Chapter 3: SQL Language

3.1, 3.2 Overview, DDL and DML

- Data-definition language (DDL): provides commands for defining relation schemas, deleting relations, and modifying relation schemas
- Data-manipulation language (DML): query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database
- Integrity: SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates violate integrity constraints are disallowed
- View definition: SQL DDL includes commands for defining views
- Transaction control: SQL includes commands for specifying the beginning and ending transactions
- Embedded SQL and Dynamic SQL: define how SQL statements can be embedded within general-purpose programming languages, e.g.: C, C++, Java
- Authorization: SQL DDL includes commands for specifying access rights to relations and views

Examples of DDL and DML? Is select-from-where DDL or DML?

DDL: create, alter, drop, truncate, comment

DML: select, insert, update, delete (relate to authorization)

DDL

SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation
- The types of values associated with each attribute, each includes null value
  - char(n): fixed length char string, "character(n)" can be used instead, if string shorter than n stored, spaces appended

- varchar(n): variable length char string with max = n, "character varying(n)" can be used instead
- int: integer, "integer" can be used instead
- smallint: machine-dependent subset of integer type
- numeric(p, d): fixed-point number with user-specified precision, consists p digits (plus a sign) and d of the p digits are to the right of the decimal point
- real, double precision: floating point and double precision, machine dependent
- float(n): floating point number with precision of at least n digits

- 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

Basic Schema Definition by DDL

- Create table command:

  Create table r

  (A_1 D_1,

  A_2 D_2,

  …,

  A_n D_n,

  <integrity-constraint$_1$>,

  …,

  <integrity-constraint$_k$>);

e.g.:

create table *department*

(*dept_name* varchar(20),

building varchar(15),

budget numeric(12, 2),

primary key(*dept_name*));

create table *course*

(*course_id* varchar(7),

title varchar(50),

dept_name varchar(2) not null,

credits numeric(2,0),

primary key(*course_id*),

foreign key (*dept_name*) references *department*);

Instantiate a relationship in this case? How relationships are instantiated?

- Loading data
  - insert into command: values are in order corresponding of attributes

    e.g.: insert into instructor

    values(10211, 'Smith', 'Biology', 66000);
- Supported integrity constraints
  - primary key (A_{j1}, A_{j2}, …, A_{jm}): attributes A_{j1} … A_{jm} from the primary key for the relation. Required to be non-null and unique.
  - foreign key (A_{k1}, A_{k2}, …, A_{km}) references s: the values of attributes (A_{k1}, A_{k2}, …, A_{km}) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s.
- Removing relation
  - drop command: deletes all tuples and the schema, no tuples can be inserted unless it is re-created with create table command, more drastic than delete

    e.g.: drop table r
  - delete command: deletes all tuples but retain the relation

    e.g.: delete from r
- Altering relation
  - alter table command: add/drop attributes from a relation

    e.g.: alter table r add A D;

    alter table r drop A;

3.3 Basic Structure of SQL Queries by DML

Consists of 3 clauses: select, from, where

- Query on single relation

  select *name*

  from *instructor*;

  - To eliminate duplicates:

    select distinct *dept_name*

    from *instructor*;

  - To explicitly not remove duplicates:

    select all *dept_name*

    from *instructor*;

  - Arithmetic operators:

    select *ID, name, dept_name, salary * 1.1*

    from *instructor*;

  - Where clause to specify a predicate:

    select *name*

    from *instructor*

    where *dept_name* = 'Comp. Sci.' and *salary* > 70000;

  - SQL allows and, or not, <. <=, >, >=, =, <> in where clause predicates

- Queries on multiple relations

  select $A_1, A_2, \ldots, A_n$

  from $r_1, r_2, \ldots, r_m$

  where A_k *operator value*;

  Select attributes desired from relations where the predicate is true

  From clause defines a Cartesian product of the relations listed, we can drop the relation-name prefix for different attribute names in the product

  Operators supported by where clause:

  - = equal
  - <> not equal
  - >, <, >=, <=

- BETWEEN between an inclusive range
- LIKE search for a pattern
- IN specify multiple possible values
- AND, OR used between predicates, logic operators

- Natural join
  - Need relations joined together
  - A natural join B ⊂ A Cartesian product B
  - Cartesian product concatenates every tuple, natural join considers only the tuples with the same value on those attributes that appear in the schemas of both relations

    How does SQL know they are the same attribute? By name and datatype?

    e.g.: select * from *instructor* natural joins *teaches*, only the tuples with the common attributes (i.e. ID) have the same value are included

    Equivalent to:

    Select * from *instructor, teaches*

    Where *instructor.ID = teaches.ID;* (all common attributes)
  - Same attribute with value only appear in one relation schema is not included
  - If no common attribute, no difference from Cartesian product
  - From clause can have a list of relation schema:

    e.g.: select * from *instructor* natural join *teaches, course*

    where *teaches.course_id = course.cousrse_id;*

    Compute the natural join first, then a Cartesian product based on predicate (note that a Cartesian product with an equal predicate is not always equivalent to natural join, every common attribute must be equal)
  - Join … using: Avoid equating attributes erroneously, specify exactly what columns should be equated:

    e.g.: select *name, title*

    from (*instructor* natural join *teaches*) join *course* using (*course_id*);

    Requires a list of attributes to be equated in using().

