**SQL**

**SQL in 5 minutes**

[Operation]{Applicable-Clause}

The Rename Operation [**as**] {SELECT}

SELECT customer-name, borrower.loan-number **as** loan-id

Tuple Variables [**as**] {FORM}-

SELECT customer-name, T.loan-number, S.amount

FROM borrower as T, loan as S

WHERE T.loan-number = S.loan-number

-They are most useful for comparing two tuples in the same relation.

Find the names of all branches that have assets greater than at least one branch located in Brooklyn.

String Operations [**like**, **not like**] {WHERE}

Find the names of all customers whose street address includes the substring ‘Main’.

WHERE customer-street like ’%Main%’

Ordering [**desc**, **asc**] {ORDER BY}

ORDER BY amount desc, loan-number asc

Set Operations**[union, intersect, except]**

(First SQL)union(Second SQL)

-Applies on two sql's. By default duplicates are removed.

Aggregate Functions [**avg, min, max, sum, count**] {select, having, Groupby}

Nested Subqueries[In, Not In, Some, All, exists, Not exists, unique, Not unique]

Theseareapplicable inside where.

WHERE column-name in (SQL-2)

SQL-2: SELECT column-name

FROM depositor

The most influential commercially marketed query language, SQL. SQL uses a combination of relational-algebra and relational-calculus constructs.

Although we refer to the SQL language as a “query language,” it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints.

Background

IBM developed the original version of SQL at its San Jose Research Laboratory (now the Almaden Research Center). IBM implemented the language, originally called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language). Many products now support the SQL language.

The SQL language has several parts:

* Data-definition language (**DDL**). The SQL DDL provides commands for **defining relation** schemas, **deleting relations**, and **modifying relation** schemas. RELSTION MEANS TABLE.
* Interactive data-manipulation language (**DML**). The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It includes also commands to **insert** tuples into, **delete** tuples from, and **modify** tuples in the database.
* View definition. The SQL DDL includes commands for defining views.
* Transaction control. SQL includes commands for specifying the beginning and ending of transactions.
* Embedded SQL and dynamic SQL. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.
* Integrity. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
* Authorization. The SQL DDL includes commands for specifying access rights to relations and views.

For examples we use banking enterprise with the following relation schemas:

*Branch-schema = (branch-name, branch-city, assets)*

*Customer-schema = (customer-name, customer-street, customer-city)*

*Loan-schema = (loan-number, branch-name, amount)*

*Borrower-schema = (customer-name, loan-number)*

*Account-schema = (account-number, branch-name, balance)*

*Depositor-schema = (customer-name, account-number)*

The basic structure of an SQL expression consists of three clauses: select, from, andwhere.

* The select clause corresponds to the projection operation of the relational algebra.It is used to list the attributes desired in the result of a query.
* The from clause corresponds to the Cartesian-product operation of the relationalalgebra. It lists the relations to be scanned in the evaluation of the expression.
* The where clause corresponds to the selection predicate of the relational algebra.It consists of a predicate involving attributes of the relations that appearin the from clause.

A typical SQL query has the form

SELECT A1, A2, . . .,An

FROM r1, r2, . . . ,rm

WHERE P

Each Ai represents an attribute, and each ri a relation. P is a predicate.

**BASIC-SQL**

**The select Clause**

*Loan-schema = (loan-number, branch-name, amount)*

simplest query **Find the names of all branches in the loan relation**:

***SELECT*** *branch-name*

***FROM*** *loan*

The result is a relation consisting of a single attribute(column) with the heading branch-name.

In practice, duplicate eliminationis time-consuming. Therefore, SQL allows duplicates in relations as well as in the results of SQL expressions. Thus, thepreceding query will list each branch-name once for every tuple in which it appears inthe loan relation.

If we want to force the elimination of duplicates, we insert thekeyword distinct after select.

*SELECT****distinct*** *branch-name*

*FROM loan*

SQL allows us to use the keyword all to specify explicitly that duplicates are not removed:

*SELECT****all*** *branch-name*

*FROM loan*

Since duplicate retention is the default, all is redundent here.

The query

*SELECT loan-number, branch-name,* ***amount \* 100***

*FROM loan*

Will return a relation [table] that is the same as the loan relation, except that the attribute [column] amount is multiplied by 100.

