

STRUCTURED QUERY LANGUAGE

HISTORY

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999
 - SQL:2003
 - SQL:2006
 - SQL:2008
 - SQL:2011
 - SQL:2016

“Not all examples here may work on your particular system”



DATA DEFINITION LANGUAGE

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.



DOMAIN TYPES IN SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*.
- **varchar(*n*)**. Variable length character strings, with user-specified maximum length *n*.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(*p*,*d*)**. Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(*n*)**. Floating point number, with user-specified precision of at least *n* digits.



CREATE TABLE CONSTRUCT

- An SQL relation is defined using the **create table** command:

```

create table  $r$  ( $A_1 D_1, A_2 D_2, ..., A_n D_n,$ 
    (integrity-constraint1),
    ...
    (integrity-constraintk))

```

- r is the name of the relation
- each A_i is an attribute name in the schema of relation r
- D_i is the data type of values in the domain of attribute A_i

- Example:

```
create table branch
  (branch_name      char(15) not null,
   branch_city    char(30),
   assets          integer)
```





INTEGRITY CONSTRAINTS IN CREATE TABLE

- **not null**
- **primary key** (A_1, \dots, A_n)

Example: Declare *branch_name* as the primary key for *branch*

```
create table branch
    (branch_name char(15),
     branch_city   char(30),
     assets         integer,
     primary key (branch_name))
```

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89



ALTER TABLE CONSTRUCTS

- The **alter table** command is used to add attributes to an existing relation:

alter table r add A D

where A is the name of the attribute to be added to relation r and D is the domain of A .

- All tuples in the relation are assigned *null* as the value for the new attribute.
- The **alter table** command can also be used to drop attributes of a relation:

alter table r drop A

where A is the name of an attribute of relation r

- The **alter table** command can also be used to Modify existing columns

alter table r modify (A <new datatype> (<new size>))

Dropping of attributes not supported by many databases



DROP TABLE CONSTRUCTS

- The **drop table** command deletes all information about the dropped relation from the database.

Drop table r

EXAMINING OBJECTS CREATED BY USER

- Finding out tables created by a user
 - *Select * from TAB;*
- Describe the Table structure
 - *Describe <table name>*



BASIC QUERY STRUCTURE

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

select A_1, A_2, \dots, A_n
from r_1, r_2, \dots, r_m
where P

- A_i represents an attribute
 - R_i represents a relation
 - P is a predicate.
- The result of an SQL query is a relation.



THE SELECT CLAUSE

- The **select** clause list the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:

```
select branch_name  
from loan
```

NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

- E.g. *Branch_Name* \equiv *BRANCH_NAME* \equiv *branch_name*
 - Some people use upper case wherever we use bold font.
- 

THE SELECT CLAUSE (CONT.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the names of all branches in the *loan* relations, and remove duplicates

```
select distinct branch_name  
from loan
```

- The keyword **all** specifies that duplicates not be removed.

```
select all branch_name  
from loan
```



THE SELECT CLAUSE (CONT.)

- An asterisk in the select clause denotes “all attributes”

select *
from *loan*

- The **select** clause can contain arithmetic expressions involving the operation, +, −, *, and /, and operating on constants or attributes of tuples.
- The query:

select *loan_number, branch_name, amount* *
100
from *loan*

would return a relation that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100.



THE WHERE CLAUSE

- The **where** clause specifies conditions that the result must satisfy
 - Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than \$1200.

```
select loan_number
from loan
where branch_name = 'Perryridge' and
amount > 1200
```

- Comparison results can be combined using the logical connectives **and**, **or**, and **not**.
- Comparisons can be applied to results of arithmetic expressions.



THE WHERE CLAUSE (CONT.)

- SQL includes a **between** comparison operator
- Example: Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, $\geq \$90,000$ and $\leq \$100,000$)

```
select loan_number  
      from loan  
      where amount between 90000 and 100000
```



THE FROM CLAUSE

- The **from** clause lists the relations involved in the query
 - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower X loan*

select *
from *borrower, loan*

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

select *customer_name, borrower.loan_number, amount*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number and*
branch_name = 'Perryridge'



MODIFICATION OF THE DATABASE – DELETION

- Delete all account tuples at the Perryridge branch
delete from *account*
where *branch_name* = 'Perryridge'



EXAMPLE QUERY

- Delete the record of all accounts with balances below the average at the bank.

```
delete from account  
  where balance < (select avg (balance)  
                    from account)
```

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
 1. First, compute **avg** balance and find all tuples to delete
 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)



MODIFICATION OF THE DATABASE – INSERTION

- Add a new tuple to *account*

```
insert into account  
values ('A-9732', 'Perryridge', 1200)
```

or equivalently

```
insert into account (branch_name, balance,  
account_number)  
values ('Perryridge', 1200, 'A-9732')
```

- Add a new tuple to *account* with *balance* set to null

```
insert into account  
values ('A-777','Perryridge', null )
```



