exist for the same query (for example, SQL essentially contains TRC as a sublanguage)

41

- The acceptance of SQL is not due to its intrinsic qualities

- Some bases of SQL language are flawed

- It was possible to design a much better, easier-to-use language

What Happened?

- Principles for designing good human computer interfaces are not well understood

- Too little awareness of programming language field (programming language design field is more mature)

- Prejudice against logic, theory, formalism, algebra, mathematics, etc. thought to be too complex

- Complexity of query language design was underestimated

- Recursive structure, orthogonality of algebra and calculus were thought to be too complex for the typical users of relational languages

- Initial incompetence followed by inertia

- Too much advertising

- Standardization goes against evolution

42

Orthogonality in Language Design

- Orthogonality in language design = **construct independence** or **generality of rules for combining concepts**

- Often, orthogonality ⇒ regularity, shorter language definition, simplicity

- Fewer general concepts combined with general rules ⇒ easier to master and remember (for most people)

43

- A small number of primitive constructs and a consistent set of rules for combining them are easier to learn and master than an equivalent larger number of primitives

- Orthogonality systematically exploited in the design of ALGOL 68 (a powerful language with a "2-level" syntax equivalent in power to general context-sensitive grammars (Chomsky type-0)

- "Orthogonal design maximizes expressive power while avoiding deleterious superfluidities" (ALGOL 68 definition)

- Orthogonality in language design =

  ◇ a relatively small set of well-defined primitive constructs

  ◇ a consistent set of rules for combining constructs

  ◇ every possible combination of constructs with the rules is legal and meaningful

  ◇ combinations of special cases of constructs is systematically prohibited

- Lack of orthogonality entails

  ◇ more complex languages (additional rules are needed to define and document special cases and exceptions; harder to learn and to teach)

  ◇ less powerful languages (the additional rules prohibit certain combinations of constructs)

Lack of Orthogonality in SQL: Relation Denotation

- Possible linguistic forms of relation denotation

  ◇ `SELECT ... FROM Project WHERE ...`

  ◇ a relation name, e.g., `Employee`

- Orthogonal definition: both expressions can be used in the same places for denoting relations, but

  ◇ `SELECT ... FROM  <relation name>  WHERE ...` is OK

  ◇ `SELECT ... FROM (SELECT ... FROM ... ) WHERE ...` is illegal in SQL

- Nesting in the FROM part of SELECT-blocks is possible through views

  `CREATE VIEW  R1 AS SELECT ... FROM ... WHERE ...`
  `SELECT ...  FROM R1 WHERE ...`

Lack of Orthogonality in SQL: Union

- `R1 UNION R2` is not allowed in SQL

- `(SELECT * FROM R1) UNION (SELECT * FROM R2)` is OK

Criticism: Rationale for GROUP BY not Obvious

- `GROUP BY`: not obviously interesting "packaging" of functionality

- `GROUP BY` is more procedural than the TRC part of SQL (more specific about how to execute the query, as opposed to stating properties of the result)

- Effect of `GROUP BY` is achieved in TRC style with variables

  List project numbers, project names and the number of employees that work on each project

  ```
  SELECT PNumber, PName, COUNT(*)
  FROM Project, WorksOn
  WHERE PNumber = PNo
  GROUP BY PNumber, PName
  ```

  $\{p.\text{PNumber}, p.\text{PName}, c \mid \text{Project}(p) \land$
  $\qquad c = count(\{w \mid \text{WorksOn}(w) \land p.\text{PNumber} = w.\text{PNo}\})\}$

46

## Criticism: GROUP BY could Have Gone Further

- `GROUP BY` could be easily adapted to express division, i.e., universal quantification

  ```
  SELECT A
  FROM R
  GROUP BY A
  HAVING  SET(B) CONTAINS S
  ```

47

Problems with Duplicate Rows

- SQL views relations as tables where duplicate rows are permitted (not so in the orthodox relational model)

- Transforming such tables into genuine relations is achieved by a `DISTINCT` function

- Problem of meaning: having identical entities in the real world is meaningless (impossible to talk about them unless something distinguishes them)

- Meaning of deleting a tuple if there are multiple copies?

