COMP3311 Week 08 Lecture (cont)

Performance Tuning

Programming with Databases

We have previously discussed that

- accessing data via SQL queries
- packaging SQL queries as views/functions
- building functions to return tables
- implementing assertions via triggers

All of the above programming

- is very close to the data
- takes place inside the DBMS

... Programming with Databases

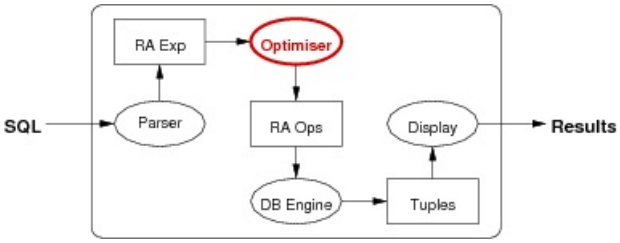
Complete applications require code outside the DBMS

- to handle the user interface (GUI or Web)
- to interact with other systems (e.g. other DBs)
- to perform compute-intensive work (vs. data-intensive)

"Conventional" programming languages (PLs) provide these.

Query Optimisation

Phase 2: mapping RA expression to (efficient) query plan



... Query Optimisation

The query optimiser start with an RA expression, then

- generates a set of equivalent expressions
- generates possible execution plans for each
- estimates cost of each plan, chooses chepaest

The cost of evaluating an RA operation is determined by:

- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- the size of intermediate results
- then, based on this, number of pages read/written

... Query Optimisation

An *execution plan* is a sequence of relational operations.

Consider execution plans for: $\sigma_c (R \bowtie_d S \bowtie_e T)$

```
tmp1   :=  hash_join[d](R,S)
tmp2   :=  sort_merge_join[e](tmp1,T)
result :=  binary_search[c](tmp2)
```

or

```
tmp1   :=  sort_merge_join[e](S,T)
tmp2   :=  hash_join[d](R,tmp1)
```

```
result :=  linear_search[c](tmp2)
```

or

```
tmp1   :=  btree_search[c](R)
tmp2   :=  hash_join[d](tmp1,S)
result :=  sort_merge_join[e](tmp2)
```

All produce same result, but have different costs.

Implementations of RA Ops

Sorting (quicksort, etc. are not applicable)

- external merge sort (cost $O(N log_B N)$ with B memory buffers)

Selection (different techniques developed for different query types)

- sequential scan (worst case, cost $O(N)$)
- index-based (e.g. B-trees, cost $O(logN)$, tree nodes are pages)
- hash-based ($O(1)$ best case, only works for equality tests)

Join (fast joins are critical to success to erlational DBMSs)

- nested-loop join (cost $O(N.M)$, buffering can reduce to $O(N+M)$)
- sort-merge join (cost $O(NlogN+MlogM)$)
- hash-join (best case cost $O(N+M.N/B)$, with B memory buffers)

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

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. 90% withdrawals; 10% deposits; 50% balance check)
- are there time constraints on queries/transactions?
  (e.g. EFTPOS payments must be approved within 7 seconds)
- are there uniqueness constraints on any attributes?
  (define indexes on attributes to speed up insertion uniqueness check)
- how frequently do updates occur?
  (indexes slow down updates, because must update table *and* index)

Denormalisation

Normalisation minimises storage redundancy.

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

Problem: queries that need to put data back together.

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

Solution: store some data redundantly

- benefit: queries needing expensive join are now cheap
- trade-off: extra maintenance effort to keep consistency
- worthwhile if joins are frequent and updates are rare

... Denormalisation

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 fast content-based access to tuples.

`CREATE INDEX` *name* `ON` *table* `(` $attr_1$, $attr_2$, ... `)`

Some considerations in applying indexes:

- is an attribute used in frequent/expensive queries?
  (note that some kinds of queries can be answered from index alone)
- is the table containing attribute frequently updated?
- (in PostgreSQL) 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/expensive operations

Examples which *may* prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 100
       -- fix by re-phrasing condition to (salary > 36500)
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

Other tricks in query tuning (effectiveness is DBMS-dependent)

- `select distinct` requires a sort ...
  is the `distinct` really necessary? (at this stage?)
- 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` 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 Students 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 Students 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

Example: Select on non-indexed attribute

```
ass2=# explain select * from Students 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 Students
ass2-#                           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

Example: Join on a primary key (indexed) attribute

```
ass2=# explain
ass2-# select s.sid,p.name
ass2-# from Students s, People 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

Example: Join on a non-indexed attribute

```
ass3=> explain select s1.code, s2.code
ass2-# from Subjects s1, Subjects s2 where s1.offerer=s2.offerer;
                        QUERY PLAN
-------------------------------------------------------------------
 Merge Join  (cost=2744.12..18397.14 rows=1100342 width=18)
   Merge Cond: (s1.offerer = s2.offerer)
   ->  Sort  (cost=1372.06..1398.33 rows=10509 width=13)
         Sort Key: s1.offerer
         ->  Seq Scan on subjects s1
                     (cost=0.00..670.09 rows=10509 width=13)
```

```
->  Sort  (cost=1372.06..1398.33 rows=10509 width=13)
      Sort Key: s2.offerer
      ->  Seq Scan on subjects s2
                  (cost=0.00..670.09 rows=10509 width=13)
```

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

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

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

Access specified in `/srvr/YOU/pgsql903/pg_hba.conf`

... Database Access Control

SQL standard doesn't specify details of users/groups/roles.

Some typical operations on users:

```
CREATE USER Name IDENTIFIED BY 'Password'
ALTER USER Name IDENTIFIED BY 'NewPassword'
ALTER USER Name WITH Capabilities
ALTER USER Name SET ConfigParameter = ...
```

Capabilities: super user, create databases, create users, etc.

Config parameters: resource usage, session settings, etc.

... Database Access Control

A user may be associated with a *group* (aka *role*)

Some typical operations on groups:

```
CREATE GROUP Name
ALTER GROUP Name ADD USER User1, User2, ...
ALTER GROUP Name DROP USER User1, User2, ...
```

Examples of groups/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 ...

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

In recent versions, USERs and GROUPs are unified by ROLEs

Older syntax is retained for backward compatibility.

... Database Access Control in PostgreSQL

PostgreSQL has two ways to create users ...

From the Unix command line, via the command

`createuser` *Name*

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

Groups are created as ROLEs via

```
CREATE ROLE GroupName
--or--
CREATE ROLE GroupName WITH USER User1, User2, ...
```

and may be subsequently modified by

```
GRANT GroupName TO User1, User2, ...
REVOKE GroupName FROM User1, User2, ...
GRANT Privileges ... TO GroupName
REVOKE Privileges ... FROM GroupName
```

SQL Access Control

SQL access control deals with

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

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

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

Accomplished via the command:

```
GRANT Privileges ON Object
TO ( ListOfRoles | 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

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

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

More privileges available for users on database objects:

- UPDATE: user can modify values stored in the table
- UPDATE(*ColName*): user can update specified column
- DELETE: user can delete rows from the table
- REFERENCES(*ColName*): user can use column as foreign key
- EXECUTE: user can execute the specified function
- TRIGGER: user is allowed to create triggers on table

Produced: 10 Apr 2019