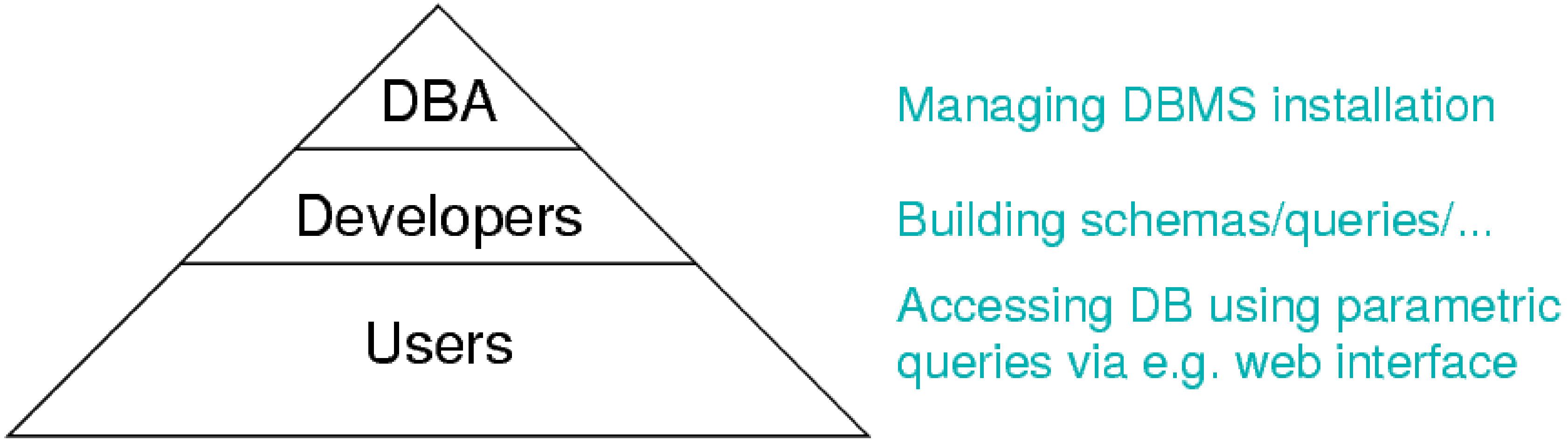
Database Administration

Database Administration

Database installations are typically ...

- used by many casual users (typically via a web interface)
- used by few developers (who build schemas and applications)
- managed by one DB administrator (or a small team of DBAs)



Database Administration (cont)

Tasks of the database administrator:

- manage the database installation
  (including configuring system parameters, running the server, back-ups, ...)
- create users and specify what they can/cannot do
  (according to the security poilicies of the organisation running the DBMS)
- tune performance of applications and overall system
- act as the owner of the system meta-data (e.g. catalog)
- back-up the contents of the database (in case of disasters)

Database Administration (cont)

The DBMS itself assists DB administration by:

- maintaining a database of schemas in the system (catalog)
- maintaining a database of users/groups and their privileges
  (which determines who can perform which operations on which objects)
- maintaining statistical information about database instances
  (used by the query optimiser in determining best query execution strategy)

Your PostgreSQL DBMSs

In the PostgreSQL installations on **grieg** you are the DBA
(aka PostgreSQL super-user, owner of postmaster process)

Data/files associated with server live under **`/srvr/`**YOU**`/pgsql903/`**

Some things that you could potentially do:

- change configuration parameters of your sever
  (e.g. more run-time buffers, more connections, **`postgresql.conf`**)
- add new users of the database and set their privileges
  (SQL commands: **`CREATE USER`**, **`CREATE ROLE`**, **`GRANT`**, **`REVOKE`**)
- modify accessibility of the databases
  (**`pg_hba.conf`**, who can access what from where and authentication)

∮ Catalogs

Catalogs

An RDBMS maintains a collection of relation instances.

To do this, it also needs information about relations:

- name, owner, primary key of each relation
- name, data type, constraints for each attribute
- authorisation for operations on each relation

Similarly for other DBMS objects (e.g. views, functions, triggers, ...)

This information is stored in the system catalog.

