PostgreSQL –**psql**

Any SQL command that has been typed but not yet sent for execution is stores in a memory buffer called *query buffer*. The contents of this buffer can be edited by invoking a configurable text editor within **psql**.

Useful online documentation
psql https://www.postgresql.org/docs/current/static/app-psql.html https://www.postgresql.org/docs/curent/static/tutorial-accessdb.html https://www.citusdata.com/blog/2017/07/16/customizing-my-postgres-shell-using-psqlrc
PostgreSQL http://www.postgresql.org/docs/current/static/index.html http://www.comp.nus.edu.sg/_cs2102/postgresql/doc/html
Nano editor https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor
pgAdmin http://www.comp.nus.edu.sg/_cs2102/using-pgadmin.pdf

Meta-commands	
\q	Quit **psql**
\h	Display all SQL commands with available syntax help
\h *COMMAND*	Display syntax of *COMMAND* (E.g. \h *create table)*
\d	List all created tables
\d *TABLE*	List information on relation named *TABLE*
\p	Display contents of query buffer (if current query buffer is empty, display the most recently executed query)
\w *FILE*	Display contents of query buffer to the file named *FILE* (if current query buffer is empty, output the most recently executed query to *FILE*)
\r	Clear query buffer
\e	Invoke text editor to edit the contents of the query buffer (if query buffer is empty, edit the most recently executed buffer)
\e *FILE*	Invoke text editor to edit the contents of a file named *FILE* (contents of edited file will be copied to the query buffer at the end of the edit session)
\o *FILE*	Enable future query results to be saved to the file named *FILE*
\g	Send contents of current query buffer to the server for execution (if current query buffer is empty, the most recently sent query is re-executed)
\i *FILE*	Reads contents from file named *FILE* and sends their contents to the server for execution (Alternative: *psql < test.sql* OR *psql -f test.sql*)
\!	Escapes from **psql** session to a sub-shell (**psql** session resumes when the sub-shell is exited)

Structured Query Language (SQL)

- SQL is not a general-purpose language but a *domain-specific language (DSL)*
- Unlike relational algebra which is a procedural language, SQL is a *declarative language* (i.e. focusing on *what* to compute – property of data to retrieve, not *how* to compute)
- SQL consists of 2 main parts:

1) Data Definition Language (DDL): create/delete/modify schemas
2) Data Manipulation Language (DML): ask queries, insert/delete/modify data

Create/Drop table

```
-- Column name, data type
create table Students (
    studentId     integer,
    name          varchar(100),
    birthdate     date
);

-- Delete/remove table
drop table Students;
```

`varchar(100)` is a variable-length string (up to a 100 characters long)

```
/* SQL also supports C-style comments beside preceding comments by two hyphens */
```

Data Types

- Built-in data types:

1) boolean
2) integer, numeric, real
3) char(50), varchar(50), text
4) date, time, timestamp

- SQL also supports user-defined data types (Refer to online documentation)
- Domain of each data type includes the special value *null*

Null values

- SQL uses a 3-valued logic system: *true, false* and *unknown*

x	y	x AND y	x OR y	NOT x
FALSE	FALSE	FALSE	FALSE	TRUE
	UNKNOWN		UNKNOWN	
	TRUE		TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
	UNKNOWN	UNKNOWN		
	TRUE		TRUE	
TRUE	FALSE	FALSE	TRUE	FALSE
	UNKNOWN	UNKNOWN		
	TRUE	TRUE		

- Result of **comparison** operation involving *null* value: *unknown*
- Result of **arithmetic** operation involving *null* value: *null*

```
-- IS NULL comparison predicate
-- Checking if a value is equal to null
-- Cannot use logical operator
x IS NULL
x IS NOT NULL

-- IS DISTINCT FROM comparison predicate
-- Treat null values as ordinary values for comparison
-- Both values null: false
-- Only one value null: true
-- Both not null: equivalent to  "x <> y"
x IS DISTINCT FROM y
```

Constraints		
Constraint Types	- Not-null constraints - Unique constraints - Primary key constraints - Foreign key constraints - General constraints	A constraint is **violated** if it evaluates to *false* (unknown is fine)
Constraint Specifications	- Column constraints (attaches constraints to column/attribute) - Table constraints - Assertions (not covered)	

Not-null constraints	`-- Every student must be a non-null value`
	
	`-- Column constraint`
	`create table Students (`
	`    studentId    integer,`
	`    name         varchar(100) not null,`
	`    birthdate    date`
	`);`
	
	`-- Table constraint`
	`-- Multiple checks can be done in a single query`
	`create table Students (`
	`    studentId    integer,`
	`    name         varchar(100),`
	`    birthdate    date`
	`    check(name is not null)`
	`);`	

