

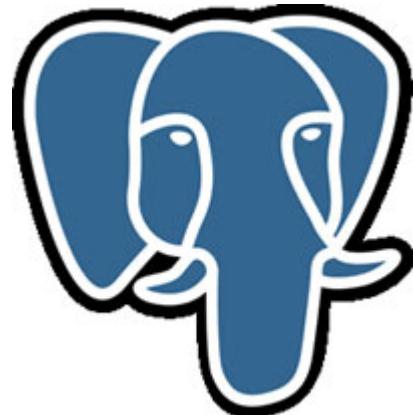
COMP9315 Course Welcome

- [Teaching Staffs](#)
- [Course Admin](#)
- [Important Links](#)
- [Problems?](#)
- [Course Goals](#)
- [Pre-requisites](#)
- [Learning/Teaching](#)
- [Rough Schedule](#)
- [Textbooks](#)
- [Prac Work](#)
- [Assessment Summary](#)
- [Assignments](#)
- [Quizzes](#)
- [Exam](#)
- [General Etiquette](#)
- [Course Outline](#)
- [Others...](#)
- [Week 01](#)
- [Text References](#)

COMP9315 23T1

DBMS Implementation

(Data structures and algorithms inside relational DBMSs)



Convenor: **Dong Wen**

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

❖ Teaching Staffs

Name: **Dong Wen**
Office: K17-507 (send me an email before coming)
Email: dong.wen@unsw.edu.au
Lectures: Weeks 1-4 (Tuesday 12:00-14:00, Thursday 12:00-14:00)
Research: Big Data Processing
Graph Mining and Processing
Distributed Algorithms

Name: **Junhua Zhang**
Lectures: Weeks 5-10 (Tuesday 12:00-14:00, Thursday 12:00-14:00)
Email: junhua.zhang@unsw.edu.au
Research: Graph Analysis
Road Network Processing

❖ Course Admin

Name: **Yiheng Hu**

Email: yiheng.hu@student.unsw.edu.au

Research: Graph Neural Networks

We also have several staffs from DKR (Data and Knowledge Research Group) for marking and consultation.

❖ Important Links

Course Website

<https://webcms3.cse.unsw.edu.au/COMP9315/23T1/>

Moodle for assessment submission

<https://moodle.telt.unsw.edu.au/course/view.php?id=72721>

Ed Forum

<https://edstem.org/au/courses/10647/discussion/>

Special consideration

<https://student.unsw.edu.au/special-consideration>

Educational Adjustments

<https://student.unsw.edu.au/els>

❖ Problems?

Ed Forum for general questions.

- Before posting, check that your query is not already answered.
- The forum entry can be found on Webcms.
- **Enroll the forum using UNSW email.**
- Enrollment link can be found from Webcms notice.
- Do not post assignment/exam solutions on forum.

Email dong.wen@unsw.edu.au for immediate and private questions.

- Technical issues (e.g. problems compiling PostgreSQL)
- Detailed assignment questions (shared-screen debugging)

❖ Course Goals

Introduce you to:

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

Develop skills in:

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

❖ Pre-requisites

We assume that you are already familiar with

- the C language and programming in 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
(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**

PostgreSQL, Assignments and Exam all involve C programming.

❖ Learning/Teaching

What's available for you:

- Textbooks: describe some syllabus topics in detail
- Lectures & Slides: introduce **topics** and show **practical cases**
- Practical/theoretical exercise
- Readings: research papers on selected topics
- General consultations

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

❖ Learning/Teaching (cont)

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)

There are **no** tute/lab classes; use Forum, Email, Consults

- debugging is best done in person (can see full context)

❖ Rough Schedule

- Week 01 welcome, relational algebra, catalogs
- Week 02 storage: disks, buffers, pages, tuples
- Week 03 RA ops: scan, sort, projection
- Week 04 selection: heaps, hashing, indexes
- Week 05 selection: N-d matching, similarity
- Week 06 *no new content, no online sessions*
- Week 07 joins: naive, sort-merge, hash join
- Week 08 query processing, optimisation
- Week 09 transactions: concurrency, recovery
- Week 10 database trends (guest lecturer?)

❖ Textbooks

No official text book; several are suitable ...