(The "system catalog" is also called "data dictionary" or "system view")

Catalogs (cont)

DBMSs typically use a hierarchy of namespaces to manage names:

Database (or **Catalog**)

- top–level namespace, contains a collection of schemas
- users typically connect to and work with a current database

Schema

- second–level namespace, contains a collection of tables, views, etc.
- users typically work with current schema, but can qualify names

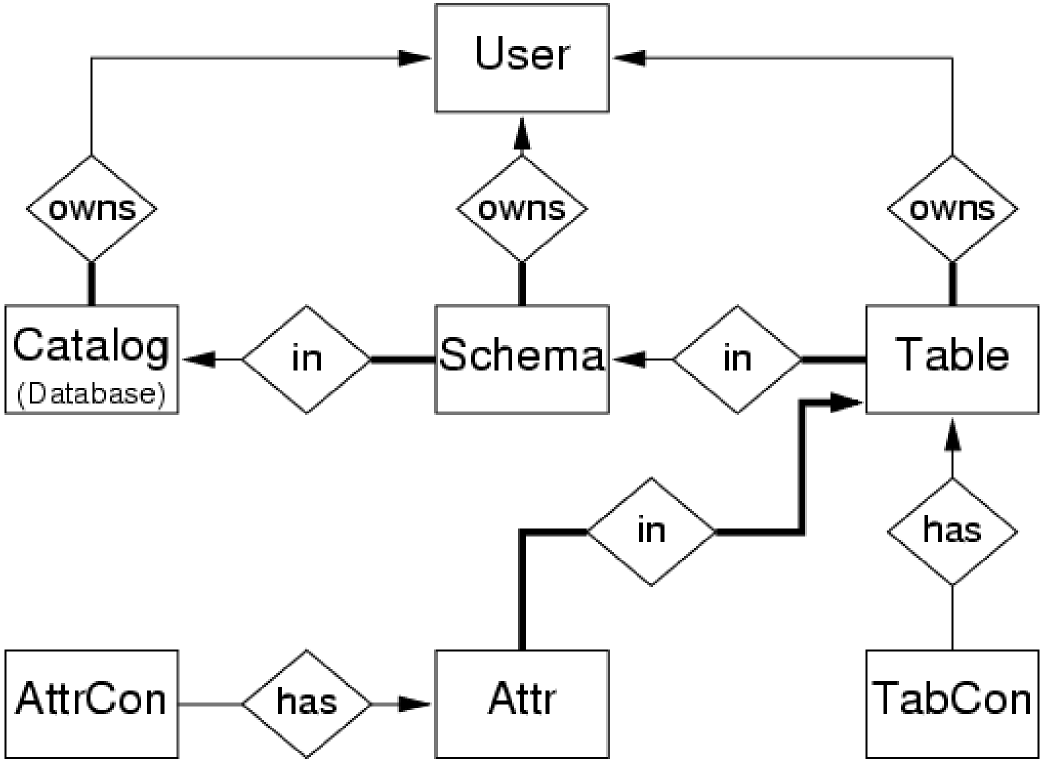
Table

- lowest–level namespace, contains a collection of attributes
- SELECT queries set a context for names; qualification often required

---

Catalogs (cont)

DBMSs store the catalog data in a collection of special tables:



(A small fragment of the meta-data tables in a typical RDBMS)

Catalogs (cont)

SQL:2003 standard view of metadata: **`INFORMATION_SCHEMA`**.

The **`INFORMATION_SCHEMA`** is available globally and includes:

Schemata(catalog_name, schema_name, schema_owner, ...)

Tables(table_catalog, table_schema, table_name, table_type, ...)

Columns(table_catalog, table_schema, table_name, column_name,
ordinal_position, column_default, data_type, ...)

Views(table_catalog, table_schema, table_name, view_definition, ...)

Table_Constraints(..., constraint_name, ..., constraint_type, ...)

etc. etc. etc.

Catalogs (cont)

DBMS internal meta-data is often different to standard, e.g.

```
Users(id:int, name:string, ...)
Databases(id:int, name:string, owner:ref(User), ...)
Schemas(id:int, name:string, owner:ref(User), ...)
Types(id:int, name:string, defn:string, size:int, ...)
Tables(id:int, name:string, owner:ref(User),
                                inSchema:ref(Schema), ...)
Attributes(id:int, name:string, table:ref(Table),
                           type:ref(Type), pkey:bool, ...)
```

```
TableConstraints(id:int, name:string, table:ref(Table),
                                         defn:string, ...)
AttrConstraints(id:int, name:string, attr:ref(Attribute),
                                         defn:string, ...)
-- etc. etc. etc.
```

Catalogs (cont)

SQL DDL operations such as

```
create table Abc (
    x integer primary key,
    y integer);
```

are implemented internally as operations on meta-data, e.g.

```
userID := current_user();
schemaID := current_schema();
tabID := nextval('tab_id_seq');
select into intID id
from Types where name='integer';
insert into Tables(id,name,owner,inSchema,...)
       values (tabID, 'abc', userID, schema, ...)
attrID := nextval('attr_id_seq');
```

```
insert into Attributes(id,name,table,type,pkey,...)
       values (attrID, 'x', tabID, intID, true, ...)
attrID := nextval('attr_id_seq');
insert into Attributes(id,name,table,type,pkey,...)
       values (attrID, 'y', tabID, intID, false, ...)
```