THE RENAME OPERATION

- The SQL allows renaming relations and attributes using the **as** clause:

old-name as new-name

- Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*.

```
select customer_name, borrower.loan_number as loan_id, amount  
from borrower, loan  
where borrower.loan_number = loan.loan_number
```



TUPLE VARIABLES

- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

```
select customer_name, T.loan_number, S.amount  
from borrower as T, loan as S  
where T.loan_number = S.loan_number
```

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

```
select distinct T.branch_name  
from branch as T, branch as S  
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

- Keyword **as** is optional and may be omitted
borrower as T \equiv *borrower T*



ORDERING THE DISPLAY OF TUPLES

- List in alphabetic order the names of all customers having a loan in Perryridge branch

```
select distinct customer_name
from    borrower, loan
where borrower loan_number =
loan.loan_number and
         branch_name = 'Perryridge'
order by customer_name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
 - Example: **order by** *customer_name desc*



SET OPERATIONS

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations \cup , \cap , $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **minus all**.

Suppose a tuple occurs m times in r and n times in s , then, it occurs:

- $m + n$ times in r **union all** s
- $\min(m, n)$ times in r **intersect all** s
- $\max(0, m - n)$ times in r **minus all** s



SET OPERATIONS

- Find all customers who have a loan, an account, or both:

```
(select customer_name from depositor)  
union  
(select customer_name from borrower)
```

- Find all customers who have both a loan and an account.

```
(select customer_name from depositor)  
intersect  
(select customer_name from borrower)
```

- Find all customers who have an account but no loan.

```
(select customer_name from depositor)  
minus  
(select customer_name from borrower)
```



AGGREGATE FUNCTIONS

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

Schema



AGGREGATE FUNCTIONS (CONT.)

- Find the average account balance at the Perryridge branch.

```
select avg (balance)  
  from account  
  where branch_name = 'Perryridge'
```

- Find the number of tuples in the *customer* relation.

```
select count (*)  
  from customer
```

- Find the number of depositors in the bank.

```
select count (distinct customer_name)  
  from depositor
```



GROUP BY CLAUSE

- **GROUP BY** takes one or more columns, and treats all rows with the same value in that column as a single group when you apply aggregate function.
- Find the number of depositors for each branch.

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

```
select branch_name, count (distinct customer_name)  
  from depositor, account  
 where depositor.account_number = account.account_number  
 group by branch_name
```



AGGREGATE FUNCTIONS – HAVING CLAUSE

HAVING is used in combination with GROUP BY to ignore groups that don't meet certain criteria.

- Find the names of all branches where the average account balance is more than \$1,200.

```
select branch_name, avg (balance)  
  from account  
 group by branch_name  
 having avg (balance) > 1200
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups



- The GROUP BY Clause SQL is used to group rows with same values.
- The GROUP BY Clause is used together with the SQL SELECT statement.
- The SELECT statement used in the GROUP BY clause can only be used contain column names, aggregate functions, constants and expressions.
- SQL Having Clause is used to restrict the results returned by the GROUP BY clause.



STRING OPERATIONS

- SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring “Main”.

```
select customer_name  
from customer  
where customer_street like '% Main%'
```

- Match the name “Main%”

```
like 'Main\%' escape '\'
```
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.



NULL VALUES

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.

- Example: Find all loan number which appear in the *loan* relation with null values for *amount*.

```
select loan_number  
from loan  
where amount is null
```

- The result of any arithmetic expression involving *null* is *null*
 - Example: $5 + \text{null}$ returns null
- However, aggregate functions simply ignore nulls
 - More on next slide



NULL VALUES AND THREE VALUED LOGIC

- Any comparison with *null* returns *unknown*
 - Example: $5 < null$ or $null <> null$ or $null = null$
- Three-valued logic using the truth value *unknown*:
 - OR: $(unknown \text{ or } true) = true$,
 $(unknown \text{ or } false) = unknown$
 $(unknown \text{ or } unknown) = unknown$
 - AND: $(true \text{ and } unknown) = unknown$,
 $(false \text{ and } unknown) = false$,
 $(unknown \text{ and } unknown) = unknown$
 - NOT: $(\text{not } unknown) = unknown$
 - “*P* is **unknown**” evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*



NULL VALUES AND AGGREGATES

- Total all loan amounts