- Handling duplicate rows suggests introducing order to distinguish them (e.g., "the 36th row")

- This is contrary to the relational model (tuple access is based on values only), volatile (updates), impractical for large relations

48

Discussion: SQL Starts Easy

- SQL provides simple query forms (e.g., without variables) for the simplest queries

- Whose life has been made simpler?
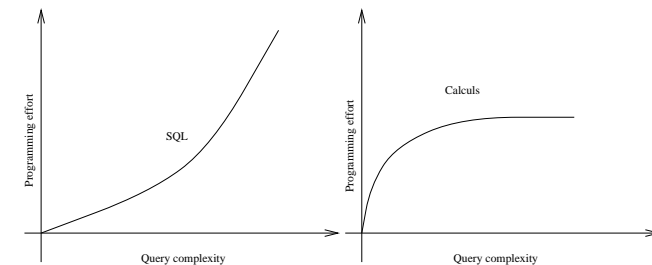
49

- This is often claimed as an advantage of SQL: short initial learning period, with gradual introduction of complexity

- The learning curve is very different for logic-based languages: the variable and existential quantification machinery has to be mastered for formulating even the simplest queries

---

Where Do Learning Curves Intersect?

- Do users exist that could satisfy their needs in the lower part of the SQL curve?

- We argue that, for many people, the calculi get simpler at some point before relational completeness

- Strategy for the more complex queries: formulate queries in calculus and translate them into SQL
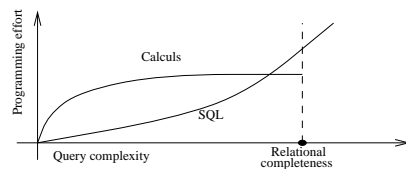
50

Data Independence
or
Quality of Query Optimizer

- Ideally, query optimizers choose the best evaluation strategy

- High level operations, unlike "navigation" programming, leave open opportunity of global optimization

- Tru optimality can only be realized on average, because true optimum in general depends on populations

- Data independence is violated when several equivalent queries are evaluated with different strategies (with different performance)

- Redundancy and nonorthogonality of SQL make full data independence hopelessly impossible, because some equivalent queries have very different forms

- Problem: knowledgeable users take advantage of violations

51

SQL History

- SQL was ill-conceived from the beginning: from a technical point of view, the history of its evolution is essentially uninteresting

- Still, SQL has become practically unescapable

- The bases for the calculi were defined with the relational model (1970)

- First version of SQUARE at IBM Research in 1974

- Numerous successive versions (SEQUEL, SEQUEL/2, SQL) with no clear rationale for evolution during 70's

- In parallel, numerous proposals for other similar languages (e.g., QUEL, QUERY BY EXAMPLE)

- First RDBMSs (1980-1986): ORACLE, SQL/DS then DB2(IBM), INGRES, SYBASE, INFORMIX, etc.

- SQL became the leading language by mid-80's

52

SQL Standardization

- ANSI RTG (Relational DB Task Group) recommendations (1982):
  - ◇ identify fundamental concepts of the relational model (representation, manipulation, constraints)
  - ◇ characterize key features of RDBMSs
  - ◇ integrate RDM and RDBMS in an architectural framework
  - ◇ idea: standardize functionality not syntax
  - ◇ hint: do not standardize SQL!
- SQL'86 by ANSI then ISO: essentially the IBM version of SQL
- SQL'89 (SQL1) with provisions for embedded SQL language
- SQL'92 or SQL2 (ANSI, ISO): emphasis on embedded SQL
- SQL3 (now called SQL'99): evolving towards a computationally complete OO programming language, with user-defined types, object functions, deduction

53