Access to System Catalog

Users typically have access to the system catalog via

- special commands (e.g. PostgreSQL's **`\d`**, **`\df`**, etc.)
- query-able views (e.g. Oracle's **`select * from tab`**)

How much is visible to each user depends on their role

- DBA can see/change anything in system catalog via SQL
- ordinary user can see only some of the system catalog and can change it only via SQL DDL statements

---

PostgreSQL Catalog

PostgreSQL stores catalog information as regular tables.

The **`\d?`** special commands in **`psql`** are just wrappers around queries on those tables, e.g.

`\dt`	list information about tables
`\dv`	list information about views
`\df`	list information about functions
`\dp`	list table access privileges
`\dT`	list information about data types
`\dd`	shows comments attached to DB objects

PostgreSQL Catalog (cont)

A PostgreSQL installation typically has several databases.

Some catalog information is global, e.g.

- databases, users, ...
- there is one copy of each "global" table for the whole PostgreSQL installation
- this copy is shared by all databases in the installation

Other catalog information is local to each database, e.g

- schemas, tables, attributes, functions, types, ...
- there is a separate copy of each "local" table in each database
- a copy of each "local" table is made when a new database is created

---

PostgreSQL Catalog (cont)

`pg_authid` contains information about database users:

`oid`	integer key to reference user
`rolname`	symbolic user name (e.g. **`jas`**)
`rolpasswd`	md5-encrypted password
`rolcreatedb`	can create new databases

rolsuper	is a superuser (owns server process)
rolcatupdate	can update system catalogs

PostgreSQL Catalog (cont)

pg_database contains information about databases:

datname	database name (e.g. **nssis**)
datdba	database owner (refs **pg_auth.oid**)
datpath	where files for database are stored (if not in the PG_DATA directory)
datacl	access permissions

PostgreSQL Catalog (cont)

pg_class contains information about tables:

relname	name of table (e.g. **employee**)
relnamespace	schema in which table defined (refs **pg_namespace.oid**)
reltype	data type corresponding to table (refs **pg_type.oid**)
relowner	owner (refs **pg_authid.oid**)
reltuples	# tuples in table
relacl	access permissions

Also holds info about objects other than tables, e.g. views, sequences, indexes.

PostgreSQL Catalog (cont)

pg_class also holds various flags/counters for each table:

relkind	what kind of object 'r' = ordinary table, 'i' = index, 'v' = view 'c' = composite type, 'S' = sequence, 's' = special

`relnatts`	# attributes in table (how many entries in **`pg_attribute`** table)
`relchecks`	# of constraints on table (how many entries in **`pg_constraint`** table)
`relhasindex`	table has/had an index?
`relhaspkey`	table has/had a primary key?

etc.

PostgreSQL Catalog (cont)

`pg_type` contains information about data types:

`typname`	name of type (e.g. **`integer`**)
`typnamespace`	schema in which type defined (refs **`pg_namespace.oid`**)
`typowner`	owner (refs **`pg_authid.oid`**)
`typtype`	what kind of data type

	'b' = base type, 'c' = complex (row) type, ...
`typlen`	how much storage used for type values (–1 for variable–length types, e.g. **`text`**)
`typrelid`	table associated to complex type (refs **`pg_class.oid`**)

etc.

PostgreSQL Catalog (cont)

`pg_attribute` contains information about attributes:

`attname`	name of attribute (e.g. **`id`**)
`attrelid`	table this attribute belongs to (refs **`pg_class.oid`**)
`atttypid`	data type of this attribute (refs **`pg_type.oid`**)
`attlen`	storage space required by attribute

(a copy of **`pg_type.typlen`** for data type)

`attnum`	attribute position (1..n, sys attrs are -ve)

PostgreSQL Catalog (cont)

`pg_attribute` also holds constraint/status information:

`attnotnull`	attribute may not be null?
`atthasdef`	attribute has a default values (value is held in **`pg_attrdef`** table)
`attisdropped`	attribute has been dropped from table

plus others related to strage properties of attribute.

PostgreSQL Catalog (cont)

`pg_proc` contains information about functions:

proname	name of function (e.g. **substr**)
pronamespace	schema in which function defined (refs **pg_namespace.oid**)
proowner	owner (refs **pg_authid.oid**)
prolang	what language function written in
proacl	access control

etc.

