Week 10 Lectures

Implementing Atomicity/Durability

Atomicity/Durability

Reminder:

Transactions are *atomic*

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

Transaction effects are *durable*

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

Implementation of atomicity/durability is intertwined.

Durability

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

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

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

... Durability

Consider following scenario:

Desired behaviour after system restart:

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

... Durability

Durabilty begins with a *stable disk storage subsystem*

- i.e. `putPage()` and `getPage()` always work as expected

We can prevent/minimise loss/corruption of data due to:

- mem/disk transfer corruption ⇒ parity checking
- sector failure ⇒ mark "bad" blocks
- disk failure ⇒ RAID (levels 4,5,6)
- destruction of computer system ⇒ off-site backups

Dealing with Transactions

6/147

The remaining "failure modes" that we need to consider:

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

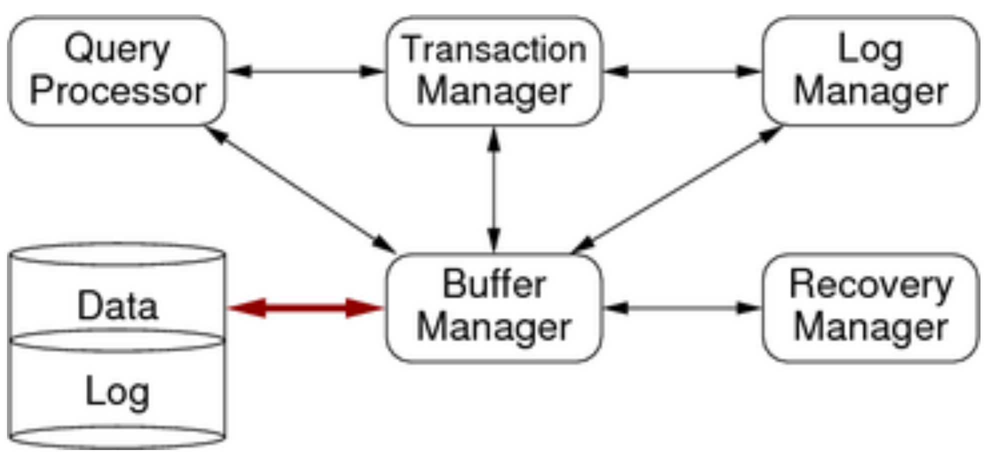
Standard technique for managing these:

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

Architecture for Atomicity/Durability

7/147

How does a DBMS provide for atomicity/durability?



Execution of Transactions

8/147

Transactions deal with three address spaces:

- stored data on the disk (representing global 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

9/147

Operations available for data transfer:

- `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.

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

`INPUT/OUTPUT` correspond to `getPage()/putPage()` mentioned above

... Execution of Transactions

Example of transaction execution:

```
-- 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
```

`READ` accesses the buffer manager and may cause `INPUT`.

`COMMIT` needs to ensure that buffer contents go to disk.

... Execution of Transactions

States as the transaction executes:

t	Action	v	Buf(A)	Buf(B)	Disk(A)	Disk(B)
(0)	BEGIN	.	.	.	8	5
(1)	READ(A,v)	8	8	.	8	5
(2)	v = v*2	16	8	.	8	5
(3)	WRITE(A,v)	16	16	.	8	5
(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
(7)	COMMIT	6	16	6	8	5
(8)	OUTPUT(A)	6	16	6	16	5
(9)	OUTPUT(B)	6	16	6	16	6
(*)	*committed*	-	-	-	16	6

If system crashes before (7), may need to undo disk changes.
If system crashes after (7), may need to redo disk changes.

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, maybe 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

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

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()`

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 first
- 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

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`

... Undo Logging

18/147

For the example transaction, we would 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>
(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>
(7)	FlushLog						
(8)	StartCommit						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)	OUTPUT(B)	6	16	6	16	6	
(11)	EndCommit						<COMMIT T>
(12)	FlushLog						

Note that T is not regarded as committed until (12) completes.

... Undo Logging

19/147

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; use checkpoints to limit scan.

... Undo Logging

20/147

Algorithmic view of recovery using UNDO logging:

```
committedTrans = abortedTrans = startedTrans = {}
for each log record from most recent to oldest {
    switch (log record) {
    <COMMIT T> : add T to committedTrans
    <ABORT T>  : add T to abortedTrans
    <START T>  : add T to startedTrans
    <T,X,v>    : if (T in committedTrans)
                     // don't undo committed changes
                 else  // roll-back changes
                     { WRITE(X,v); OUTPUT(X) }
}   }
for each T in startedTrans {
    if (T in committedTrans) ignore
    else if (T in abortedTrans) ignore
    else write <ABORT T> to log
}
```

flush log

Checkpointing

21/147

Simple view of recovery implies reading entire log file.

Since log file grows without bound, this is infeasible.

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

22/147

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

23/147

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

24/147

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

Requirement for redo logging: *write-ahead rule*.

Data must be written to disk as follows:

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

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

... Redo Logging

For the example transaction, we would 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,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,6>
(7)	COMMIT						<COMMIT T>
(8)	FlushLog						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)	OUTPUT(B)	6	16	6	16	6	

Note that T is regarded as committed as soon as (8) completes.

... Redo Logging

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

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

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

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

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

(`PGDATA/pg_wal` used to be called `PGDATA/pg_xlog`)

Database Trends (overview)

Future of Database

Core "database" goals:

- deal with very large amounts of data (petabyes, exabytes, ...)
- very-high-level languages (deal with data in uniform ways)
- fast query execution (evaluation too slow ⇒ useless)

At the moment (and for the last 30 years) RDBMSs dominate ...

- simple/clean data model, backed up by theory
- high-level language for accessing data
- 40 years development work on RDBMS engine technology

RDBMSs work well in domains with uniform, structured data.

... Future of Database

Limitations/pitfalls of classical RDBMSs:

- NULL is ambiguous: unknown, not applicable, not supplied
- "limited" support for constraints/integrity and rules
- no support for uncertainty (data represents *the* state-of-the-world)
- data model too simple (e.g. no direct support for complex objects)
- query model too rigid (e.g. no approximate matching)
- continually changing data sources not well-handled
- data must be "molded" to fit a single rigid schema
- database systems must be manually "tuned"
- do not scale well to some data sets (e.g. Google, Telco's)

... Future of Database

How to overcome (some) RDBMS limitations?

Extend the relational model ...

- add new data types and query ops for new applications
- deal with uncertainty/inaccuracy/approximation in data

Replace the relational model ...

- object-oriented DBMS ... OO programming with persistent objects
- XML DBMS ... all data stored as XML documents, new query model
- noSQL data stores (e.g. *(key,value)* pairs, json or rdf)

... Future of Database

How to overcome (some) RDBMS limitations?

Performance ...

- new query algorithms/data-structures for new types of queries
- parallel processing
- DBMSs that "tune" themselves

Scalability ...

- distribute data across (more and more) nodes
- techniques for handling streams of incoming data

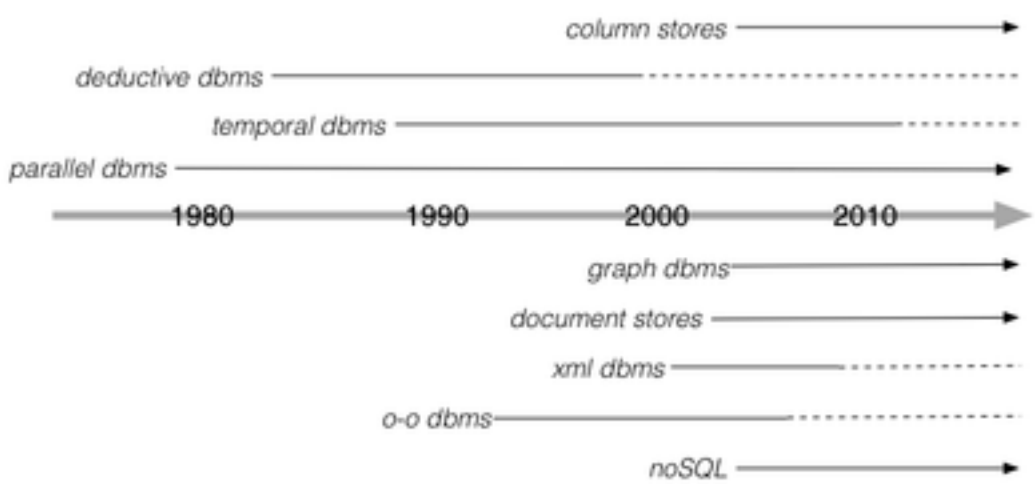
... Future of Database

An overview of the possibilities:

