# COMP9315 Introduction

## COMP9315 DBMS Implementation

( Data structures and algorithms inside relational DBMSs )

Lecturer: **John Shepherd**

Web Site: http://www.cse.unsw.edu.au/~cs9315/

(If WebCMS unavailable, use http://www.cse.unsw.edu.au/~cs9315/20T1/)

## Lecturer

| | |
|---|---|
| Name: | John Shepherd |
| Office: | K17-410 (turn right from lift) |
| Phone: | 9385 6494 |
| Email: | jas@cse.unsw.edu.au |
| Consults: | still working out the details |
| Research: | Information Extraction/Integration |
| | Information Retrieval/Web Search |
| | e-Learning Technologies |
| | Multimedia Databases |
| | Query Processing |

## Course Admin

| | |
|---|---|
| Email: | cs9315@cse.unsw.edu.au |
| Reasons: | Enrolment problems |
| | Special consideration |
| | Detailed assignment questions |
| | Technical issues |

## What this Course is NOT

Official course title: *Database Systems Implementation*

More accurate: *Implementation of Database Engines*

This is a course about

- the internal workings of database management systems
- their data structures, algorithms, techniques

It is **not** a course about

- how to be a database administrator, or
- how to build (advanced) database applications

---

## Course Goals

Introduce you to:

- architecture of relational DBMSs (e.g. PostgreSQL)
- algorithms/data-structures for data-intensive computing
- representation of relational database objects
- implementation of relational operators (sel,proj,join)
- techniques for processing SQL queries
- techniques for managing concurrent transactions
- ?concepts in distributed and non-relational databases?

Develop skills in:

- analysing the performance of data-intensive algorithms
- the use of C to implement data-intensive algorithms

---

### ... Course Goals

A major course goal is to give you exposure to:

- the inner workings of a complete RDBMS (**PostgreSQL**)
- a large software system (and accompanying apparatus)

Concepts will also be illustrated via their PostgreSQL implementation.

PostgreSQL is a good vehicle for this purpose, because:

- open-source, unlike Oracle, SQL-server, etc.
- good-quality, consistent code base, unlike MySQL, etc.

---

### ... Course Goals

At the end of this course you should understand:

- internal structure and functioning of relational DBMSs
- how SQL queries are translated/optimised/evaluated
- techniques for implementing transactions and reliable storage

At the end of this course you should be able to:

- analyse the cost/tradeoffs of relational operation implementations
- select appropriate tools to solve data-intensive computing problems
- administer (install/tune/manage) a PostgreSQL installation
- (maybe) contribute modifications to the PostgreSQL project
- (maybe) build a relational database management system "from scratch"

---

## Pre-requisites

We assume that you are already familiar with

- the C language and programming in C (or C++)
  (e.g. completed ≥ 1 programming course in C)

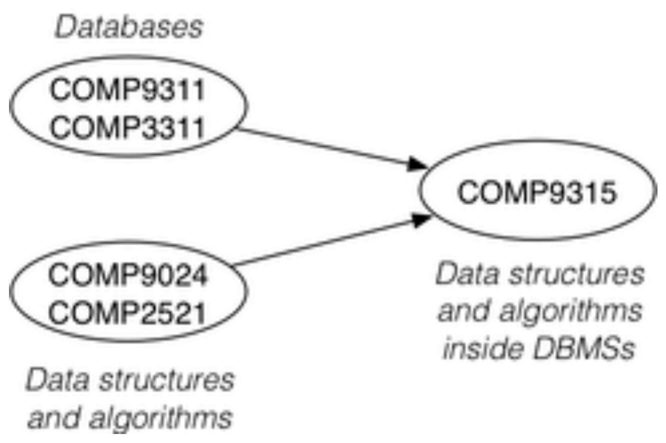
- developing applications on RDBMSs
  (SQL, relational algebra   e.g. an intro DB course)
- basic ideas about file organisation and file manipulation
  (e.g. Unix `open, close, lseek, read, write, flock`)
- sorting algorithms, data structures for searching
  (sorting, trees, hashing   e.g. a data structures course)

If you don't know this material very well, don't take this course.

---

**... Pre-requisites**

How you might meet the pre-reqs ...


Databases
COMP9311
COMP3311
COMP9315
COMP9024
COMP2521
Data structures and algorithms inside DBMSs
Data structures and algorithms

... via courses at CSE.

---

## Syllabus Overview

1. Relational DBMS Architecture
   - including details of the PostgreSQL architecture (case study)
2. Storage Management
   - disks, file organisation, buffer pool management
3. Relation Alegbra Operations
   - implementation of selection, projection, join, sort, ...
4. Query Processing
   - mapping SQL to query plans, query plan optimisation
5. Transaction Processing
   - concurrency, locking, crash recovery
6. Parallel and Distributed Databases
   - moving beyond one data store
7. Object Data, Document Data, Graph Data
   - beyond relations and tuples

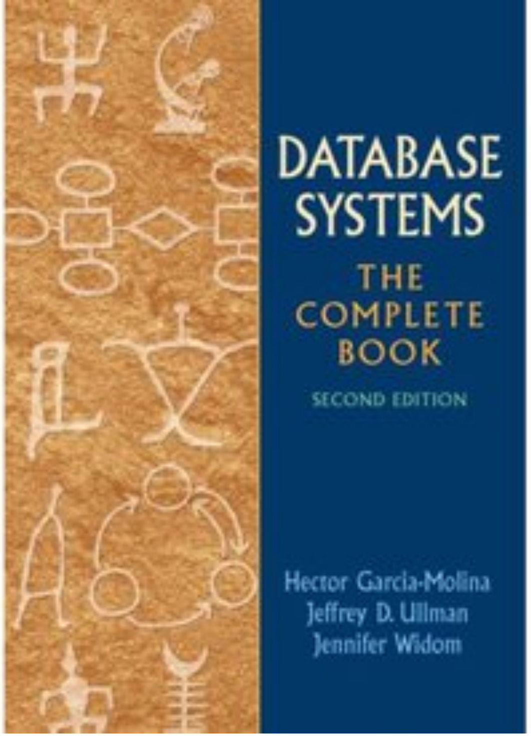
---

## Textbooks

No official text book; several are suitable ...

- Garcia-Molina, Ullman, Widom
  "Database Systems: The Complete Book"
- Ramakrishnan, Gehrke
  "Database Systems Management"
- Silberschatz, Korth, Sudarshan
  "Database System Concepts"
- Kifer, Bernstein, Lewis
  "Database Systems: An algorithmic-oriented approach"
- Elmasri, Navathe
  "Database Systems: Models, languages, design ..."

but not all cover all topics in detail



## Teaching/Learning

What's available for you:

- Textbooks: describe syllabus in lots of detail
- Notes: describe syllabus topics in some detail
- Lectures: summarise Notes and contain examples/exercises
- Lecture videos: for review (or if you miss a lecture, or are in WEB stream)
- Readings: research papers on selected topics

The onus is on *you* to use all of this material.

Note: Lecture slides, exercises and videos will be available only *after* the lecture.

---

### ... Teaching/Learning

Things that you need to **do**:

- Exercises: tutorial-like questions
- Prac work: lab-class-like exercises
- Assignments: large/important practical exercises
- On-line quizzes: for self-assessment

Dependencies:

- Exercises → Exam (theory part)
- Prac work → Assignments → Exam (prac part)

---

### ... Teaching/Learning

Scheduled classes?

- there are **no** tute classes or lab classes
- lectures are thus more important than usual

What to do if you have problems understanding stuff?

- ask a question in/after the lecture
- come to a *consultation* (M, T, W, F)
- ask about it on the *MessageBoard* (under WebCMS)
- send me email (essential for detailed assignment questions)

Debugging is most easily done in person.

---

## ... Teaching/Learning

The course web site site is where you can:

- find out the latest course news/announcements
- collect the course notes, and view lecture slides/videos
- get all of the information about theory/prac exercises
- get your questions answered (via the MessageBoard)

URL: `http://www.cse.unsw.edu.au/~cs9315/`

If WebCMS is ever down, most material is accessible via:

`http://www.cse.unsw.edu.au/~cs9315/20T1/index.php`

---

## Prac Work

Prac Work requires you to compile PostgreSQL from source code

- instructions explain how to do this on Linux at CSE
- also works easily on Linux and Mac OSX at home
- PostgreSQL docs describe how to compile for Windows

Make sure you do the first Prac Exercise when it becomes available.

Sort out any problems ASAP (preferably at a consultation).

You can do prac work in groups, if you wish.

---

## Assignments

Schedule of assignment work:

| Ass | Description | Due | Marks |
|---|---|---|---|
| 1 | Storage Management | Week 5 | 10% |
| 2 | Query Processing | Week 11 | 15% |

Assignments will be carried out either individually or in pairs (see WebCMS)

Choose your own online tools to share code (e.g. git,DropBox).

Ultimately, submission is via CSE's `give` system.

Will spend some time in lectures reviewing assignments.

Assignments will require up-front code-reading (see Pracs).

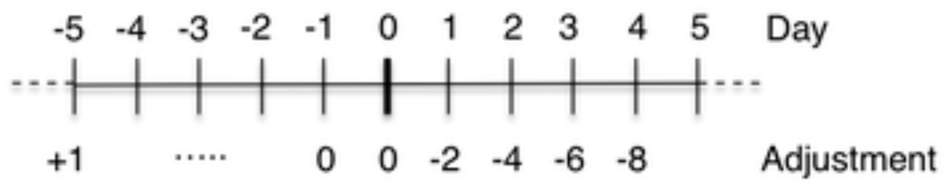
---

## ... Assignments

Don't leave assignments to the last minute; you *can* submit early.

As a "carrot", bonus marks are available for early submissions.

As a "stick", marks deducted (from max) for late submissions.



## ... Assignments

More comments on assignments ...

*You* are responsible for managing your own time, and for committing enough time to complete your work in this course.

"Work pressure" is not an acceptable excuse for late assignments.

Plagiarism will be checked for and punished.

Slacking off and letting your partner do the work is unhelpful.

The exam will contain questions related to the assignment work.

## Quizzes

Over the course of the semester ...

- six online quizzes
- taken in your own time (but there are deadlines)
- each quiz is worth a small number of marks

Quizzes are primarily a review tool to check progress.

But they contribute 15% of your overall mark for the course.

## Exam

There will be a three-hour exam in the August exam period.

Exam is held in CSE Labs (learn the environment, VLab)

The Course Notes (only) will be available in the exam.

Things that we can't reasonably test in the exam:

- writing *large* C programs, running *major* experiments, drawing diagrams

Everything else is potentially examinable.

