Implementing Durability

- Atomicity/Durability
- Durability
- Dealing with Transactions
- Architecture for Atomicity/Durability
- Execution of Transactions
- Transactions and Buffer Pool

❖ 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

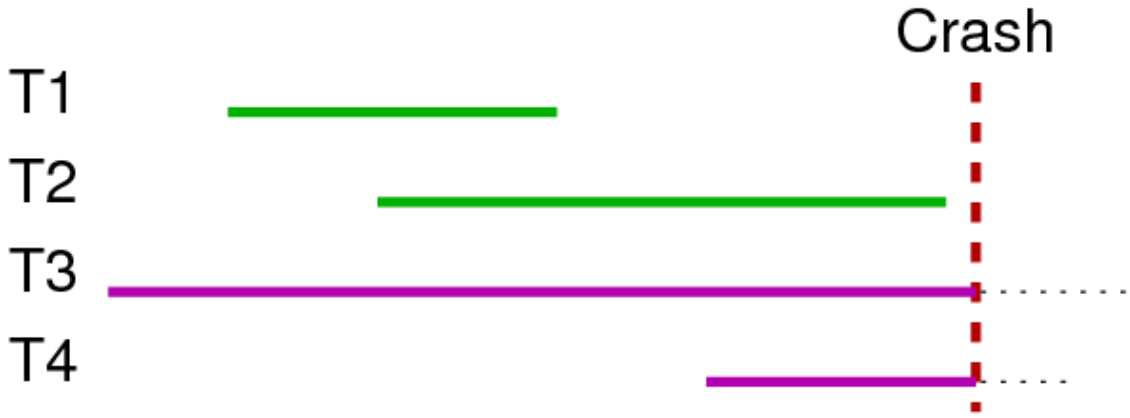
What kinds of "system failures" do we need to deal with?

- 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 (cont)

Consider following scenario:



Desired behaviour after system restart:

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

❖ Durability (cont)

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

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

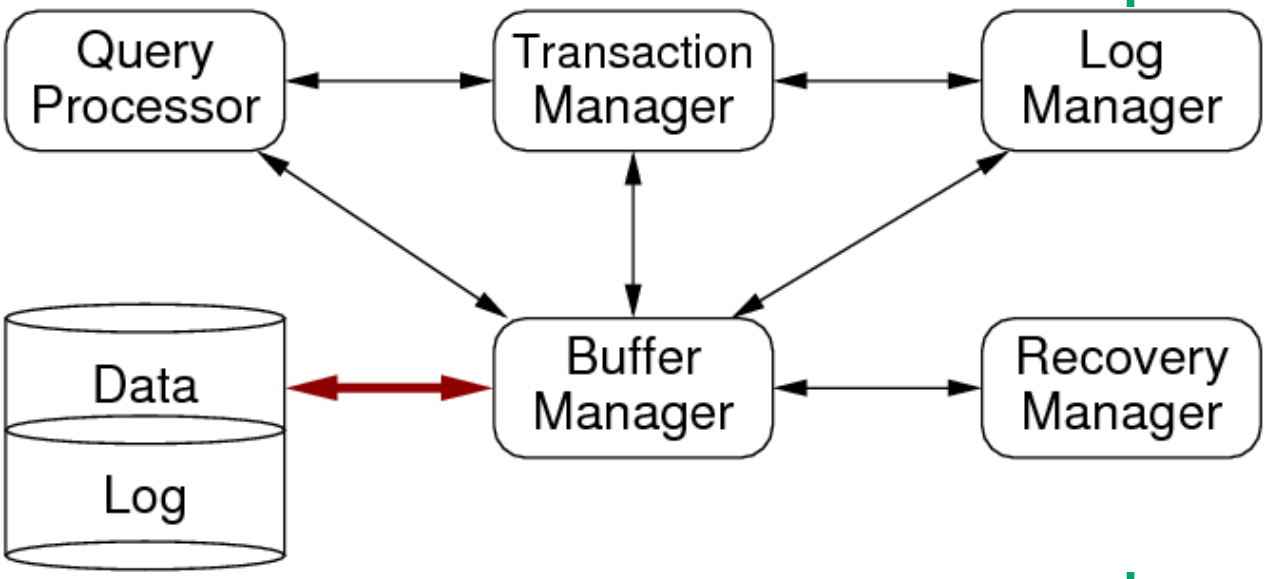
- failure of DBMS processes or operating system
- failure of transactions (**`ABORT`**)