- *"classical" RDBMS* (e.g. PostgreSQL, Oracle, SQLite)
- *parallel DBMS* (e.g. XPRS)
- *distributed DBMS* (e.g. Cohera)
- *deductive databases* (e.g. Datalog)
- *temporal databases* (e.g. MariaDB)
- *column stores* (e.g. Vertica, Druid)
- *object-oriented DBMS* (e.g. ObjectStore)
- *key-value stores* (e.g. Redis, DynamoDB)
- *wide column stores* (e.g. Cassandra, Scylla, HBase)
- *graph databases* (e.g. Neo4J, Datastax)
- *document stores* (e.g. MongoDB, Couchbase)
- *search engines* (e.g. Google, Solr)

... Future of Database

Historical perspective



Large Data

Some modern applications have massive data sets (e.g. Google)

- far too large to store on a single machine/RDBMS
- query demands far too high even if could store in DBMS

Approach to dealing with such data

- distribute data over large collection of nodes (also, redundancy)
- provide computational mechanisms for distributing computation

Often this data does not need full relational selection

- represent data via *(key,value)* pairs
- unique *key*s can be used for addressing data
- *values* can be large objects (e.g. web pages, images, ...)

... Large Data

Popular computational approach to such data: *map/reduce*

- suitable for widely-distributed, very-large data
- allows parallel computation on such data to be easily specified
- distribute (map) parts of computation across network
- compute in parallel (possibly with further *map*ping)
- merge (reduce) multiple results for delivery to requestor

Some large data proponents see no future need for SQL/relational ...

- depends on application (e.g. hard integrity vs eventual consistency)

Humour: Parody of noSQL fans (strong language warning)

- Compares MySQL (not PostgreSQL) to noSQL systems

Information Retrieval

DBMSs generally do precise matching (although `like`/regexps)

Information retrieval systems do approximate matching.

E.g. documents containing a set of keywords (Google, etc.)

Also introduces notion of "quality" of matching
(e.g. tuple T_1 is a *better* match than tuple T_2)

Quality also implies *ranking* of results.

Ongoing research in incorporating IR ideas into DBMS context.

Goal: support database exploration better.

Multimedia Data

Data which does not fit the "tabular model":

- image, video, music, text, ... (and combinations of these)

Research problems:

- how to specify queries on such data? ($image_1 \cong image_2$)
- how to "display" results? (synchronize components)

Solutions to the first problem typically:

- extend notions of "matching"/indexes for querying
- require sophisticated methods for capturing data features

Sample query: find other songs *like* this one?

Uncertainty

Multimedia/IR introduces approximate matching.

In some contexts, we have approximate/uncertain data.

E.g. witness statements in a crime-fighting database

"I think the getaway car was red ... or maybe orange ..."

"I am 75% sure that John carried out the crime"

Work by Jennifer Widom at Stanford on the *Trio* system

- extends the relational model (ULDB)
- extends the query language (TriQL)

Stream Data Management Systems

Makes one addition to the relational model

- *stream* = infinite sequence of tuples, arriving one-at-a-time

Applications: news feeds, telecomms, monitoring web usage, ...

RDBMSs: run a variety of queries on (relatively) fixed data

StreamDBs: run fixed queries on changing data (stream)

One approach: *window* = "relation" formed from a stream via a rule

E.g. StreamSQL

```
select avg(price)
from examplestream [size 10 advance 1 tuples]
```

Graph Data

Uses *graphs* rather than tables as basic data structure tool.

Applications: social networks, ecommerce purchases, interests, ...

Many real-world problems are modelled naturally by graphs

- can be represented in RDBMSs, but not processed efficiently
- e.g. recursive queries on `Nodes`, `Properties`, `Edges` tables

Graph data models: flexible, "schema-free", inter-linked

Typical modeling formalisms: XML, JSON, RDF

More details later ...

Dispersed Databases

Characteristics of dispersed databases:

- very large numbers of small processing nodes
- data is distributed/shared among nodes

Applications: environmental monitoring devices, "intelligent dust", ...

Research issues:

- query/search strategies (how to organise query processing)
- distribution of data (trade-off between centralised and diffused)

Less extreme versions of this already exist:

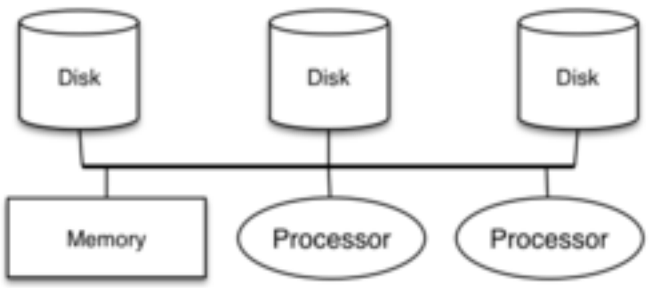
- grid and cloud computing
- database management for mobile devices

Parallelism in Databases

Parallel DBMSs

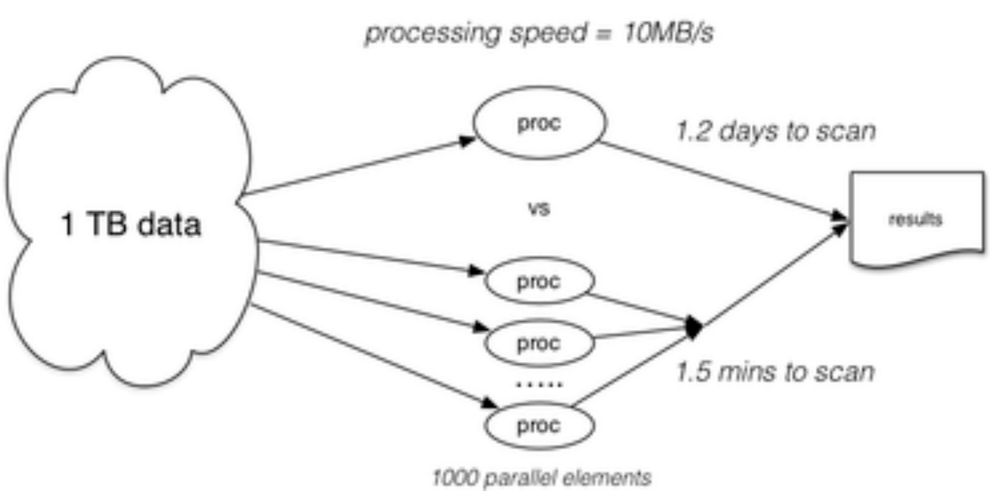
RDBMS discussion so far has revolved around systems

- with a single or small number of processors
- accessing a single memory space
- getting data from one or more disk devices



... Parallel DBMSs

Why parallelism? ... Throughput!



... Parallel DBMSs

DBMSs are a success story in application of parallelism

- can process many data elements (tuples) at the same time
- can create pipelines of query evaluation steps
- don't require special hardware

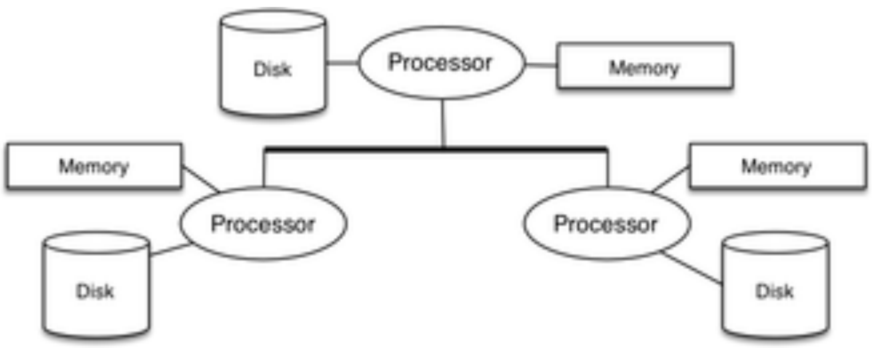
- can hide paralleism within the query evaluator
    - application programmers don't need to change habits

Compare this with effort to do parallel programming.