Unique constraints	- *null* values do not violate constraints - *unique* constraints are usually bundled with a "*not null*" constraint	

	`create table Students (`
	`    studentId    integer unique,`
	`    name         varchar(100),`
	`    birthdate    date`
	`);`
		Unique constraint is violated if there exists 2 records
		x, y in *Students*,
		where
		"*x.studentId* <> *y.studentId*"
		evaluates to *false*
		(i.e. unique *studentIds* wanted)

	`create table Census (`
	`    city          varchar(50),`
	`    state         char(2),`
	`    population    integer,`
	`    unique(city, state)`
	`);`
		Generally, table constraints are applied when there needs to be more than 1 attribute with a *unique* identity

Primary key constraints	create table Students (studentId integer primary key, name varchar(100), birthdate date); -- Equivalent definition create table Students (studentId integer unique not null, name varchar(100), birthdate date);	
	create table Enrolls (sid integer, cid integer, grade char(2), primary key(sid, cid));	
Foreign key constraints	- Note: **Reference tables** need to be declared first before declaring foreign key - Strictly, the attribute being referenced should be a *primary key* or *unique* - But in SQL, the rules are relaxed – attributes referenced just need to be *unique*, not necessarily a primary key - Referencing attributes need not be unique	
	create table Enrolls (-- Column constraint -- Students: table referenced, studentId: primary key of table sid integer references Students(studentId), cid integer, grade char(2), primary key(sid, cid), -- Equivalent: Table constraint foreign key(cid) references Courses(courseId));	
General constraints	create table Movies (title integer, director integer, releaseYear char(2), -- Values with 3 digits with 2 decimal points (E.g. 0.00) rating numeric(3, 1), primary key(title), -- Not able to put as column constraint, else -- it becomes an AND constraint check(releaseYear > 2010 or rating > 8.0));	

Database Modifications

Insert

```
create table Students (
    studentId    integer primary key,
    name         varchar(100) not null,
    birthDate    date,
    -- If value is not specified/missing value during insertion,
    -- the record for that attribute is replaced with a default set
    -- The default default set is  ‘null’
    dept         varchar(20) default  ‘CS’
);

insert into Students
values (12345,  ‘Alice’ ,  ‘1999-12-25’ ,  ‘Maths’ );

-- To specify attributes which need non-null values
insert into Students (name, studentId)
values ( ‘Bob’ , 67890);
```

Delete

Note: Table *still* exists, just empty

```
create table Students (
    studentId    integer primary key,
    name         varchar(100) not null,
    birthDate    date,
    dept         varchar(20) default  ‘CS’
);

-- Remove all students
delete from Students;
-- Remove all students from Maths department
delete from Students
WHERE dept =  ‘Maths’ ;
```

Update

Changing values of certain records/contents

```
create table Accounts (
    accountId    integer primary key,
    name         varchar(100) not null,
    birthDate    date,
    balance      numeric(10,2) default 0.00
);

-- Add 2% interest to all accounts
update Accounts
set balance = balance * 1.02;
-- Add $500 to account 12345
update Accounts
set balance = balance + 500
where accountId = 12345;
```

- While executing the update, if constraints are violated, the system will reject the update
- The *where* condition need not involve a primary key

<table>
<tr><td><b>Modifying schema</b></td><td>-- Add/remove/modify columns
alter table Students alter column dept drop default;
alter table Students drop column dept;
alter table Students add column faculty varchar(20);
-- etc.

-- Add/remove constraints
-- etc, for more details: refer to documentation</td></tr>
<tr><td>Checking of constraints</td><td>- By default, constraints are checked immediately at the end of SQL statement execution
▪ A violation will cause the statement to be <b>rollbacked</b>
- Constraint checking could also deferred to the end of transaction execution (there may be times when constraints are violated <b><i>during</i></b> execution but at the <b><i>end</i></b>, there may not be a violation in constraint)
▪ A violation will cause the transaction to be <b>aborted</b>
- Specify type of constraint checking as part of constraint declaration/configure: use <i>set constraints</i> command</td></tr>
<tr><td>Handling <b>foreign key constraint violations</b></td><td>- Deletion/update of a referenced tuple could violate a foreign key constraint

<table>
<tr><th colspan="2"><i>FOREIGN KEY … REFERENCES … ON DELETE/UPDATE action</i></th></tr>
<tr><td>NO ACTION</td><td>Rejects <i>DELETE/UPDATE</i> if it violates constraint (default option)</td></tr>
<tr><td>RESTRICT</td><td>Similar to <i>NO ACTION</i> except that constraint checking can’t be deferred</td></tr>
<tr><td>CASCADE</td><td>Propagates <i>DELETE/UPDATE</i> to referencing tuples (propagating records action)</td></tr>
<tr><td>SET DEFAULT</td><td>Updates foreign keys of referencing tuples to some default value</td></tr>
<tr><td>SET NULL</td><td>Updates foreign keys of referencing tuples to <i>null</i> values</td></tr>
</table></td></tr>
<tr><td></td><td>create table Enrolls (
    sid integer, cid integer, grade char(2),
    primary key (sid, cid),
    foreign key (sid) references Students
        on delete cascade
        on update no action,
    foreign key (cid) references Courses
        on update cascade
        on delete set null
);</td></tr>
</table>

