>>

Week 10 Online Sessions

∧ >>

❖ Week 10

Things to note for this week and next week and ...

- Assignment 1 marking queries all investigated
- Assignment 2 due TODAY
- Quiz 5 due before Friday 22 April at 9pm
- MyExperience due before April 28 at midnight
- Final Exam ... Thursday 12 May, 1pm - 5pm

In this session ...

- durability, recovery, exam

<< ^ >>

❖ Atomicity/Durability

Reminder:

Transactions are atomic

- if a tx commits, all of its changes occur in DB
- if a tx aborts, none of its changes occur in DB

Transaction effects are durable

- if a tx commits, its effects persist in DB
  (even in the event of subsequent (catastrophic) system failures)

Implementation of atomicity/durability is intertwined.

They require stable storage and recovery mechanisms

<< ^ >>

❖ Durability

What kinds of "system failures" do we need to deal with?

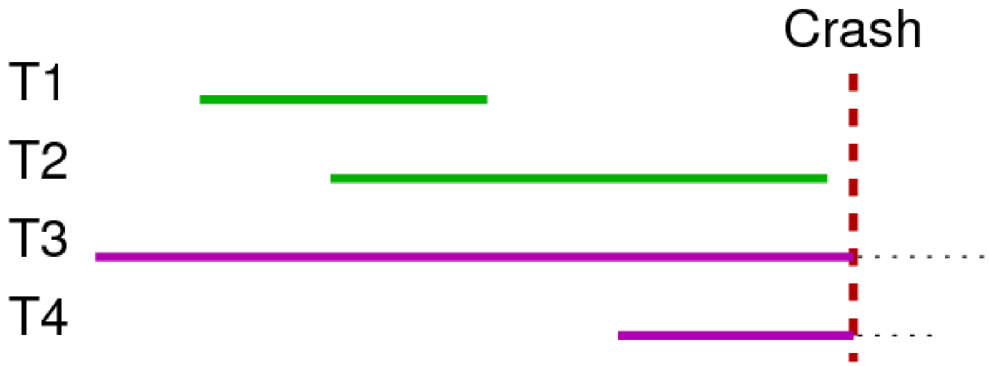
- single-bit inversion during transfer mem-to-disk (parity)
- decay of storage medium on disk (some data changed) (bad block list)
- failure of entire disk device (data no longer accessible) (RAID)
- failure of DBMS processes (e.g. **`postgres`** crashes)
- operating system crash; power failure to computer room
- complete destruction of computer system running DBMS

The first three can be handled outside the DBMS, and invisible to it.

The last requires off-site backup; all others should be locally recoverable.

❖ Durability (cont)

Consider following scenario:



Desired behaviour after system restart:

- all effects of T1, T2 persist
- as if T3, T4 were aborted (no effects remain)

❖ Dealing with Transactions

The "failure modes" that we need to consider:

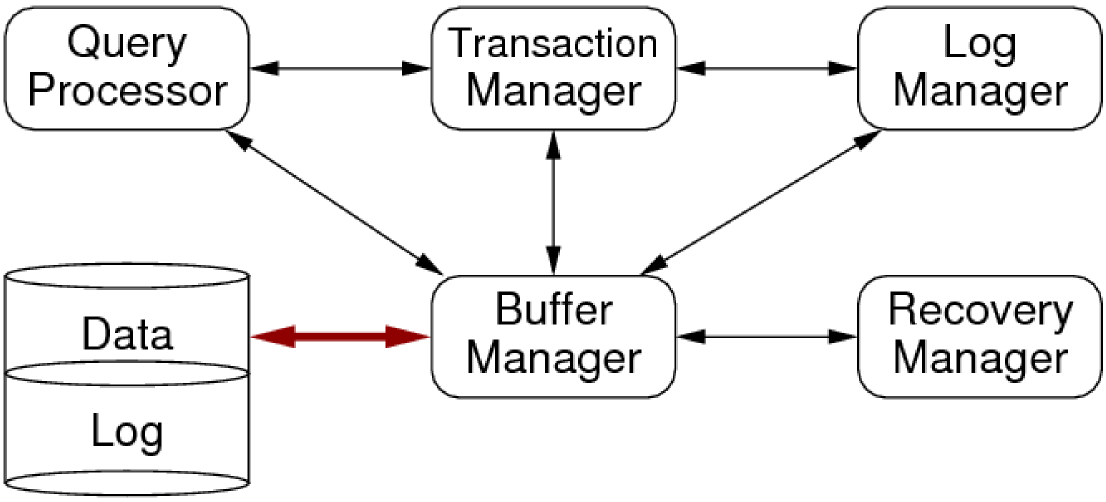
- failure of DBMS processes or operating system
- failure of transactions (**`ABORT`**)

Standard technique for managing these:

- keep a log of changes made to database
- use this log to restore state in case of failures

❖ Architecture for Atomicity/Durability

How does a DBMS provide for atomicity/durability?



