

Extending SQL

1/45

Limitations of Basic SQL

2/45

What we have seen of SQL so far:

- data definition language (`create table(...)`)
- constraints (domain, key, referential integrity)
- query language (`select...from...where...`)
- views (give names to SQL queries)

This is not sufficient to write complete applications.

Need to write *programs* to manipulate database.

Extending database functionality would also help.

Extending SQL

3/45

Ways in which standard SQL might be extended:

- new data types (incl. constraints, I/O, indexes, ...)
- more operations/aggregates for use in queries
- more powerful constraint checking
- event-based triggered actions
- different kinds of queries (e.g. recursive)

All are potentially useful in application development.

New Data Types

4/45

SQL data definition language provides:

- atomic types: integer, float, character, boolean
- ability to define tuple types (`create table`)

SQL also provides mechanisms to define new types, e.g.

```
create domain Positive as integer check (value > 0);
create type Rating as enum ('poor', 'ok', 'excellent');
create type Pair as (x integer, y integer);
```

CREATE TYPE also allows new basic types to be defined.

New Functions

5/45

SQL provides for new functions via *stored procedures*.

PostgreSQL has functions in: SQL, PLpgSQL, Python, ...

```
create function
  f(arg1 type1, arg2 type2, ...) returns type
as $$ ... function body ... $$
language Language [ mode ];
```

Possible modes (can make *big* performance difference)

- `immutable` ... does not access database (fast)
- `stable` ... does not modify the database

- **volatile ... may change the database** (slow, default)

Exercise: functions on (sets of) integers

6/45

Write PLpgSQL functions:

```
-- factorial n!
function fac(n integer) returns integer

-- returns integers 1..hi
function iota(hi integer) returns setof integer

-- returns integers lo..hi
function iota(lo integer, hi integer)
    returns setof integer

-- returns integers lo,lo+inc,..hi
function iota(lo integer, hi integer, step integer)
    returns setof integer
```

[\[Solution\]](#)

Queries

7/45

Advanced Query Types

8/45

Many specialised types of query have been identified.

We have seen: select/project/join, aggregation, grouping

Many modern queries (e.g. skyline) come from OLAP.

Two important standard query types:

- *recursive* ... e.g. to manage hierarchies, graphs (1999)
- *window* ... e.g. to "spread" group-by summaries (2003)

Window Functions

9/45

Group-by allows us to

- summarize a set of tuples
- that have common values for a set of attributes

E.g. average mark for each student (not the WAM)

```
select student, avg(mark)
from   CourseEnrolments
group  by student;
```

Produces a single summary tuple for each group.

... Window Functions

10/45

Window functions allow us to

- compute summary values for a group
- append the summary value to each tuple in the group

E.g. attach student's average mark to each enrolment

```
select *, avg(mark)
over  (partition by student)
```

from CourseEnrolments;

... Window Functions

11/45

Comparison of group by and partition by:

```
select student, avg(mark) ... group by student
```

student	avg
46000936	64.75
46001128	73.50

```
select *, avg(mark) over (partition by student) ...
```

student	course	mark	grade	stueval	avg
46000936	11971	68	CR	3	64.75
46000936	12937	63	PS	3	64.75
46000936	12045	71	CR	4	64.75
46000936	11507	57	PS	2	64.75
46001128	12932	73	CR	3	73.50
46001128	13498	74	CR	5	73.50
46001128	11909	79	DN	4	73.50
46001128	12118	68	CR	4	73.50

Exercise: Underachievers

12/45

Using window functions, write an SQL function to find:

- all of the students in a given course
- whose mark is < 60% of average mark for course

```
create or replace function
  under(integer) returns setof CourseEnrolments
as $$
...
$$ language sql;
```

WITH Queries

13/45

We often break a complex query up into views, e.g.

```
create view V as
select a,b,c from ... where ...;
```

```
create view W as
select d,e from ... where ...;
```

```
create view Result(x,y,z) as
select V.a, V.b, W.e
from V join W on (v.c = W.d);
```

But we don't always need/want the views to persist.