**The where Clause**

**Find all loan numbers 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*

SQL uses the logical connectives **and, or,** and **not** [rather than the mathematicalsymbols ∧, ∨, and ¬] in the where clause. The operands of the logical connectivescan be expressions involving the comparison operators <, <=, >, >=, =, and <>.

SQL allows comparison operators to compare strings and arithmeticexpressions, as well as special types, such as date types.

SQL includes a **between** comparison operator to simplify where clauses.

**Find the loan number of those loans with loan amounts between $90,000 and $100,000.**

SELECT loan-number

FROM loan

WHERE amount between 90000 and 100000

Similarly, we can use the **not between** comparison operator.

**The from Clause**

The from clause by itself defines a Cartesian product [**natural join**] of the relations in the clause. It is a relatively simple matter to write an SQL expression for the natural join.

**For all customers who have a loan from the bank, find their names, loan numbers and loan amount.**

*SELECT customer-name, borrower.loan-number, amount*

*FROM borrower, loan*

*WHEREborrower.loan-number = loan.loan-number*

Notice that SQL uses the notation relation-name.attribute-name, to avoid ambiguity in cases where an attribute appears in the schema of more than one relation.

Consider a more complicated case

**Find the customer names, loan numbers, and loan amounts for all loans at the Perryridge branch.**

*SELECT customer-name, borrower.loan-number, amount*

*FROM borrower, loan*

*WHEREborrower.loan-number = loan.loan-number and*

*branch-name = ’Perryridge’*

SQL includes extensions to perform natural joins and outer joins in the from clause.

**The Rename Operation**

old-name as new-name

The names of the attributes in the result [of query] are derived from the names of the attributesin the relations in the from clause.

We cannot, always derive names in this way, for following reasons:

* Two relations in the “from clause” may have attributes with the same name, in which Case an attribute name is duplicated in the result.
* If we used an arithmeticexpression in the select clause, the resultant attribute does not have a name.
* Even if an attribute name can be derived from the base relations, we may want to change the attribute name in the result.

For example, if we want the attribute name loan-number to be replaced with the name loan-id, in query

For all customers who have a loan from the bank, find their names, loan numbers and loan amount.

SELECT customer-name, borrower.loan-number as loan-id, amount

FROM borrower, loan

WHEREborrower.loan-number = loan.loan-number

**Tuple Variables**

A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the from clause by way of the asclause.

**For all customers who have a loan from the bank, find their names, loan numbers, and loan amount.**

SELECT customer-name, T.loan-number, S.amount

FROM borrower as T, loan as S

WHERET.loan-number = S.loan-number

We define a tuple variable in the from clause by placing it after the name ofthe relation with which it is associated, with the keyword as in between.

When we write expressions of the form relation-name.attribute-name,the relation name is, an implicitly defined tuple variable.

Tuple variables are most useful for comparing two tuples in the same relation.

**Find the names of all branches that have assets greater than at least one branch located in Brooklyn.**

*SELECT distinct T.branch-name*

*FROM branch as T, branch as S*

*WHERET.assets>S.assets and S.branch-city = ’Brooklyn’*

Observe that we could not use the notation branch.asset, since it would not be clear which reference to branch is intended.

**String Operations [like]**

SQL specifies strings by enclosing them in single quotes, for example,**’**Perryridge**’**.

A single quote character that is part of a string can be specified by using two single quote characters; for example the string **“It’s right”** can be specifiedby**’It”s right’.**

The most commonly used operation on strings is pattern matching using the operator**like**.

We describe patterns by using two special characters:

* Percent (**%**): The % character matches any substring.
  + ’%idge%’ matches any string containing “idge” as a substring,
  + for example, ’Perryridge’, ’Ridgeway’ etc..
* Underscore (**\_**): The character matches any character.
  + ’\_ \_ \_ ’ matches any string of exactly three characters.
  + ’\_ \_ \_ %’ matches any string of at least three characters.

SQL expresses patterns by using the **like** comparison operator.

**Find the names of all customers whose street address includes the substring ‘Main’.**

*SELECT customer-name*

*FROM customer*

*WHERE customer-street* ***like****’%Main%’*

Remember Patterns are case sensitive

For patterns to include the special pattern characters (that is, % and ), SQL allowsthe specification of an escape character(\). The escape character is used immediatelybefore a special pattern character to indicate that the special pattern character is to betreated like a normal character.