```
select sum (amount )  
from loan
```


- Above statement ignores null amounts
- Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.



JOINED RELATIONS**

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<i>Join types</i>	<i>Join Conditions</i>
inner join left outer join right outer join full outer join	natural on <predicate> using (A_1, A_1, \dots, A_n)



JOINED RELATIONS – DATASETS FOR EXAMPLES

- Relation *loan*

- Relation *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- Note: borrower information missing for L-260 and loan information missing for L-155



JOINED RELATIONS – EXAMPLES

- *loan inner join borrower on
loan.loan_number = borrower.loan_number*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan left outer join borrower on
loan.loan_number = borrower.loan_number*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	<i>null</i>	<i>null</i>

Select <col name1>,<colname2>,...<colname n> from <Table name1> Inner Join<Table name2> ON < Table name1> .<col name1>=< Table name2> .<col name2> where<condition> order by <Col name1> ,<col name2> ...<col name n>



JOINED RELATIONS – EXAMPLES

- *loan natural inner join borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- *loan natural right outer join borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	<i>null</i>	<i>null</i>	Hayes



JOINED RELATIONS – EXAMPLES

- *loan* **full outer join** *borrower* **using** (*loan_number*)

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes

- Find all customers who have either an account or a loan (but not both) at the bank.

```
select customer_name  
      from (depositor natural full outer join borrower)  
      where account_number is null or loan_number is null
```



○ Cross Joins

- *Select <col name1>,<colname2>,..<colname n> from <Table name1> Cross Join<Table name2>*

○ Self Joins

- *from <tablename> [<Alias>],<Tablename>[<alias>]....*



VIEWS

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
(select customer_name, borrower.loan_number,  
branch_name  
from borrower, loan  
where borrower.loan_number = loan.loan_number )
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.



VIEW DEFINITION

- A view is defined using the **create view** statement which has the form

create view v as < query expression >

where <query expression> is any legal SQL expression. The view name is represented by v .

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.



- There are 2 types of Views in SQL:
- **Simple View**
 - **Simple views** can only contain a single base table.
- **Complex views**
 - can be constructed on more than one base table.
 - complex views can contain: join conditions, a group by clause, an order by clause.



COMPARISON

Simple View

Contains only one single base table or is created from only one table.

We cannot use group functions like MAX(), COUNT(), etc.

Does not contain groups of data.

DML operations could be performed through a simple view.

INSERT, DELETE and UPDATE are directly possible on a simple view.

Simple view does not contain group by, distinct, pseudocolumn like rownum, columns defined by expressions.

Does not include NOT NULL columns from base tables.

Complex View

Contains more than one base tables or is created from more than one tables.

We can use group functions.

It can contain groups of data.

DML operations could not always be performed through a complex view.

We cannot apply INSERT, DELETE and UPDATE on complex view directly.

It can contain group by, distinct, pseudocolumn like rownum, columns defined by expressions.

NOT NULL columns that are not selected by simple view can be included in complex view.



EXAMPLE QUERIES

- A view consisting of branches and their customers

create view *all_customer* **as**

 (**select** *branch_name, customer_name*
 from *depositor, account*

where *depositor.account_number =*
 account.account_number)

union

 (**select** *branch_name, customer_name*
 from *borrower, loan*

where *borrower.loan_number = loan.loan_number*)

- Find all customers of the Perryridge branch

select *customer_name*

from *all_customer*

where *branch_name = 'Perryridge'*



UPDATES THROUGH VIEWS (CONT.)

- Some updates through views are impossible to translate into updates on the database relations
 - **create view v as**
 select *loan_number*, *branch_name*,
 amount
 from *loan*
 where *branch_name* = 'Perryridge'
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation



UPDATING/INSERTING/DELETING A VIEW

- Conditions for updating/Inserting/Deleting a View
 - The SELECT clause may not contain the keyword DISTINCT.
 - The SELECT clause may not contain summary functions.
 - The SELECT clause may not contain set functions.
 - The SELECT clause may not contain set operators.
 - The SELECT clause may not contain an ORDER BY clause.
 - The FROM clause may not contain multiple tables.
 - The WHERE clause may not contain subqueries.
 - The query may not contain GROUP BY or HAVING.
 - Calculated columns may not be updated.
 - All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.



- If a view satisfies all the conditions mentioned, then view can be updated/inserted/deleted.
- This would ultimately update the base table from which the view was created and the same would reflect in the view itself.
- Dropping Views
 - DROP VIEW view_name;



TRANSACTION CONTROL COMMANDS

- The following commands are used to control transactions.
 - **COMMIT** – to save the changes.
 - **ROLLBACK** – to roll back the changes.
 - **SAVEPOINT** – creates points within the groups of transactions in which to ROLLBACK.
- used only with the **DML Commands**
- Not applicable for DDL Commands since its autocommitted



THE COMMIT COMMAND

- The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.
- The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.
- The syntax for the COMMIT command is as follows.
 - COMMIT;



THE ROLLBACK COMMAND

- The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database.
- This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.
- The syntax for a ROLLBACK command is as follows:
 - ROLLBACK;



THE SAVEPOINT COMMAND

- A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.
- The syntax for a SAVEPOINT command is as shown below.
 - `SAVEPOINT SAVEPOINT_NAME;`
- This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions.
- The syntax for rolling back to a SAVEPOINT
 - `ROLLBACK TO SAVEPOINT_NAME;`