... WITH Queries

14/45

WITH allows scoped/temporary views, e.g.

```
with V as (select a,b,c from ... where ...),
     W as (select d,e from ... where ...)
select V.a as x, V.b as y, W.e as z
from V join W on (v.c = W.d);
```

The views V and W

- only exist while this query is evaluated
- are not accessible in any other context

V and W are also called "common table expressions" (CTEs)

... WITH Queries

15/45

Note that named subqueries achieve the same effect:

```
select V.a as x, V.b as y, W.e as z
from   (select a,b,c from ... where ...) as V,
       (select d,e from ... where ...) as W
where  V.c = W.d;
```

For this purpose, WITH is a syntactic convenience.

However, WITH also provides recursive queries.

Recursive Queries

16/45

Recursive queries are defined as:

```
with recursive T( $a_1$ ,  $a_2$ , ...) as
(
    non-recursive select
  union
    recursive select involving T
)
select ... from T where ...
```

$T(a_1, a_2, \dots)$ is a recursively-defined view.

... Recursive Queries

17/45

Example: generate sum of first 100 integers

```
with recursive nums(n) as (
  select 1
  union
    select n+1 from nums where n < 100
)
select sum(n) from nums;
-- which produces ...
sum
-----
5050
```

... Recursive Queries

18/45

The subqueries generating T cannot be arbitrary

- non-recursive select
 - does not refer to T
 - generates an initial set of tuples for T (*initialisation*)
- recursive select
 - must be a query involving T
 - must include a where condition
 - must eventually return an empty result (*termination*)

... Recursive Queries

19/45

Semantics of recursive query yielding $T(a_1, a_2, \dots)$:

```
-- res, work, tmp are all temporary tables
res = result of non-recursive query
work = res
while (work is not empty) {
  -- using work as the value for T ...
  tmp = result of recursive query
  res = res + tmp
  work = tmp
}
return res
```

Exercise: Recursive Queries

20/45

Re-write the `iota(lo,hi)` function so that it uses a recursive query.

function `iotar(integer,integer)` returns integer

Re-write the `facultyOf(oid)` function so that it uses a recursive query.

function `facOf(integer)` returns integer

[\[Solution\]](#)

Aggregates

21/45

Aggregates

22/45

Aggregates reduce a collection of values into a single result.

Examples: `count(Tuples)`, `sum(Numbers)`, etc.

The action of an aggregate function can be viewed as:

```
AggState = initial state
for each item V {
  # update AggState to include V
  AggState = newState(AggState, V)
}
return final(AggState)
```

... Aggregates

23/45

Aggregates are commonly used with `GROUP BY`.

In that context, they "summarise" each group.

Example:

<p>R</p> <table border="1"> <thead> <tr> <th>a</th> <th>b</th> <th>c</th> </tr> </thead> <tbody> <tr><td>1</td><td>2</td><td>x</td></tr> <tr><td>1</td><td>3</td><td>y</td></tr> <tr><td>2</td><td>2</td><td>z</td></tr> <tr><td>2</td><td>1</td><td>a</td></tr> <tr><td>2</td><td>3</td><td>b</td></tr> </tbody> </table>	a	b	c	1	2	x	1	3	y	2	2	z	2	1	a	2	3	b	<pre>select a,sum(b),count(*) from R group by a</pre> <table border="1"> <thead> <tr> <th>a</th> <th>sum</th> <th>count</th> </tr> </thead> <tbody> <tr><td>1</td><td>5</td><td>2</td></tr> <tr><td>2</td><td>6</td><td>3</td></tr> </tbody> </table>	a	sum	count	1	5	2	2	6	3
a	b	c																										
1	2	x																										
1	3	y																										
2	2	z																										
2	1	a																										
2	3	b																										
a	sum	count																										
1	5	2																										
2	6	3																										

User-defined Aggregates

24/45

SQL standard does not specify user-defined aggregates.

But PostgreSQL provides a mechanism for defining them.