The exam will be a mixture of descriptive questions, quantitative analysis, and small programming exercises.

The exam contributes 65% of the overall mark for this course.

## Supplementary Assessment Policy

Everyone gets **exactly one chance** to pass the Exam.

If you attend the Exam

- I assume that you are fit/healthy enough to take it
- you won't get a 2nd chance, even with a medical certificate

If you're sick just before or on the day of the Exam

- do *not* attend the Exam
- get documentation to support your claim

---

### ... Supplementary Assessment Policy

All Special Consideration requests:

- must *document* how *you* were affected
- must be submitted to Student Central (useful to email me as well)

Supplementary Exams are in mid-December (near Xmas)

- UNSW requires you to be available at that time, if needed
- if granted a Supp and don't attend, your exam mark = 0

Excuses like "I have already bought a plane ticket home" are not acceptable.

---

## Passing this Course

There is only one way to pass this course:

- *learn* the material
    - by listening in lectures and reading the Notes
    - by doing the exercises and prac work
- *perform* in the assignments/exam

You are assessed based on your *demonstrated* competence.

Pleading for a pass on compassionate grounds won't work.

---

## Assessment Summary

Your final mark/grade will be computed according to the following:

```
ass1    = mark for assignment 1      (out of 10)
ass2    = mark for assignment 2      (out of 15)
quiz    = mark for on-line quizzes   (out of 15)
exam    = mark for final exam        (out of 60)
okExam = exam > 24/60            (after scaling)

mark    = ass1 + ass2 + quiz + exam
grade   = HD|DN|CR|PS,  if mark ≥ 50 && okExam
        = FL,           if mark < 50 && okExam
        = UF,           if !okExam
```

---

## Reading Material

All of these textbooks have relevant material (to varying depths):

- Elmasri, Navathe,
  Fundamentals of Database Systems (6th ed),
  Addison-Wesley, 2010. (General DB book, conventional approach)
- Garcia-Molina, Ullman, Widom,
  Database Systems: The Complete Book (2nd ed),
  Prentice Hall, 2009. (General DB book, slightly more formal view)
- Silberschatz, Korth, Sudarshan,
  Database System Concepts (6th ed),

McGraw-Hill, 2010. (General DB book, conventional approach)

- Kifer, Bernstein, Lewis,
  Database Systems: Application-Oriented Approach (2nd ed)
  Addison-Wesley, 2006. (General DB book, application and transaction focus)
- Ramakrishan, Gehrke,
  Database Management Systems (3rd ed),
  McGraw-Hill, 2002. (General DB book, emphasises "systems" aspects)

---

## ... Reading Material

Useful material can also be found in:

- Hellerstein, Stonebraker, Hamilton,
  Architecture of a Database System,
  Foundations and Trends in Database Systems, 141-259, 2007.
  (Overview of modern DBMS architectures)
- Graefe,
  Query Evaluation Techniques for Large Databases,
  ACM Computing Surveys, 25(2), 73-170, 1993.
  (Overview of advanced query processing)
- Gaede and Gunther,
  Multidimensional Access Methods,
  ACM Computing Surveys, 30(2), 170-231, 1998.
  (Overview of indexing techniques)

These (and others) are available via the course web site.

---

## PostgreSQL

In this course, we will be using PostgreSQL v12.1 (compulsory)

PostgreSQL tarball is available for copying from the course web site.
(Don't waste your IP quota downloading it from www.postgresql.org)

Install/modify your own PostgreSQL server at CSE (from source code).

(This is **not** the same setup as for COMP3311; COMP3311 used a shared binary)

Working on a home PC is simple if you run Linux or Mac OSX.

Alternatively, use `putty` to connect to CSE from home.

However, you can do it all on Windows (see Chap.17 PostgreSQL manual)

---

## ... PostgreSQL

PostgreSQL is a *large* software system:

- > 2000 source code files in the core engine/clients
- > 1,500,000 lines of C code in the core

You won't be required to understand all of it :-)

You will need to learn to navigate this code effectively.

Will discuss relevant parts in lectures to help with this.

---

## ... PostgreSQL

Some comments on PostgreSQL books:

- they tend to be expensive and short-lived
- many just provide the manual, plus a bit extra
- generally, anything published by O'Reilly is useful
- does the book describe the right version? (PostgreSQL 12)

There's no need to buy a PostgreSQL reference book ...

---

## Relational Database Revision

---

## Relational Databases

*Relational databases* build on relational theory

- all data is modelled as *relations* (tables) and *tuples*
- *constraints* define consistency of data
- normalisation theory *validates* data designs
- *relational algebra* describes manipulation of data

Relational theory provides the foundation, plus ...

- 40 years of technology development
- standardised declarative language (SQL)

leading to modern relational DBMSs.

---

#### ... Relational Databases

We begin by looking at relational databases top-down

- starting from SQL
- through relational operations
- to file structures and storage devices

For the remainder of the semster, we work bottom-up.

---

## Database Management Systems

Relational DBMSs provide critical infrastructure for modern computing

- enterprises commit mission-critical data to DBMSs (e.g. CBA)
- many major Web-sites are based on them (e.g. Wikipedia)
- even embedded in other software (e.g. web browsers)

Nowadays, some very large web sites are developing their own large-scale distributed solutions

- Google (GFS), Amazon (SimpleDB), ...

But even they run relational DBMSs in their "back-office".

---

## DBMS History

| | |
|---|---|
| 1960s | Files, Hierachical and network databases |
| 1970 | Relational data model (Ted Codd) |

| | |
|---|---|
| 1975 | First RDBMS and SQL (IBM Almaden) |
| 1979 | First version of Oracle |
| 1980s | Refinement of technology, distributed systems, new data types (objects, Prolog) |
| 1990s | Object-relational DBMSs, OLAP, data mining, data warehousing, multimedia data, SQL standards |
| 2000s | Databases for XML, bioinformatics, telecomms, SQL3 |
| 2000s | also, very-large, distributed, relaxed-consistency storage |

---

## DBMS Functionality

DBMSs provide a variety of functionalities:

- storing/modifying *data* and *meta-data* ~>(data defintions)
- *constraint* definition/storage/maintenance/checking
- declarative manipulation of data (via *SQL*)
- extensibility via *views, triggers, procedures*
- query re-writing (*rules*), optimisation (*indexes*)
- *transaction* processing, concurrency/recovery
- etc. etc. etc.

Common feature of all relational DBMSs: relational model, SQL.

---

## DBMS for Data Definition

Critical function of DBMS: defining relational data (DDL sub-language)

Relational data: relations/tables, tuples, values, types, constraints.

E.g.

```
create domain WAMvalue float
   check (value between 0.0 and 100.0);

create table Students (
   id          integer,  -- e.g. 3123456
   familyName  text,     -- e.g. 'Smith'
   givenName   text,     -- e.g. 'John'
   birthDate   date,     -- e.g. '1-Mar-1984'
   wam         WAMvalue, -- e.g. 85.4
   primary key (id)
);
```

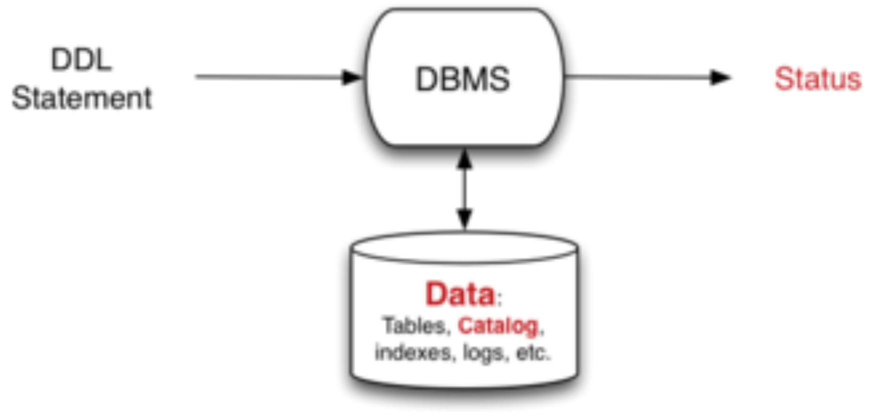
Executing the above adds *meta-data* to the database.

DBMSs typically store meta-data as special tables (catalog).

---

### ... DBMS for Data Definition

Input: DDL statements



Result: meta-data in catalog is modified

---

### ... DBMS for Data Definition

Specifying *constraints* is an important aspect of data definition:

- attribute (column) constraints
- tuple constraints
- relation (table) constraints
- referential integrity constraints

Examples:

```
create table Employee (
   id      integer primary key,
   name    varchar(40),
   salary  real,
   age     integer check (age > 15),
   worksIn integer
              references Department(id),
   constraint PayOk check (salary > age*1000)
);
```

---

## DBMS for Data Modification

Critical function of DBMS: manipulating data   (DML sub-language)

- `insert` new tuples into tables
- `delete` existing tuples from tables
- `update` values within existing tuples

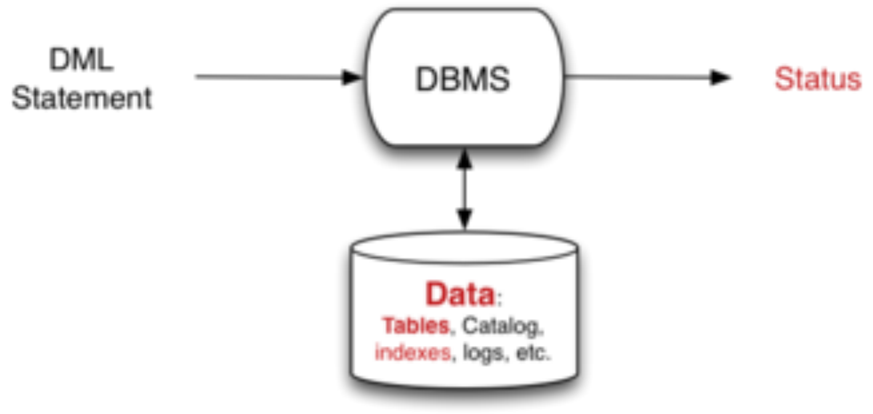
E.g.

```
insert into Enrolments(student,course,mark)
values (3312345, 5542, 75);

update Enrolments set mark = 77
where  student = 3354321 and course = 5542;

delete Enrolments where student = 331122333;
```

---

### ... DBMS for Data Modification

Input: DML statements



Result: tuples are added, removed or modified

---

### ... DBMS for Data Modification

Most DBMSs also provide bulk download/upload mechanisms:

- `dump` gives text copy of data/schema
- `load` reads data/schema info into DB

For PostgreSQL:

- `pg_dump` application dumps database
- `copy` SQL command loads entire tables

---

## DBMS as Query Evaluator

Most common function of relational DBMSs

- read an SQL query
- return a table giving result of query

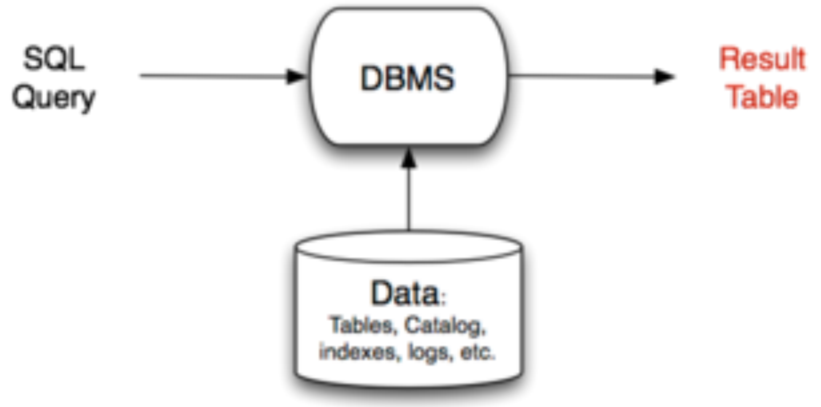
E.g.

```
select s.id, c.code, e.mark
from   Students s
       join Enrolments e on s.id = e.student
       join Courses c on e.course = c.id;
```

---

### ... DBMS as Query Evaluator

Input: SQL query



Output: table (displayed as text)

## DBMS Architecture

The aim of this course is to

- look inside the DBMS box
- discover the various mechanisms it uses
- understand and analyse their performance

Why should we care? (apart from passing the exam)

Practical reason:

- if we understand how query processor works
  ⇒ we can (maybe) do a better job of writing efficient queries

Educational reason:

- DBMSs contain interesting data structures + algorithms
- these may be useful outside the (relational) DBMS context

---

### ... DBMS Architecture

Fundamental tenets of DBMS architecture:

- data is stored permanently on large slow devices**
- data is processed in small fast memory

Implications:

- data structures should minimise storage utilisation
- algorithms should minimise memory/disk data transfers

In the past, DBMSs attempted to solve this by completely controlling disks themselves.

Modern DBMSs interact with storage via the O/S file-system.

** SSDs change things a little, but most high volume bulk storage still on disks

---

### ... DBMS Architecture

Implementation of DBMS operations is complicated by

- potentially multiple concurrent accesses to data structures
  (not just data tables, but indexes, buffers, catalogues, ...)
- transactional requirements (atomicity, rollback, ...)
- requirement for high reliability of raw data (recovery)

Locking helps with concurrency, but may degrade performance.

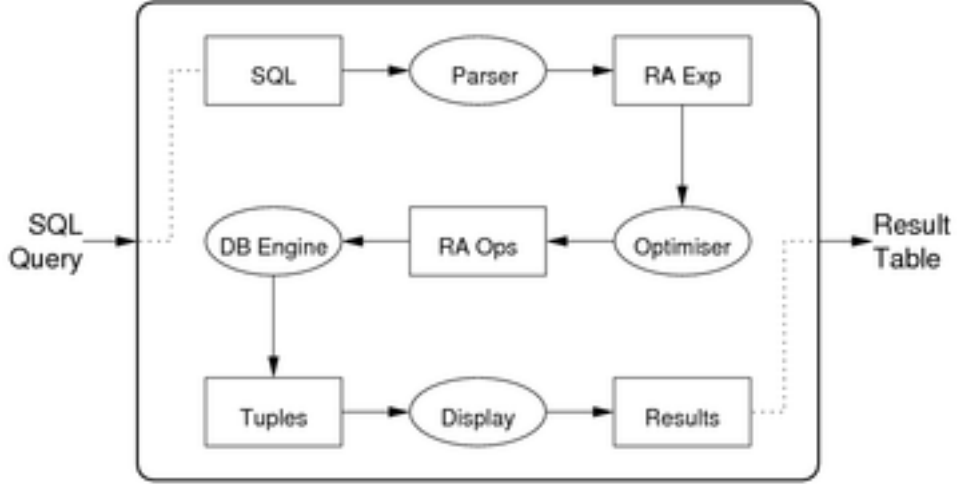
In practice, may need new "concurrency-tolerant" data structures.

Transactions/reliability require some form of logging.

---

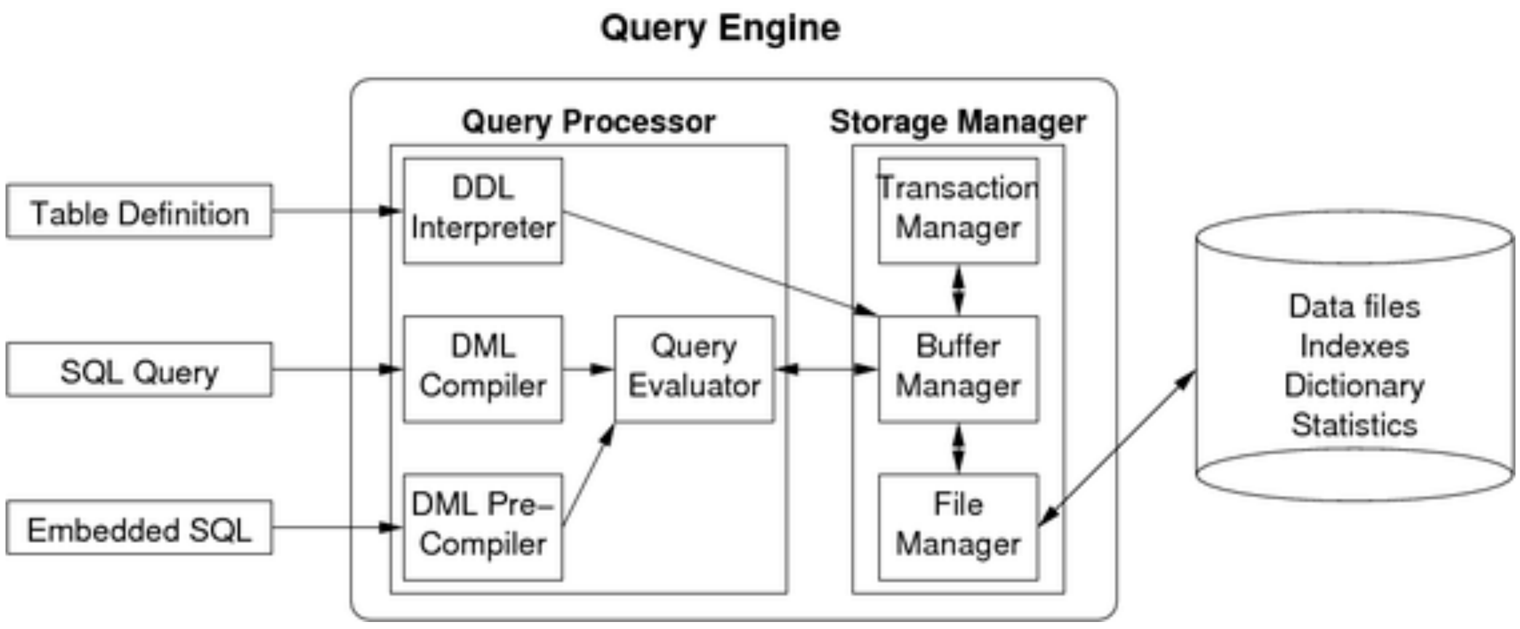
### ... DBMS Architecture

Path of a query through a typical DBMS:



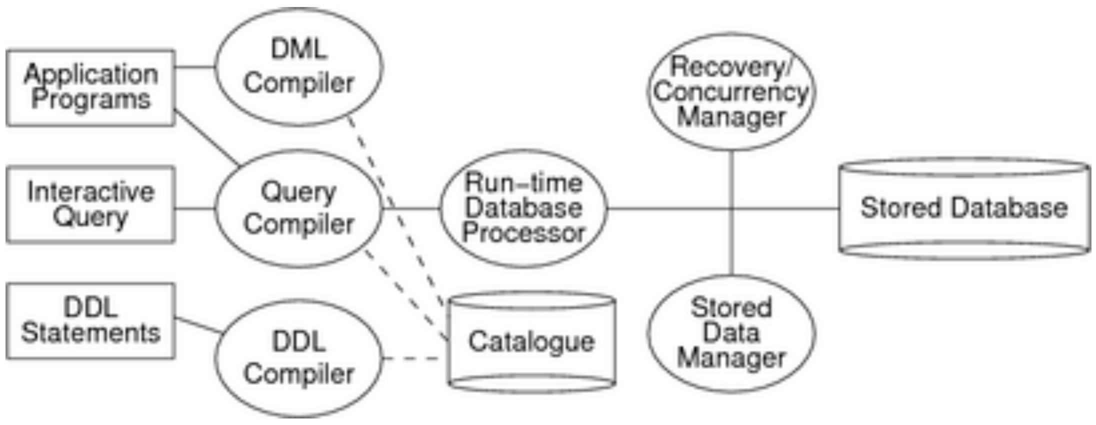
## ... DBMS Architecture

Accoring to Silberschatz/Korth/Sudarshan (SKS) ...



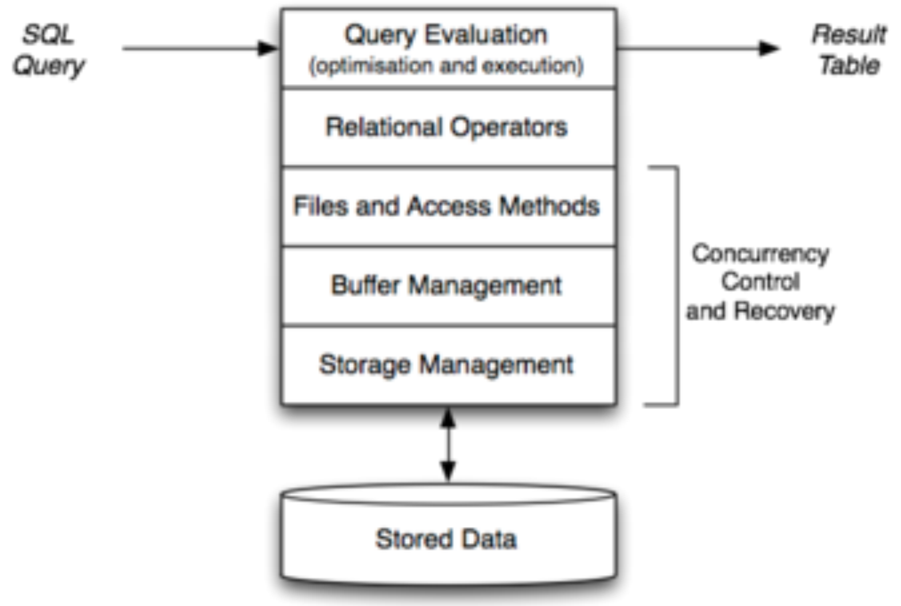
## ... DBMS Architecture

Accoring to Elmasri/Navathe (EN) ...