    Both schemas must have the attributes in the list.

Difference from natural join: only requires attributes in the list equal, regardless of other common attributes even if they are unequal

3.4 Additional Basic Operations

- Rename by as clause

  Names of attributes in result are derived from the names of the attributes in the relations in the from clause

  Reasons to rename:

  - Replace a long relation name with a shortened version

    e.g.: For all instructors who have taught some course, find their names and the course ID of all courses they taught

    select *name* as *instructor_name, course_id*

    from *instructor, teaches*

    where *instructor.ID = teaches.ID*;

    The same as:

    select *T.name, S.course_id*

    from *instructor* as *T*, *teaches* as *S*

    where *T.ID = S.ID*;

    The result has two attributes: name and course_id

  - To compare tuples in the same relation

    e.g.: Find the names of all instructors whose salary is greater than at least one instructor in the Biology department (the minimum salary in the Biology department)

    select distinct *T.name*

    from *instructor* as *T*, *instructor* as *S*

    where *T.salary* > *S.salary* and *S.dept_name* = 'Biology';

    Cannot use instructor.salary here, due to ambiguity

  T and S above are correlation name, a. k. a. table alias, correlation variable, tuple variable. They are identifiers that is used to rename a relation to be referred to.

  Ways to declare correlation name?

- As clause for tuples with more than one element

  (*A1*, *A2*) as *R* (*B1*, *B2*)

- String Operations
  - Enclosed in single quotes, case sensitive
  - Concatenation: s1 || s2
  - Substring:
  - Finding length of strings:
  - Converting strings to upper/lower case: upper(s), lower(s)
  - Removing spaces at the end: trim(s)
  - Pattern matching: like, using % to match any substring, _ to match any character, \ before '%' '_' '\' to denote escape characters

    e.g.: 'Intro%' = any string begin with 'Intro'

    '___' = any string with exactly length of 3

    In where clause:

    where *building* like '%Watson%';

- Attribute Specification in Select Clause

  Use select* to specify all attributes in result of from clause, use T.* to specify all attributes in relation T.

  e.g.: select *instructor.**

  from *instructor, teaches*

  where *instructor.ID = teaches.ID*;

  Indicates all attributes in instructor will be selected

- Ordering the Display of Tuples

  Order by clause lists the items in ascending order by an attribute. To specify descending order, use desc (or ascending use asc)

  e.g.: List all instructors order by salary in descending order, if more than one has the same salary, list by name in ascending order:

  select *

  from *instructor*

  order by *salary* desc, *name* asc;

- Where Clause Predicates
  - Between: inclusive comparison, equivalent to but simpler than "<= and >="
    Also we can use not between
  - And, or
  - Two tuples with all attributes the same can be evaluated as equal

3.5 Set Operations

Union, intersect, except respectively correspond to ∪, ∩, -

- Union

  e.g.: (select *course_id* from *section*
  where *semester* = 'Fall' and *year* = 2009)
  union
  (select *course_id* from *section*
  where *semester* = 'Spring' and *year* = 2010);

  If want to retain all duplicates in union, use "union all" at the place of "union"]

  Number of duplicates = total number of duplicates that appear in both sets
- Intersect

  Similar to union, return the intersection of two sets

  If duplicates need to be retained, use "intersect all"
- Except

  Similar to union and intersect, return the difference of two sets

  If duplicates need to be retained, use "except all"

  Some SQL implementations use keyword "minus", equivalent to "except"

3.6 NULL Values

NULL appears in relational operations including: arithmetic operations, comparison operations, and set operations

- Arithmetic operations: result is NULL if any input is NULL
- Comparison operations: result is unknown if any input is NULL (third logical value other than true and false), thus boolean operations are extended:

Treat unknown as a boolean variable X, result is X if it is still a variable

- And: true and unknown = unknown
  false and unknown = false
  unknown and unknown = unknown
- Or: true or unknown = true
  False or unknown = unknown
  Unknown or unknown = unknown
- Not: not unknown = unknown

If a tuple evaluates a predicate to either false or unknown, the tuple is not added

New predicates:

- is null, is not null
- Some implementations allow is unknown, is not unknown

3.7 Aggregate Functions

Definition: functions that take a collection (a set or multiset) of values as input and return a single value

SQL offers 5 built-in aggregate functions: avg, min, max, sum, count (also some, every for boolean operations)

- Average

  e.g.: Find the average salary of instructors in the Computer Science department

  ```
  select avg(salary) as avg_salary
  from instructor
  where dept_name = 'Comp. Sci.';
  ```

  Usually we give a name of the result of aggregate function to prevent SQL gives an arbitrary name. The result is a single tuple with a single attribute.