Parallel Architectures

Types: *shared memory*, *shared disk*, *shared nothing*

Example shared-nothing architecture:

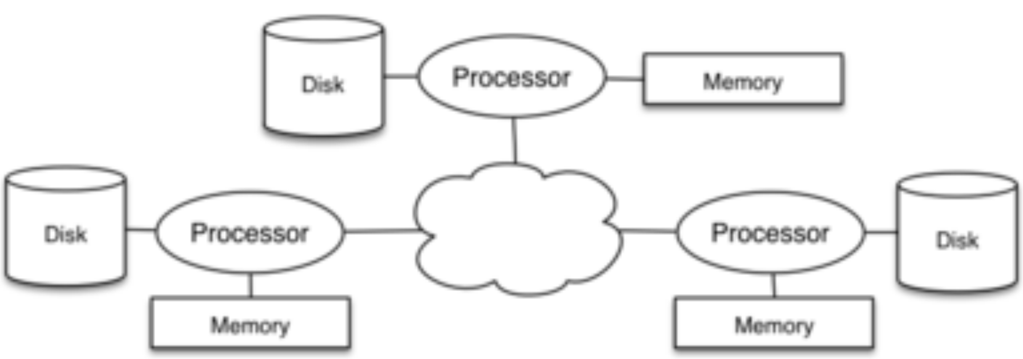


Typically same room/LAN (data transfer cost ~ 100's of μsecs .. msecs)

Distributed Architectures

Distributed architectures are ...

- effectively shared-nothing, on a global-scale network



Typically on the Internet (data transfer cost ~ secs)

Parallel Databases (PDBs)

Parallel databases provide various forms of parallelism ...

- process parallelism can speed up query evaluation
- processor parallelism can assist in speeding up memory ops
- processor parallelism introduces cache coherence issues
- disk parallelism can assist in overcoming latency
- disk parallelism can be used to improve fault-tolerance (RAID)
- one limiting factor is congestion on communication bus

... Parallel Databases (PDBs)

Types of parallelism

- *pipeline parallelism*
    - multi-step process, each processor handles one step
    - run in parallel and pipeline result from one to another

- *partition parallelism*
    - many processors running in parallel
    - each performs same task on a subset of the data
    - results from processors need to be merged

Data Storage in PDBs

Assume that each table/relation consists of pages in a file

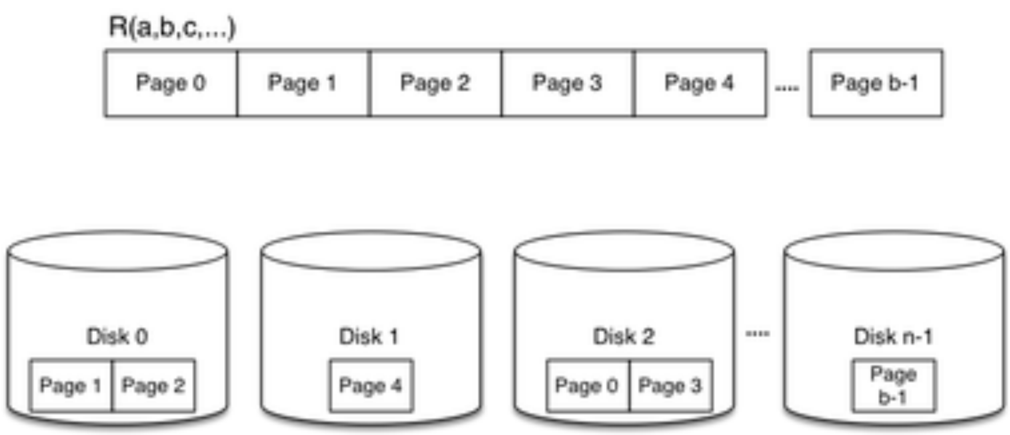
Can distribute data across multiple storage devices

- duplicate all pages from a relation (replication)
- store some pages on one store, some on others (partitioning)

Replication example:

... Data Storage in PDBs

Data-partitioning example:



... Data Storage in PDBs

A table is a collection of *pages* (aka blocks).

Page addressing on single processor/disk: *(Table, File, Page)*

- *Table* maps to a set of files (e.g. named by tableID)
- *File* distinguishes primary/overflow files
- *PageNum* maps to an offset in a specific file

If multiple nodes, then addressing depends how data distributed

- partitioned: *(Node, Table, File, Page)*
- replicated: *({Nodes}, Table, File, Page)*

... Data Storage in PDBs

Assume that partitioning is based on one attribute

Data-partitioning strategies for one table on *n* nodes:

- *round-robin*, *hash-based*, *range-based*

Round-robin partitioning

- cycle through nodes, new tuple added on "next" node
- e.g. i^{th} tuple is placed on $(i \bmod n)^{th}$ node
- balances load on nodes; no help for querying

... Data Storage in PDBs

Hash partitioning

- use hash value to determine which node and page
- e.g. *i = hash(tuple)* so tuple is placed on i^{th} node
- helpful for equality-based queries on hashing attribute

Range partitioning

- ranges of attr values are assigned to processors
- e.g. values 1-10 on $node_0$, 11-20 on $node_1$, ..., 99-100 $node_{n-1}$
- potentially helpful for range-based queries

In both cases, data skew may lead to unbalanced load

Parallelism in DB Operations

Different types of parallelism in DBMS operations

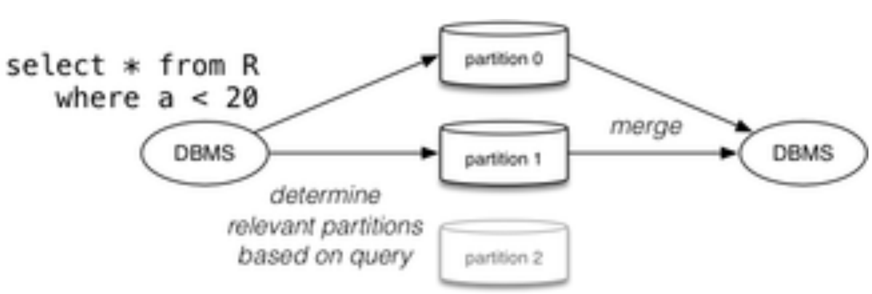
- intra-operator parallelism
  - get all machines working to compute a given operation (scan, sort, join)
- inter-operator parallelism
  - each operator runs concurrently on a different processor (exploits pipelining)
- Inter-query parallelism
  - different queries run on different processors

... Parallelism in DB Operations

Parallel scanning

- scan partitions in parallel and merge results
- maybe ignore some partitions (e.g. range and hash partitioning)
- can build indexes on each partition

Effectiveness depends on query type vs partitioning type



... Parallelism in DB Operations

Parallel sorting

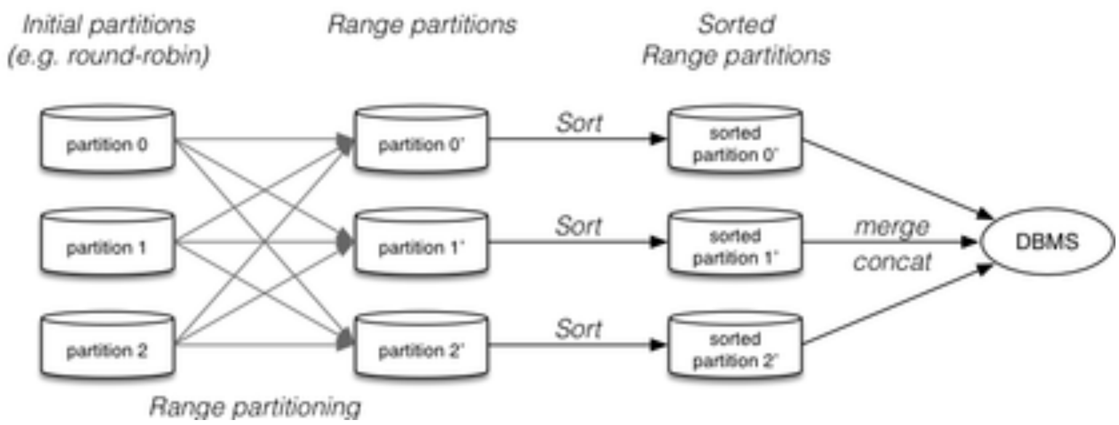
- scan in parallel, range-partition during scan
- pipeline into local sort on each processor
- merge sorted partitions in order