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

but not all cover all topics in detail

❖ Prac Work

In this course, we use PostgreSQL v15

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 MacOS at home
- PostgreSQL docs describe how to compile for Windows (WSL)

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

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

❖ Prac Work (cont)

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.

We discuss relevant parts in lectures to help with this.

PostgreSQL books?

- tend to add little to the manual, and cost a lot

❖ Assessment Summary

Your final mark/grade is computed according to the following:

```
ass1    = mark for assignment 1      (out of 15)
ass2    = mark for assignment 2      (out of 20)
quiz    = mark for on-line quizzes   (out of 15)
exam    = mark for final exam       (out of 50)
okExam  = exam > 20/50             (after scaling)

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

❖ Assignments

Schedule of assignment work:

Ass	Description	Due	Marks
1	Storage Management	Week 5	15%
2	Query Processing	Week 9	20%

Assignments will be done individually.

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

Test cases available before submission (extra tests after submission)

Ultimately, submission is via Moodle.

Late penalties apply; **plagiarism checking** will be used

❖ Quizzes

Over the course of the semester ...

- five online quizzes
- taken in your own time (but there are deadlines)
- each quiz is worth a small number of marks
- release on Monday and due on Friday

Quizzes are primarily a review tool to check progress.

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

❖ Exam

Three-hour exam in the exam period.

Exam is held in CSE Labs; **online exam is not available**

The exam is totally open-book ("open-web").

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

- writing **large** programs, running **major** experiments

Everything else is potentially examinable.

Contains: descriptive questions, analysis, small programming exercises.

Exam contributes 50% of the overall mark for this course.

**** 3-hours worth of work; 4-hours allowed to complete**

❖ Exam (cont)

If you cannot attend the final exam ...

- because of documented illness/misadventure
- and you apply for special consideration (within 3 working days)

then you will be offered a Supplementary Exam.