- Count

  e.g.: Find the total number of instructors who teach a course in Spring 2010 semester

  ```
  select count (distinct ID)
  from teaches
  ```

where *semester* = 'Spring' and *year* = 2010;

To count the total number of tuples without constraint, use count(*)

Distinct is illegal to go with count(*), but legal to go with max and min though result will not change. The default is all.

- Aggregation with Grouping

  Apply the aggregate function not only to a single set of tuples, but a group of sets of tuples

  Use group by clause, tuples with the same value on all attributes in the group by clause are placed in one group

  The return value is the aggregate function result on all groups, in one column

- The Having Clause

  Having vs. Where: constraints on group vs. constraints on tuples

  The meaning of a query containing aggregation, group by or having clauses is defined by the following sequence of operations:

  Aggregate -> from -> where -> group by -> having -> select (aggregate)

  - If no aggregation, the from clause is the first evaluated to get a relation
  - If a where clause is present, the predicate in the where clause is applied on the result relation of the from clause
  - Tuples satisfying the where predicate then placed into groups by group by clause if there is one, otherwise treated as one group
  - The having clause, if present, is applied to each group, groups do not satisfy the having clause are removed
  - The select clause uses the remaining groups to generate tuples of the result query, applying the aggregate functions to get a single result tuple for each group

- Aggregation with NULL and Boolean Values

  - All aggregate function except for count(*) ignore null values in inputs

    Count of an empty collection is 0

  - Some(<Boolean expression>) and every(<Boolean expression>) are used as aggregate functions for Boolean values, treated as existential and universal

quantifier

3.8 Nested Subqueries

The select clause itself creates a set, and we can do further manipulation based on such a set

in, not in, <comparison> some, <comparison> all, exist, not exist, unique, not unique provide predicate functions in where clause

- Set Membership

  e.g.: select distinct *course_id*
  from *section*
  where *semester* = 'Fall' and *year* = 2009 and
  course_id in (select *course_id*
  from *section*
  where *semester* = 'Spring' and *year* = 2010);

  The phrase "in" and "not in" are followed by a set in parentheses, the set can be a select-from-where clause, or enumerated sets, etc., as long as it is a set

  Is example above equivalent to union? If it is "not in", equivalent to difference?

- Set Comparison

  e.g.: Greater than at least one problem:

  select *name*
  from *instructor*
  where *salary* > some(select *salary*
  from *instructor*
  where *dept_name* = 'Biology');

  This some is different from aggregate function some(), the former produces a set while the latter produces a boolean value

  - SQL allows < some, <= some, >= some, = some and <> some
  - If it is "all", then it is greater than all

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

  Why no > some and > all?

- <> all is identical to not in, = some is identical to in

- Test for Empty Relations

  Using exists(subquery), a predicate which returns true if the subquery is not empty, false if it is empty

  e.g.: Find all courses taught in both Fall 2009 and Spring 2010

  ```
  select course_id
  from section as S
  where semester = 'Fall' and year = 2009 and
      exists(select *
              from section as T
              where semester = 'Spring' and year = 2010 and
                  S.course_id = T.course_id);
  ```

  - A correlation name from an outer query can be used in a subquery.

    A subquery uses a correlation name from an outer query is a correlated subquery.

  - Scoping rule: if same name defined globally and locally, local one applies; it is legal to use only correlation names defined in the subquery itself or its outer queries

  - *A* contains *B* ⇔ not exists (*B* except *A*)

- Test for the Absence of Duplicate Tuples

  The unique() function is similar to exist(), a predicate returns true if the argument subquery inside contains no duplicate tuples.

  - Evaluate true on empty result
  - 1 <= select(count(*A*)) ⇔ unique(select(*A*))
  - Also have not unique

- Subqueries in the From Clause

  Typical use: subquery contains "group by", outer query contains "where" ⇔ "having" without nested, because the subquery forms each group as a tuple

- The with Clause

  A collection of temporary definitions, for clearer writing

- with *R1*(*B1, B2, B3*…) as (<set or single value defined by subquery>) R2(*C1, C2, C3*…) as (<set or single value defined by subquery>) … *Rn*(…) select… from… where…

  Here R is a temporary relation

  e.g.: Find maximum budget of all departments

  with *max_budget*(*value*) as

  (select max(*budget*) from *department)*

  select *budget*

  from *department, max_budget*

  where *department.budget = max_budget.value;*

- Nested subqueries in from and where clause make the query harder to read and understand, and with clause can simplify it