Potential problem:

- data skew because of unfortunate choice of partition points
- resolve by initial data sampling to determine partitions

... Parallelism in DB Operations

Parallel sort:



... Parallelism in DB Operations

Parallel nested loop join

- each outer tuple needs to examine each inner tuple
- but only if it could potentially join
- range/hash partitioning reduce partitions to consider

Parallel sort-merge join

- as noted above, parallel sort gives range partitioning
- merging partitioned tables has no parallelism (but is fast)

... Parallelism in DB Operations

Parallel hash join

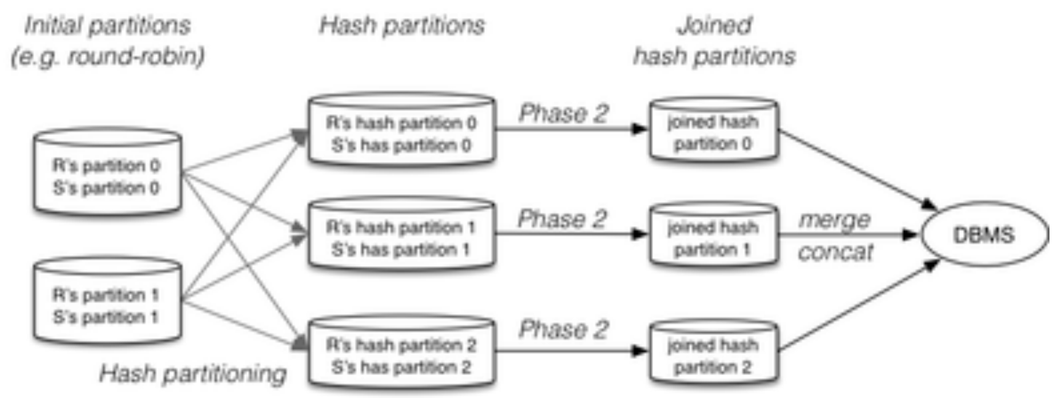
- distribute partitions to different processors
- partition 0 of R goes to same node as partition 0 of S
- join phase can be done in parallel on each processor
- then results need to be merged
- very effective for equijoin

Fragment-and-replicate join

- outer relation R is partitioned (using any partition scheme)
- inner relation S is copied to all nodes
- each node computes join with R partition and S

... Parallelism in DB Operations

Parallel hash join:



PostgreSQL and Parallelism

PostgreSQL assumes

- shared memory space accessable to all back-ends
- files for one table are located on one disk

PostgreSQL allows

- data to be distributed across multiple disk devices

So could run on ...

- shared-memory, shared-disk architectures
- hierarchical architectures with distributed virtual memory

... PostgreSQL and Parallelism

PostgreSQL can provide

- multiple servers running on separate nodes
- application #1: high availability
    - "standby" server takes over if primary server fails
- application #2: load balancing
    - several servers can be used to provide same data
    - direct queries to least loaded server

Both need *data synchronisation* between servers

PostgreSQL uses notion of *master* and *slave* servers.

... PostgreSQL and Parallelism

High availability ...

- updates occur on master, recorded in tx log
- tx logs shipped/streamed from master to slave(s)
- slave uses tx logs to maintain current state
- configuration controls frequency of log shipping
- bringing slave up-to-date is "fast" (~1-2secs)

Note: small window for data loss (committed tx log records not sent)

Load balancing ...

- not provided built-in to PostgreSQL, 3rd-party tools exist

Distributed Databases

A *distributed database* (DDB) is

- collection of multiple logically-related databases
- distributed over a network (generally a WAN)

A *distributed database management system* (DDBMS) is

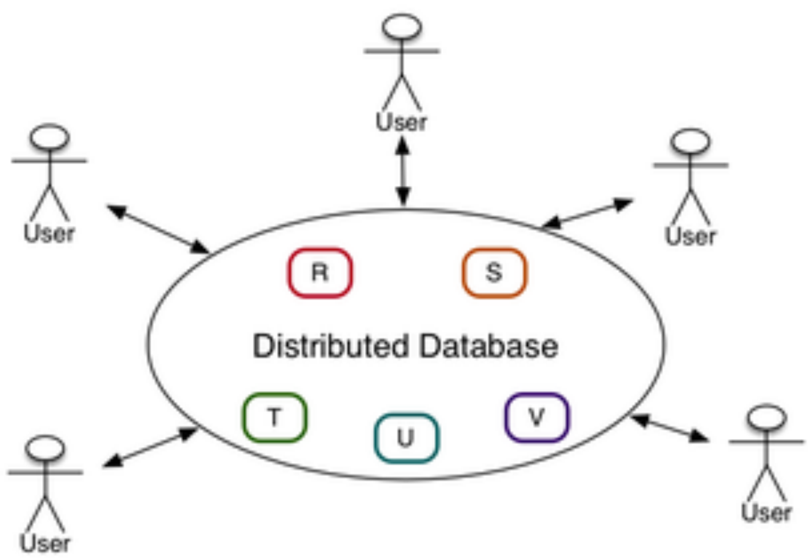
- software that manages a distributed database
- providing access that hides complexity of distribution

A DDBMS may involve

- instances of a single DBMS (e.g. ≥1 PostgreSQL servers)
- a layer over multiple different DBMSs (e.g. Oracle+PostgreSQL+DB2)

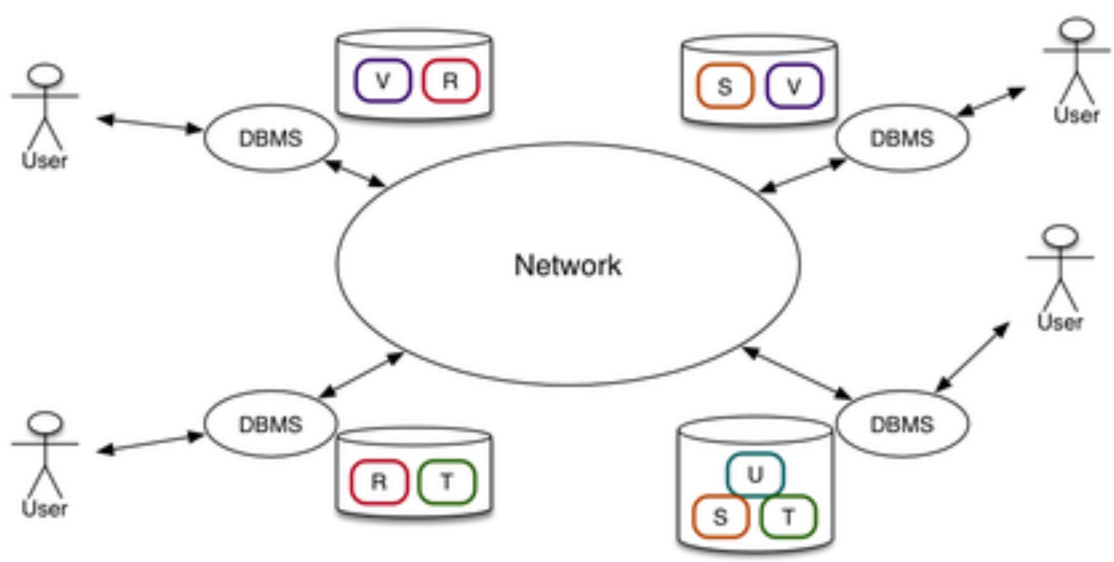
... Distributed Databases

User view:



... Distributed Databases

Arhcitecture:



... Distributed Databases

Two kinds of distributed databases

- parallel database on a distributed architecture
    - single schema, homogeneous DBMSs
- independent databases on a distributed architecture
    - independent schemas, heterogeneous DBMSs

The latter are also called *federated* databases

Ultimately, the distributed database (DDB) provides

- global schema, with mappings from constituent schemas
- giving the impression of a single database

... Distributed Databases

Advantages of distributed databases

- allow information from multiple DBs to be merged
- provide for replication of some data (resilience)
- allow for possible parallel query evaluation

Disadavtanges of distributed databases

- cost of mapping between different schemas (federated)
- communication costs (write-to-network vs write-to-disk)
- maintaining ACID properties in distributed transactions

... Distributed Databases

Application examples:

- bank with multiple branches
    - local branch-related data (e.g. accounts) stored in branch
    - corporate data (e.g. HR) stored on central server(s)
    - central register of credit-worthiness for customers
- chain of department stores
    - store-related data (e.g. sales, inventory) stored in store
    - corporate data (e.g. customers) stored on central server(s)
    - sales data sent to data warehouse for analysis

... Distributed Databases

In both examples

- some data is conceptually a single table at corporate level
- but does not physically exist as a table in one location

E.g. `account(acct_id, branch, customer, balance)`

- each branch maintains its own data (for its accounts)
- set of tuples, all with same `branch`
- bank also needs a view of *all* accounts

Data Distribution

Partitioning/distributing data

- where to place (parts of) tables
    - determined by usage of data (locality, used together)
    - affects communication cost ⇒ query evaluation cost

- how to partition data within tables
    - no partitioning ... whole table stored on ≥1 DBMS
    - *horizontal partitioning* ... subsets of rows
    - *vertical partitioning* ... subsets of columns

Problem: maintaining consistency

... Data Distribution

Consider table R horizontally partitioned into R_1, R_2, ..., R_n

Fragmentation can be done in multiple ways, but need to ensure ...

Completeness

- decompostion is complete iff each $t \in R$ is in some R_i

Reconstruction

- original R can be produced by some relational operation

Disjoint

- if item $t \in R_i$, then $t \notin R_k$, $k \neq i$ (assuming no replication)

Query Processing

Query processing typically involves *shipping* data

- e.g. reconstructing table from distributed partitions
- e.g. join on tables stored on separate sites

Aim: minimise shipping cost   (since it is a networking cost)

Shipping cost becomes the "disk access cost" of DQOpt

Can still use cost-based query optimisation

- consider possible execution plans, choose cheapest

... Query Processing

Distributed query processing

- may require query ops to be executed on different nodes
    - node provides only source of some data
    - some nodes may have limited set of operations
- needs to merge data received from different nodes
    - may require data transformation (to fit schemas together)

Query optimisation in such contexts is *complex* ...

- larger space of possibilities than single-node database

Transaction Processing

Distribution of data complicates tx processing ...

- potential for multiple copies of data to become inconsistent
- commit or abort must occur consistently on all nodes

Distributed tx processing handled by *two-phase commit*

- initiating site has transaction coordinator C_i ...
    - waits for all other sites executing tx `T` to "complete"
    - sends `<prepare T>` message to all other sites
    - waits for `<ready T>` response from all other sites
    - if not received (timeout), or `<abort T>` received, flag abort
    - if all other sites respond `<ready T>`, flag commit
    - write `<commit T>` or `<abort T>` to log
    - send `<commit T>` or `<abort T>` to all other sites
- non-initiating sites write log entries before responding

Non-classical DBMSs

Classical DBMSs

Assumptions made in conventional DBMSs:

- data is sets of tuples; tuples are lists of atomic values
- data values can be compared precisely (via =, >, <, ...)
- filters can be described via boolean formulae
- SQL is a suitable language for all data management
- transaction-based consistency is critical
- data stored on disk, processed in memory
- data transferred in blocks of many tuples
- disk ↔ memory cost is most expensive in system
- disks are connected to processors via fast local bus

Modern DBMSs

Demands from modern applications

- more flexible data structuring mechanisms
- very large data objects/values (e.g. music, video)
- alternative comparisons/filters (e.g. similarity matching)
- massive amounts of data (too much to store "locally")
- massive number of clients (thousands tx's per second)
- solid-state storage (minimal data latency)
- data required globally (network latency)

Clearly, not all of these are relevant for every modern application.

... Modern DBMSs

Some conclusions:

- relational model doesn't work *for all* applications
- SQL is not appropriate *for all* applications
- hard transactions not essential *for all* applications

Some "modernists" claim that

- "for all" is really "for any"
- ⇒ relational DBMSs and SQL are dinosaurs
- ⇒ NoSQL is the new way

... Modern DBMSs

Some approaches:

- storage systems: Google FS, Hadoop DFS,   Amazon S3
- data structures: BigTable, HBase, Cassandra,   XML, RDF
- data structures: column-oriented DBMSs e.g. C-store
- data structures: graph databases e.g. Neo4j
- operations: multimedia similarity search e.g. Shazam
- operations: web search e.g. Google
- transactions: eventual consistency
- programming: object-relational mapping (ORM)
- programming: MapReduce
- languages: Sawzall, Pig, Hive,   SPARQL
- DB systems: CouchDB, MongoDB, F1, Cstore

Scale, Distribution, Replication

Data for modern applications is very large (TB, PB, XB)

- not feasible to store on a single machine
- not feasible to store in a single location

Many systems opt for massive networks of simple nodes

- each node holds moderate amount of data
- each data item is replicated on several nodes
- nodes clustered in different geographic sites

Benefits:

- reliability, fault-tolerance, availability
- proximity ... use data closest to client
- scope for parallel execution/evaluation

Schema-free Data Models

Many new DBMSs provide *(key,value)* stores

- *key* is a unique identifier (cf. URI)
- *value* is an arbitrarily complex "object"
    - e.g. a text document (often structured, e.g. Wiki, XML)
    - e.g. a JSON object: *(property,value)* list
    - e.g. an RDF triple (e.g. `<John,worksFor,UNSW>`)
- objects may contain *key*s to link to other objects

Tables can be simulated by a collection of "similar" objects.

Eventual Consistency

RDBMSs use a strong transactional/consistency model

- if a tx commits, changes take effect "instantly"
- all tx's have a strong guarantee about data integrity

Many new DBMSs applications do not need strong consistency

- e.g. doesn't matter if catalogue shows yesterday's price

Because of distribution/replication

- update is initiated on one node
- different nodes may have different versions of data

- after some time, updates propagate to all nodes

... Eventual Consistency

If different nodes have different versions of data

- conflicts arise, and need to be resolved (when noticed)
- need to decide which node has "the right value"

Levels of consistency (from Cassandra system)

- ONE: at least one node has committed change (weakest)
- QUORUM: at least half nodes holding data have committed
- ALL: changes propagated to all copies (strongest)

MapReduce

MapReduce is a *programming model*

- suited for use on large networks of computers
- processing large amounts of data with high parallelism
- originally developed by Google; Hadoop is open-source implementation

Computation is structured in two phases:

- **Map** phase:
  - master node partitions work into sub-problems
  - distributes them to worker nodes (who may further distribute)
- **Reduce** phase:
  - master collects results of sub-problems from workers
  - combines results to produce final answer

... MapReduce

MapReduce makes use of *(key,value)* pairs

- *key* values identify parts of computation

$Map(key_1,val_1) \rightarrow list(key_2,val_2)$

- applied in parallel to all (key_1,val_1) pairs
- results with common key_2 are collected in group for "reduction"

$Reduce(key_2,list(val_2)) \rightarrow val_3$

- collects all values tagged with key_2
- combines them to produce result(s) val_3

... MapReduce

"Classic" MapReduce example (word frequency in set of docs):