<< ^ >>

❖ Execution of Transactions

Transactions deal with three address/memory spaces:

- stored data on the disk (representing persistent DB state)
- data in memory buffers (where held for sharing by tx's)
- data in their own local variables (where manipulated)

Each of these may hold a different "version" of a DB object.

PostgreSQL processes make heavy use of shared buffer pool

⇒ transactions do not deal with much local data.

❖ Execution of Transactions (cont)

Being more precise about data flow within DBMS ...

- **`INPUT(X)`** ... read page containing **`X`** into a buffer
- **`READ(X,v)`** ... copy value of **`X`** from buffer to local var **`v`**
- **`WRITE(X,v)`** ... copy value of local var **`v`** to **`X`** in buffer
- **`OUTPUT(X)`** ... write buffer containing **`X`** to disk

`READ/WRITE` are issued by transaction (e.g. **`R(X)`** in schedules)

`INPUT/OUTPUT` are issued by buffer manager and log manager)

<< ^ >>

❖ Exercise: Data Flow in Transaction

Consider the following transaction:

```
-- implements A = A*2; B = B+1;
BEGIN
READ(A,v); v = v*2; WRITE(A,v);
READ(B,v); v = v+1; WRITE(B,v);
COMMIT
```

Show how the following change values after each statement

- **`v`** ... value of local variable
- **`Buf(A)`** ... value of A stored in memory buffer
- **`Buf(B)`** ... value of B stored in memory buffer
- **`Disk(A)`** ... value of A stored on disk
- **`Disk(B)`** ... value of B stored on disk

What is the final state after the **`COMMIT`** completes?

<< ^ >>

❖ Exercise: Failure in Transaction

Consider the previous transaction.

```
(0) BEGIN
(1) READ(A,v); (2) v = v*2; (3) WRITE(A,v);
(4) READ(B,v); (5) v = v+1; (6) WRITE(B,v);
(7) COMMIT     (8) OUTPUT(A)   (9) OUTPUT(B)
```

What happens if ...

- the tx is aborted before (4)
- the tx is aborted before (7)
- the system crashes before (8)
- the system crashes before (9)

<< ^ >>

❖ Transactions and Buffer Pool

Two issues arise w.r.t. buffers:

- forcing ... **`OUTPUT`** buffer on each **`WRITE`**
  - ensures durability; disk always consistent with buffer pool
  - poor performance; defeats purpose of having buffer pool
- stealing ... replace buffers of uncommitted tx's
  - if we don't, poor throughput (tx's blocked on buffers)
  - if we do, seems to cause atomicity problems?

Ideally, we want stealing and not forcing.

❖ Transactions and Buffer Pool (cont)

Handling stealing:

- transaction T loads page P and makes changes
- T_2 needs a buffer, and P is the "victim"
- P is output to disk (it's dirty) and replaced
- if T aborts, some of its changes are already "committed"
- must log values changed by T in P at "steal-time"
- use these to UNDO changes in case of failure of T

<< ^ >>

❖ Transactions and Buffer Pool (cont)

Handling no forcing:

- transaction T makes changes & commits, then system crashes
- but what if modified page P has not yet been output?
- must log values changed by T in P as soon as they change
- use these to support REDO to restore changes

Above scenario may be a problem, even if we are forcing

- e.g. system crashes immediately after requesting a **`WRITE()`**

<< ^ >>

❖ Recovery

For a DBMS to recover from a system failure, it needs

- a mechanism to record what updates were "in train" at failure time
- methods for restoring the database(s) to a valid state afterwards

Assume multiple transactions are running when failure occurs

- uncommitted transactions need to be rolled back (**`ABORT`**)
- committed, but not yet finalised, tx's need to be completed

A critical mechanism in achieving this is the transaction (event) log

<< ∧ >>

❖ Logging

Three "styles" of logging

- undo ... removes changes by any uncommitted tx's
- redo ... repeats changes by any committed tx's
- undo/redo ... combines aspects of both

All approaches require:

- a sequential file of log records
- each log record describes a change to a data item
- log records are written *before* changes to data
- actual changes to data are written later

Known as write-ahead logging (PostgreSQL uses WAL)

<< ∧ >>

❖ Undo Logging

Simple form of logging which ensures atomicity.