- Scalar Subqueries

  A subquery returns only 1 tuple containing 1 attribute (a scalar) is a scalar subquery

  Hence can be used in select, where, and having clauses as values of an attribute

  e.g.: Count the number of instructors in each department

  select *dept_name*,

  (select count(*)

  from *instructor*

  where *department.dept_name = instructor.dept_name*)

  as *num_instructors*

  from *department*;

  - Technically a scalar subquery is still a 1 * 1 relation, but SQL implicitly extracts that scalar value out when it is used in an expression

3.9 Modification of the Database

- Deletion

  delete from *r* where *P*;

  If there is no where clause predicate, all tuples in the relation are deleted. The relation still exists but is empty (unlike drop).

e.g.: delete from *instructor*

where *salary* < (select avg(*salary*)

from *instructor*);

- Insertion

To insert data into a relation, either specify the values of a tuple, or write a query whose result is a set of tuples to be inserted

- Insert a specified tuple:

insert into *R*

values(v1, v2, …, vn);

- Insert a tuple resulted from a query:

insert into R

select … from … where …

e.g.: insert into *instructor*

select *ID, name, dept_name*, 18000

from *student*

where *dept_name* = 'Music' and *tot_cred* > 144;

Here we use select to specify the value of one attribute in all tuples

For both cases, if number / datatype of values does not match, insertion fail?

If the inserted tuple only havs partial attributes, assign NULL

Need to evaluate the select statement fully, i.e.: no *, or may have infinite loop

e.g.: insert into *student*

select *

from *student*;

Infinite loop in this case. If select is fully evaluated, or primary key constraint is added, infinite loop can be prevented.

Why primary key constraint prevents infinite loop?

- Updates

Update … set clause chooses the tuples to be updated by a query

update *R* set … where *P*;

e.g.: update *instructor*

set *salary* = *salary* * 1.05;

where *salary* <= 100000;

Equivalent to:

```
update instructor
set salary = case
                when salary <= 100000 then salary * 1.05
                else salary
             end;
```

- Case construct to perform updates with a single update statement:

```
update R
set A = case
            when P1 then Result1
            when P2 then Result2
            …
            when Pn then Resultn
            else Result0
        end
```

Chapter 4: Intermediate SQL

4.1 Join Expressions

A join operation is a Cartesian product of two relations that satisfies some join conditions, typically used in from clauses

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

Join conditions: defines which tuples in the two relations match, and what attributes are present in the result of the join

Join types	Join conditions
Inner join	natural
Left outer join	On <predicate>
Right outer join	Using (A1, A2, …, An)
Full outer join	

- Difference between the conditions:
  - A inner join B on (common attributes are the same) has reoccurrence of common attributes, but natural join does not
  - Using, like natural, omits the reoccurrence of common attributes

  When are the reoccurrence of attributes omitted?

  All joins omit reoccurrence of common attributes

  What is the difference between "natural left outer join" and "left outer join"?

  The same
- On Conditions: *A* join *B* on *P*;

  e.g.: select *

  from *student* join *takes* on *student.ID* = *takes.ID*;

  Equivalent to:

  select *

  from *student*, *takes*

where *student.ID = takes.ID*;

Different from:

select *

from *student* natural join *takes*; (Suppose ID is the only common attribute)

Since the natural join makes the common attribute as a single attribute, but on condition and where condition list two ID's, one for student and one for takes

Reasons for using on condition:

- Can take any SQL predicate, richer class of join conditions than natural join
- Do behave differently from where conditions
- More readable with on clause than where clause

- Outer Joins: method to prevent information loss

  Tuples may be lost in natural join, since some tuples in relation A may never appear in relation B, thus never satisfy the condition A natural join B

  Outer join preserves those tuples by creating tuples in the result containing NULL

  Three types of outer join:

  - Left outer join: preserves tuples only in the relation named before operation

    All tuples from left relation which has no match, padded with NULL, union with natural join
  - Right outer join: preserves tuples only in the relation named after operation

    All tuples from right relation which has no match, padded with NULL, union with natural join
  - Full outer join: preserves both

    All tuples from left and right relation which has no match in each other, padded with NULL, union with natural join
  - In contrast, those joins do not preserve non-matched tuples are inner joins (the default of "join" is inner join, thus "natural join" means "natural inner join"?)

  For instance, a left outer join is:

  - The inner join of two relations
  - Union with tuples such that:
  - The attributes derived from the LHS are filled in, as well as the remain ones

filled in with NULL

- Theta Join

  Allows arbitrary comparison relationships

  Example?

- Equijoin

  A theta join which uses equality operator

  A natural join is an equijoin on attributes with the same name (common attributes)

  Is join … using() an equijoin?