We define the escape character for a**like** comparisonusing the **escape** keyword.

Following patterns, uses a backslash (\) as the escape character:

* **like** ’ab\%cd%’ **escape** ’\’ matches all strings beginning with “ab%cd”.
* **like** ’ab\\cd%’ **escape** ’\’ matches all strings beginning with “ab\cd”.

SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator.

SQL also permits a variety of functions on character strings, such as concatenating(using “||”), extracting substrings, finding the length of strings, converting betweenuppercase and lowercase, and so on.

SQL:1999 also offers a similar to operation,which provides more powerful pattern matching than the like operation.

**Ordering**[*order by ,*desc, asc]

The order by clause causes the tuples in the result of a query to appear in sorted order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. By default, the order by clause lists items in ascending order.

**List all customers in alphabetic order who have a loan at the Perryridge branch.**

*SELECT distinct customer-name*

*FROM borrower, loan*

*WHEREborrower.loan-number = loan.loan-number and branch-name = ’Perryridge’*

*order by customer-name*

Ordering can be performed on multiple attributes.

**List the entire loan relation in descending order of amount. If several loans have the same amount, order them in ascending order by loan number.**

*SELECT \**

*FROM loan*

*ORDER BY amount desc, loan-number asc*

To fulfill an order by request, SQL must perform a sort. Since sorting a large number of tuples may be costly, it should be done only when necessary.

**Duplicates**

Using relations with duplicates offers advantages in several situations. Accordingly, SQL formally defines not only what tuples are in the result of a query, but also how many copies of each of those tuples appear in the result.

**Set Operations[**union, intersect, except**]**

The SQL operations **union**, **intersect**, and **except**operate on compatible [that is, they must have the same set of attributes] relations. The set operation automatically eliminates duplicates, unlike the select clause.

The Union Operation

**Find all customers having a loan, an account, or both at the bank.**

*(SELECT customer-name*

*FROM depositor)*

***union***

*(SELECT customer-name*

*FROM borrower)*

If we want to retain all duplicates, we must write union all in place of union:

(SELECT customer-name

FROM depositor)

**union all**

(SELECT customer-name

FROM borrower)

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be five tuples with the name Jones in the result.

**The Intersect Operation**

To find all customers who have both a loan and an account at the bank, we write

*(SELECT distinct customer-name*

*FROM depositor)*

***intersect***

*(SELECT distinct customer-name*

*FROM borrower)*

The number of duplicate tuples[in case of intersect all] that appear in the result is equal to the minimum

number of duplicates in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be two tuples with the name Jones in the result.

**The Except Operation**

To find all customers who have an account but no loan at the bank

*(SELECT distinct customer-name*

*FROM depositor)*

***except***

*(SELECT customer-name*

*FROM borrower)*

If we want to retain all duplicates, we must write except all in place of except:

(SELECT customer-name

FROM depositor)

**except all**

(SELECT customer-name

FROM borrower)

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies of the tuple in d minus the number of duplicate copies of the tuple in b, provided that the difference is positive. Thus, if Jones has three accounts and one loan at the bank, then there will be two tuples with the name Jones in the result. If, instead, this customer has two accounts and three loans at the bank, there will be no tuple with the name Jones in the result.

**Aggregate Functions [**avg, min, max, sum, count, *group by, having***]**

Aggregate functions are functions that take a collection (a set) of values asinput and return a single value. SQL offers five built-in aggregate functions:

* Average: avg
* Minimum: min
* Maximum: max
* Total: sum
* Count: count

The input to **sum** and **avg** must be a collection of **numbers**, but the other operatorscan operate on collections of non-numeric data types, such as strings, as well.

**Find the average account balance at the Perryridge branch.**

*SELECT avg (balance)*

*FROM account*

*WHERE branch-name = ’Perryridge’*

The result of this query is a relation with a single attribute, containing a single tuplewith a numerical value corresponding to the average balance at the Perryridge branch. Optionally, we can give a name to the attribute of the result relation by usingthe as clause.

Find the average account balance at each branch.