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

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)

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 (cont)

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 (cont)

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)	OUTPUT(A)	6	16	6	16	5
(8)	OUTPUT(B)	6	16	6	16	6

After tx completes, we must have either Disk(A)=8, Disk(B)=5 or Disk(A)=16, Disk(B)=6

If system crashes before (8), may need to undo disk changes.
If system crashes after (8), 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, 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()`

Implementing Recovery

- Recovery
- Logging
- Undo Logging
- Checkpointing
- Redo Logging
- Undo/Redo Logging
- Recovery in PostgreSQL

❖ 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**

❖ Undo Logging (cont)

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 (cont)

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 (cont)

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

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 (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 log records indicating changes
3. then commit log record (**OUTPUT**)
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**.

❖ Redo Logging (cont)

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 (cont)

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

Database Trends

- Future of Database
- Large Data
- Information Retrieval
- Multimedia Data
- Uncertainty
- Stream Data Management Systems
- Graph Data
- Dispersed Databases

❖ 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 (cont)

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 (cont)

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 (cont)

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

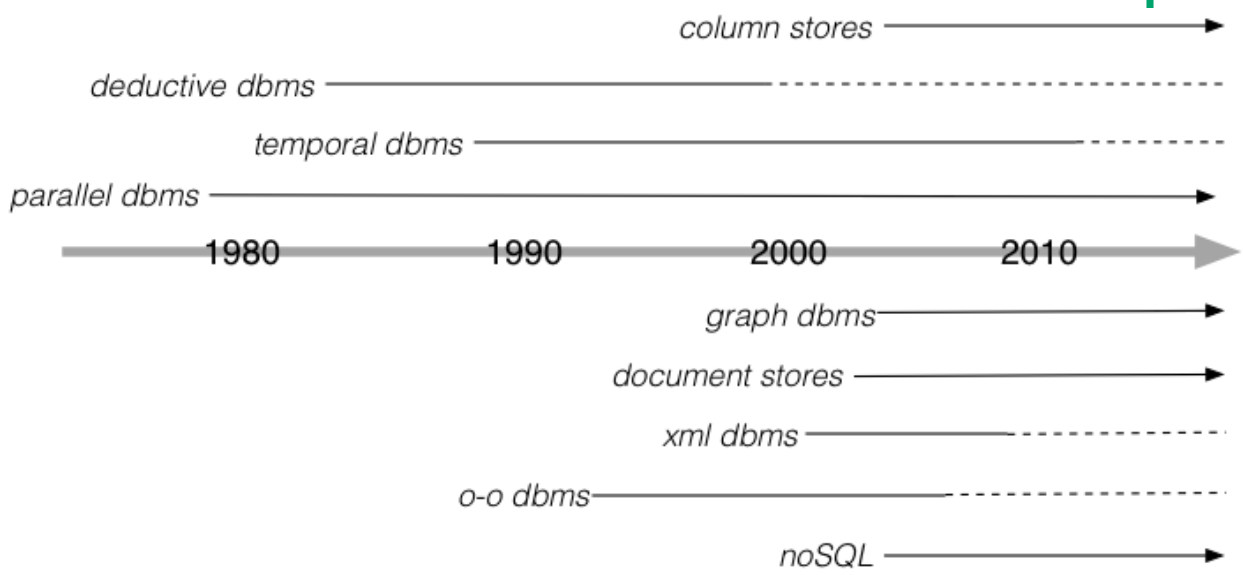
❖ Future of Database (cont)

