DBMS Architecture and Implementation

DBMS Architecture and Implementation

Aims:

- examine techniques used in implementation of DBMSs:
  - query processing (QP), transaction processing (TP)
- use QP knowledge to make DB applications efficient
- use TP knowledge to make DB applications safe

COMP3311: overview of the above; how to use them in app development
COMP9315: explore the above in detail; implement (bits of) a DBMS.

Database Application Performance

In order to make DB applications efficient, we need to know:

- what operations on the data does the application require

  (which queries, updates, inserts and how frequently is each one performed)

- how these operations might be implemented in the DBMS

  (data structures and algorithms for select, project, join, sort, ...)

- how much each implementation will cost

  (in terms of the amount of data transferred between memory and disk)

and then, as much as the DBMS allows, "encourage" it to use the most efficient methods.

Database Application Performance (cont)

Application programmer choices that affect query cost:

- how queries are expressed
  - generally join is faster than subquery
  - especially if subquery is correlated
  - avoid producing large intermediate tables *then* filtering
  - avoid applying functions in where/group–by clasues
- creating indexes on tables
  - index will speed–up filtering based on indexed attributes
  - indexes generally only effective for equality, gt/lt
  - indexes have update–time and storage overheads
  - only useful if filtering much more frequent than update

Database Application Performance (cont)

Whatever you do as a DB application programmer

- the DBMS will transform your query
- to make it execute as efficiently as possible

Transformation is carried out by query optimiser

- which assesses possible query execution approaches
- evaluates likely cost of each approach, chooses cheapest

You have no control over the optimisation process

- but choices you make can block certain options
- limiting the query optimiser's chance to improve

Database Application Performance (cont)

Example: query to find sales people earning more than $50K

```
select name from Employee
where  salary > 50000 and
```

```
        empid in (select empid from Worksin
                  where  dept = 'Sales')
```

A query optimiser might use the strategy

```
SalesEmps = (select empid from WorksIn where dept='Sales')
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

Needs to examine *all* employees, even if not in Sales

Database Application Performance (cont)

A different expression of the same query:

```
select name
from   Employee join WorksIn using (empid)
where  Employee.salary > 5000 and
       WorksIn.dept = 'Sales'
```

Query optimiser might use the strategy

```
SalesEmps = (select * from WorksIn where dept='Sales')
foreach e in (Employee join SalesEmps) {
    if (e.salary > 50000)
        add e to result set
}
```

Only examines Sales employees, and uses a simpler test

Database Application Performance (cont)

A very poor expression of the query (correlated subquery):

```
select name from Employee e
where  salary > 50000 and
       'Sales' in (select dept from Worksin where empid=e.id)
```

A query optimiser would be forced to use the strategy:

```
foreach e in Employee {
    Depts = (select dept from WorksIn where empid=e.empid)
```

```
        if ('Sales' in Depts && e.salary > 50000)
            add e to result set
    }
```

Needs to run a query for *every* employee ...

Database Transaction Processing

In most applications ...

- application–level operations comprise multiple DB operations
- multiple users interact with the database concurrently

Example: bank funds transfer requires at least two operations

```
Transfer(Amount, Source, Dest)
-- deduct funds from source account
update Account set balance = balance - Amount
where  account = Source
-- add funds to destination account
```

```
update Account set balance = balance + Amount
where  account = Dest
```

May also require checks on validity of **`Source`** and **`Dest`** ...

Database Transaction Processing (cont)

Where things can go wrong in the funds transfer example:

```
(1) update Account set balance = balance - Amount where account = Source
(2) update Account set balance = balance + Amount where account = Dest
```

1. system could crash after first operation but before second
   - money lost from source account
   - solution requires us to ensure 0 or 2 ops are completed
2. two users could do first operation simultaneously
   - only one deduction from source account
   - solution requires us to control "simultaneous" access

Database Transaction Processing (cont)

DBMSs provide two inter-related mechanisms for transactions ...

- transactions
  - syntax: **`BEGIN`** ... *SQL statements* ... **`END`**
  - treats a group of DB operations as an atomic unit
  - all operations happen, or none do (may requie rollback)
- locking
  - syntax: **`LOCK`** *DBobject* (**`SHARE`**|**`EXCLUSIVE`**)
  - block second transaction trying to obtain exlcusive lock
  - transaction continues once lock released by first transaction

---

Database Transaction Processing (cont)

Locking is critical in many contexts

- but has overheads (lock objects, reduced concurrency)

If need for locking can be reduced ⇒ better throughput

Many DBMSs provide MVCC (*multi-version concurrency control*)

- stores multiple versions of each tuple
- each transaction accesses "appropriate" version
- reduces locking requirements (and sometimes eliminates locking)

Disadvantages: storage overhead, each tuple access requires relevance check

---

Data Representation

A DBMS provides representations for:

- values, tuples, tables, indexes (native representation)
- transactions, locks (native representations)
- functions, views, triggers (via tuples in meta-data tables)

Value representations use underlying data types and byte arrays

- e.g. **`int`** for **`integer`**, **`char[`***N***`]`** for **`varchar(`***N***`)`**

Tuple representations are like **`structs`** with header info

- values stored in a "chunk of bytes"
- plus **`oid`**, MVCC values, attribute offsets
- large data is stored out–of–band (TOAST tables)

Data Representation (cont)

Many ways to represent database relations:

- one relation = one file
- one file contains data from many relations
- one relation is implemented as multiple files

Examples of the above:

- MiniSQL: each relation+indexes stored in a single file

- SQLite: all tables+indexes+metadata in one file
- Oracle: all table+indexes+catalog in a set of large files
- PostgreSQL: each relation+indexes in several files

Data Representation (cont)

PostgreSQL's data layout:

- all data held under **`PGDATA/base`** directory
- each database is a subdirectory (named after oid)
- each table or index is stored in a separate file
- large data values are compressed and stored separately
  (a TOAST file associated with each table containing large values)
- **`pgsql_tmp`** subdirectory holds temp query proc files
  (used when intermediate results are too large to fit in mem buffers)

Data Representation (cont)

DB data is (obviously) persistent ⇒ resides on disk

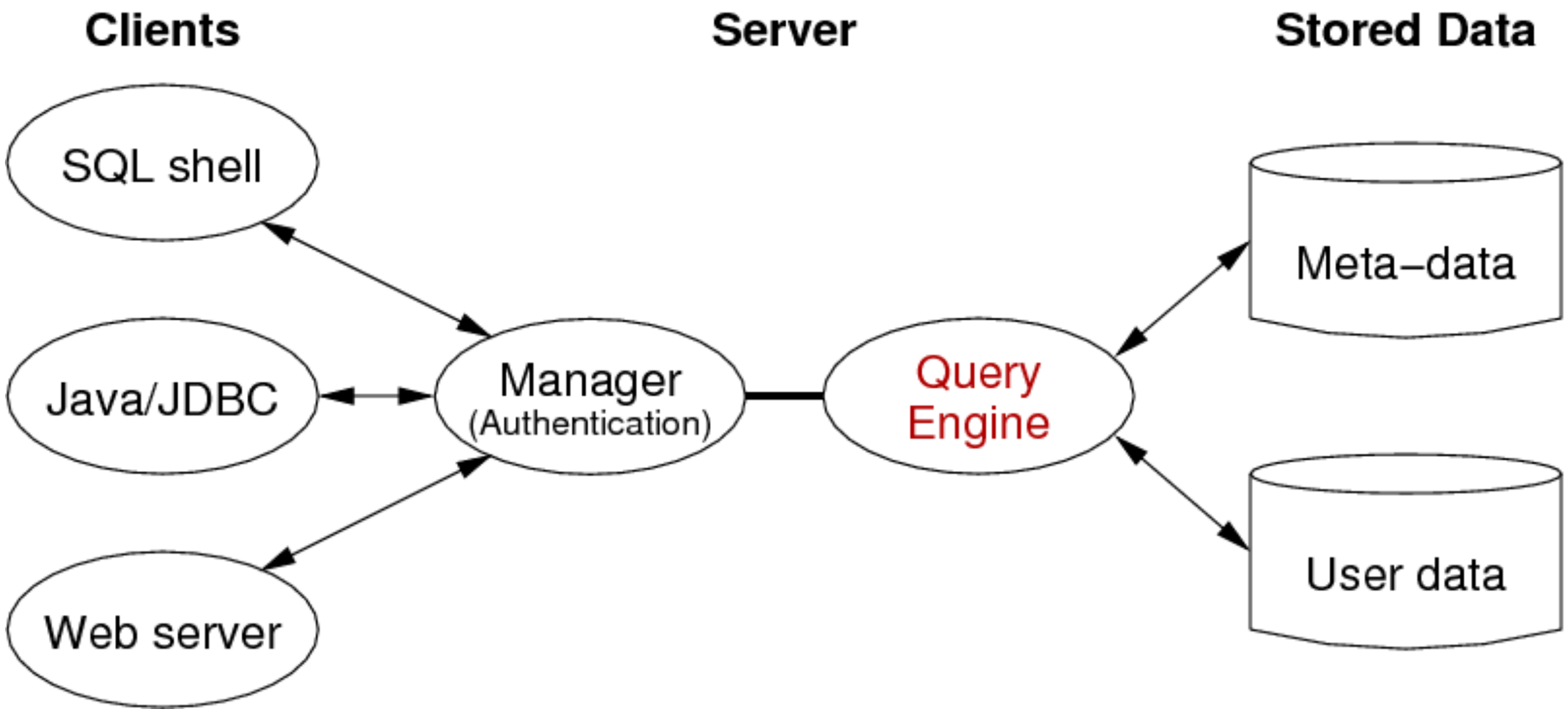
To maximise efficiency of disk transfers

- data is grouped into disk-friendly chunks (pages)
- each page contains .e.g many tuples
- all disk transfers are in units of pages

To avoid disk reads/writes where possible, DBMSs typically have a very large in-memory cache of database pages.

DBMS Architecture

Layers in a DB Engine (Ramakrishnan's View)



DBMS Components

File manager	manages allocation of disk space and data structures
Buffer manager	manages data transfer between disk and main memory
Query optimiser	translates queries into efficient sequence of

	relational ops
Recovery manager	ensures consistent database state after system failures
Concurrency manager	controls concurrent access to database
Integrity manager	verifies integrity constraints and user privileges

---

Produced: 13 Sep 2020