Log file consists of a sequence of small records:

- **`<START T>`** ... transaction **`T`** begins
- **`<COMMIT T>`** ... transaction **`T`** completes successfully
- **`<ABORT T>`** ... transaction **`T`** fails (no changes)
- **`<T,X,v>`** ... transaction **`T`** changed value of **`X`** from **`v`**

Notes:

- we refer to **`<T,X,v>`** generically as **`<UPDATE>`** log records
- update log entry created for each **`WRITE`** (not **`OUTPUT`**)
- update log entry contains *old* value (new value is not recorded)

❖ Undo Logging (cont)

Data must be written to disk in the following order:

1. **`<START>`** transaction log record
2. **`<UPDATE>`** log records indicating changes
3. the changed data elements themselves
4. **`<COMMIT>`** log record

Note: sufficient to have **`<T,X,v>`** output before **`X`**, for each **`X`**

<< ^ >>

❖ Exercise: UNDO Log

Recall the example transaction:

```
(0) BEGIN
(1) READ(A,v); (2) v = v*2; (3) WRITE(A,v);
(4) READ(B,v); (5) v = v+1; (6) WRITE(B,v);
(7) COMMIT     (8) OUTPUT(A)   (9) OUTPUT(B)
```

Show the UNDO log that would be produced while this tx executes?

Where is it important to ensure that the log is written to disk?

❖ Using UNDO Logs

Simplified view of recovery using UNDO logging:

- scan backwards through log
  - if **`<COMMIT T>`**, mark **`T`** as committed
  - if **`<T,X,v>`** and **`T`** not committed, set **`X`** to **`v`** on disk
  - if **`<START T>`** and **`T`** not committed, put **`<ABORT T>`** in log

Assumes we scan entire log; need some way to limit scan.

<< ^ >>

❖ Exercise: Recovery with UNDO Log

Show how the UNDO log would be used if the previous tx

- failed at (5)
- failed at (10)
- failed after (12)

<< ^ >>

❖ Checkpointing

Simple view of recovery implies reading entire log file.

Eventually we can delete "old" section of log.

- i.e. where all prior transactions have committed

This point is called a checkpoint.

- all of log prior to checkpoint can be ignored for recovery

❖ Checkpointing (cont)

Problem: many concurrent/overlapping transactions.

How to know that all have finished?

1. periodically, write log record **`<CHKPT (T1,..,Tk)>`**
   (contains references to all active transactions ⇒ active tx table)
2. continue normal processing (e.g. new tx's can start)
3. when all of **`T1,..,Tk`** have completed,
   write log record **`<ENDCHKPT>`** and flush log

Note: tx manager maintains chkpt and active tx information

<< ∧ >>

❖ Checkpointing (cont)

Recovery: scan backwards through log file processing as before.

Determining where to stop depends on ...

- whether we meet **`<ENDCHKPT>`** or **`<CHKPT...>`** first

If we encounter **`<ENDCHKPT>`** first:

- we know that all incomplete tx's come after prev **`<CHKPT...>`**
- thus, can stop backward scan when we reach **`<CHKPT...>`**

If we encounter **`<CHKPT (T1,..,Tk)>`** first:

- crash occurred *during* the checkpoint period
- any of **`T1,..,Tk`** that committed before crash are ok
- for uncommitted tx's, need to continue backward scan

<< ∧ >>

❖ Redo Logging

Problem with UNDO logging:

- all changed data must be output to disk before committing
- conflicts with optimal use of the buffer pool

Alternative approach is redo logging:

- allow changes to remain only in buffers after commit
- write records to indicate what changes are "pending"
- after a crash, can apply changes during recovery

<< ∧ >>

❖ Redo Logging (cont)

Requirement for redo logging: write-ahead rule.

Data must be written to disk as follows:

1. **`<START>`** transaction log record
2. **`<UPDATE>`** update log records indicating changes
3. **`<COMMIT>`** log record (flushed)
4. then **`OUTPUT`** changed data elements themselves

Note that update log records now contain **`<T,X,v'>`**, where **`v'`** is the *new* value for **`X`**.