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 (cont)

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 (cont)

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 mapping)
- 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)

❖ 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:

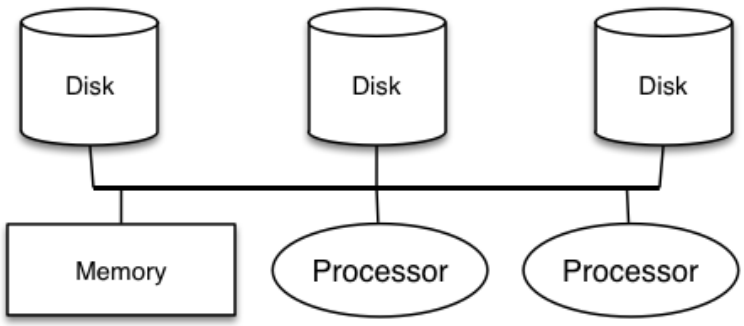
- grid and cloud computing
- database management for mobile devices

Parallelism in Databases

❖ Parallel DBMSs

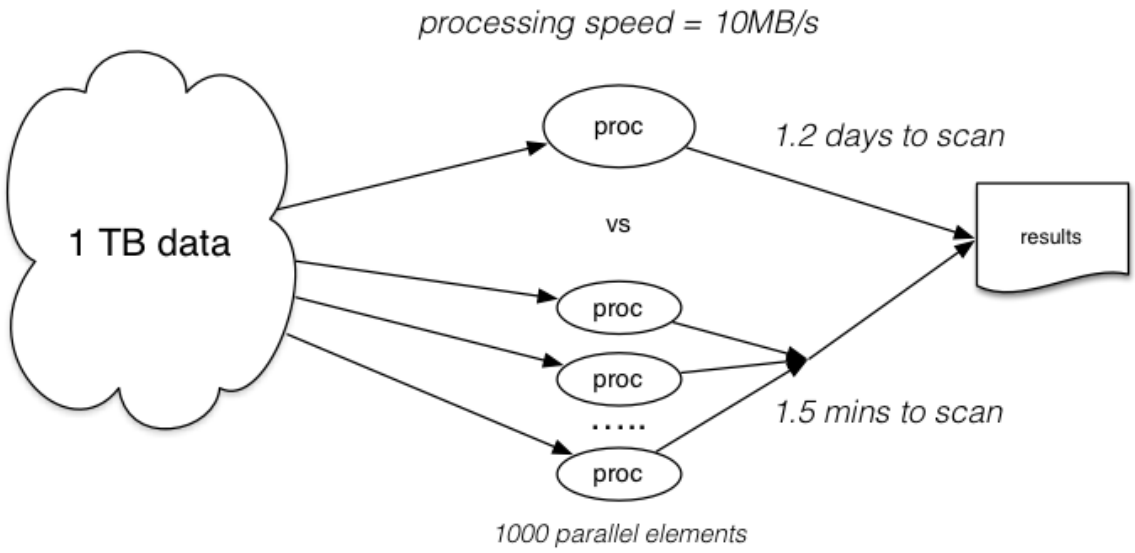
RDBMS discussion so far has revolved around systems

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



❖ Parallel DBMSs (cont)

Why parallelism? ... Throughput!