## ... DBMS Architecture

According to Ramakrishnan/Gerhke (RG) ...



## ... DBMS Architecture

| | |
|---|---|
| Query optimiser | translates queries into efficient sequence of relational ops |
| Query executor | controls execution of sequence of relational ops |
| Access methods | basis for implementation of relational operations |
| Buffer manager | manages data transfer between disk and main memory |
| Storage manager | manages allocation of disk space and data structures |
| Concurrency manager | controls concurrent access to database |
| Recovery manager | ensures consistent database state after system failures |
| Integrity manager | verifies integrity constraints and user privileges |

## Database Engine Operations

DB engine = "relational algebra virtual machine":

| | | |
|---|---|---|
| selection ($\sigma$) | projection ($\pi$) | join ($\bowtie$) |
| union ($\cup$) | intersection ($\cap$) | difference (-) |
| sort | group | aggregate |

For each of these operations:

- various data structures and algorithms are available
- DBMSs may provide only one, or may provide a choice

## ... Database Engine Operations

Different implementations of Selection:

- a hash-structured file is good for queries like:

  ```
  select * from Students where id = 3312345;
  ```

  where `id` is the hashing attribute

- a B-tree file is good for queries like:

```
select * from Employees where age > 55;
```

## Relational Algebra

*Relational algebra* (RA) can be viewed as ...

- mathematical system for manipulating relations, or
- data manipulation language (DML) for the relational model

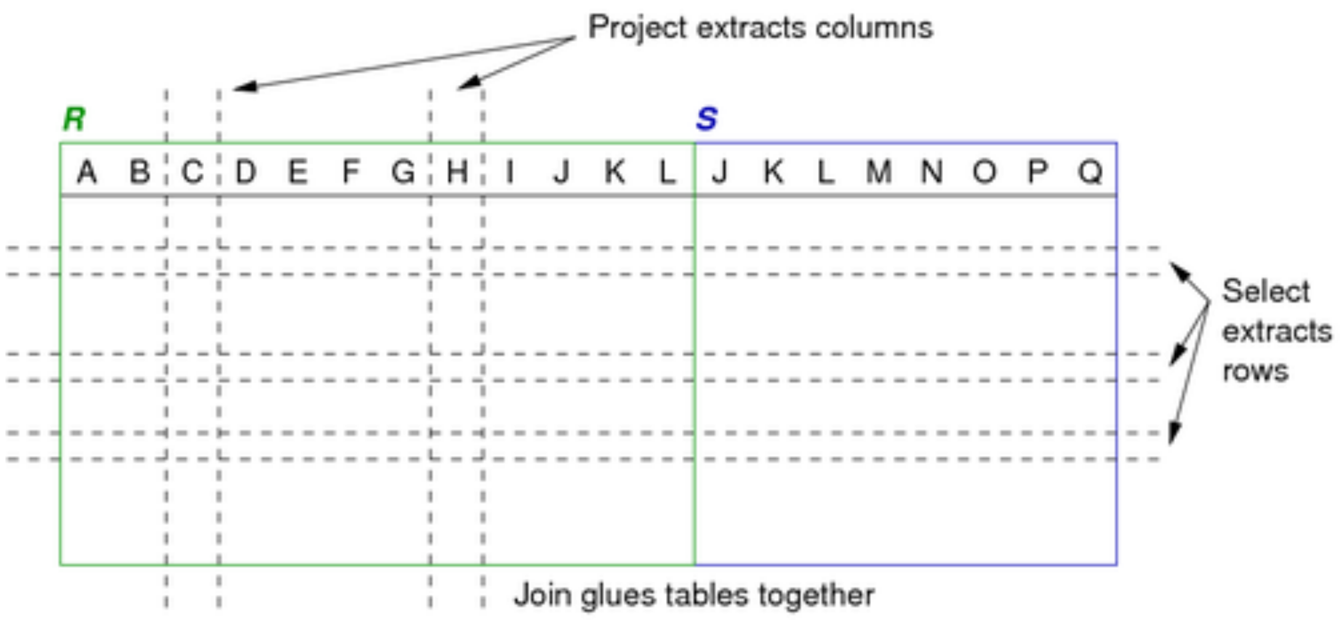
Relational algebra consists of:

- *operands*: relations, or variables representing relations
- *operators* that map relations to relations
- rules for combining operands/operators into expressions
- rules for evaluating such expressions

RA can be viewed as the "machine language" for RDBMSs

### ... Relational Algebra

Select, project, join provide a powerful set of operations for constructing relations and extracting relevant data from them.



Adding set operations and renaming makes RA *complete*.

## Notation

Standard treatments of relational algebra use Greek symbols.

We use the following notation (because it is easier to reproduce):

| **Operation** | **Standard Notation** | **Our Notation** |
|---|---|---|
| Selection | $\sigma_{expr}(Rel)$ | *Sel[expr](Rel)* |
| Projection | $\pi_{A,B,C}(Rel)$ | *Proj[A,B,C](Rel)* |
| Join | $Rel_1 \bowtie_{expr} Rel_2$ | $Rel_1$ *Join[expr]* $Rel_2$ |
| Rename | | *Rename[schema](Rel)* |

$_{schema}Rel$

For other operations (e.g. set operations) we adopt the standard notation.

## ... Notation

We define the semantics of RA operations using

- regular "conditional set" expressions   e.g. *{ x | condition }*
- tuple notations:
    - *t[ab]*   (extracts attributes *a* and *b* from tuple *t*)
    - *(x,y,z)*   (enumerated tuples; specify attribute values)
- quantifiers, set operations, boolean operators

## ... Notation

All RA operators return a result relation (no DB updates).

For convenience, we can name a result and use it later.

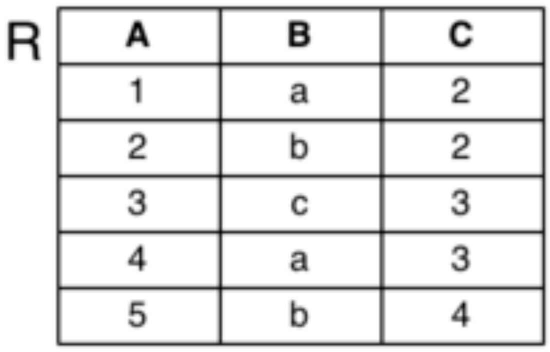
E.g.

```
Temp = R op1 S op2 T
Res  = Temp op3 Z
-- which is equivalent to
Res  = (R op1 S op2 T) op3 Z
```

Each "intermediate result" has a well-defined schema.

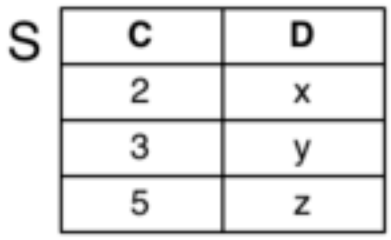
## Sample Relations

Example database #1 to demonstrate RA operators:



R

| A | B | C |
|---|---|---|
| 1 | a | 2 |
| 2 | b | 2 |
| 3 | c | 3 |
| 4 | a | 3 |
| 5 | b | 4 |



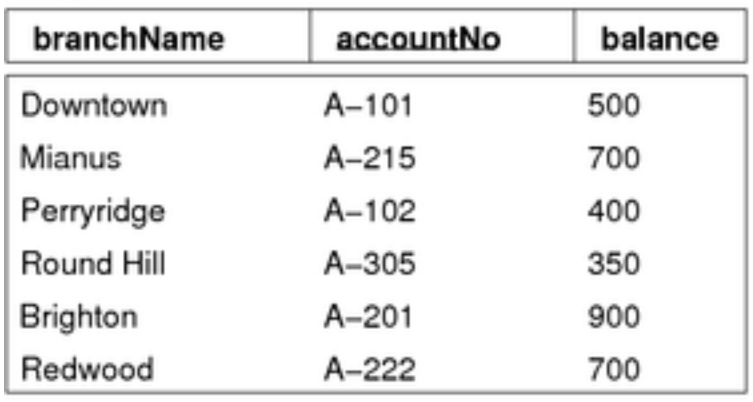
S

| C | D |
|---|---|
| 2 | x |
| 3 | y |
| 5 | z |

## ... Sample Relations

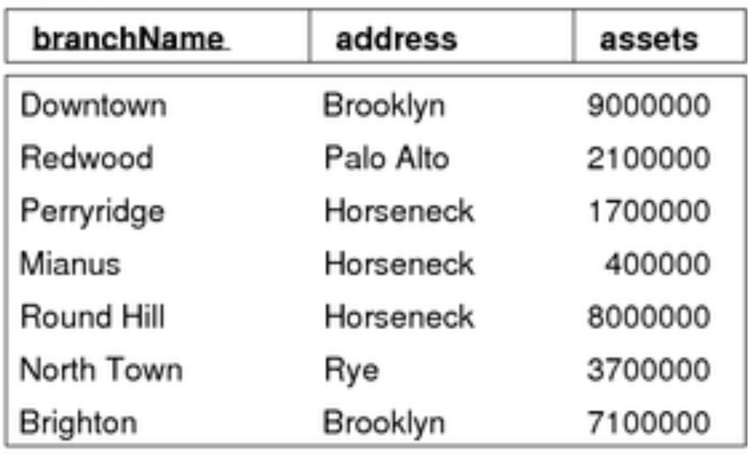
Example database #2 to demonstrate RA operators:

*Account*



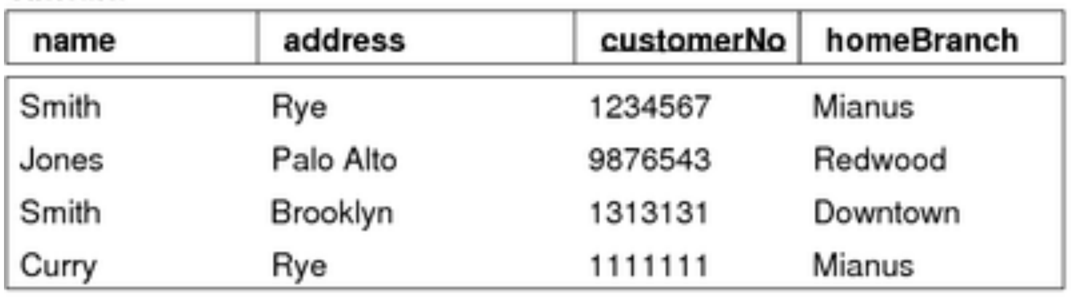
| branchName | accountNo | balance |
|---|---|---|
| Downtown | A–101 | 500 |
| Mianus | A–215 | 700 |
| Perryridge | A–102 | 400 |
| Round Hill | A–305 | 350 |
| Brighton | A–201 | 900 |
| Redwood | A–222 | 700 |

*Branch*



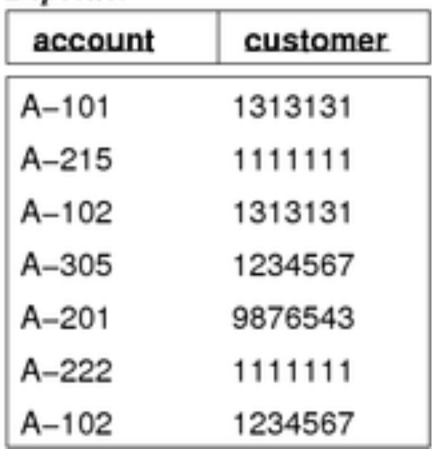
| branchName | address | assets |
|---|---|---|
| Downtown | Brooklyn | 9000000 |
| Redwood | Palo Alto | 2100000 |
| Perryridge | Horseneck | 1700000 |
| Mianus | Horseneck | 400000 |
| Round Hill | Horseneck | 8000000 |
| North Town | Rye | 3700000 |
| Brighton | Brooklyn | 7100000 |

*Customer*



| name | address | customerNo | homeBranch |
|---|---|---|---|
| Smith | Rye | 1234567 | Mianus |
| Jones | Palo Alto | 9876543 | Redwood |
| Smith | Brooklyn | 1313131 | Downtown |
| Curry | Rye | 1111111 | Mianus |

*Depositor*



| account | customer |
|---|---|
| A–101 | 1313131 |
| A–215 | 1111111 |
| A–102 | 1313131 |
| A–305 | 1234567 |
| A–201 | 9876543 |
| A–222 | 1111111 |
| A–102 | 1234567 |

## Selection

*Selection* returns a subset of the tuples in a relation *r* that satisfy a specified condition *C*.

$\sigma_C(r) \;=\; Sel[C](r) \;=\; \{\, t \mid t \in r \wedge C(t) \,\}$,   where $r(R)$

*C* is a boolean expression on attributes in *R*.

Result size: $|\sigma_C(r)| \leq |r|$

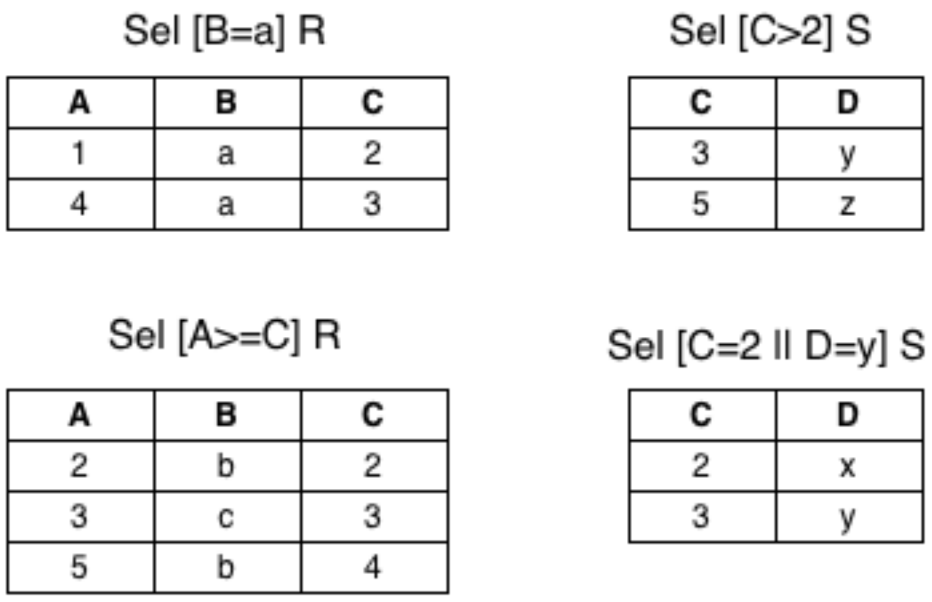
Result schema: same as the schema of *r* (i.e. *R*)

Computational view:

```
result = {}
for each tuple t in relation r
    if (C(t)) { result = result ∪ {t} }
```

### ... Selection

Example selections:



Sel [B=a] R

| A | B | C |
|---|---|---|
| 1 | a | 2 |
| 4 | a | 3 |

Sel [C>2] S

| C | D |
|---|---|
| 3 | y |
| 5 | z |

Sel [A>=C] R

| A | B | C |
|---|---|---|
| 2 | b | 2 |
| 3 | c | 3 |
| 5 | b | 4 |

Sel [C=2 || D=y] S

| C | D |
|---|---|
| 2 | x |
| 3 | y |

---

### ... Selection

Example queries:

- Find details about the Perryridge branch?
    - *Sel [branchName=Perryridge] (Branch)*
- Which accounts are overdrawn?
    - *Sel [balance<0] (Account)*
- Which Round Hill accounts are overdrawn?
    - *Sel [branchName=Round Hill ∧ balance<0] (Account)*

---

## Projection

*Projection* returns a set of tuples containing a subset of the attributes in the original relation.

$$\pi_X(r) \;=\; Proj[X](r) \;=\; \{\, t[X] \mid t \in r \,\}, \quad \text{where } r(R)$$

*X* specifies a subset of the attributes of *R*.

Note that removing key attributes can produce duplicates.

In RA, duplicates are removed from the result *set*.
(In many RDBMS's, duplicates are retained (i.e. they use bag, not set, semantics))

Result size: $|\pi_X(r)| \le |r|$ Result schema: *R'(X)*

Computational view:

```
result = {}
for each tuple t in relation r
    result = result ∪ {t[X]}
```

---

### ... Projection

Example projections:

R

| A | B | C |
|---|---|---|
| 1 | a | 2 |
| 2 | b | 2 |
| 3 | c | 3 |
| 4 | a | 3 |
| 5 | b | 4 |

Proj [B] R

| B |
|---|
| a |
| b |
| c |

Proj [A,C] R

| A | C |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 3 |
| 4 | 3 |
| 5 | 4 |

---

**... Projection** 67/130

Example queries:

- What branches are there?
    - *Proj [branchName] (Branch)*
- Which branches actually hold accounts?
    - *Proj [branchName] (Account)*
- What are the names and addresses of all customers?
    - *Proj [name,address] (Customer)*
- Generate a list of all the account numbers
    - *Proj [accountNo] (Account)* or
    - *Proj [account] (Depositor)* (if we assume every account has a depositor)

---

## Union

68/130

*Union* combines two *compatible* relations into a single relation via set union of sets of tuples.

$$r_1 \cup r_2 \ = \ \{ t \mid t \in r_1 \vee t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

Compatibility = both relations have the same schema

Result size: $|r_1 \cup r_2| \ \leq \ |r_1| + |r_2|$ Result schema: $R$

Computational view:

```
result = $r_1$
for each tuple $t$ in relation $r_2$
    result = result ∪ {t}
```

---

**... Union** 69/130

Example queries:

- Which suburbs have either customers or branches?
    - *Proj[address](Customer) ∪ Proj[address](Branch)*
- Which branches have either customers or accounts?
    - *Proj[homeBranch](Customer) ∪ Proj[branchName](Account)*

The union operator is symmetric i.e. $R \cup S \ = \ S \cup R$.

---

## Intersection

70/130

*Intersection* combines two *compatible* relations into a single relation via set intersection of sets of tuples.

$$r_1 \cap r_2 \ = \ \{ t \mid t \in r_1 \wedge t \in r_2 \}, \quad \text{where } r_1(R), r_2(R)$$

Uses same notion of relation compatibility as union.

Result size: $|r_1 \cup r_2| \le \min(|r_1|,|r_2|)$   Result schema: *R*

Computational view:

```
result = {}
for each tuple t in relation $r_1$
    if ($t \in r_2$) { result = result ∪ {t} }
```

---

### ... Intersection

Example queries:

- Which suburbs have both customers and branches?
    - *Proj[address](Customer) ∩ Proj[address](Branch)*
- Which branches have both customers and accounts?
    - *Proj[homeBranch](Customer) ∩ Proj[branchName](Account)*

The intersection operator is symmetric i.e. $R \cap S = S \cap R$.

---

## Difference

*Difference* finds the set of tuples that exist in one relation but do not occur in a second *compatible* relation.

$$r_1 - r_2 = \{ t \mid t \in r_1 \land \neg\, t \in r_2 \}, \quad \text{where } r_1(R),\ r_2(R)$$

Uses same notion of relation compatibility as union.

Note: tuples in $r_2$ but not $r_1$ do not appear in the result

i.e. set difference != complement of set intersection

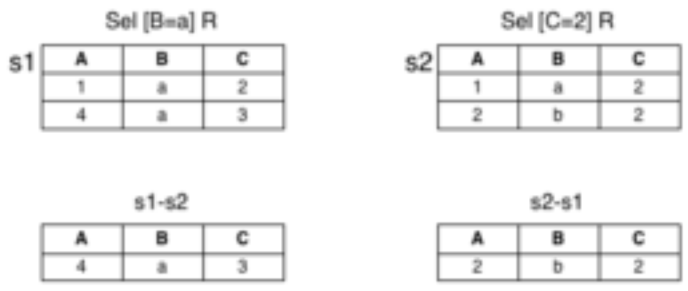
Computational view:

```
result = {}
for each tuple t in relation $r_1$
    if (!($t \in r_2$)) { result = result ∪ {t} }
```

---

### ... Difference

Example difference:



Sel [B=a] R

s1

| A | B | C |
|---|---|---|
| 1 | a | 2 |
| 4 | a | 3 |

Sel [C=2] R

s2

| A | B | C |
|---|---|---|
| 1 | a | 2 |
| 2 | b | 2 |

s1-s2

| A | B | C |
|---|---|---|
| 4 | a | 3 |

s2-s1

| A | B | C |
|---|---|---|
| 2 | b | 2 |

*s1 = Sel [B = 1] (r1)*

*s2 = Sel [C = x] (r1)*

*s1 - s2*

*s2 - s1*

## ... Difference

Example queries:

- Which customers have no accounts?
  - *AllCusts = Proj[customerNo](Customer)*
    *CustsWithAccts = Proj[customer](Depositor)*
    *Result = AllCusts - CustsWithAccts*
- Which branches have no customers?
  - *AllBranches = Proj[branchName](Branch)*
    *BranchesWithCusts = Proj[homeBranch](Customer)*
    *Result = AllBranches - BranchesWithCusts*

---

## Natural Join

*Natural join* is a specialised product:

- containing only pairs that match on their *common* attributes
- with one of each pair of common attributes eliminated

Consider relation schemas *R(ABC..JKLM)*, *S(KLMN..XYZ)*.