4.2 Views

Compute and store data in new relations and present the relations to users may work, but the underlying data may be affected. Thus we need a "virtual relation", not precomputed and stored, but instead computed every time whenever it is used.

- View: a virtual relation that is not part of the logical model, but computed to be visible for users
- Define a view with "create view" command

  create view *v* as <query expression>;
- Materialized View: views that are allowed to be stored, and must be updated if the actual relations used in the view definition change
  - The process of keeping up-to-date is called materialized view maintenance
  - SQL does not define materialization, but some extensions do
- Update a view / Modify relation through view

  e.g.: create view *faculty(A1, A2, A3)* based on *instructor(A1, A2, A3, A4)*

  If an insertion into faculty occurs:

  insert into *faculty* values(a1, a2, a3);

  Can be handled in two ways: 1, Reject the insertion; 2, Insert (a1, a2, a3, null)

  Since modifications on views can generate various problems, they are usually not permitted, except in limited cases. An SQL view is updatable (can be inserted, updated, or deleted) if they following conditions are satisfied by the query defining the view:

- The from clause has only one database relation
- The select clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification
- Any attribute not listed in the select clause can be set to null; that is, it does not have a not null constraint and is not part of a primary key
- The query does not have a group by or having clause

- Views Defined by Other Views
  - A view relation v1 is said to depend directly on a view relation v2 if v2 is used in the expression defining v1
  - A view relation v1 is said to depend on view relation v2 if either v1 depends directly to v2 or there is a path of dependencies from v1 to v2
  - A view relation v is recursive if it depends on itself
  - As long as the dependency is not recursive, the loop for view expansion on an expression will terminate

4.3 Transactions

A transaction is a unit of work, consisting of a sequence of query and/or update statements. It is isolated from concurrent transactions.

Starts implicitly when an SQL statement is executed; must end with one of the following SQL statements:

- Commit work: commits the current transaction, that is, makes the updates permanent in the database. After the transaction is committed, a new transaction is automatically started.
- Rollback work: causes the current transaction to be rolled back, that is, undoes all the updates. The database state is restored to before the first statement. It is useful when handling errors in the middle of a transaction.
- An abstraction of a transaction is atomic: either all the effects of the transaction are reflected in the database (committed), or none are (rollback).
- If program terminates without executing either termination command, it depends on the implementation that either it is committed or rollback

- Usually, by default, each statement itself is committed after it is executed (Automatic commitment for individual)
- SQL:1999 supports "begin atomic … end", not supported on most databases now

4.4 Integrity Constraints

Ensure changes made to the database by authorized users do not result in a loss of data consistency

- Constraints on a Single Relation

  Added with the create table command, in addition to primary key and foreign key constraints, including:

  - not null

    e.g.: *name* varchar(20) not null
  - unique

    unique(*A1, A2, A3, …, An*), that (*A1, A2, …, An*) is a candidate key, that no two tuples can be equal on all the listed attributes

    However candidate key attributes can be null unless be declared not null
  - check(<predicate>)

    At the end of create table, the predicate cannot contain a subquery

    What if the predicate evaluates false/unknown? Create table fails?
- Referential Integrity

  To ensure that a value appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation

  - Foreign Key

    Let r1, r2 be relations whose set of attributes are R1 and R2 respectively, with primary keys K1, K2. A subset alpha of R2 is a foreign key referencing K1 in relation r1 if it is required that, for every tuple t2 in r2, there must be a tuple t1 in r1 such tat t1.K1 = t2.alpha

    Do alpha = K1 for all cases? Can K1 be part of a super key?
  - Such requirements are referential-integrity constraints, or subset dependencies
- Cascading Actions in Referential Integrity

e.g.: create table *course*

(…

foreign key (*dept_name*) references department

on delete cascade

on update cascade,

…)

Clause "on delete cascade" and "on update cascade" means that if a tuple in the referred relation (department) is deleted or updated, the system does not reject for its violation of referential constraint, but delete or update the correspond tuple in course to satisfy it.

Alternative cascade actions: set null, set default

- Integrity Constraint Violation During a Transaction

  Three ways to avoid violations:

  - Insert its foreign key before the tuple
  - Set its foreign key to NULL then insert the tuple, then update the foreign key
  - Defer constraint checking

  Defer constraint checking is specified as:

  - Such transaction is allowed with clause "initially deferred" to be added to a constraint specification. Such constraint is deferrable.
  - Default is to check constraint immediately, reject any violation

- Complex Check Conditions and Assertions
  - For more complex custom constraints, use check(<predicate>)
  - An assertion is a predicate expressing a condition that we wish the database always to satisfy

    create assertion <assertion name> check (<predicate>);

4.5 More SQL Datatypes

4.6 Authorization

- Four kinds of privileges
  - Authorization to read data

- Authorization to insert new data
- Authorization to update data
- Authorization to delete data

Each type of authorization is called a privilege, analogous to superuser, administrator or operator for an operating system

What is the default privilege of a user?