PostgreSQL Catalog (cont)

pg_proc contains information about arguments:

pronargs	how many arguments
prorettype	return type (refs **pg_type.oid**)
proargtypes	argument types (vector of refs **pg_type.oid**)
proisstrict	returns null if any arg is null

`prosrc`	source code if interpreted (e.g. PLpgSQL)

PostgreSQL Catalog (cont)

`pg_constraints` contains information about constraints:

`conname`	name of constraint (not unique)
`connamespace`	schema containing this constraint
`contype`	kind of constraint 'c' = check, 'u' = unique, 'p' = primary key, 'f' = foreign key
`conrelid`	which table (refs **`pg_class.oid`**)
`conkey`	which attributes (vector of values from **`pg_attribute.attnum`**)
`consrc`	check constraint expression

PostgreSQL Catalog (cont)

For full details on PostgreSQL catalogs:

PostgreSQL 9.0.3 Developer's Guide

Part VII, Chapter 45, System Catalogs

Part IV, Chapter 34, The Information Schema

∮ Security, Privilege, Authorisation

Database Security

Database security has to meet the following objectives:

- Secrecy: information not disclosed to unauthorised users

  e.g. a student should **not** be able to examine other students' marks
- Integrity: only authorised users are allowed to modify data

  e.g. a student should not be able to modify anybody's marks

- Availability: authorised users should not be denied access

  e.g. the LIC should be able to read/changes marks for their course

Goal: prevent unauthorised use/alteration/destruction of mission-critical data.

Database Security (cont)

Security mechanisms operate at a number of levels

- within the database system (SQL-level privileges)

  e.g. specific users can query/modify/update only specified database objects
- accessing the database system (users/passwords)

  e.g. users are required to authenticate themselves at connection-time
- operating system (access to DB clients)

  e.g. users should not obtain access to the DBMS superuser account
- network (most DB access nowadays is via network)

  e.g. results should not be transmitted unencrypted to Web browsers
- human/physical (conventional security mechanisms)

  e.g. no unauthorised physical access to server hosting the DBMS

Database Access Control

Access to DBMSs involves two aspects:

- having execute permission for a DBMS client (e.g. **`psql`**)
- having a username/password registered in the DBMS

Establishing a connection to the database:

- user supplies database/username/password to client
- client passes these to server, which validates them
- if valid, user is "logged in" to the specified database

Database Access Control (cont)

Note: we don't need to supply username/password to **`psql`**

- **`psql`** works out which user by who ran the client process
- we're all PostgreSQL super-users on our own servers

- servers are configured to allow super-user direct access

Note: access to databases via the Web involves:

- running a script on a Web server
- using the Web server's access rights on the DBMS

Database Access Control (cont)

Database users are set up by the DBA via the command:

```
CREATE USER Name IDENTIFIED BY 'Password'
```

Various privileges can be assigned at user-creation time, e.g.

- ability to create new users or databases or tables

User properties may be subsequently changed via:

```
ALTER USER Name IDENTIFIED BY 'NewPassword'
```

This command is also used to change privileges, quotas, etc.

Database Access Control (cont)

A user may be associated with a role or group

- which typically gives them additional specific privileges

Roles are also set up by the DBA via the command:

```
CREATE ROLE RoleName
```

Examples of roles:

- AcademicStaff ... has privileges to read/modify marks
- OfficeStaff ... has privilege to read all marks
- Student ... has privilege to read own marks only

Database Access Control in PostgreSQL

In older versions of PostgreSQL ...

- **`USER`**s and **`GROUP`**s were distinct kinds of objects
- **`USER`**s were added via **`CREATE USER`** *UserName*
- **`GROUP`**s were added via **`CREATE GROUP`** *GroupName*
- **`GROUP`**s were built via **`ALTER GROUP ... ADD USER ...`**

In recent versions, **`USER`**s and **`GROUP`**s are unified by **`ROLE`**s

Older syntax is retained for backward compatibility.

Database Access Control in PostgreSQL (cont)

PostgreSQL has two ways to create users ...

From the Unix command line, via the command

```
createuser [ -a | -d ] Name
```

(**`-a`** allows user to create other users, **`-d`** allows user to create databases)

From SQL, via the statement:

```
CREATE ROLE UserName Options
-- where Options include ...
PASSWORD 'Password'
CREATEDB | NOCREATEDB
CREATEUSER | NOCREATEUSER
IN GROUP GroupName
VALID UNTIL 'TimeStamp'
```

Database Access Control in PostgreSQL (cont)

Groups are created as **ROLE**s via

```
CREATE ROLE GroupName
--or--
CREATE ROLE GroupName
WITH USER UserName1, UserName2, ...
```

and may be subsequently modified by

```
GRANT GroupName TO UserName1, UserName2, ...
--and
REVOKE GroupName FROM UserName1, UserName2, ...
```

Database Access Control in PostgreSQL (cont)

PostgreSQL stores user information in a table **`pg_authid`**.