To define a new aggregate, first need to supply:

- *BaseType* ... type of input values
- *StateType* ... type of intermediate states
- state mapping function: *sfunc(state,value) → newState*
- [optionally] an initial state value (defaults to null)
- [optionally] final function: *ffunc(state) → result*

... User-defined Aggregates

25/45

New aggregates defined using CREATE AGGREGATE statement:

```
CREATE AGGREGATE AggName(BaseType) (  
    sfunc      = NewStateFunction,  
    stype      = StateType,  
    initcond   = InitialValue,  
    finalfunc  = FinalResFunction,  
    sortop     = OrderingOperator  
);
```

- initcond (type *StateType*) is optional; defaults to NULL
- finalfunc is optional; defaults to identity function
- sortop is optional; needed for min/max-type aggregates

... User-defined Aggregates

26/45

Example: defining the count aggregate (roughly)

```
create aggregate myCount(anyelement) (  
    stype      = int,      -- the accumulator type  
    initcond   = 0,       -- initial accumulator value  
    sfunc      = oneMore  -- increment function  
);  
  
create function  
    oneMore(sum int, x anyelement) returns int  
as $$  
begin return sum + 1; end;  
$$ language plpgsql;
```

... User-defined Aggregates

27/45

Example: sum2 sums two columns of integers

```
create type IntPair as (x int, y int);  
  
create function  
    AddPair(sum int, p IntPair) returns int  
as $$  
begin return p.x + p.y + sum; end;  
$$ language plpgsql;  
  
create aggregate sum2(IntPair) (  
    stype      = int,  
    initcond   = 0,  
    sfunc      = AddPair  
);
```

Exercise: Defining Aggregates

28/45

Define a concat aggregate that

- takes a column of string values
- returns a comma-separated string of values

Example:

```
select count(*), concat(name) from Employee;
-- returns e.g.
count |          concat
-----+-----
4 | John,Jane,David,Phil
```

Use it to get a list of beers liked by each drinker.

Constraints

29/45

Constraints

30/45

So far, we have considered several kinds of constraints:

- **attribute** (column) constraints
- **relation** (table) constraints
- **referential integrity** constraints

Examples:

```
create table Employee (
  id      integer primary key,
  name    varchar(40),
  salary  real,
  age     integer check (age > 15),
  worksIn integer
          references Department(id),
  constraint PayOk check (salary > age*1000)
);
```

... Constraints

31/45

Column and table constraints ensure validity of one table.

RI constraints ensure connections between tables are valid.

However, specifying validity of entire database often requires constraints involving multiple tables.

Simple example (from banking domain):

```
for all Branches b
  b.assets == (select sum(acct.balance)
               from   Accounts acct
               where  acct.branch = b.location)
```

i.e. assets of a branch is sum of balances of accounts held at that branch

Assertions

32/45

Assertions are schema-level constraints

- typically involving multiple tables
- expressing a condition that must hold at all times
- need to be checked on each change to relevant tables
- if change would cause check to fail, reject change

SQL syntax for assertions:

```
CREATE ASSERTION name CHECK (condition)
```

The *condition* is expressed as "there are no violations in the database"

Implementation: ask a query to find all the violations; check for empty result

Example: #students in any UNSW course must be < 10000

```
create assertion ClassSizeConstraint check (
  not exists (
    select c.id from Courses c, CourseEnrolments e
    where c.id = e.course
    group by c.id having count(e.student) > 9999
  )
);
```

Needs to be checked after every change to either Course or CourseEnrolment

Example: assets of branch = sum of its account balances

```
create assertion AssetsCheck check (
  not exists (
    select branchName from Branches b
    where b.assets <>
      (select sum(a.balance) from Accounts a
       where a.branch = b.location)
  )
);
```

Needs to be checked after every change to either Branch or Account

On each update, it is expensive

- to determine which assertions need to be checked
- to run the queries which check the assertions

A database with many assertions would be way too slow.

So, most RDBMSs do not implement general assertions.