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
```

```
// partialCounts: list of aggregated partial counts
sum = 0
for each c in partialCounts:
  sum += c
emit (word, sum)
```

... MapReduce

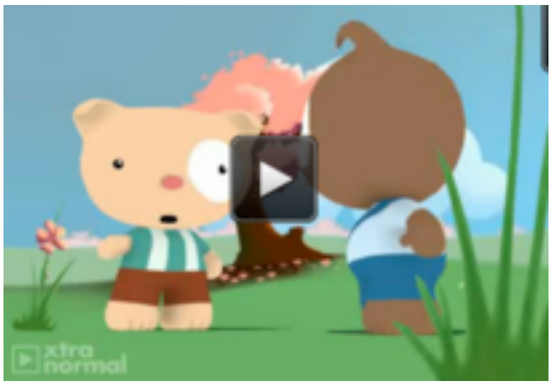
MapReduce as a "database language"

- some advocates of MapReduce have oversold it (replace SQL)
- DeWitt/Stonebraker criticised this
  - return to low-level model of data access
  - all done before in distributed DB research
  - misses efficiency opportunities afford by DBMSs
- concensus is emerging
  - SQL/MapReduce good for different kinds of task
  - MapReduce as a basis for SQL-like languages (e.g. Apache HiveQL)

Modern vs Classical

Some criticisms of the NoSQL approach:

- DeWitt/Stonebraker: *MapReduce: A major step backwards*
- Online parody of noSQL advocates (strong language warning)



Hadoop DFS

Apache Hadoop Distributed File System

- a hierarchical file system (directories & files a la Linux)
- designed to run on large number of commodity computing nodes
- supporting very large files (TB) distributed/replicated over nodes
- providing high reliability (failed nodes is the norm)

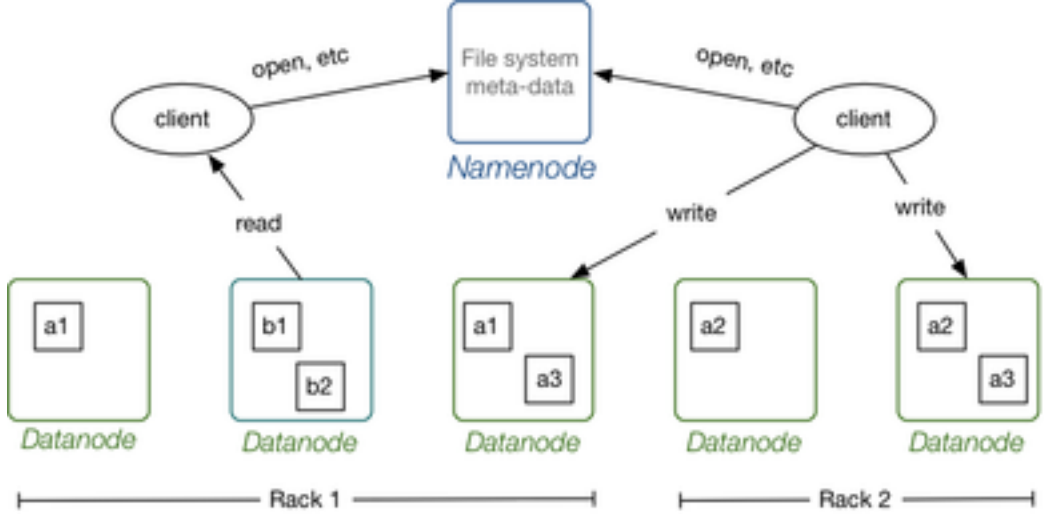
Provides support for Hadoop map/reduce implementation.

Optimised for write-once-read-many apps

- simplifies data coherence
- aim is maximum throughput rather than low latency

... Hadoop DFS

Architecture of one HDFS cluster:



... Hadoop DFS

Datanodes ...

- provide file read/write/append operations to clients
  - under instruction from Namenode
- periodically send reports to Namenode

A Hadoop *file*

- is a collection of fixed-size blocks
- blocks are distributed/replicated across nodes

Datanode → Namenode reports

- *Heartbeat* ... Datanode still functioning ok
- *Blockreport* ... list of all blocks on DataNode

... Hadoop DFS

Namenodes ...

- hold file-system meta-data (directory structure, file info)
  - e.g. file info: (filename, block#, #replicas, nodes)
  - e.g. (/data/a, 1, 2, {1,3}), (/data/a, 2, 2, {4,5}), (/data/a, 3, 2, {3,5})
- provides file open/close/rename operations to clients
- determine replication and mapping of data blocks to DataNodes
- select Datanodes to serve client requests for efficient access
  - e.g. node in local rack > node in other rack > remote node

Namenode knows file ok if all relevant Datanodes sent Bockreport

- if not ok, replicate blocks on other Datanodes & update meta-data

---

Two Case Studies

Consider two variations on the DBMS theme ...

Column Stores

- still based on the relational model
- but with a variation in how data is stored
- to address a range of modern query types

Graph Databases

- based on a graph model of data
- emphasising explicit representation of relationships

- relevant to a wide range of application domains

Column Stores

(Based on material by Daniel Abadi et al.)

Column Stores

Column-oriented Databases (CoDbs):

- are based on the relational model
- store data column-by-column rather that row-by-row
- leading to performance gains for analytical applications

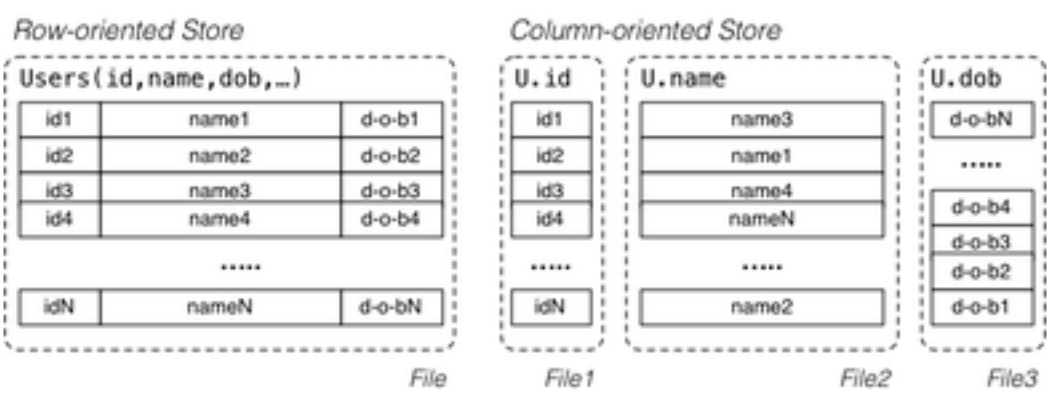
Ideas for CoDbs have been around since the 1970's

Rose to prominence via Daniel Abadi's PhD thesis (MIT, 2008)

Commercial systems have now been developed (e.g. Vertica)

... Column Stores

File structures for row-store vs column-store:



Values in individual columns are related by extra tuple id (cf. oid)

... Column Stores

Stored representation of logical (relational) tables

- each table is stored as a set of projections (slices)
- each projection consists of a different set of columns
- each column appears in at least one projection
- "rows" can be ordered differently in each projection

Example: Enrolment(course,student,term,mark,grade)

- $projection_1$: (course,student,grade) ordered by course
- $projection_2$: (term,student,mark) ordered by student
- $projection_3$: (course,student) ordered by course

Rows vs Columns

Workload for different operations

- *insert* requires more work in CoDbs
  - row: update one page;   column: update multiple pages
- *project* comes "for free" in CoDbs
  - row: extract fields from each tuple;   column: merge columns
- *select* may require less work in CoDbs
  - row: read whole tuples;   column: read just needed columns
- *join* may require less work in CoDbs
  - row: hash join;   column: scan columns for join attributes

---

... Rows vs Columns

Which is more efficient depends on mix of queries/updates

- RDBMSs are, to some extent, write-optimized
  - effective for OLTP applications (e.g. ATM, POS, ...)
- when RDBMSs might be better ...
  - when query requires all attributes
  - might read more data, but less seek-time (multiple files)
- when CoDbs might be better ...
  - smaller intermediate "tuples"
  - less competition for access to pages (locking)

... Rows vs Columns

Storing sorted columns leads to

- potential for effective compression
  - compression ⇒ more projections in same space
  - no need to compress all columns (if some aren't "compressible")
- sorted data is useful in some query evaluation contexts
  - e.g. terminating scan once unique match found
  - e.g. sort-merge join

Only one column in each projection will be sorted

- but if even one projection has a column sorted how you need ...

Query Evaluation in CoDbs

Projection is easy if one slice contains all required attributes.

If not ...

- sequential scan of relevant slices in parallel
- combine values at each iteration to form a tuple

Example: `select a,b,c from R(a,b,c,d,e)`