<< ^ >>

❖ Exercise: REDO Log

Recall the example transaction:

```
(0) BEGIN
(1) READ(A,v); (2) v = v*2; (3) WRITE(A,v);
(4) READ(B,v); (5) v = v+1; (6) WRITE(B,v);
(7) COMMIT     (8) OUTPUT(A)  (9) OUTPUT(B)
```

Show the REDO log that would be produced while this tx executes?

Where is it important to ensure that the log is written to disk?

<< ∧ >>

❖ Using REDO logs

Simplified view of recovery using REDO logging:

- identify all committed tx's (backwards scan)
- scan forwards through log
  - if **`<T,X,v>`** and **`T`** is committed, set **`X`** to **`v`** on disk
  - if **`<START T>`** and **`T`** not committed, put **`<ABORT T>`** in log

Assumes we scan entire log; use checkpoints to limit scan.

<< ^ >>

❖ Undo/Redo Logging

UNDO logging and REDO logging are incompatible in

- order of outputting **`<COMMIT T>`** and changed data
- how data in buffers is handled during checkpoints

Undo/Redo logging combines aspects of both

- requires new kind of update log record
  `<T,X,v,v'>` gives both old and new values for **`X`**
- removes incompatibilities between output orders

As for previous cases, requires write-ahead of log records.

Undo/redo loging is common in practice; Aries algorithm.

❖ Undo/Redo Logging (cont)

For the example transaction, we might get:

t	Action	v	B(A)	B(B)	D(A)	D(B)	Log
(0)	BEGIN	.	.	.	8	5	<START T>
(1)	READ(A,v)	8	8	.	8	5	
(2)	v = v*2	16	8	.	8	5	
(3)	WRITE(A,v)	16	16	.	8	5	<T,A,8,16>
(4)	READ(B,v)	5	16	5	8	5	
(5)	v = v+1	6	16	5	8	5	
(6)	WRITE(B,v)	6	16	6	8	5	<T,B,5,6>
(7)	FlushLog						
(8)	StartCommit						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)							<COMMIT T>
(11)	OUTPUT(B)	6	16	6	16	6	

Note that T is regarded as committed as soon as (10) completes.

<< ∧ >>

❖ Undo/Redo Logging (cont)

Simplified view of recovery using UNDO/REDO logging:

- scan log to determine committed/uncommitted txs
- for each uncommitted tx **`T`** add **`<ABORT T>`** to log
- scan backwards through log
  - if **`<T,X,v,w>`** and **`T`** is not committed, set **`X`** to **`v`** on disk
- scan forwards through log
  - if **`<T,X,v,w>`** and **`T`** is committed, set **`X`** to **`w`** on disk

<< ∧ >>

❖ Undo/Redo Logging (cont)

The above description simplifies details of undo/redo logging.

Aries is a complete algorithm for undo/redo logging.

Differences to what we have described:

- log records contain a sequence numnber (LSN)
- LSNs used in tx and buffer managers, and stored in data pages
- additional log record to mark **<END>** (of commit or abort)
- **<CHKPT>** contains only a timestamp
- **<ENDCHKPT..>** contains tx and dirty page info

❖ Recovery in PostgreSQL

PostgreSQL uses write-ahead undo/redo style logging.

It also uses multi-version concurrency control, which

- tags each record with a tx and update timestamp

MVCC simplifies some aspects of undo/redo, e.g.

- some info required by logging is already held in each tuple
- no need to undo effects of aborted tx's; use old version

<< ^ >>

❖ Recovery in PostgreSQL (cont)

Transaction/logging code is distributed throughout backend.

Core transaction code is in **`src/backend/access/transam`**.

Transaction/logging data is written to files in **`PGDATA/pg_wal`**

- a number of very large files containing log records
- old files are removed once all txs noted there are completed
- new files added when existing files reach their capacity (16MB)
- number of tx log files varies depending on tx activity

❖ Week 10 Thursday

Things to note for this week and next week and ...