*SELECT branch-name, avg (balance)*

*FROM account*

*GROUP BY branch-name*

**Groupby**

There are circumstances where we would like to apply the aggregate function not to a single set of tuples, but to a group of sets of tuples; we specify this in SQL using the group by clause.

The attribute or attributes given in the group by clause are used to form groups. Tuples with the same value on all attributes in the group by clause are placed in one group.

Retaining duplicates is important in computing an average. There are cases where we must eliminate duplicates before computing an aggregate function.

**Find the number of depositors for each branch.**

*SELECT branch-name, count (distinct customer-name)*

*FROM depositor, account*

*WHERE depositor.account-number = account.account-number*

*GROUP BY branch-name*

When the aggregate function is applied to a group of sets of tuples [instead a single set of tuples] we specify this in SQL using the group by clause.

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword distinct in the aggregate expression.

A depositor counts only once, regardless of the number of accounts that depositor may have.

**Find those branches where the average account balance is more than $1200**

This condition does not apply to a single tuple;rather, it applies to each group constructed by the group by clause. To express such aquery, we use the having clause of SQL.

*SELECT branch-name, avg (balance)*

*FROM account*

*GROUP BY branch-name*

*HAVING avg (balance) > 1200*

SQL applies predicates in the having clause after groups have been formed.

Find the average balance for all accounts.

*SELECT avg (balance)*

*FROM account*

Find the number of tuples in the customer relation

*SELECT count (\*)*

*FROM customer*

SQL does not allow the use of distinct with count(\*).

**Find the average balance for each customer who lives in Harrison and has at least three accounts**

SELECT depositor.customer-name, avg (balance)

*FROM*depositor, account, customer

WHERE depositor.account-number = account.account-number and

depositor.customer-name = customer.customer-name and

customer-city = ’Harrison’

*GROUP BY*depositor.customer-name

*HAVING*count (distinct depositor.account-number) >= 3

If a WHERE clause and a HAVING clause appear in the same query, SQL applies

1. The predicate in the where clause first.
2. Tuples satisfying the where predicate are then placed into groups by the group by clause.
3. SQL then applies the having clause, if it is present, to each group;
4. It removes the groups that do not satisfy the having clause predicate.
5. The select clause uses the remaining groups to generate tuples of the result of the query.

**Find all loan numbers that appear in the loan relation with null values for amount**

SELECT loan-number

FROM Loan

WHERE amount is null

SQL allows the use of null values to indicate absence of information about the value of an attribute. We use the special keyword null in a predicate to test for a null value.

**Advanced SQL**

**[In/Not In, Some/All, exists/Not exists, unique/Not unique]**

**NestedSubqueries**

**Find those customers who are borrowers from the bank and who appear in the list of account holders.**

SELECT customer-name

FROM Borrower

WHERE customer-name in

(SELECT customer-name

FROM depositor)

The **in connective** tests for set membership, where the set is a collection of values produced by a select clause.

The **not in connective** tests for the absence of set membership.

The above query is same as

Find all customerswho have both a loan and an account at the bank.

This example shows that it is possible to write the same query several ways in SQL.

**Find all customers who have both an account and a loan at the Perryridge branch**

SELECT distinct customer-name

FROM Depositor, Account

WHERE Depositor.loan-number=Account.loan-number and customer-name **IN** (

Select customer-name

FROM Borrower, Loan

WHERE Borrower.loan-number Loan.loan-number and branch-name='Perryridge')

**Find all customers who do have a loan at the bank, but do not have an account at the bank.**

SELECT distinct customer-name

FROM Depositor

WHERE customer-name **NOT IN** (

SELECT distinct customer-name

FROM Borrower

)

**Set Comparison[Some, all]**

**Find the names of all branches that have assets greater than those of at least one branch located in Brooklyn.**

SELECT branch-name

FROM Branch as T, Branch as S

WERE T.assets>S.assets and S.branch-city = 'Brooklyn'

Alternative style.

The phrase “greater than at least one” is represented in SQL by **> some**. This construct allows us to rewrite

SELECT branch-name

FROM Branch

WHERE assets **> SOME** (SELECT assets

FROM Branch

WHERE branch-city = 'Brooklyn')

SQL also allows < some, <= some, >= some, = some, and<> some comparisons.