```
Assume: each column contains N values
for i in 0 .. N-1 {
   x = a[i]   // i'th value in slice containing a
   y = b[i]   // i'th value in slice containing b
   z = c[i]   // i'th value in slice containing c
   add (x,y,z) to Results
}
```

... Query Evaluation in CoDbs

If slices are sorted differently, more complicated

- scan based on tid values

- at each step, look up relevant entry in slice

Example: `select a,b,c from R(a,b,c,d,e)`

```
Assume: each column contains N values
for tid in 0 .. N-1 {
   x = fetch(a,tid)  // entry with tid in slice containing a
   y = fetch(b,tid)  // entry with tid in slice containing b
   z = fetch(c,tid)  // entry with tid in slice containing c
   add (x,y,z) to Results
}
```

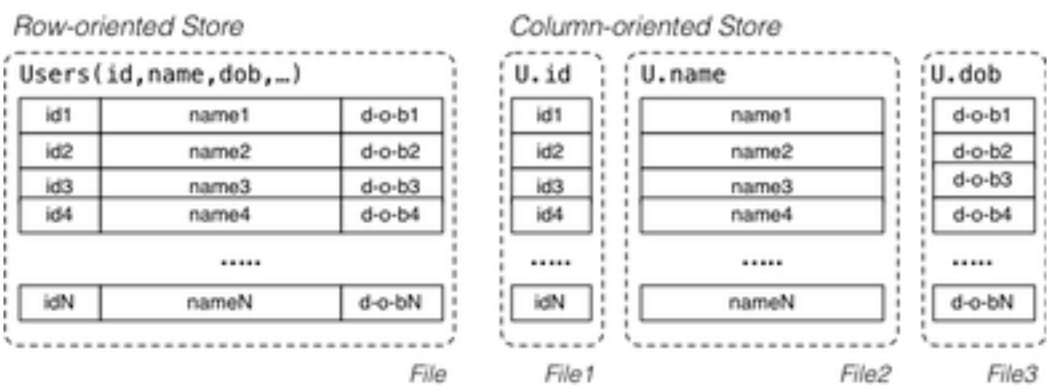
Potentially slow, depending on how `fetch()` works.

... Query Evaluation in CoDbs

113/147

For remaining discussion, assume

- each slice has 1 attribute, and a[i].tid = b[i].tid = c[i].tid



... Query Evaluation in CoDbs

114/147

Consider typical multi-attribute SQL query

```
select a,b,c from R where b > 10 and d < 7
```

Query operation on individual column is done in one slice

Mark index of each matching entry in a bit-vector

Combine (AND) bit-vectors to get indexes for result entries

For each index, merge result entry columns into result tuple

Known as *late materialization*.

... Query Evaluation in CoDbs

115/147

Example: `select a,b,c from R where b = 5`

```
// Assume: each column contains N values
matches = all-zero bit-string of length N
for i in 0 .. N-1 {
   x = b[i]  // i'th value in b column
   if (x == 5)
      matches[i] = 1  // set bit i in matches
}
for i in 0 .. N-1 {
   if (matches[i] == 0) continue
   add (a[i], b[i], c[i]) to Results
```

}

Fast sequential scanning of small (compressed?) data

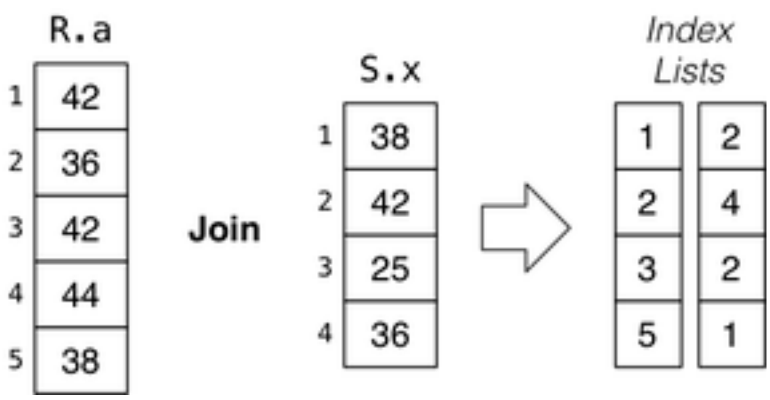
... Query Evaluation in CoDbs

Example: `select a,b,c from R where b>10 and d<7`

```
// Assume: each column contains N values
matches1 = all-zero bit-string of length N
matches2 = all-zero bit-string of length N
for i in 0 .. N-1 {
   if (b[i] > 10) matches1[i] = 1
   if (d[i] < 7) matches2[i] = 1
}
matches = matches1 AND matches2
for i in 0 .. N-1 {
   if (matches[i] == 0) continue
   add (a[i], b[i], c[i]) to Results
}
```

... Query Evaluation in CoDbs

Join on columns, set up for late materialization



Note: the left result column is always sorted

... Query Evaluation in CoDbs

Example:
```
select R.a, S.b
from   R join S on R.a = S.x
```

```
// Assume: N tuples in R, M tuples in S
for i in 0 .. N-1 {
   for j in 0 .. M-1 {
      if (a[i] == x[j])
         append (i,j) to IndexList
   }
}
for each (i,j) in IndexList {
   add (a_R[i], b_S[j]) to Results
}
```

... Query Evaluation in CoDbs

Aggregation generally involves a single column

- multiple aggregations could be carried out in parallel

E.g.