You get **one chance** at passing the exam

- unsw's new fit-to-sit rule applies

Exam hurdle = 20/50 (which is only 40%)

❖ General Etiquette

The course website is a *workplace* platform

- make all communication professional and respectful

Summary: work hard and be nice to each other.

❖ Course Outline

All of the above is described in detail in the Course Outline.

Read it.

It forms a contract between you and me on how this course will run.

Additional resources:

- The Nucleus, in the Library
- Forms for various requests: unsw.to/webforms
- Student Counselling: student.unsw.edu.au/counselling

❖ Others...

We are seeking self-motivated and excellent students (e.g., Mphil and PhDs).

- Big Data (Graph) Processing;
- Distributed Algorithms;
- Streaming/Dynamic Data Processing;
- GPU Acceleration;
- AI4DB, GNN;
- ...

Send me an email with your CV and transcript.

Our group: <https://unswdb.github.io/>

❖ Week 01

Things to Note ...

- Slides available Mon/Tue each week
- Prac P01 now available ... install PostgreSQL

This Week's Topics ...

- DBMS Revision (Architecture, SQL, RelAlg, Catalogs)
- PostgreSQL introduction and installation
- Storage: devices/files ([E16](#), [G11](#), [K9.1](#), [S11](#))

Next Week's Topics ...

- buffer pool, page internals, tuple representation

❖ Text References

References to material in texts use format:

- BookChapter.Section.... (e.g. E2.1.1)

Books are denoted by:

- G = Garcia-Molina/Ullman/Widom
- S = Silberschatz/Korth/Sudarshan
- E = Elmasri/Navathe
- K = Kifer/Bernstein/Lewis

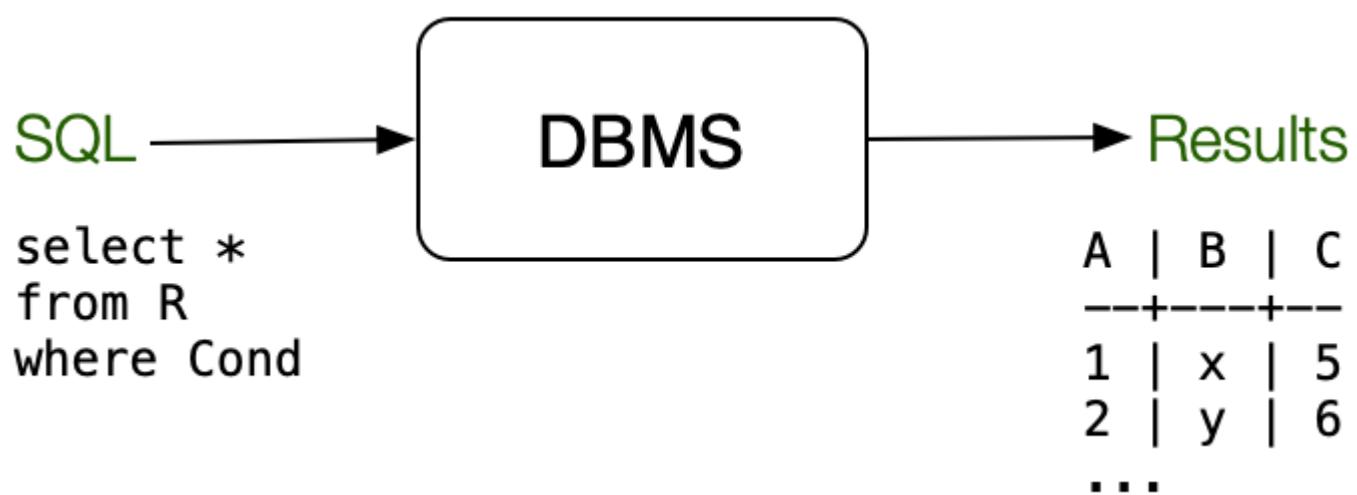
DBMS Overview

- [DBMSs](#)
- [Query Evaluation](#)
- [Mapping SQL to RA](#)
- [Mapping Example](#)
- [Query Cost Estimation](#)
- [Implementations of RA Ops](#)
- [DBMS Architecture](#)

❖ DBMSs

DBMS = DataBase Management System

Our view of the DBMS so far ...



A machine to process SQL queries.

❖ DBMSs (cont)

One view of DB engine: "relational algebra virtual machine"

Machine code for such a machine:

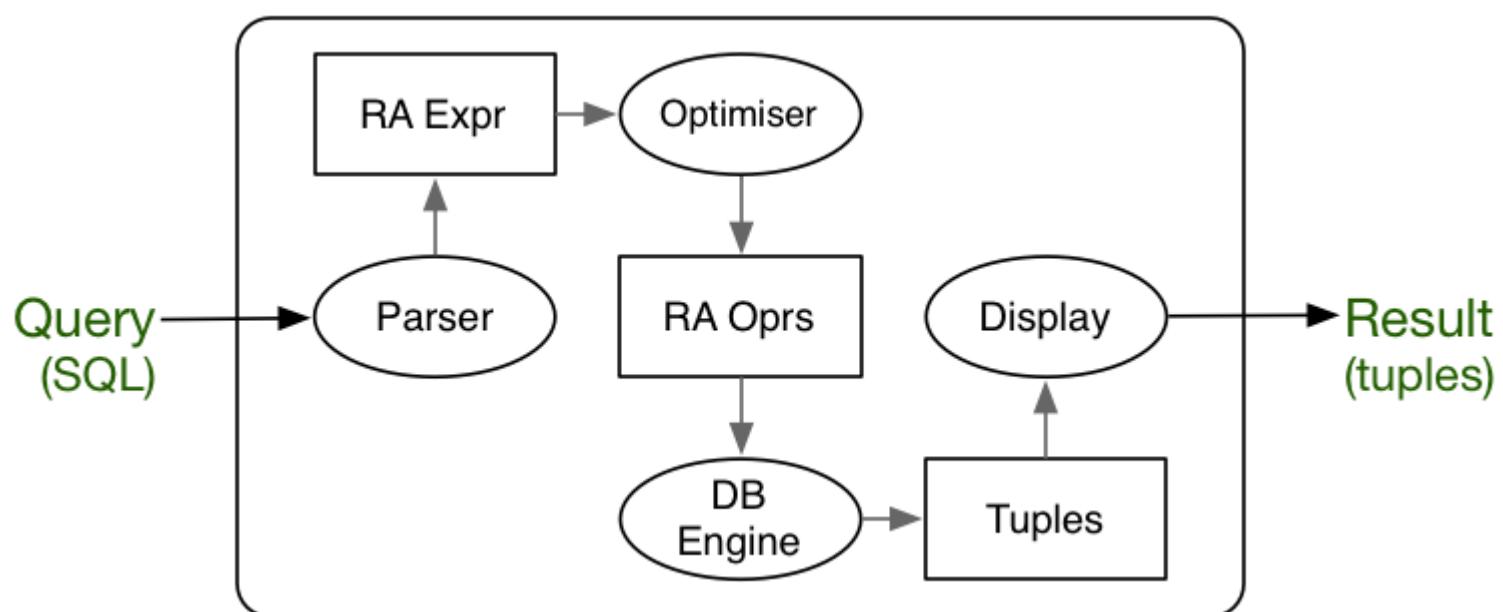
selection (σ)	projection (π)	join (\bowtie, \times)
union (\cup)	intersection (\cap)	difference (-)
sort	insert	delete

For each of these operations:

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

❖ Query Evaluation

The path of a query through its evaluation:



❖ Mapping SQL to RA

Mapping SQL to relational algebra, e.g.