Verify that **= some** is identical to **in**, whereas **<> some** is **not the sameas not in**.

The keyword **any** is synonymous to **some** in SQL. Early versions of SQL allowed only any. Later versions added the alternative some to avoid the linguistic ambiguity of the word any in English.

**Find the names of all branches that have an asset value greater than that of each branch in Brooklyn.**

Select branch-name

From Branch

Where assets > all (Select assets

From Branch

Where branch-city=’Brooklyn’

)

Alternative style.

Select branch-name

From Branch as S, Branch as T

Where S.assets>T.assets and T.branch-city="Brooklyn"

SQL allows < all, <= all, >= all, = all, and <> all comparisons.

As an exercise, verify that **<> all** is identical to **not in.**

**Find the branch that has the highest average balance.**

SELECT branch-name

FROM Account

group by branch-name

havingavg (balance) >= all (SELECT avg(balance)

FROM Account

GROUBY branch-name

)

TBC …

**Complex Queries**

Complex queries are often hard or impossible to write as a single SQL block or a

union/intersection/difference of SQL blocks.

There are two ways of composing multiple SQL blocks to express a complex query: derived

relations and the with clause.

**Modification of the Database**

**Deletion**

We can delete whole tuples; we cannot delete values on only particular attributes.

SQL expresses a deletion by

delete from r

where P

The where clause can be omitted, in which case all tuples in r are deleted.

Note that a delete command operates on only one relation. If we want to delete tuples from several relations, we must use one delete command for each relation.

**Delete all account tuples in the Perryridge branch.**

DELETE FROM Account

WHERE branch-name='Perryridge'

**Delete all loans with loan amounts between $1300 and $1500.**

DELETE FROM Loan

WHERE amount between $1300 and $1500

**Delete all account tuples at every branch located in Needham.**

delete from account

where branch-name in (select branch-name

from branch

where branch-city = ’Needham’)

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

DELETE FROM Account

WHERE balance < (select avg (balance)

from account)

The delete statement first tests each tuple in the relation account to check whether the account has a balance less than the average at the bank. Then, all tuples that fail the test are deleted.

Performing all the tests before performing any deletion is important—if some tuples are deleted before other tuples have been tested, the average balance may change, and the final result of the delete would depend on the order in which the tuples were processed!

**Insertion**

Insert an account A-9732 at the Perryridge branch with balance of $1200.

INSERT INTO account

values (’A-9732’, ’Perryridge’, 1200)

INSERT INTO account (branch-name, account-number, balance)

values (’Perryridge’, ’A-9732’, 1200)

Suppose that we want to present a new $200 savings acocunt as a gift to all loan customers of the Perryridge branch, for each loan they have. Let the loan number serve as the account number for the savings account.

INSERT INTO account

select loan-number, branch-name, 200

from loan

where branch-name = ’Perryridge’

**Updates**

change a value in a tuple without changing all values in the tuple.

payments of all balances are to be increased by 5 percent

update account

set balance = balance \* 1.05

**Data-Definition Language**

The SQL DDL allows specification of a set of relations, and information about each relation, including

* The schema for each relation
* The domain of values associated with each attribute
* The integrity constraints
* The set of indices to be maintained for each relation
* The security and authorization information for each relation
* The physical storage structure of each relation on disk

We define an SQL relation by using the create table command:

create table r(A1D1,A2D2, . . . , AnDn,

<<integrity-constraint1>>,

. . . ,

<<integrity-constraintk>>

)

Where r is the name of the relation, each Ai is the name of an attribute in the schema of relation r, and Di is the domain type of values in the domain of attribute Ai.

The allowed integrity constraints include

* primary key (Aj1,Aj2, . . .,Ajm): Attributes Aj1,Aj2, . . .,Ajm form the primary key for the relation. The primarykey attributes are required to be non-null and unique; Although the primary key specification is optional, it is generally a good idea to specify a primary key for eachrelation.
* check(P): The check clause specifies a predicate P that must be satisfied by every tuple in the relation.

To remove a relation from an SQL database, we use the drop table command.

drop table r

Use the alter table command to add attributes to an existing relation.

alter table r add A D

All tuples in the relation are assigned null as the value for the new attribute.

We can drop attributes from a relation by the command

alter table r drop A

Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.