Transactions	- A *transaction* consists of one or more *update/retrieval* operations (i.e. SQL statements) - Good for multiple updates - Abstraction for representing a logic unit of work - The *begin* command starts a new transaction - Each transaction must end with either a *commit* or a *rollback* command - A *rollback* is **aborting** the execution of a command where the original state before the transfer is restored	
	ACID Properties: - **A**tomicity: Either all the effects of the transactions are reflected in the database or none are - **C**onsistency: The execution of a transaction in isolation preserves the consistency of the database - **I**solation: The execution of a transaction is isolated from the effects of other concurrent transaction executions - **D**urability: The effects of a committed transaction persists in the database even in the presence of system failures	

	`-- Performing bank transfer`
	`begin;`
	`update Accounts`
	`set balance = balance + 100`
	`where accountId = 456;`
	
	`update Accounts`
	`set balance = balance – 100`
	`where accountId = 123;`
	`commit;`
		- "All or nothing" – mainly, when using *begin,commit*, the entire block of commands between these keywords has to go through successfully or none is executed (i.e. whole transaction is aborted)
		- E.g. If a system fails, it will execute a rollback

<table>
<tr><td rowspan="2"><b>Simple queries</b></td><td colspan="2">- Basic form of SQL query consists of 3 clauses:
1) from-list (from): specifies list of relations
2) qualification (where): specificies conditions on relations
3) select-list (select): specifies columns to be included in output table
- Output relation could contain duplicate records if distinct is not used in the select clause</td></tr>
<tr><td>select distinct a1, a2, … am
from r1, r2, …, rn
where c;</td><td>Equivalent to:
$\pi_{a1, a2, \ldots am}(\sigma_c(r1 \times r2 \times \ldots \times rn))$</td></tr>
<tr><td>Removing duplicate records (distinct)</td><td>select distinct A, C
from R;</td><td>Two tuples – (a1, c1) and (a2, c2) - are <i>distinct</i> when true of:
<i>(a1 is distinct from a2)</i>
<i>or</i>
<i>(c1 is distinct from c2)</i></td></tr>
<tr><td>Renaming column (as)</td><td>select item, price*qty as cost
from Orders;</td><td>Similar to <i>mutate</i> in <i>dplyr</i></td></tr>
<tr><td>String concatenation (||)</td><td colspan="2">-- String concatenation with ||
select 'Price of' ||pizza|| 'is' ||round(price/1.3)|| 'USD' as menu
from Sells
where rname = 'Corleone Corner' ;</td></tr>
<tr><td>Pattern matching (like)</td><td>-- Underscore (_) symbol matches any single character
-- Percentage (%) symbol matches any sequence of 0 or more characters
select cname
from Customers
where cname like '_ _ _%e'</td><td>Finds customers names ending with "e" that consists of at least 4 characters</td></tr>
<tr><td>Set operations</td><td colspan="2">- Let Q1 and Q2 denote SQL queries that output union-compatible relations
1) Q1 union Q2 = Q1 ∪ Q2
2) Q1 intersect Q2 = Q1 ∩ Q2
3) Q1 except Q2 = Q1 - Q2
- union, intersect, except: eliminates duplicate records
- union all, intersect all, except all: preserves duplicate records</td></tr>
<tr><td>∪

(union, union all)</td><td>select cname from Customers
union
select rname from Restaurants;</td><td></td></tr>
<tr><td>∩

(intersect, intersect all)</td><td colspan="2">select pizza from Contains where ingredient = 'cheese'
intersect
select pizza from Contains where ingredient = 'chilli' ;</td></tr>
<tr><td>-

(except, except all)</td><td>-- No duplicates
select B from R
except
select B from S;
-- Keeps duplicates
select B from R
except all
select B from S;</td><td>Using <i>except all</i> is like literally minus-ing value by value the values (based on quantity)</td></tr>
</table>

Multi-relation queries	`select cname, rname`
	`from Customers, Restaurants`
	`where`
	`Customers.area = Restaurants.area;`
		Respective referencing:
		cname references to *Customers* and *rname* references to *Restaurants*

	`select cname, rname`
	`from Customers as C, Restaurants as R`
	`where C.area = R.area;`
		Renaming the tables using the *as* clause