❖ Parallel DBMSs (cont)

DBMSs are a success story in application of parallelism

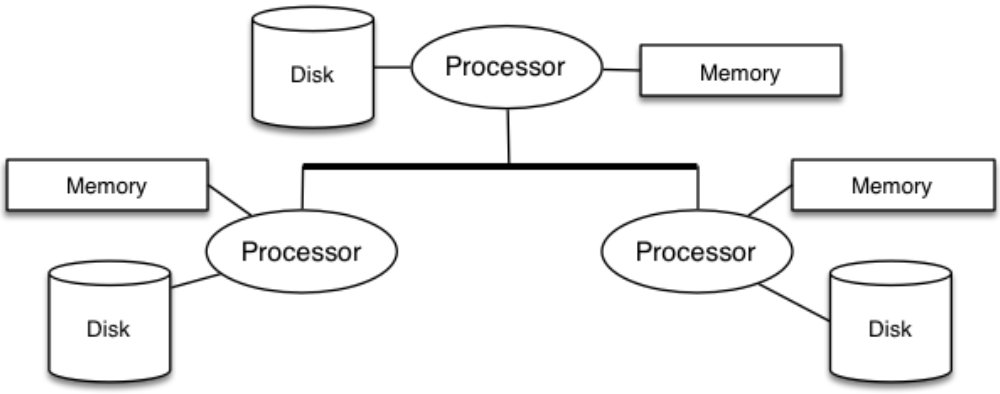
- can process many data elements (tuples) at the same time
- can create pipelines of query evaluation steps
- don't require special hardware
- can hide paralleism within the query evaluator
  - application programmers don't need to change habits

Compare this with effort to do parallel programming.

❖ Parallel Architectures

Types: shared memory, shared disk, shared nothing

Example shared-nothing architecture:

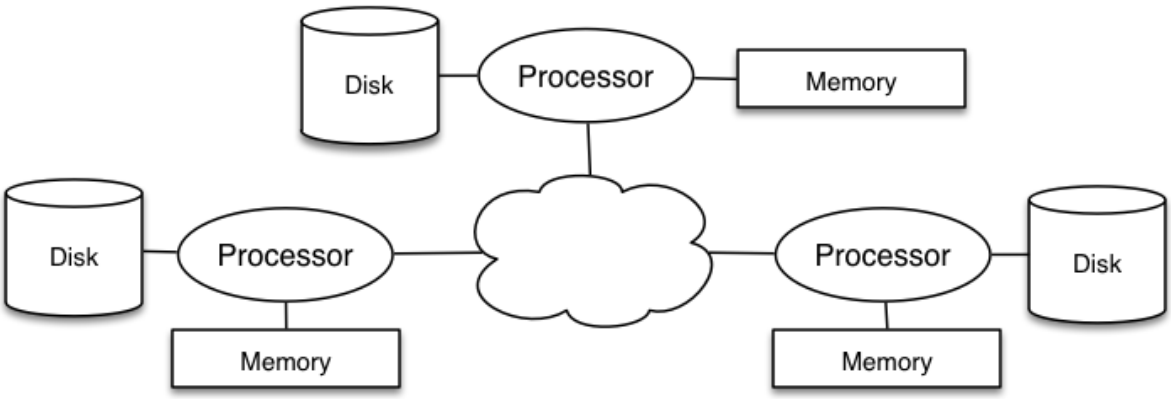


Typically same room/LAN (data transfer cost ~ 100's of μsecs .. msecs)

❖ 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) (cont)