Each user is associated with a unique identifying number.

Every PostgreSQL is created with a default user (id=1, superuser).

Some fields in the **`pg_authid`** table:

- **`oid`**: unqiue user id (e.g. 1, 100, ...)
- **`rolname`**: string for user name (e.g. **`jas`**)
- **`rolpassword`**: encrypted version of user's password

Database Access Control in PostgreSQL (cont)

PostgreSQL uses a file called **`pg_hba.conf`** to determine how to authenticate users (**`hba`** stands for host-based authentication).

This file contains a sequence of entries where each entry has:

- connection method (Unix-domain socket, TCP/IP, encrypted)
- list of databases (or **`all`**)
- user/group name (or **`all`**)
- IP address and IP mask
- authentication method (plus options)

PostgreSQL uses first four items to determine authentication method.

Database Access Control in PostgreSQL (cont)

Possible authentication methods:

- **`md5`**: get password from user, check against **`pg_shadow`**
- **`krb5`**: use Kerberos-based authentication

- **reject**: do not allow the user to access DBMS this way
- **trust**: allow them to log in as any user without password
- **ident**: allow them access based on Unix user name info
  - **ident**+map: see if UnixName is member of map list
  - **ident+sameuser**: log in as user with name UnixName

Database Access Control in PostgreSQL (cont)

Examples from **pg_hba.conf** file:

```
# Allow any user on the local system to connect to any database
# as any user if accessing from local machine via Unix socket.
#
#TYPE   DATABASE USER IP-ADDRESS    IP-MASK           METHOD
local   all      all                                  trust

# Reject all connection from 192.168.54.1
#
#TYPE   DATABASE   USER IP-ADDRESS    IP-MASK           METHOD
host    all        all  192.168.54.1  255.255.255.255   reject

# Allow any user from any host with IP address 192.168.93.x
```

```
# to connect to database "mydb" as the same user name that
# ident reports for the connection (typically Unix user name).
#
#TYPE   DATABASE USER IP-ADDRESS    IP-MASK          METHOD
host    mydb     all  192.168.93.0  255.255.255.0    ident sameuser

# Allow a user from host 192.168.12.10 to connect to database
# "mydb" only if the user's password is correctly supplied.
#
#TYPE   DATABASE USER IP-ADDRESS    IP-MASK            METHOD
host    mydb     all  192.168.12.10 255.255.255.255    md5
```

SQL Access Control

SQL access control deals with

- privileges on database objects (e.g. tables, view, functions, ...)
- allocating such privileges to users and/or roles

The user who creates an object is automatically assigned:

- ownership of that object
- a privilege to modify (**ALTER**) the object

- a privilege to remove (**DROP**) the object
- along with all other privileges specified below

SQL Access Control (cont)

The owner of an object can assign privileges on that object to other users.

Accomplished via the command:

```
GRANT Privileges ON Object
TO list of (Users|Roles) | PUBLIC
[ WITH GRANT OPTION ]
```