- Granting/Revoking of Privileges

Use the grant statement:

grant <privilege list>

on <relation name or view name>

to <user / role list>

Privilege list includes:

- select: to read tuples
- update: to update tuples
- insert: to insert tuples
- delete: to delete tuples

Special user names:

- public: all current and future users of the system

On contrary, use the revoke statement in the place of grant to revoke privileges

- Roles

To specify the same level of privilege to a group of users

Roles can be granted to users as well as other roles

e.g.: create role *instructor*;

create role *dean*;

grant *instructor* to Amit;

grant *instructor* to Satoshi;

grant *instructor* to *dean*;

Thus the privilege of a user/role consist of:

- The privilege granted to the user/role itself
- The privilege granted to the roles that have been granted to the user/role

Grant Option, Cascade, Restrict, Authorization Graph

Grant option: the granted user can grant others

Chapter 6: Formal Relational Query Languages

6.1 The Relational Algebra

Relational algebra is a procedural query language, consisting of a set of operations that take one or two relations as input and produce a new relation.

Basic Relational-Algebra Operations:

- Select

  Use σ to denote a selection, predicate as its subscript

  e.g.: $\sigma_{dept\_name = \text{“Physics”} \wedge salary > 90000}(instructor)$

  Note that “select” in relational algebra does not correspond to “select” in SQL, but “where” clause

- Project

  A unary operation that returns its argument relation with certain attributes left out

  Use Π to denote a projection, attributes to appear as its subscript

  e.g.: $\Pi_{ID, name, salary}(instructor)$

- Composition of Relational Operations

  Operations can be composed together into a relational-algebra expression

  e.g.: Find the names of all instructors in physics department

  $\Pi_{name}(\sigma_{dept\_name = \text{“Physics”}}(instructor))$

- Union

  Also denote by ∪ as in set operations, requires the relations to be compatible:

  - The two relations must be of the same arity, that is, the same number of

attributes

- The domains of the i^{th} attribute for both relations must be the same, for all i

- Set difference

  Also denote by – as in set operations, requires the relations to be compatible

- Cartesian product

  Also denote by × as in set operations

  The prefix of relation, e.g.: instructor.name, can be removed if there is no ambiguity

- Rename

  Denote by ρ, the new name is its subscript

  - e.g.: $\rho_X(E)$ gives name X to the expression E
  - Or give n names to the attributes for an expression of arity n

    e.g.: $\rho_{X(A1, A2, A3, \ldots, An)}(E)$ rename expression E as X and its attributes A1, A2, …, An

- Intersection

  Denote by ∩

  $r \cap s = r - (r - s)$

- Natural join

  Denote by |×|

  $r \,|\times|\, s = \Pi_{R \cup S}(\sigma_{r.A1 = s.A1 \wedge r.A2 = s.A2 \wedge \ldots \wedge r.An = s.An}(r \times s))$

  where $r \cap s = \{A1, A2, \ldots, An\}$

  $r \cap s = \emptyset \Rightarrow r \,|\times|\, s = r \times s$

  Natural join is associative

- Theta join

  Denote by $|\times|_\theta$

  $r \,|\times|_\theta\, s = \sigma_\theta(r \times s)$

- Outer joins

  - Left outer join: $r \,]\times|\, s = r \,|\times|\, s \cup (r - \Pi_r(r \,|\times|\, s)) \times \{(NULL, NULL, \ldots, NULL)\}$
  - Right outer join: $r \,|\times[\, s = r \,|\times|\, s \cup (s - \Pi_s(r \,|\times|\, s)) \times \{(NULL, NULL, \ldots, NULL)\}$
  - Full outer join: $r \,]\times[\, s = r \,|\times|\, s \cup (r - \Pi_R(r \,|\times|\, s)) \times \{(NULL, NULL, \ldots, NULL)\}$

$\cup$ (s - Π_S(r |×| s)) × {(NULL, NULL, …, NULL)}

- Assignment

  Denote by ←

  e.g.: Natural join can be rewritten as:

  temp1 ← r × s

  temp2 ← $\sigma_{r.A1 = s.A1 \wedge r.A2 = s.A2 \wedge \ldots \wedge r.An = s.An}$ (temp1)

  result = $\Pi_{R \cup S}$(temp2)

Extended Relational-Algebra Operations:

- Generalized-projection:

  $\Pi_{F1, F2, \ldots, Fn}$ (E)

  Where F1, F2, … Fn are attribute names with arithmetic operations involved

  e.g.: $\Pi_{name, dept\_name, salary * 2}$ (*instructor*)