```
-- schema: R(a,b) S(c,d)
select a as x
from   R join S on (b=c)
where   d = 100
-- could be mapped to
Tmp1(a,b,c,d) = R Join[b=c] S
Tmp2(a,b,c,d) = Sel[d=100](Tmp1)
Tmp3(a)         = Proj[a](Tmp2)
Res(x)          = Rename[Res(x)](Tmp3)
```

In general:

- **SELECT** clause becomes *projection*
- **WHERE** condition becomes *selection* or *join*
- **FROM** clause becomes *join*

❖ Mapping Example

Consider the database schema:

```
Person(pid, name, address, ...)  
Subject(sid, code, title, uoc, ...)  
Terms(tid, code, start, end, ...)  
Courses(cid, sid, tid, ...)  
Enrolments(cid, pid, mark, ...)
```

and the query: *Courses with more than 100 students in them?*

which can be expressed in SQL as

```
select s.sid, s.code  
from Course c join Subject s on (c.sid=s.sid)  
      join Enrolment e on (c.cid=e.cid)  
group by s.sid, s.code  
having count(*) > 100;
```

❖ Mapping Example (cont)

The SQL might be compiled to

```
Tmp1(cid,sid,pid) = Course Join[c.cid = e.cid] Enrolment  
Tmp2(cid,code,pid) = Tmp1 Join[t1.sid = s.sid] Subject  
Tmp3(cid,code,nstu) = GroupCount[cid,code](Tmp2)  
Res(cid,code)        = Sel[nstu > 100](Tmp3)
```

or, equivalently

```
Tmp1(cid,code)      = Course Join[c.sid = s.sid] Subject  
Tmp2(cid,code,pid) = Tmp1 Join[t1.cid = e.cid] Enrolment  
Tmp3(cid,code,nstu) = GroupCount[cid,code](Tmp2)  
Res(cid,code)        = Sel[nstu > 100](Tmp3)
```

Which is better?

❖ Query Cost Estimation

The cost of evaluating a query is determined by

- the operations specified in the query execution plan
- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- the size of intermediate results
- then, based on this, cost of secondary storage accesses

Accessing data from disk is the dominant cost in query evaluation

❖ Query Cost Estimation (cont)

Consider **execution plans** for: $\sigma_c(R \bowtie_d S \bowtie_e T)$ where $R(c,d), S(d,e), T(e)$

```
Tmp1(c,d,e) := hash_join[d](R,S)
Tmp2(c,d,e) := sort_merge_join[e](tmp1,T)
Res(c,d,e)   := binary_search[c](Tmp2)
```

or

```
Tmp1(d,e) := sort_merge_join[e](S,T)
Tmp2(c,d,e) := hash_join[d](R,Tmp1)
Res(c,d,e)   := linear_search[c](Tmp2)
```

or