The natural join of relations *r(R)* and *s(S)* is defined as:

$$r \bowtie s \;=\; r\ Join\ s \;=$$
$$\{\ (t_1[ABC..J] : t_2[K..XYZ])\ \mid\ t_1 \in r \wedge t_2 \in s \wedge match\ \}$$

$$\text{where}\quad match \;=\; t_1[K] = t_2[K] \wedge t_1[L] = t_2[L] \wedge t_1[M] = t_2[M]$$

Computational view:

```
result = {}
for each tuple $t_1$ in relation r
   for each tuple $t_2$ in relation s
      if (matches($t_1$,$t_2$))
         result = result ∪ {combine($t_1$,$t_2$)}
```

---

## ... Natural Join

Natural join can also be defined in terms of other relational algebra operations:

$$r\ Join\ s \;=\; Proj[R \cup S]\ (\ Sel[match]\ (\ r \times s)\ )$$

We assume that the union on attributes eliminates duplicates.

If we wish to join relations, where the common attributes have different names, we rename the attributes first.

E.g. *R(ABC)* and *S(DEF)* can be joined by

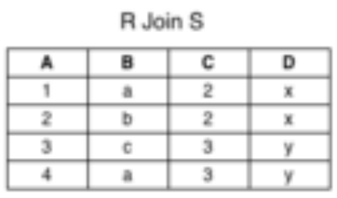
*R Join Rename[S(DCF)](S)*

Note: $|r \bowtie s| \ll |r \times s|$, so *join* not implemented via *product*.

---

## ... Natural Join

Example natural join:


R Join S

| A | B | C | D |
|---|---|---|---|
| 1 | a | 2 | x |
| 2 | b | 2 | x |
| 3 | c | 3 | y |
| 4 | a | 3 | y |

## ... Natural Join

Example queries:

- Who is the owner of account A101?
  - *Proj[name](Sel[account=A101](Customer ⋈ Depositor))*
- Which accounts are held in branches in Horseneck?
  - *tmp1 = Sel[address=Horseneck](Account ⋈ Branch)*
    *res   = Proj[accountNo](tmp1))*
- Which customers hold accounts at a Brooklyn branch?
  - *tmp1 = Account ⋈ Branch ⋈ Customer ⋈ Depositor*
    *res   = Proj[name](Sel[address=Brooklyn](tmp1))*

## Theta Join

The *theta join* is a specialised product containing only pairs that match on a supplied condition *C*.

$$r \bowtie_C s \;=\; \{ (t_1 : t_2) \mid t_1 \in r \wedge t_2 \in s \wedge C(t_1 : t_2) \},$$
where $r(R),s(S)$

Examples:   *(r1 Join[B>E] r2)* ... *(r1 Join[E<D ∧ C=G] r2)*

Can be defined in terms of other RA operations:

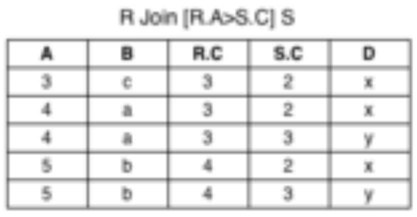
$$r \bowtie_C s \;=\; r\ Join[C]\ s \;=\; Sel[C]\ (\ r \times s\ )$$

Unlike natural join, "duplicate" attributes are not removed.

Note that   $r \bowtie_{true} s \;=\; r \times s$.

## ... Theta Join

Example theta join:


R Join [R.A>S.C] S

| A | B | R.C | S.C | D |
|---|---|---|---|---|
| 3 | c | 3 | 2 | x |
| 4 | a | 3 | 2 | x |
| 4 | a | 3 | 3 | y |
| 5 | b | 4 | 2 | x |
| 5 | b | 4 | 3 | y |

## ... Theta Join

Comparison between join operations:

- *theta join* allows arbitrary tests in the condition
  (and leaves all attributes from the original relations in the result)
- *equijoin* has only equality tests in the condition
  (and leaves all attributes from the original relations in the result)
- *natural join* has only equality tests on common attributes
  (and removes one of each pair of matching attributes)

Equijoin is a specialised theta join; natural join is like theta join followed by projection.

---

## Outer Join

*r Join s* eliminates all *s* tuples that do not match some *r* tuple.

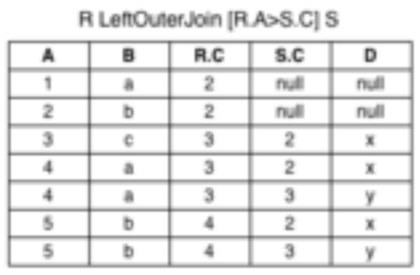
Sometimes, we wish to keep this information, so *outer join*

- includes all tuples from each relation in the result
- for pairs of matching tuples, concatenate attributes as for standard join
- for tuples that have no match, assign null to "unmatched" attributes

---

### ... Outer Join

Example outer join:



R LeftOuterJoin [R.A>S.C] S

| A | B | R.C | S.C | D |
|---|---|---|---|---|
| 1 | a | 2 | null | null |
| 2 | b | 2 | null | null |
| 3 | c | 3 | 2 | x |
| 4 | a | 3 | 2 | x |
| 4 | a | 3 | 3 | y |
| 5 | b | 4 | 2 | x |
| 5 | b | 4 | 3 | y |

Contrast this to the result for theta-join presented earlier.

---

### ... Outer Join

There are three variations of outer join *R OuterJoin S*:

- left outer join (*LeftOuterJoin*) includes all tuples from *R*
- right outer join (*RightOuterJoin*) includes all tuples from *S*
- full outer join (*OuterJoin*) includes all tuples from *R* and *S*

Which one to use depends on the application e.g.

If we want to know about all Branches, regardless of whether they have Customers as their homeBranch:

*Branches LeftOuterJoin[branchName=homeBranch] Customer*

---

### ... Outer Join

Computational description of *r(R) LeftOuterJoin s(S)*:

```
result = {}
for each tuple $t_1$ in relation r
   nmatches = 0
   for each tuple $t_2$ in relation s
      if (matches($t_1$,$t_2$))
         result = result ∪ {combine($t_1$,$t_2$)}
         nmatches++
   if (nmatches == 0)
      result = result ∪
                 {combine($t_1$,$S_{null}$)}
```

where $S_{null}$ is a tuple from *S* with all atributes set to NULL.

## Aggregation

Two types of aggregation are common in database queries:

- accumulating summary values for data in tables
  - typical operations *Sum*, *Average*, *Count*
  - many operations work on a single column
    (e.g. *Sum[assets](Branch)*)
- grouping sets of tuples with common values
  - *GroupBy[$A_1$...$A_n$](R)*
  - typically we group using only a single attribute

---

## Generalised Projection

In standard projection, we select values of specified attributes.

In *generalised projection* we perform some computation on the attribute value before placing it in the result tuple.

Examples:

- Display branch assets in Aus$ rather than US$.
  - *Proj [branchname,address,assets*0.75] (Branch)*
- Display employee records using age rather than birthday.
  - *Proj [id,name,(today-birthdate)/365,salary] (Employee)*

---

# PostgreSQL

---

## PostgreSQL

*PostgreSQL* is a full-featured open-source (O)RDBMS.

- provides a relational engine with:
  - efficient implementation of relational operations
  - very good transaction processing (concurrent access)
  - good backup/recovery (from application/system failure)
  - novel query optimisation (genetic algorithm-based)
  - replication, JSON, extensible indexing, etc. etc.
- already supports several non-standard data types
- allows users to define their own data types
- supports most of the SQL3 standard

---

## Brief History of PostgreSQL

| | |
|---|---|
| 1977-1985 | *Ingres* (Stonebraker)<br>research prototype<br>→ Relational Technologies<br>→ bought by Computer Associates |
| 1986-1994 | *Postgres* (Stonebraker)<br>research prototype<br>→ Illustra → bought by Informix |
| 1994-1995 | *Postgres95* (Chen, Yu)<br>added SQL, spawned PostgreSQL |
| 1996-... | *PostgreSQL* (Momjian,Lane,...)<br>open-source DBMS with Oracle-level functionality, platform for experiments with new DBMS implementation ideas |

## PostgreSQL Online

Web site: www.postgresql.org

Key developers: Bruce Momjian, Tom Lane, Marc Fournier, ...

Full list of developers: www.postgresql.org/developer/bios

Local copy of source code:

```
/home/cs9315/web/20T1/postgresql/src.tar.bz2
```

Documentation is available via WebCMS menu.

## User View of PostgreSQL

Users interact via SQL in a *client* process, e.g.

```
$ psql webcms
SET
psql (11.3)
Type "help" for help.
webcms2=# select * from calendar;
 id | course |   evdate   |       event
----+--------+------------+---------------------------
  1 |      4 | 2001-08-09 | Project Proposals due
 10 |      3 | 2001-08-01 | Tute/Lab Enrolments Close
 12 |      3 | 2001-09-07 | Assignment #1 Due (10pm)
 ...
```

or

```
$dbconn = pg_connect("dbname=webcms");
$result = pg_query($dbconn,"select * from calendar");
while ($tuple = pg_fetch_array($result))
   { ... $tuple["event"] ... }
```

## PostgreSQL Functionality

PostgreSQL systems deal with various kinds of entities:

- *users* ... who can use the system, what they can do
- *groups* ... groups of users, for role-based privileges
- *databases* ... collections of schemas/tables/views/...
- *namespaces* ... to uniquely identify objects (schema.table.attr)
- *tables* ... collection of tuples (standard relational notion)
- *views* ... "virtual" tables (can be made updatable)
- *functions* ... operations on values from/in tables
- *triggers* ... operations invoked in response to events
- *operators* ... functions with infix syntax
- *aggregates* ... operations over whole table columns
- *types* ... user-defined data types (with own operations)
- *rules* ... for query rewriting (used e.g. to implement views)
- *access methods* ... efficient access to tuples in tables

### ... PostgreSQL Functionality

PostgreSQL's dialect of SQL is mostly standard (but with extensions).