Typically, *triggers* are provided as

- a lightweight mechanism for dealing with assertions
 - a general event-based programming tool for databases
-

Triggers

Triggers

Triggers are

- procedures stored in the database
- activated in response to database events (e.g. updates)

Examples of uses for triggers:

- maintaining summary data
 - checking schema-level constraints (assertions) on update
 - performing multi-table updates (to maintain assertions)
-

Triggers provide event-condition-action (ECA) programming:

- an *event* activates the trigger
- on activation, the trigger checks a *condition*
- if the condition holds, a procedure is executed (the *action*)

Some typical variations on this:

- execute the action before, after or instead of the triggering event
- can refer to both old and new values of updated tuples
- can limit updates to a particular set of attributes
- perform action: for each modified tuple, once for all modified tuples

... Triggers

39/45

SQL "standard" syntax for defining triggers:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE} Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

Possible *Events* are INSERT, DELETE, UPDATE.

FOR EACH ROW clause ...

- if present, code is executed on each modified tuple
- if not present, code is executed once after all tuples are modified, just before changes are finally COMMITed

Trigger Semantics

40/45

Triggers can be activated BEFORE or AFTER the event.

If activated BEFORE, can affect the change that occurs:

- NEW contains "proposed" value of changed tuple
- modifying NEW causes a different value to be placed in DB

If activated AFTER, the effects of the event are visible:

- NEW contains the current value of the changed tuple
- OLD contains the previous value of the changed tuple
- constraint-checking has been done for NEW

Note: OLD does not exist for insertion; NEW does not exist for deletion.

... Trigger Semantics

41/45

Consider two triggers and an INSERT statement

```
create trigger X before insert on T Code1;
create trigger Y after insert on T Code2;
insert into T values (a,b,c,...);
```

Sequence of events:

- execute Code1 for trigger X
- code has access to (a,b,c,...) via NEW
- code typically checks the values of a,b,c,...
- code can modify values of a,b,c,... in NEW
- DBMS does constraint checking as if NEW is inserted
- if fails any checking, abort insertion and rollback
- execute Code2 for trigger Y
- code has access to final version of tuple via NEW
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: there is no OLD tuple for an INSERT trigger.

Consider two triggers and an UPDATE statement

```
create trigger X before update on T Code1;
create trigger Y after update on T Code2;
update T set b=j,c=k where a=m;
```

Sequence of events:

- execute Code1 for trigger X
- code has access to current version of tuple via OLD
- code has access to updated version of tuple via NEW
- code typically checks new values of b,c,...
- code can modify values of a,b,c,... in NEW
- do constraint checking as if NEW has replaced OLD
- if fails any checking, abort update and rollback
- execute Code2 for trigger Y
- code has access to final version of tuple via NEW
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: both OLD and NEW exist in UPDATE triggers.

Consider two triggers and an DELETE statement

```
create trigger X before delete on T Code1;
create trigger Y after delete on T Code2;
delete from T where a=m;
```

Sequence of events:

- execute Code1 for trigger X
- code has access to (a,b,c,...) via OLD
- code typically checks the values of a,b,c,...
- DBMS does constraint checking as if OLD is removed
- if fails any checking, abort deletion (restore OLD)
- execute Code2 for trigger Y
- code has access to about-to-be-deleted tuple via OLD
- code typically does final checking, or modifies other tables in database to ensure constraints are satisfied

Reminder: tuple NEW does not exist in DELETE triggers.

Triggers in PostgreSQL

PostgreSQL triggers provide a mechanism for

- INSERT, DELETE or UPDATE events
- to automatically activate PLpgSQL functions

Syntax for PostgreSQL trigger definition:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE} Event1 [OR Event2 ...]
ON TableName
[ WHEN ( Condition ) ]
FOR EACH {ROW|STATEMENT}
EXECUTE PROCEDURE FunctionName(args...);
```

There is no restriction on what code can go in the function.

However a BEFORE function must contain one of:

```
RETURN old;      or      RETURN new;
```

depending on which version of the tuple is to be used.

If BEFORE trigger returns old, no change occurs.

If exception is raised in trigger function, no change occurs.