- Aggregation

  $_{G1, G2, \ldots, Gn}\mathcal{G}_{F1(A1), F2(A2), \ldots, Fm(Am)}$ (E)

  Where the left side of $\mathcal{G}$ is a list of attribute to be grouped by

  The right side of $\mathcal{G}$ is a list of aggregate functions

  e.g.: $\mathcal{G}_{avg(salary)}$(*instructor*)

  $\mathcal{G}_{count\text{-}distinct(ID)}(\ldots)$

  Note that "$\mathcal{G}$" is "calligraphic G", signifying an aggregate function is applied

6.2 The Tuple Relational Calculus

The tuple relational calculus is a nonprocedural query language, unlike relational-algebra expressions. It is expressed as:

{t | P(t)}, the set of tuples t that make predicate (formula) P true

A tuple variable t is a free variable if it has no quantifier, otherwise a bound variable

Use t[A] to denote the value of tuple t on attribute A, t ∈ r to denote t is in relation r

P(t) is built by atoms:

- s ∈ r where s is a tuple variable and r is a relation
- s[x] Θ u[y] where Θ is a comparison operator
- s[x] Θ c, where c is a constant

Formulae are built by following rules:

- An atom is a formula
- If P1 is a formula, then so are ¬P1 and (P1)
- If P1 and P2 are formulae, then so are P1 ∨ P2, P1 ∧ P2, P1 ⇒ P2
- If P1(s) is a formula containing a free variable s, and r is a relation, then

  ∃s ∈ r (P1(s)) and ∀s ∈ r (P1(s)) are also formulae

- Safety of Expressions

  The domain of a tuple t cannot be infinite, thus add constraints if the domain is infinite

6.3 The Domain Relational Calculus

Uses domain variables that take on values from an attribute's domain, rather than the entire tuple, a theoretical basis for QBE language, with form:

{<x1, x2, …, xn> | P(x1, x2, …, xn)}

where x1, x2, …, xn are domain variables, P is a formula composed of atoms, including:

- <x1, x2, …, xn> ∈ r, where x1…xn are domain variables or domain constants
- x Θ y where x, y are domain variables, require attributes x and y have comparable domains
- x Θ c where x is a domain variable and c is a constant in the domain of the attribute for which x is a domain variable

Build-up rules are the same as tuple relational calculus

e.g.: Find the instructor's ID, name, dept_name and salary for instructors whose salary is greater than 80000:

{<i, n, d, s> | <i, n, d, s> ∈ instructor ∧ s > 80000}

The formula provides a constraint on the variable's domain

- Safety of expressions requires
  - All values in the tuples of the expression are values from dom(P)
  - For every subformula contains existential quantifier, the subformula is true iff

there is a value x in dom(P1) such that P1(x) is true

- For every subformula contains universal quantifier, the subformula is true iff P1(x) is true for all values x from dom(P1)

Chapter 8: Normalization

First Normal Form (1NF): depends on key

- All values are atomic (data cannot be further broken down)
- No repeating groups
- Each tuple can be uniquely identified by the primary key

Second Normal Form (2NF): depends on whole key

- In 1NF
- All non-prime attributes depend on the whole primary key (cannot be partially dependent on proper subsets of any candidate key)

Third Normal Form (3NF): depends on nothing but whole key

- In 2NF
- Any non-prime attribute is contained in a candidate key
- In other words, all non-prime attributes are non-transitively dependent on every candidate key

Boyce-Codd Normal Form (BCNF, 3.5NF)

- In 3NF
- Difference from 3NF: In 3NF, if RHS is prime, don't care LHS, if RHS is non-prime, LHS must be superkey; In BCNF, either RHS is prime or not prime, LHS must be a superkey.
- All prime attributes are superkeys

To Build a Normal Form:

- Start from unnormalized data
- Divide non-atomic data, concatenate repeating groups, create separate tables for each tuple can be determined by a key -> 1NF
- Put partially dependent data and its key to another schema -> 2NF
- Put transitively dependent data and its direct key to another schema -> 3NF

Check functional dependency of decomposition, then

Do/Check decomposition in any form

3NF check and decomposition:

3 constraints check:

1, Trivial

2, LHS is a key, by closure of LHS

3, RHS – LHS is a prime attribute (part of a key, so we need to find all keys first)

Check dependencies one by one, if one not in the three, relation is not in 3NF

Need further decomposition:

Find the Canonical Cover and recreate the relations

To find the C Cover:

1, Combine the similar LHS's together

2, Reduce LHS: if LHS has more than one attribute, try if one can be removed

3, Reduce RHS: if RHS has more than one attribute, try if one can be removed

Take each functional dependency from the C Cover, create relations for each

e.g.:

F = {A -> BC, B -> C, A -> B, AB -> C}, R(A, B, C, D)