Differences visible at the user-level:

- attributes containing arrays of atomic values
- table type inheritance, table-valued functions, ...

Differences at the implementation level:

- referential integrity checking is accomplished via triggers
- views are implemented via query re-writing *rules*

Example:

```
create view myview as select * from mytab;
-- is implemented as
create type as myview (same fields as mytab);
create rule myview as on select to myview
            do instead select * from mytab;
```

---

## ... PostgreSQL Functionality

PostgreSQL stored procedures differ from SQL standard:

- only provides functions, not procedures (but functions can return `void`)
- allows function overloading (same function name, diff argument types)
- defined at different "lexical level" to SQL
- provides own PL/SQL-like language for functions

Example:

```
create or replace function
    barsIn(suburb text) returns setof Bars
as $$
declare
    r record;
begin
    for r in
        select * from Bars where location = suburb
    loop
       return next r;
    end loop;
end;
$$ language plpgsql;
used as e.g.
select * from barsIn('Randwick');
```

---

## ... PostgreSQL Functionality

Concurrency is handled via *multi-version concurrency control* (MVCC)

- multiple "versions" of the database exist together
- a transaction sees the version that was valid at its start-time
- readers don't block writers; writers don't block readers
- this significantly reduces the need for locking

Disadvantages of this approach:

- extra storage for old versions of tuples   (`vacuum` fixes this)

---

## ... PostgreSQL Functionality

Allows transactions to specify a consistency level for concurrency

- read-committed (allows some inconsistency), serializable (no inconsistency)
- default isolation level is read-committed   (⇒ potential portability issues)

Explicit locking is also available.

- different varieties: share/exclusive, row/table
- deadlock detection via time-out

Access methods need to implement their own concurrency control.

---

## ... PostgreSQL Functionality

PostgreSQL has a well-defined and open extensibility model:

- stored procedures are held in database as strings
    - allows a variety of languages to be used
    - language interpreters can be integrated into PostgreSQL engine
- new data types, operators, aggregates, indexes can be added
    - typically requires code written in C, following defined API
    - for new data types, need to write input/output functions, ...
    - for new indexes, need to implement file structures

---

## ... PostgreSQL Functionality

Because of its extensibility, PostgreSQL has extra data types:

- built-in:   geometric (line,point,...), network address (macaddr,...)
- contributed:   complex number, ISBN/ISSN, encrypted password, ...

Also has a wider-than-usual range of access methods:

- B-trees, linear hashing, R-trees, GiST/GIN indexes
- full-text indexing

And provides a range of replication services.

---

## Installing PostgreSQL

PostgreSQL is available via the COMP9315 web site.

Provided as tarball and zip in `~cs9315/web/20T1/postgresql/`

Brief summary of installation:

```
$ configure --prefix=~/your/pgsql/directory
$ make
$ make install
$ source ~/your/environment/file
   # set up environment variables
$ initdb
   # set up postgresql configuration
$ pg_ctl start
   # do some work with PostgreSQL databases
$ pg_ctl stop
```

---

## ... Installing PostgreSQL

Simplified version of running the server:

```
$ source ~/your/environment/file
   # set up environment variables
$ pgs setup
   # runs initdb and fixes configuration
```

```
$ pgs start
   # do some work with PostgreSQL databases
$ pgs stop
   # stops server
$ pgs cleanup
   # removes all data files
```

`pgs` is a shell script in `/home/cs9315/bin`

---

## PostgreSQL Configuration

PostgreSQL configuration parameters (some important ones):

- `PGHOME` = directory where PostgreSQL resides
  Same as value given in `configure --prefix=$PGHOME`
  (typical value: /usr/local/pgsql)
- `PGBIN` = directory where client applications reside
  (typical value: $PGHOME/bin)
- `PGDATA` = directory where data files reside
  (typical value: $PGHOME/data)
- `PGHOST` = host where server is running (if using TCP/IP)
- `PGPORT` = port address for server (default: 5432)

Note: if not using TCP/IP, `PGHOST` holds name of directory where Unix socket files reside.

---

### ... PostgreSQL Configuration

A typical envrionment setup for COMP9315:

```
# Set up environment for running PostgreSQL
# Must be "source"d from sh, bash, ksh, ...

PGHOME=/home/jas/srvr/pgsql
export PGDATA=$PGHOME/data
export PGHOST=$PGDATA
export PGPORT=5432
export PATH=$PGHOME/bin:$PATH
export PGDATA PGHOST PATH

alias p0="$D/bin/pg_ctl stop"
alias p1="$D/bin/pg_ctl -l $PGDATA/log start"
```

---

### ... PostgreSQL Configuration

Other configuration files live in `$PGDATA`.

`postgresql.conf`: server configuration

- must change `unix_socket_directory` to match `$PGHOST`
- may change `max_connections` to e.g. 10
- assignments may require other changes
- changes typically need server re-start to take effect

`pg_hba.conf`: authorisation/user access

- which users can access which database from which hosts
- ignore, since we do everything as PostgreSQL super-user

---

## Using PostgreSQL for Assignments

You will need to modify then re-start the server:

```
# edit source code to make changes
$ pg_ctl stop
$ make
$ make install
# restore postgresql configuration
$ pg_ctl start
# run tests and analyse results
```

Assumes no changes that affect storage structures.

I.e. existing databases will continue to work ok.

---

### ... Using PostgreSQL for Assignments

If you change storage structures ...

- old database will not work with the new server
- need to dump, re-run `initdb`, then restore

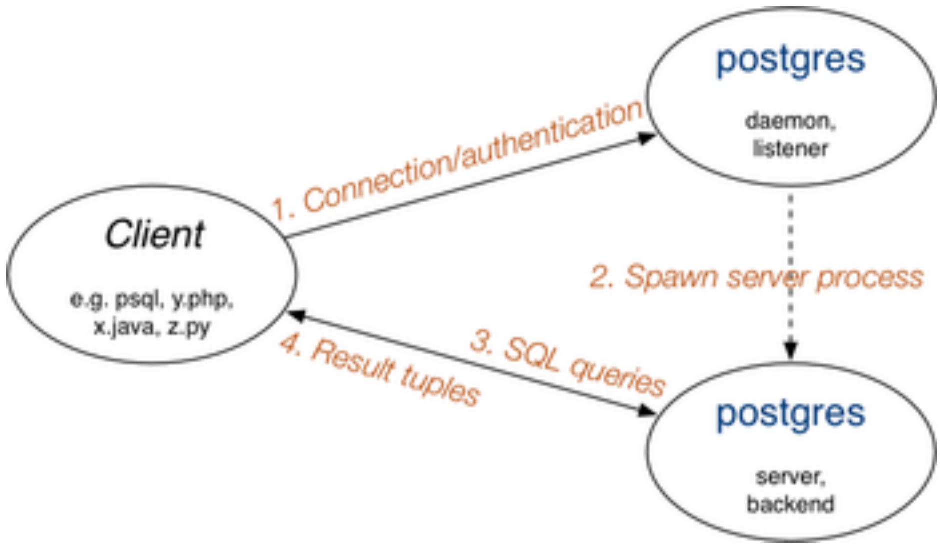
```
# edit source code to make changes
$ pg_dump testdb > testdb.dump
$ make
$ pg_ctl stop
$ rm -fr /your/pgsql/directory/data
$ make install
$ initdb
# restore postgresql configuration
$ pg_ctl start
$ createdb testdb
$ psql testdb -f testdb.dump
# run tests and analyse results
```

Need to save a copy of `postgresql.conf` before re-installing.

---

## PostgreSQL Architecture

Client/server architecture:



Note: nowadays the `postmaster` process is also called `postgres`.

---

### ... PostgreSQL Architecture

Notes:

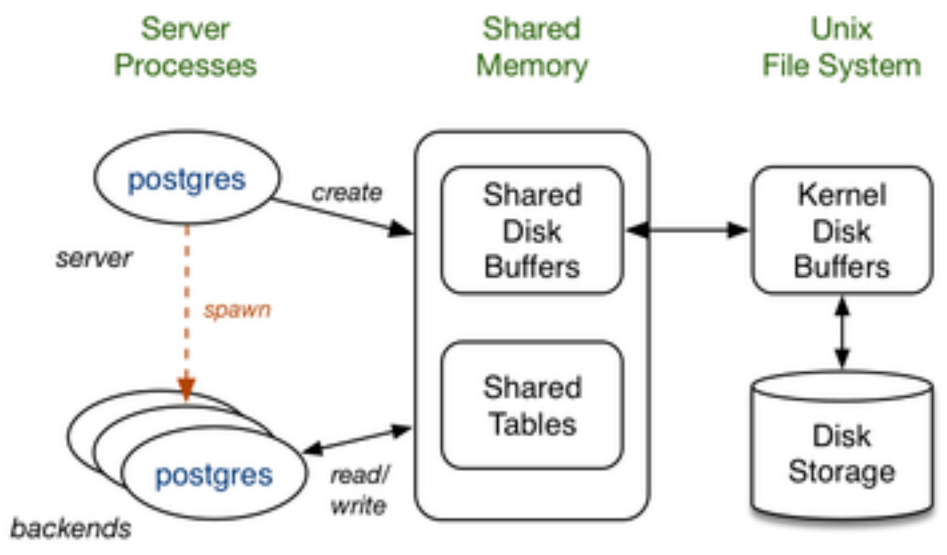
- exactly one postmaster; many clients; many servers

- each client has its own server process
- client/server communication via TCP/IP or Unix sockets
- uses PostgreSQL-specific frontend/backend protocol
- client/server separation good for security/reliability
- client/server connection overhead is significant
  (generally solved by client-side pooling of persistent conenctions)

---

## ... PostgreSQL Architecture

Memory/storage architecture:



---

## ... PostgreSQL Architecture

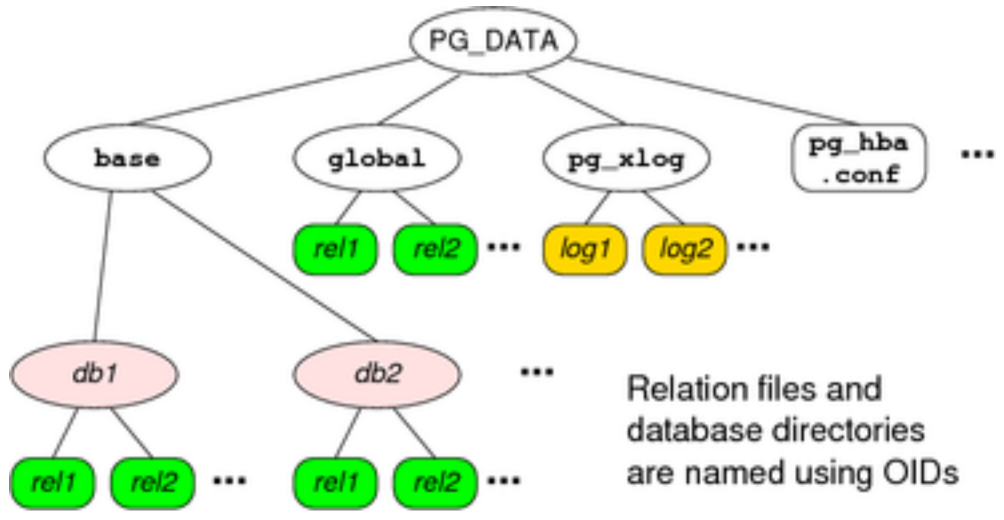
Notes:

- all servers access database files via buffer pool
  (thus, all servers get a consistent view of data ... essential!)
- Unix kernel provides additional buffering (useful?)
- use of shared memory limits distribution/scalability
  (all server processes must run on the same machine)
- shared tables are "global" system catalog tables
  (hold user/group/database info for entire PostgreSQL installation)

---

## ... PostgreSQL Architecture

File-system architecture:



---

## ... PostgreSQL Architecture

Interesting files in $PGDATA:

|  |  |
|---|---|
| PG_VERSION | which server version made this directory |
| pg_hba.conf | who can access which databases from where |
| postgresql.conf | server parameters (e.g. max connections) |
| postmaster.opts | how was current postmaster invoked |
| postmaster.pid | process id of current postmaster |

---

# PostgreSQL Source Code

---

## PostgreSQL Source Code

Top-level of PostgreSQL distribution contains:

- *README,INSTALL*: overview and installation instructions
- *config**: scripts to build localised Makefiles
- *Makefile*: top-level script to control system build
- *src*: sub-directories containing system source code
- *doc*: FAQs and documentation (in various formats)
- *contrib*: source code for contributed extensions

---

### ... PostgreSQL Source Code

How to get started understanding the workings of PostgreSQL:

- become familiar with the user-level interface (psql, pg_dump, pg_ctl, etc.)
- start with the ***.h** files, then move to ***.c** files
  (note that: ***.c** files live under src/backend/*, ***.h** files live under src/include)
- start globally, then work one subsystem-at-a-time

Some helpful information is available via:

- PostgreSQL link on web site
- Readings link on web site (but old docs)

---

### ... PostgreSQL Source Code

The source code directory (*src*) contains:

- *include*: ***.h** files with global definitions (constants, types, ...)
- *backend*: code for PostgreSQL database engine (server)
- *bin*: code for clients (e.g. psql, pg_ctl, pg_dump, ...)
- *pl*: stored procedure language interpreters (e.g. plpgsql)
- *interfaces* code for low-level C interfaces (e.g. libpq)

along with Makefiles to build system and other directories not relevant for us Code for backend (DBMS engine)

- 1300 files (800.c,500.h,4.y,5.l), $10^6$ lines of code

---

### ... PostgreSQL Source Code

We introduce the code

- by following the execution of a query
- tracing which functions are involved
- pointing out the files containing these functions

PostgreSQL manual has detailed description of internals:

- Section VII, Chapters 46,47,53-57
- Ch.46 is an overview; a good place to start
- other chapters discuss specific components

See also "How PostgreSQL Processes a Query"

- `src/tools/backend/index.html`

---

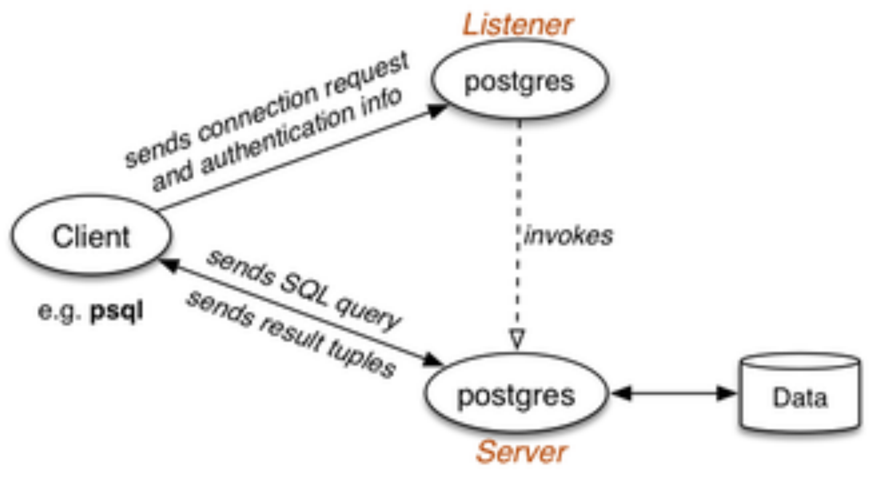
## Life-cycle of a PostgreSQL query

How a PostgreSQL query is executed:

- SQL query string is produced in client
- client establishes connection to PostgreSQL
- dedicated server process attached to client
- SQL query string sent to server process
- *server parses/plans/optimises query*
- server executes query to produce result tuples
- tuples are transmitted back to client
- client disconnects from server

---

### ... Life-cycle of a PostgreSQL query



---

## PostgreSQL server

**`PostgresMain(int argc, char *argv[], ...)`**

- defined in `src/backend/tcop/postgres.c`
- PostgreSQL server (`postgres`) main loop
- performs much setting up/initialisation
- reads and executes requests from client
- using the frontend/backend protocol (Ch.46)
- on `Q` request, evaluates supplied query
- on `X` request, exits the server process

---

### ... PostgreSQL server

As well as handling SQL queries, `PostgresqlMain` also

- handles "utility" commands e.g. `CREATE TABLE`
  - most utility commands modify catalog (e.g. `CREATE` *X*)
  - other commands affect server (e.g. `vacuum`)
- handles `COPY` command
  - special `COPY` mode; context is one table
  - reads line-by-line, treats each line as tuple

- inserts tuples into table; at end, checks constraints

---

## PostgreSQL Data Types

Data types defined in `*.h` files under `src/include/`

Two important data types: **`Node`** and **`List`**

- `Node` provides generic structure for nodes
    - defined in `src/include/nodes/nodes.h`
    - specific node types defined in `src/include/nodes/*.h`
    - functions on nodes defined in `src/backend/nodes/*.c`
    - `Node` types: parse trees, plan trees, execution trees, ...
- `List` provides generic singly-linked list
    - defined in `src/include/nodes/pg_list.h`
    - functions on lists defined in `src/backend/nodes/list.c`

---

## PostgreSQL Query Evaluation

**`exec_simple_query(const char *query_string)`**

- defined in `src/backend/tcop/postgres.c`
- entry point for evaluating SQL queries
- assumes `query_string` is one or more SQL statements
- performs much setting up/initialisation
- parses the SQL string (into one or more parse trees)
- for each parsed query ...
    - perform any rule-based rewriting
    - produces an evaluation plan (optimisation)
    - execute the plan, sending tuples to client

---

### ... PostgreSQL Query Evaluation

**`pg_parse_query(char *sqlStatements)`**

- defined in `src/backend/tcop/postgres.c`
- returns list of parse trees, one for each SQL statement

**`pg_analyze_and_rewrite(Node *parsetree, ...)`**

- defined in `src/backend/tcop/postgres.c`
- converts parsed queries into form suitable for planning

---

### ... PostgreSQL Query Evaluation

Each query is represented by a **`Query`** structure

- defined in `src/include/nodes/parsenodes.h`
- holds all components of the SQL query, including
    - required columns as list of `TargetEntry`s
    - referenced tables as list of `RangeTblEntry`s
    - `where` clause as node in `FromExpr` struct
    - sorting requirements as list of `SortGroupClause`s
- queries may be nested, so forms a tree structure

---

### ... PostgreSQL Query Evaluation

**`pg_plan_queries(querytree_list, ...)`**

- defined in `src/backend/tcop/postgres.c`
- converts analyzed queries into executable "statements"
- uses `pg_plan_query()` to plan each `Query`
  - defined in `src/backend/tcop/postgres.c`
- uses `planner()` to actually do the planning
  - defined in `optimizer/plan/planner.c`

---

## ... PostgreSQL Query Evaluation

Each executable query is represented by a **`PlannedStmt`** node

- defined in `src/include/nodes/plannodes.h`
- contains information for execution of query, e.g.
  - which relations are involved, output tuple structure, etc.
- most important component is a tree of **`Plan`** nodes

Each `Plan` node represents one relational operation

- types: `SeqScan`, `IndexScan`, `HashJoin`, `Sort`, ...
- each `Plan` node also contains cost estimates for operation

---

## ... PostgreSQL Query Evaluation

```
PlannedStmt *planner(Query *parse, ...)
```

- defined in `optimizer/plan/planner.c`
- `subquery_planner()` performs standard transformations
  - e.g. push selection and projection down the tree
- then invokes a cost-based optimiser:
  - choose possible plan (execution order for operations)
  - choose physical operations for this plan
  - estimate cost of this plan (using DB statistics)
  - do this for *sufficient* cases and pick cheapest

---

## ... PostgreSQL Query Evaluation

Queries run in a **`Portal`** environment containing

- the planned statement(s) (trees of `Plan` nodes)
- run-time versions of `Plan` nodes (under `QueryDesc`)
- description of result tuples (under `TupleDesc`)
- overall state of scan through result tuples (e.g. `atStart`)
- other context information (transaction, memory, ...)

`Portal` defined in `src/include/utils/portal.h`

`PortalRun()` function also requires

- destination for query results (e.g. connection to client)
- scan direction (forward or backward)

---

## ... PostgreSQL Query Evaluation

How query evaluation happens in `exec_simple_query()`:

- parse, rewrite and plan ⇒ `PlannedStmts`
- for each `PlannedStmt` ...
- create `Portal` structure
- then insert `PlannedStmt` into `portal`
- then set up `CommandDest` to receive results
- then invoke `PortalRun(portal,...,dest,...)`

- `PortalRun...()` invokes `ProcessQuery(plan,...)`
- `ProcessQuery()` makes `QueryDesc` from `plan`
- then invoke `ExecutorRun(qdesc,...)`
- `ExecutorRun()` invokes `ExecutePlan()` to generate result

---