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

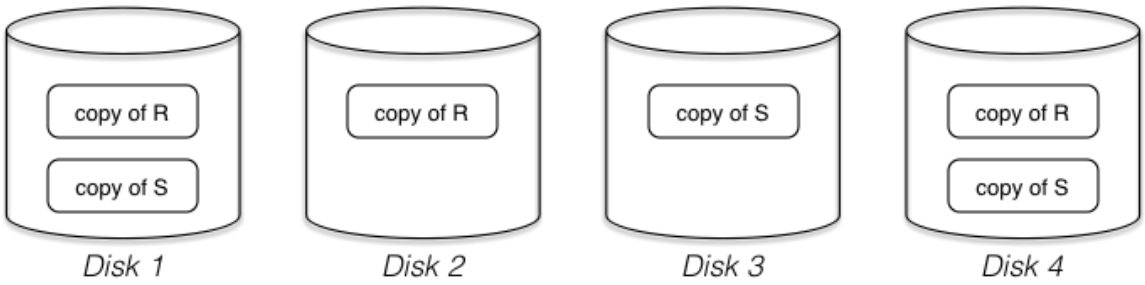
❖ Data Storage in PDBs

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

Can distribute data across multiple storage devices

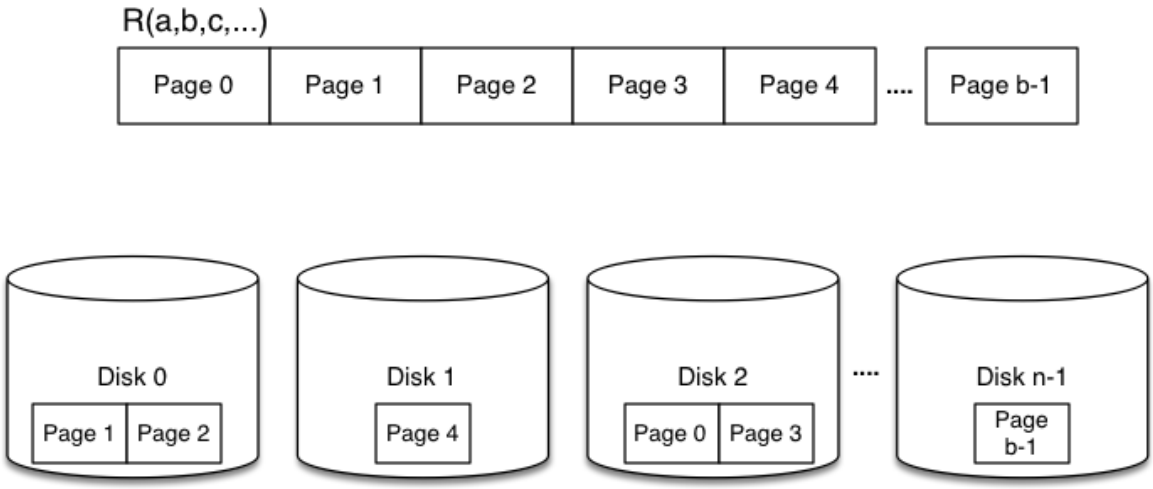
- duplicate all pages from a relation (replication)
- store some pages on one store, some on others (partitioning)

Replication example:



❖ Data Storage in PDBs (cont)

Data-partitioning example:



❖ Data Storage in PDBs (cont)

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 (cont)

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 (cont)

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

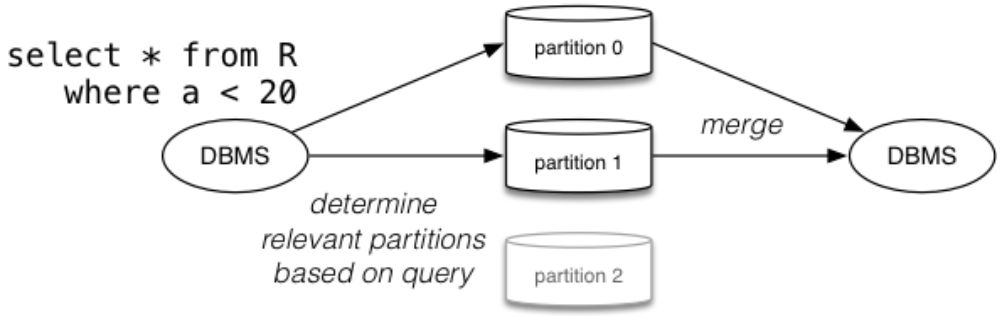
- intra-operator parallelism
  - get all machines working to compute a given operation
    (scan, sort, join)
- inter-operator parallelism
  - each operator runs concurrently on a different processor
    (exploits pipelining)
- Inter-query parallelism
  - different queries run on different processors

❖ Parallelism in DB Operations (cont)