- All marking finalised by end next week
- Quiz 5 due before 9pm tomorrow
- MyExperience due before April 28 at midnight
- Final Exam ... Thursday 12 May, 1pm - 5pm

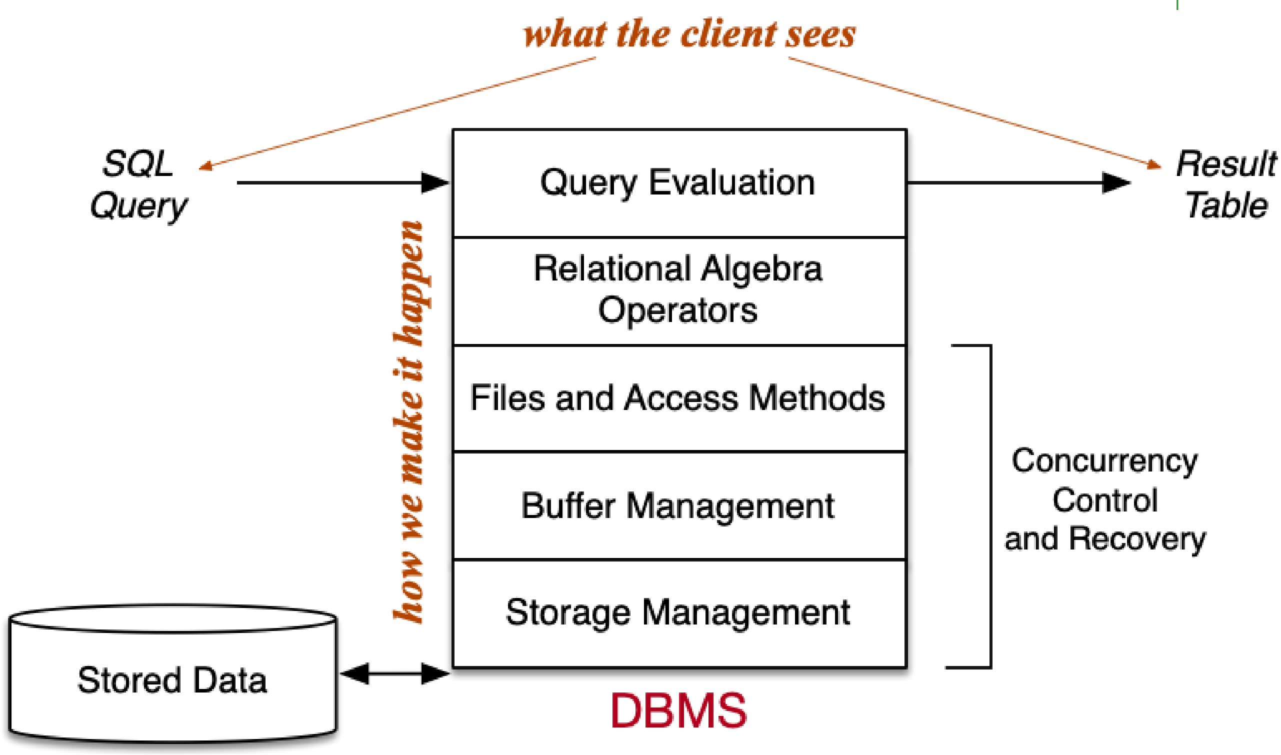
In this session ...

- course review, exam

<< ∧ >>

❖ What's this Course about?

Understanding how relational DBMSs work ...



<< ^ >>

❖ Syllabus

View of DBMS internals from the bottom-up:

- storage subsystem (disks,pages)
- buffer manager, representation of data
- implementing RA operations (sel,proj,join,...)
- combining RA operations (iterators/execution)
- query translation, optimization, execution
- transactions, concurrency, durability

<< ^ >>

❖ Exam

Thursday 12 May, 1pm - 5pm AEST

Held in the comfort of your own home.

All answers are typed and submitted on-line.

Can submit answers any time in this period (1-5).

Exam paper (questions, working files) are available at 1pm.

Submissions after 5pm are ignored (unless you have ELS provisions).

You can submit each question multiple times; last submission is marked.

Let me know ASAP if you are in an unusual time zone (e.g. Europe, USA).

<< ^ >>

❖ Exam (cont)

Is this a 3-hour exam?

If it was running in a CSE lab, it would have 3-hours duration.

If you know your stuff, it should take around 3 hours to complete.

You have up to 4 hours if you need it, and if you start at 1pm.

Environments: VLab or **`ssh`** or **`putty`** or work locally

Learn to use the shell, a text editor and on-screen calculator.

We monitor Chegg, WeChat, *etc*; any posts there will be prosecuted.

We run plagiarism checking on submissions; plagiarists will be prosecuted.