	`select distinct S1.rname, S2.rname`
	`from Sells S1, Sells S2`
	`where S1.rname < S2.rname`
	`and S1.pizza = S2.pizza;`
		- Cartesian product
		- *as* clause is optional to rename table

Joins	
Join operators	- A *join operator* combines cross-product, selection and possibly projection operators - More convenient to use than plain cross-product operator
Natural join ($R \bowtie S$) (`natural join`)	- *Natural join* of R and S, $R \bowtie S = \pi_l (\sigma_c(R \times S))$ where A = common attributes between R and S = {a1, a2, … an} c = (R.a1 = S.a1) ∧ (R.a2 = S.a2) ∧ … ∧ (R.an = S.an) l = list of attributes in A, followed by those in R (excluding those in A) and those in R (excluding those in A) - Equality condition is imposed on common attributes

	`select R.rname, R.area, S.pizza, S.price`
	`from Restaurants R, Sells S`
	`where R.rname = S.rname;`
	
	`-- OR/equivalent to --`
	
	`select *`
	`from Restaurants natural join Sells;`

Inner join ($R \bowtie_c S$) (`inner join`)	- *Inner join* of R and S, $R \bowtie_c S = \sigma_c(R \times S)$ - Especially used in cases when joins between the same table is made (unable to use *natural join* since all attributes are common)

	`select distinct L1.cname, L2.cname`
	`from Likes L1, Likes L2`
	`where L1.cname < L2.cname`
	`and L1.pizza = L2.pizza;`
	
	`-- OR/equivalent to --`
	
	`select distinct L1.cname, L2.cname`
	`from Likes L1 inner join Likes L2`
	`on (L1.pizza = L2.pizza) and (L1.cname < L2.cname);`

Left outer join
($R \rightarrow_c S$)

(`left outer join`, `natural left outer join`)

- *Left outer join* of R and S, $R \rightarrow_c S = (R \bowtie_c S) \cup ((R \triangleright_c S) \times \{null(S)\})$

where $R \triangleright_c S = R - (R \ltimes_c S)$ is the *left anti-join* of R and S

$R \ltimes_c S = \pi_{attr(R)} (R \bowtie_c S)$ us the *left semi-join* of R and S

attr(R) = list of attributes in the schema of R (i.e. column names)

null(R) = n-component tuple of null values (n is the arity of relation R)

- *Left anti-join* of R and S computes all tuples in R that do not join with any tuple in S

- *Left semi-join* of R and S finds all tuples in R that joins with some tuples in S (equivalent to *right outer join*)

- *Left outer join* preserves everything in the left operand even if it is not found in the right operand

- Use *natural left outer join* when you recognise that

1) The only common attributes between the 2 tables are the ones you wish to join on

2) The only condition imposed is the equality condition on the common attributes

Question:
Find customers and the pizzas they like; include also customers who don't like any pizza.

```
select C.cname, L.pizza
from Customers C left outer join Likes L
on C.cname = L.cname;

-- OR/equivalent to --

select C.cname, L.pizza
from Customers C natural left outer join Likes L;
```

Customers

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North

Likes

cname	pizza
Homer	Hawaiian
Homer	Margherita
Lisa	Funghi
Maggie	Funghi
Moe	Funghi
Moe	Sciliana
Ralph	Diavola

(Output relation)

cname	pizza
Homer	Hawaiian
Homer	Margherita
Lisa	Funghi
Maggie	Funghi
Moe	Funghi
Moe	Sciliana
Ralph	Diavola
Willie	*null*

Left outer join
($R \leftrightarrow_c S$)

(`full outer join, natural full outer join`)

- *Full outer join* of R and S, $R \leftrightarrow_c S = (R \rightarrow_c S) \cup (\{null(R)\} \times (S \triangleright_c R))$
- Both left and right relation preserved (preserves everything and use *null* wherever applicable)
- It is not all the time that a *full outer join* can be translated to a *natural full outer join*

Question:
Find customer-restaurant pairs (C, R) where C and R are located in the same area. Include customers that are not co-located with any restaurant, and include restaurants that are not co-located with any customers.

```
select C.cname, R.rname
from Customers C full outer join Restaurants R
on C.area = R.area;

-- OR/equivalent to --

select C.cname, R.rname
from Customers C natural full outer join Restaurants R;
```

Customers

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North

Restaurants

rname	area
Corleone Corner	West
Gambino Oven	East
Lorenzo Tavern	Central
Mamma's Place	South
Pizza King	East

(Output relation)

cname	rname
Homer	*null*
Lisa	Mamma's Place
Maggie	Gambino Oven
Maggie	Pizza King
Moe	Lorenzo Tavern
Ralph	Lorenzo Tavern
Willie	*null*
null	Corleone Corner