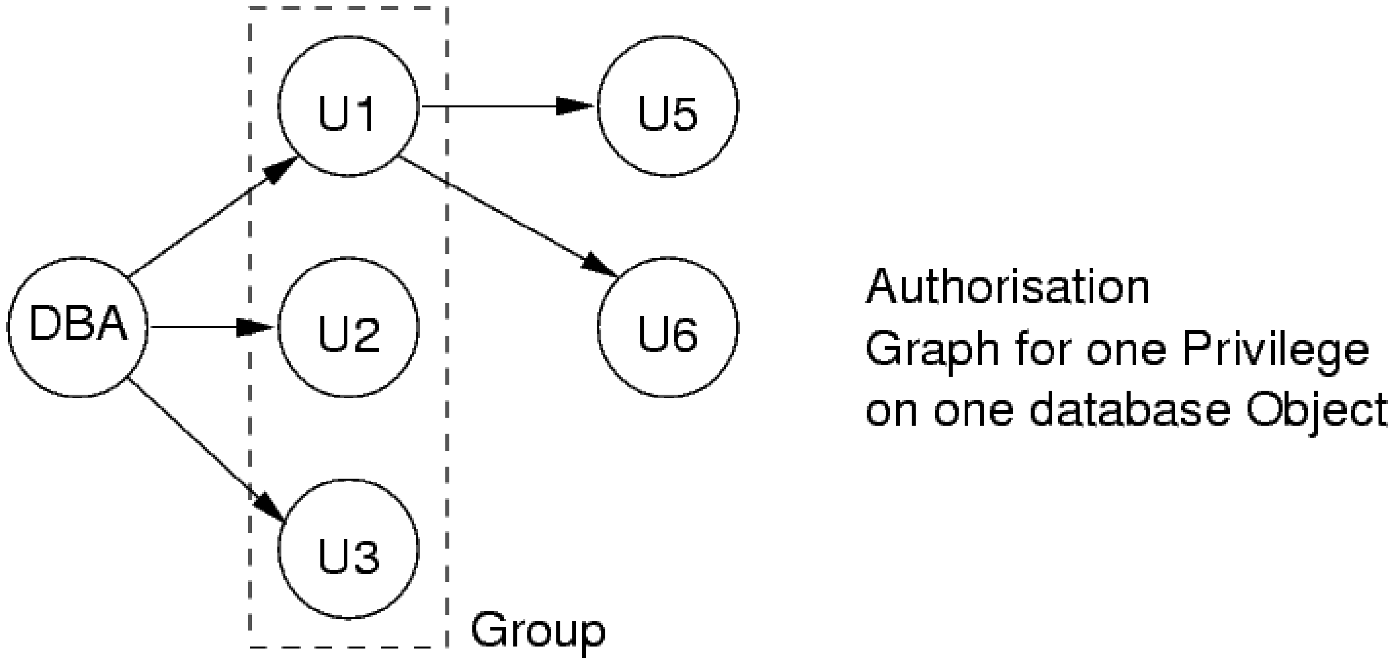
Privileges can be **ALL** (giving everything but **ALTER** and **DROP**)

WITH GRANT OPTION allows a user who has been granted a privilege to pass the privilege on to any other user.

SQL Access Control (cont)

Effects of privilege granting

- are sometimes subtle (possible conflicts?)
- can be represented by an authorisation graph



SQL Access Control (cont)

Privileges can be withdrawn via the command:

```
REVOKE Privileges ON Object
FROM ListOf (Users|Roles) | PUBLIC
CASCADE | RESTRICT
```

Normally withdraws Privileges from just specified users/roles.

`CASCADE` ... also withdraws from users they had granted to.

E.g. revoking from U1 also revokes U5 and U6

`RESTRICT` ... fails if users had granted privileges to others.

E.g. revoking from U1 fails, revoking U5 or U2 succeeds

SQL Access Control (cont)

Privileges available for users on database objects:

`SELECT`:

- user can read all rows and columns of table/view

- this includes columns added later via **`ALTER TABLE`**

`INSERT` or **`INSERT(`***ColName***`)`**:

- user can insert rows into table
- if *ColName* specified, can only set value of that column

SQL Access Control (cont)

More privileges available for users on database objects:

`UPDATE` or **`UPDATE(`***ColName***`)`**:

- user can modify values stored in the table
- if *ColName* specified, can only set value of that column

`DELETE`:

- user can delete rows from the table
- this does *not* imply permission to remove table itself
  (this is the **`DROP`** privilege automatically assigned to the object creator)

SQL Access Control (cont)

More privileges available for users on database objects:

`REFERENCES(`*ColName***`)`**:

- user can use *ColName* as foreign key in their tables

`EXECUTE`:

- user can execute the specified function

`TRIGGER`:

- user is allowed to create triggers on a table
- note that tiggers always execute with creator's privileges

SQL Access Control in PostgreSQL

PostgreSQL follows the above with some minor variations:

- group names must be preceded by **`GROUP`**
- there is an additional privilege for **`RULE`**s

See the PostgreSQL manual for full details.

Problems with SQL Access Control

Allowing users to assign privileges to others can be exploited.

Example: student S wants access to table of marks M

- M is owned by lecturer L, S has no SELECT privilege on M
- S creates a new table SS and grants INSERT privilege to L
- S modifies code of some PLpgSQL function F used by L
- the modifications to F copy the data from M into SS
- S restores F to its original state (in case L gets suspicious)
- S now has access to the data from M in table SS

Mandatory Access Control

Above approach is called discretionary access control

- relies on individual users to assign privileges
- very fine-grained ⇒ tedious to specify privileges

An alternative approach: manadatory access control (MAC)

- global approach to access control; simple to specify
- currently under development (not yet available in DBMSs)
- a popular MAC model is Bell-LaPadula (see next page)

Mandatory Access Control (cont)

Access control is described in terms of ...