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 (cont)

Parallel sorting

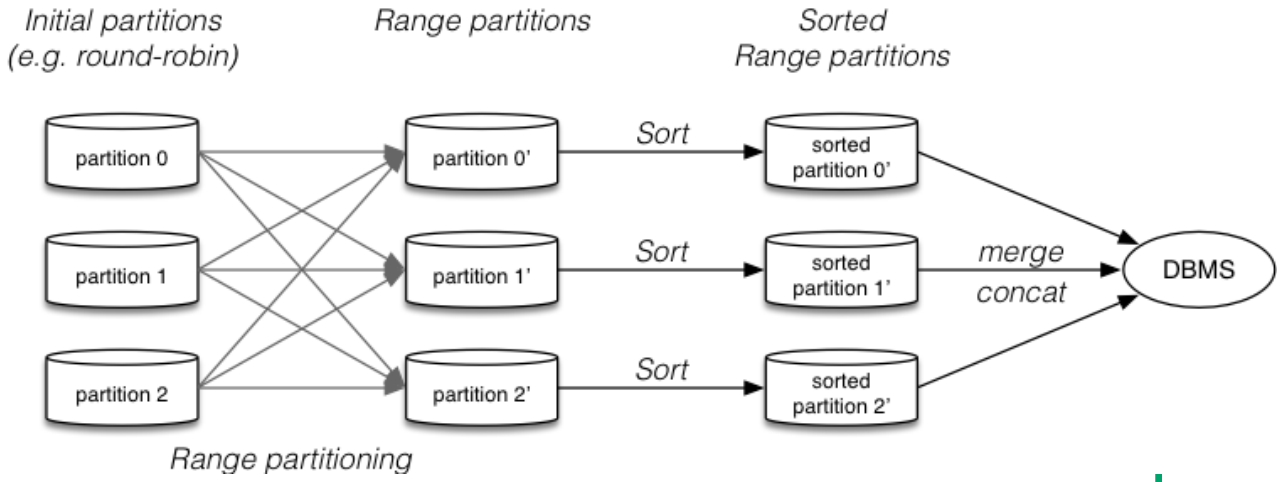
- scan in parallel, range-partition during scan
- pipeline into local sort on each processor
- merge sorted partitions in order

Potential problem:

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

❖ Parallelism in DB Operations (cont)

Parallel sort:



❖ Parallelism in DB Operations (cont)

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 (cont)

Parallel hash join

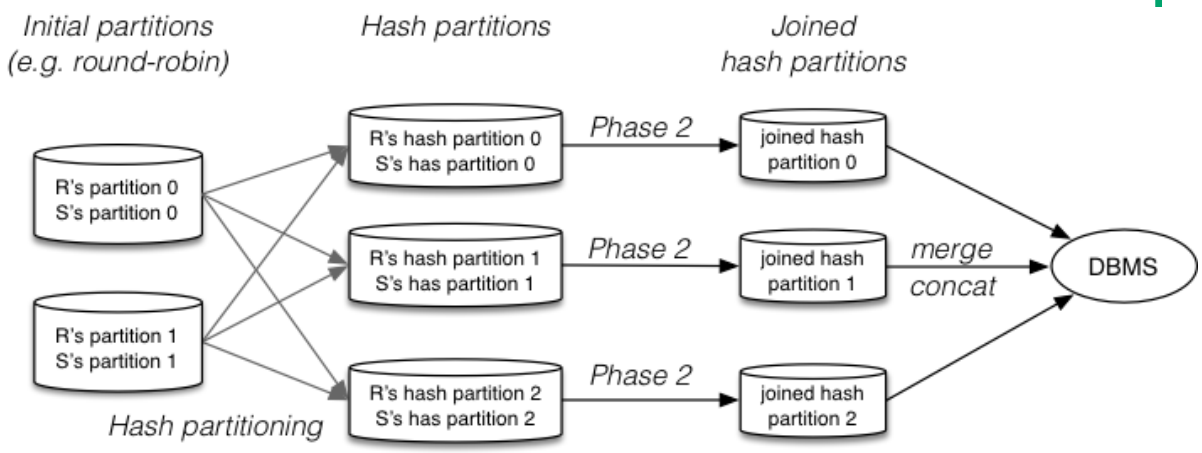
- distribute partitions to different processors
- partition 0 of R goes to same node as partition 0 of S
- join phase can be done in parallel on each processor
- then results need to be merged
- very effective for equijoin