<< ^ >>

❖ Exam (cont)

Resources available during exam:

- exam questions (collection of web pages)
- program directories and question templates (**`exam-work.zip`**)
- PostgreSQL manual (collection of web pages)
- C programming reference (collection of web pages)
- Course web site (all, including submission pages)

And you can access the whole of the Internet.

Except, **do not** communicate with anyone else.

<< ^ >>

❖ Exam (cont)

Tools available during the exam on the CSE servers

- C compiler (**`gcc`**, **`make`**)
- text editors (e.g. **`vim`**, **`emacs`**, **`gedit`**, **`nano`**, ...)
- code editor (e.g. **`code`**)
- on-screen calculators (e.g. **`bc`**, **`gcalctool`**, **`xcalc`**)
- all your favourite Linux tools (e.g. **`ls`**, **`grep`**, ...)
- Linux manual (**`man`**)

Answers are submitted either via **`give`** or Webcms3

❖ Exam (cont)

Minimal tool set to work at home during the exam

- C compiler (**`gcc`**, **`make`**)
- text editors (**`vim`**, **`emacs`**, **`gedit`**, **`nedit`**, **`nano`**, ...)
- a calculator (**`bc`**, **`gcalctool`**, **`xcalc`**)

And, yes, you can use VScode on your own machine, if you insist.

<< ^ >>

❖ Before the exam ...

Practice with the sample exams

Use **`VLab`**

- especially if you haven't used it during term

or

Set up a working environment on your computer

- need a C compiler (+ **`make`**), and text editor
- learn how to use **`scp`**, **`ssh`**, the Unix shell
  - **`scp`** can be replaced by any file-transfer tool
  - **`ssh`** can be replaced by **`putty`**

❖ At the start of the exam ...

Read the **Instructions** page (see sample exams)

Make an exam working directory (either on VLab or home machine)

Unzip **`exam-work.zip`** in that directory

Optionally, if working at home, unzip **`paper.zip`** into another directory

Read the exam font cover and the questions

Plan which questions to answer first

Go!

<< ^ >>

❖ During the exam ...

Do not spend more than 45 mins on any question.

- if stuck submit what you've got, do other questions, return to this one later

If you need clarification on some question

- send email to cs9315@cse.unsw.edu.au
- email will be monitored for duration of exam

If we need to change/correct an exam question

- we will change the version on the CSE servers
- we will post a Notice on the Webcms3 site
  (for people working on a downloaded copy of the exam paper)

❖ During the exam ... (cont)

If there are technical difficulties with the CSE servers/Webcms3

- send email to alert us
- we will attempt to fix within 30 mins

If you have technical difficulties with your machine

- try to fix (e.g. reboot) and email us (with photo) if long delay

If your machine/network dies during exam, apply for Special Consideration

- will need convincing evidence of the equipment failure

<< ∧ >>

❖ What's on the Exam?

Potential topics to be examined: anything under "Videos and Slides"

Questions will have the following "flavours" ...

- write a small C program to do V
- describe what happens when we execute method W
- how many page accesses occur if we do X on Y
- explain the numbers in the following output
- describe the characteristics of Z

There will be **no** SQL/PLpgSQL code writing.

You will **not** have to modify PostgreSQL during the exam.

<< ∧ >>

❖ Exam Structure

There will be 8 questions

- 3 x C programming questions (60%, ~2 hours)
- 5 x written answer questions (40%, ~1 hour)

Reminder:

- exam contributes 50% of final mark
- hurdle requirement: must score > 20/50 on exam

❖ Special Consideration

Reminder: this is a one-chance exam.

- attempting the Exam is treated as "I am fit and well" (fit-to-sit)
- subsequent claims of "I failed because I felt sick" are ignored

If you're sick, get documentation and do not attempt the exam.

Special consideration requests must clearly show

- how you were personally affected
- that your ability to study/take-exam was impacted

Other factors are not relevant (e.g. "I can't afford to repeat")

<< ^ >>

❖ Revision

Things you can use for revision:

- past exams
- theory exercises
- prac exercises
- course notes
- textbooks

If you have questions before the exam

- post them on the forum, or
- send email to cs9315@cse.unsw.edu.au

<< ^

❖ And that's all folks ...

End of COMP9315 22T1 Lectures

Good luck with the exam ...

And keep on using PostgreSQL ...

And don't forget to give feedback via MyExperience!

Produced: 21 Apr 2022