- objects (e.g. tables, rows, views, columns, ...)
- subjects (e.g. users, programs, ...)

- security classes (an ordered collection, least to most secure)

Example: *Unclassified < Confidential < Secret < Top Secret*

Subjects and objects are assigned to security classes.

Impose these restrictions on every data access:

- subject S can read object O only if class(S) ⩾ class(O)
- subject S can write object O only if class(S) ⩽ class(O)

Security in Statistical Databases

Statistical databases

- contain data about a group of individuals
- but allow access only to summary data
- also provide controls to select subsets of data

Example of such a database: population census

- information is stored about individuals
- users are only allowed to examine trends/summaries
- e.g. can find out average age of people living in Sydney
  but cannot ask the question "How old is John?"

Privacy is protected, but useful information is still available.

Security in Statistical Databases (cont)

Consider: anonymous surveys in an on-line survey system

- can see overall results (e.g. tutorials rated at 7/10)
- but do not have access to individual student responses
- can also summarize results for subgroups
  (e.g. CE students rated tutorials at 6/10, CS students rated tutorials at 8/10)

Problem: subset controls may allow selection of individuals, e.g.

- we know that there's only one Law student in the class

- ask for a "summary" of responses given by Law students

Security in Statistical Databases (cont)

How to solve this problem?

- restrict subsets to being larger than a minimum size *N*

But still doesn't quite work, e.g.

- system gives summaries for several groups (e.g. SE+CE)
- get summary for Law+CE (presumably with >*N* responses)
- get summary just for CE (assume count of reponses given)
- determine Law response from the "difference"

Security is difficult to enforce in statistical databases, but activity can be logged.

∮ Performance Tuning

Performance Tuning

Schema design:

- devise data structures to represent application information

Performance tuning:

- devise data structures to achieve good performance

Good performance may involve any/all of:

- making applications run faster
- lowering response time of queries/transactions
- improving overall transaction throughput

Performance Tuning (cont)

Tuning requires us to consider the following:

- which queries and transactions will be used?

  (e.g. check balance for payment, display recent transaction history)

- how frequently does each query/transaction occur?

  (e.g. 99% of transactions are EFTPOS payments; 1% are print balance)

- are there time constraints on queries/transactions?

  (e.g. payment at EFTPOS terminals must be approved within 7 seconds)

- are there uniqueness constraints on any attributes?

  (therefore, define index on attributes to speed up insertion uniqueness check)

- how frequently do updates occur?

  (indexes slow down updates, because must update table *and* index)

Performance Tuning (cont)

Performance can be considered at two times:

- *during* schema design
  - typically towards the end of schema design process

  - requires schema transformations such as denormalisation
- *after* schema design
  - requires adding extra data structures such as indexes

Denormalisation

Normalisation structures data to minimise storage redundancy.

- achieves this by "breaking up" the data into logical chunks
- requires minimal "maintenance" to ensure data consistency

Problem: queries that need to put data back together.

- need to use a (potentially expensive) join operation
- if an expensive join is frequent, system performance suffers

Solution: store some data redundantly

- benefit: queries needing expensive join are now cheap
- trade-off: extra maintenance effort to maintain consistency

- worthwhile if joins are frequent and updates are rare

Denormalisation (cont)

Example: Courses = Course Subject Term

If we frequently need to refer to course "standard" name

- add extra **`courseName`** column into **`Course`** table
- cost: trigger before insert on **`Course`** to construct name
- trade-off likely to be worthwhile: **`Course`** insertions infrequent

```
-- can now replace a query like:
select s.code||t.year||t.sess, e.grade, e.mark
from   Course c, CourseEnrolment e, Subject s, Term t
where  e.course = c.id and c.subject = s.id and c.term = t.id
-- by a query like:
select c.courseName, e.grade, e.mark
from   Course c, CourseEnrolment e
where  e.course = c.id
```

Indexes

Indexes provide efficient content-based access to tuples.

Can build indexes on any (combination of) attributes.

Defining indexes:

```
CREATE INDEX name ON table ( attr1, attr2, ... )
```

attr$_i$ can be an arbitrary expression (e.g. **`upper(name)`**).

`CREATE INDEX` also allows us to specify

- that the index is on **`UNIQUE`** values
- an access method (**`USING`** btree, hash, rtree, or gist)

Indexes (cont)

Indexes can make a huge difference to query processing cost.

On the other hand, they introduce overheads (storage, updates).