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 (cont)

Parallel hash join:



Graph Databases

❖ Graph Databases

Graph Databases (GDbs):

- DBMSs that use graphs as the data model

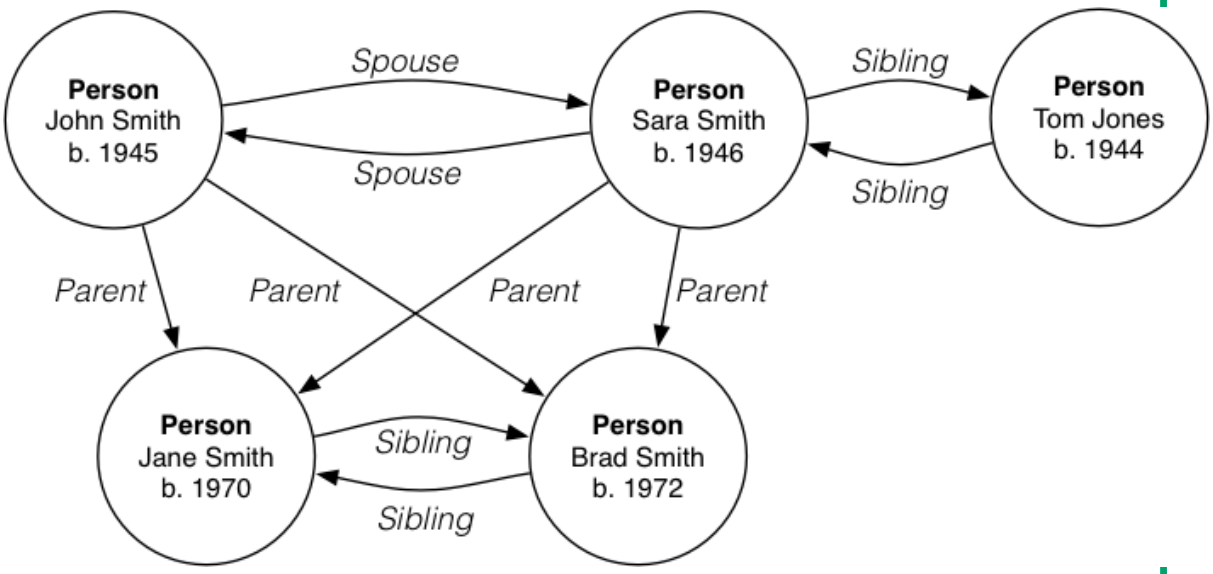
But what kind of "graphs"?

- 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 (cont)