D is independent of any other, so none of A, B, C is a key

But combine one with D together, e.g.: {A, D} is a key, A is a prime attribute

Found that R is not in 3NF because B->C violates all constraints

Find C Cover:

1, Combine: F = {A -> BC, B -> C, AB-> C}

2, Reduce LHS: F = {A-> BC, B-> C, B->C} thus F = {A-> BC, B-> C}

3, Reduce RHS: F = {A -> B, B-> C}, thus it is the C Cover

Thus R1(A, B), R2(B, C) from each dependency

BCNF check and decomposition:

Two constraints, check one by one on the dependencies:

1, The dependency is trivial, RHS is subset of LHS

2, For all transitions, LHS is a key

A key has a closure that includes all attributes

If both are not satisfied, the relation is not in BCNF

Need to decomposed more to 2 relations:

1, R1 = LHS and RHS of an unsatisfied dependency (alpha and beta)

2, R2 = R – (beta - alpha)

Check again until all are in BCNF

e.g.:

F = {A-> B, B->C, B->D}, R(A, B, C, D)

A+ = {A, B, C, D}

B+ = {B, C, D}

Thus B is not a key, the second and third dependencies make R not in BCNF

Take away both sides of an unsatisfied dependency to form a new relation R1

In this case, alpha -> beta = B -> C, take R1(B, C)

The other relation is R – (beta - alpha) = R – {C}, thus R2(A, B, D)

Check again that R2 is not in BCNF because B->D is still not satisfied

Thus decompose R2 into R3(B, D) and R4 = R2 – (D - B) = R2 – D = (A, B)

Consequently, R1(B, C), R3(B, D), R4(A, B)

Chapter 11: Indexing and Hashing

Advantage of using index:

Better for searching, worse for update (due to keeping the B+ tree balanced)

Primary index vs. secondary index:

Each data set has only one primary index (for values of each record)

But can have many secondary index, e.g.: for each attribute

Dense vs. sparse index:

Dense: Each value of a record has a match in the index

Sparse: Note each value of a record has a match in the index

If an attribute is indexed, to check if a select-from-where clause performs better, compare without index (sequentially go down the records) and with index (sequentially go down the sorted index in B+ tree). Usually the B+ tree is better for range query because its node is sorted.

B+ Tree

Pointer, value pairs, in sorted sequence

Left pointer of a value points to the child where all values smaller than it

Right pointer opposite

N = n means the maximum children of each node is n, max value is n – 1

Split: when max value number exceeds n – 1, split into 2 parts, if even number, they are same size, if odd, left is 1 fewer than right

After split, copy the first element of right to the last spot in parent

- Search, find record with search-key value V

  C = root

  While C is not leaf {

Let I be min value such that V <= Ki

If no such exists, C = last non-null pointer in C

Else{ if V == Ki, C = Pi+ 1 else C = Pi}

}

Let I be min value such that Ki = V

If there is such a value I, follow pointer Pi to the desired record

Else no record with the search-key value exists

- Insertion

Find the leaf node in which the search key value would appear

If the search key value already present in the leaf node

Add record to file

If necessary, add a pointer to the bucket

B Tree

Hashing

Comparison between B+ Tree, B Tree and Hashing

Chapter 13 Optimization Query

When optimizing an expression tree, take projection after each selection, project the attributes that are needed in next intermediate states and in final projection. That means, filter out useless attributes.

Avoid useless tuples repeatedly occur in selected results

Draw initial tree, place single predicates (for one relation only) immediately above leaves. Find equality queries in predicate (sigma), then convert Cartesian product + equality to join. Projection after each selection.

B tree and B+ tree are almost the same, but B tree can have value pointers in internal nodes, while B+ tree only has value pointers at the leaves. B+ tree leaves nodes are

sorted, so B+ tree is good for range queries by just going through the leaves. Also implementation for B+ tree is easier than B tree. B+ tree on average is better than B tree.

Hash only works for equality, while trees can answer both equal and range queries. Hard to find a good hash function.

Query optimization in physical level? E.g. 12.5

Choose form: Nested loop, blocked nested loop, indexed nested loop, merge join, hash join

Indexed loop: available only if there is index

Hash join: need to build a hash table by reading the whole file, then hash it to another file, for both, then join the hashed files, very costly

Merge join: sort both first, then join the sorted tables

Block nested loop: in the case of limited memory, (especially limited memory that can hold one block) use this, load block by block, join each block, load next block

Nested loop: the same as block nested loop if infinite memory, but usually cannot be done for limited memory

How to use indices to make arbitrary queries faster?

Without other assumptions, use B+ tree for range query, use hash for equality query.

If data file is clustered (sorted), directly jump to beginning of gpa=3.5, linearly go.

If not clustered, use B+ tree and choose the fewer seeks (> 3.5 or < 3.5).