Creating indexes to maximise performance benefits:

- apply to attributes used in equality/range conditions, e.g.

```
select * from Employee where id = 12345
select * from Employee where age > 60
select * from Employee where salary between 10000 and 20000
```

- but only in queries that are frequently used
- and on tables that are not updated frequently

Indexes (cont)

Considerations in applying indexes:

- is an attribute used in frequent/expensive queries?
  (note that some kinds of queries can be answered from index alone)

- should we create an index on a collection of attributes?

  (yes, if the collection is used in a frequent/expensive query)

- can we exploit a clustered index? (only one per table)

- should we use B-tree or Hash index?

```
-- use hashing for (unique) attributes in equality tests, e.g.
select * from Employee where id = 12345
-- use B-tree for attributes in range tests, e.g.
select * from Employee where age > 60
```

Query Tuning

Sometimes, a query can be re-phrased to affect performance:

- by helping the optimiser to make use of indexes
- by avoiding (unnecessary) operations that are expensive

Examples which *may* prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 10.0
       -- fix by re-phrasing condition to (salary > 3650)
select name from Employee where name like '%ith%'
select name from Employee where birthday is null
       -- above two are difficult to "fix"
select name from Employee
where  dept in (select id from Dept where ...)
       -- fix by using Employee join Dept on (e.dept=d.id)
```

Query Tuning (cont)

Other factors to consider in query tuning:

- **`select distinct`** requires a sort; is **`distinct`** necessary?
- if multiple join conditions are available ...
  choose join attributes that are indexed, avoid joins on strings

```
select ... Employee join Customer on (s.name = p.name)
vs
select ... Employee join Customer on (s.ssn = p.ssn)
```

- sometimes **`or`** in condition prevents index from being used ...
  replace the **`or`** condition by a union of non-**`or`** clauses

```
select name from Employee where dept=1 or dept=2
vs
(select name from Employee where dept=1)
union
(select name from Employee where dept=2)
```

PostgreSQL Query Tuning

PostgreSQL provides the **`explain`** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

```
EXPLAIN [ANALYZE] Query
```

Without **`ANALYZE`**, **`EXPLAIN`** shows plan with estimated costs.

With **ANALYZE**, **EXPLAIN** executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

EXPLAIN Examples

Example: Select on indexed attribute

```
ass2=# explain select * from student where id=100250;
                                QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using student_pkey on student  (cost=0.00..5.94 rows=1 width=17)
   Index Cond: (id = 100250)

ass2=# explain analyze select * from student where id=100250;
                                QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using student_pkey on student  (cost=0.00..5.94 rows=1 width=17)
                                 (actual time=31.209..31.212 rows=1 loops=1)
   Index Cond: (id = 100250)
 Total runtime: 31.252 ms
```

EXPLAIN Examples (cont)

Example: Select on non-indexed attribute

```
ass2=# explain select * from student where stype='local';
                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
   Filter: ((stype)::text = 'local'::text)

ass2=# explain analyze select * from student where stype='local';
                              QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)
             (actual time=0.061..4.784 rows=2512 loops=1)
   Filter: ((stype)::text = 'local'::text)
 Total runtime: 7.554 ms
```

EXPLAIN Examples (cont)

Example: Join on a primary key (indexed) attribute

```
ass2=# explain
ass2-# select s.sid,p.name from Student s, Person p where s.id=p.id;
                               QUERY PLAN
```

```
------------------------------------------------------------------------------
 Hash Join  (cost=70.33..305.86 rows=3626 width=52)
   Hash Cond: ("outer".id = "inner".id)
   ->  Seq Scan on person p  (cost=0.00..153.01 rows=3701 width=52)
   ->  Hash  (cost=61.26..61.26 rows=3626 width=8)
         ->  Seq Scan on student s  (cost=0.00..61.26 rows=3626 width=8)
```

EXPLAIN Examples (cont)

Join on a primary key (indexed) attribute:

```
ass2=# explain anaylze
ass2-# select s.sid,p.name from Student s, Person p where s.id=p.id;
                                QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=70.33..305.86 rows=3626 width=52)
            (actual time=11.680..28.242 rows=3626 loops=1)
   Hash Cond: ("outer".id = "inner".id)
   ->  Seq Scan on person p  (cost=0.00..153.01 rows=3701 width=52)
                       (actual time=0.039..5.976 rows=3701 loops=1)
   ->  Hash  (cost=61.26..61.26 rows=3626 width=8)
             (actual time=11.615..11.615 rows=3626 loops=1)
         ->  Seq Scan on student s  (cost=0.00..61.26 rows=3626 width=8)
                              (actual time=0.005..5.731 rows=3626 loops=1)
 Total runtime: 32.374 ms
```

Produced: 13 Sep 2020