Typical graph modelled by a GDb



❖ Graph Data Models

RDF = Resource Description Framework

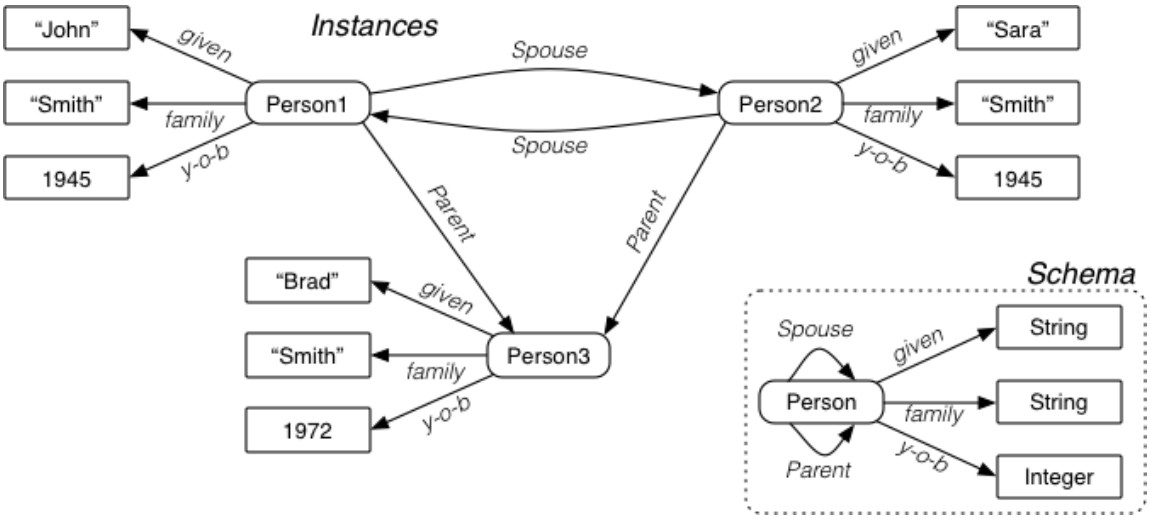
- directed, labelled graphs
- nodes have identifiers (constant values, incl. URIs)
- edges are labelled with the relationship
- can have multiple edges between nodes (diff. labels)
- can store multiple graphs in one database
- datatypes based on W3C XML Schema datatypes

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

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

❖ Graph Data Models (cont)

RDF model of part of earlier graph:



❖ Graph Data Models (cont)

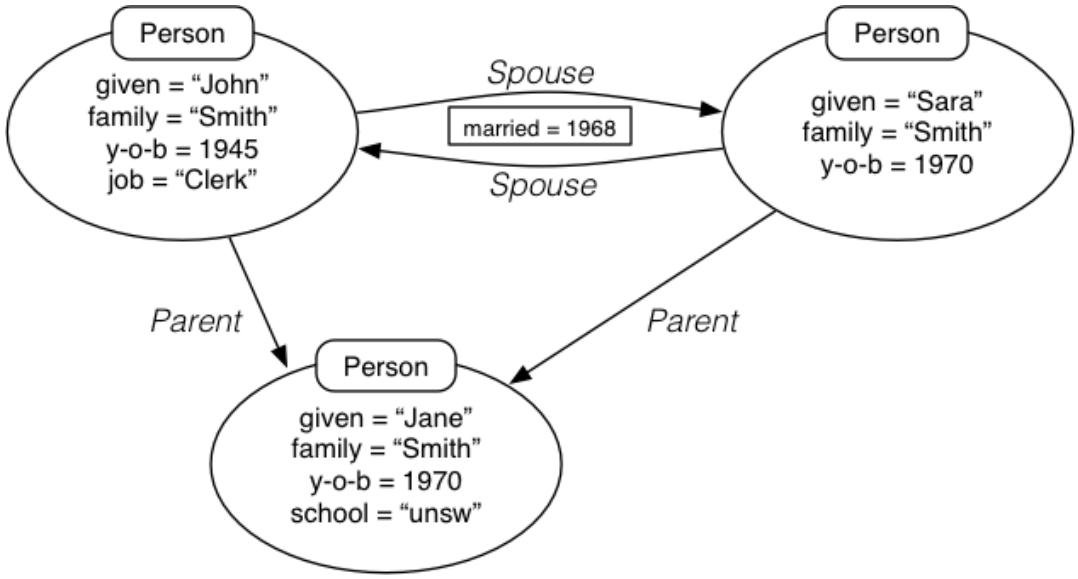
Property Graph

- 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 (cont)

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 (cont)

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 (cont)

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 (cont)

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 (cont)

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 (cont)

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
```