```
select avg(mark), count(student) from Enrolments
```

Operations involving groups of columns

- may require early materialization ⇒ slower

Graph Databases

(Based on material by Markus Krotzsch, Renzo Angles, Claudio Gutierrez)

Graph Databases

Graph Databases (GDbs):

- DBMSs that use *graphs* as the data model

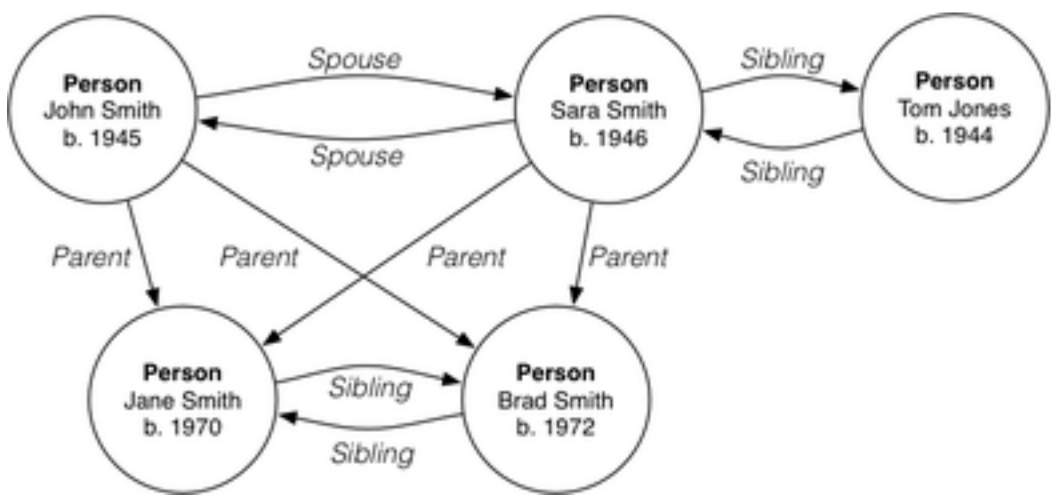
But what kind of "graphs"?

- all graphs have nodes and edges, but are they ...
- directed or undirected, labelled or unlabelled?
- what kinds of labels? what datatypes?
- one graph or multiple graphs in each database?

Two major GDb data models: RDF, Property Graph

... Graph Databases

Typical graph modelled by a GDb



Graph Data Models

RDF = Resource Description Framework

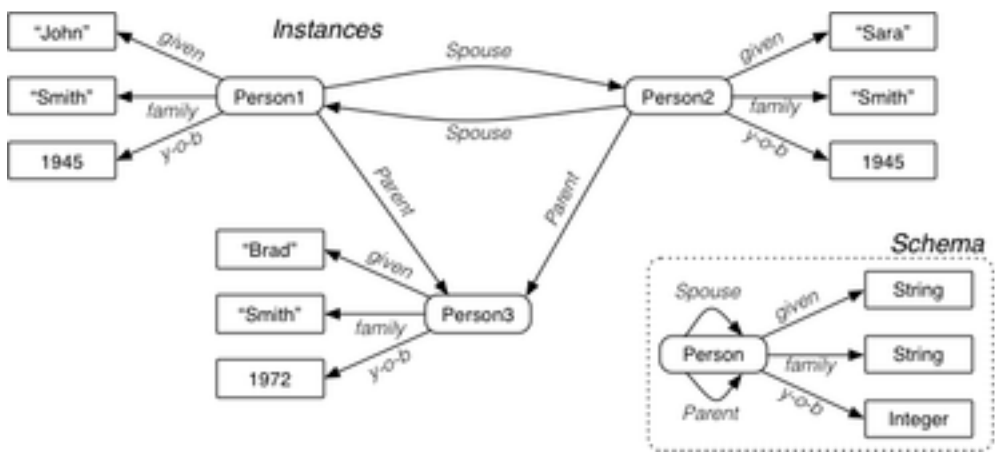
- directed, labelled graphs
- nodes have identifiers (constant values, incl. URIs)
- edges are labelled with the relationship
- can have multiple edges between nodes (diff. labels)
- can store multiple graphs in one database
- datatypes based on W3C XML Schema datatypes

Data as triples, e.g. <Person1,given,"John">, <Person1,parent,Person3>

RDF is a W3C standard; supported in many prog. languages

... Graph Data Models

RDF model of part of earlier graph:



... Graph Data Models

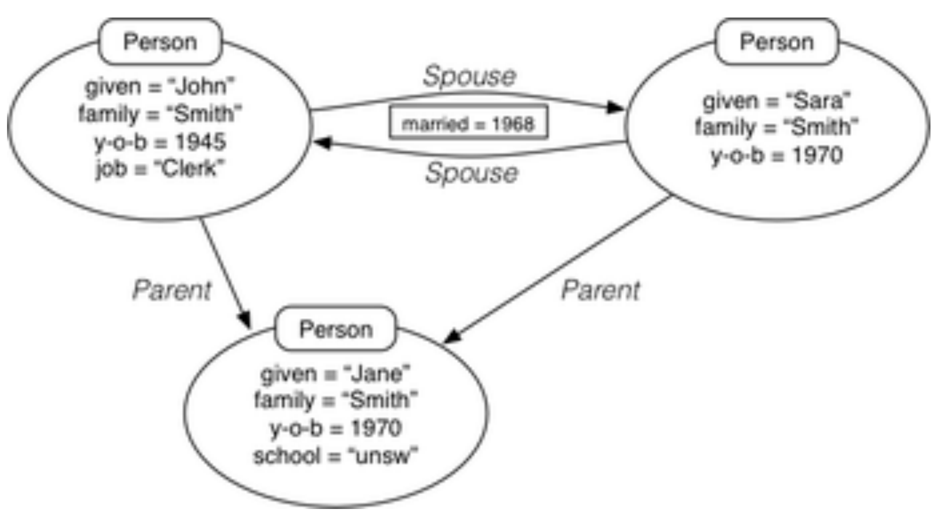
Property Graph

- directed, labelled graphs
- properties are (key/label, value) pairs
- nodes and edges are associated with a list of properties
- can have multiple edges between nodes (incl same labels)

Not a standard like RDF, so variations exist

... Graph Data Models

Property Graph model of part of earlier graph:



GDb Queries

Graph data models require a graph-oriented query framework

Types of queries in GDbs

- node properties   (like SQL `where` clauses)
  - e.g.   is there a Person called John?   how old is John?
- adjacency queries
  - e.g. is John the parent of Jane?
- reachability queries
  - e.g. is William one of John's ancestors?
- summarization queries   (like SQL aggregates)
  - e.g. how many generations between William and John?

... GDb Queries

Graphs contain arbitrary-length paths

Need an expression mechanism for describing such paths

- path expressions are regular expressions involving edge labels
- e.g. `L*` is a sequence of one or more connected `L` edges

GDb query languages:

- SPARQL = based on the RDF model   (widely available via RDF)
- Cypher = based on the Property Graph model   (used in Neo4j)

Example Graph Queries

Example: Persons whose first name is James

SPARQL:

```
PREFIX p: <http://www.people.org>
SELECT ?X
WHERE { ?X p:given "James" }
```

Cypher:

```
MATCH (person:Person)
WHERE person.given="James"
RETURN person
```

... Example Graph Queries

Example: Persons born between 1965 and 1975

SPARQL:

```
PREFIX p: <http://www.people.org/>
SELECT ?X
WHERE {
 ?X p:type p:Person . ?X p:y-o-b ?A .
 FILTER (?A ≥ 1965 && ?A ≤ 1975)
 }
```

Cypher:

```
MATCH (person:Person)
WHERE person.y-o-b ≥ 1965 and person.y-o-b ≤ 1975
RETURN person
```

... Example Graph Queries

Example: pairs of Persons related by the "parent" relationship

SPARQL:

```
PREFIX p: <http://www.people.org/>
SELECT ?X ?Y
WHERE { ?X p:parent ?Y }
```

Cypher:

```
MATCH (person1:Person)-[:parent]->(person2:Person)
```

```
RETURN person1, person2
```

... Example Graph Queries

Example: Given names of people with a sibling called "Tom"

SPARQL:

```
PREFIX p: <http://www.people.org/>
SELECT ?N
WHERE { ?X p:type p:Person . ?X p:given ?N .
        ?X p:sibling ?Y . ?Y p:given "Tom"   }
```

Cypher:

```
MATCH (person:Person)-[:sibling]-(tom:Person)
WHERE tom.given="Tom"
RETURN person.given
```

... Example Graph Queries

Example: All of James' ancestors

SPARQL:

```
PREFIX p: <http://www.socialnetwork.org/>
SELECT ?Y
WHERE { ?X p:type p:Person . ?X p:given "James" .
        ?Y p:parent* ?X }
```

Cypher:

```
MATCH (ancestor:Person)-[:parent*]->(james:Person)
WHERE james.given="James"
RETURN DISTINCT ancestor
```

Course Review + Exam

Syllabus

View of DBMS internals from the bottom-up:

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

Exam

Thursday 7 May, 11:00am - 8 May 1:00am

Held in the comfort of your own home.

All answers are typed and submitted on-line.

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

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

... Exam

Resources available during exam:

- exam questions (collection of web pages)
- 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 other students.

... Exam

Tools available during the exam on the CSE servers

- C compiler (`gcc`, `make`)
- text editors (e.g. `vim`, `emacs`, `gedit`, ...)
- 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`)

... Exam

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`)

Before the exam ...

Practice with the Sample Exam

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`

During the exam ...

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)

If there are technical difficulties with the CSE servers/Webcms3

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

If you have technical difficulties with your machine

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

What's on the Exam?

Potential topics to be examined ...

- ~~A - Course Introduction, DBMS Revision,~~ PostgreSQL
- B - Storage: Devices, Files, Pages, Tuples, Buffers, Catalogs
- C - Cost Models, Implementing Scan, Sort, Projection
- D - Implementing Selection on One Attribute
- E - Implementing Selection on Multiple Attributes
- F - Similarity-based Selection (only first 15 slides)
- G - Implementing Join
- H - Query Translation, Optimisation, Execution
- I - Transactions, Concurrency, Recovery
- ~~J - Non-classical DBMSs~~

... What's on the Exam?

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

- 2 x C programming questions (40%)
- 6 x written answer questions (60%)

Reminder:

- exam contributes 60% of final mark
- hurdle requirement: must score > 24/60 on exam

Special Consideration

Reminder: this is a one-chance exam.

- attempting the Exam is treated as "I am fit and well"
- 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

Pre-exam consultations leading up to exam (see course web site)

And that's all folks ...

End of COMP9315 20T1 Lectures

Good luck with the exam ...

And keep on using PostgreSQL ...