```
Tmp1(c,d) := btree_search[c](R)
Tmp2(c,d,e) := hash_join[d](Tmp1,S)
Res(c,d,e)   := sort_merge_join[e](Tmp2,T)
```

All produce same result, but have different costs.

❖ Implementations of RA Ops

Sorting (quicksort, etc. are not applicable)

- external merge sort (cost $O(N \log_B N)$ with B memory buffers)

Selection (different techniques developed for different query types)

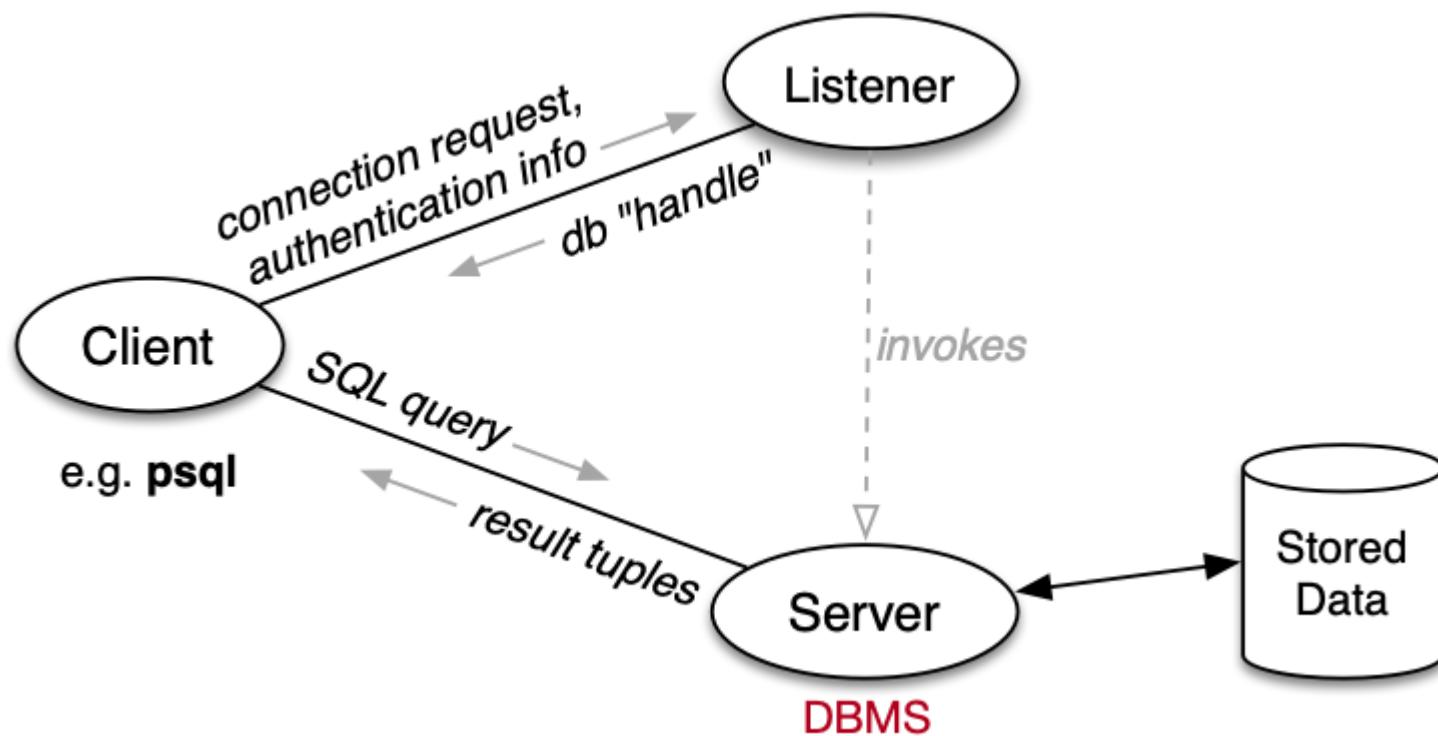
- sequential scan (worst case, cost $O(N)$)
- index-based (e.g. B-trees, cost $O(\log N)$, tree nodes are pages)
- hash-based ($O(1)$ best case, only works for equality tests)

Join (fast joins are critical to success of relational DBMSs)

- nested-loop join (cost $O(N \cdot M)$, buffering can reduce to $O(N+M)$)
- sort-merge join (cost $O(N \log N + M \log M)$)
- hash-join (best case cost $O(N+M \cdot N/B)$, with B memory buffers)

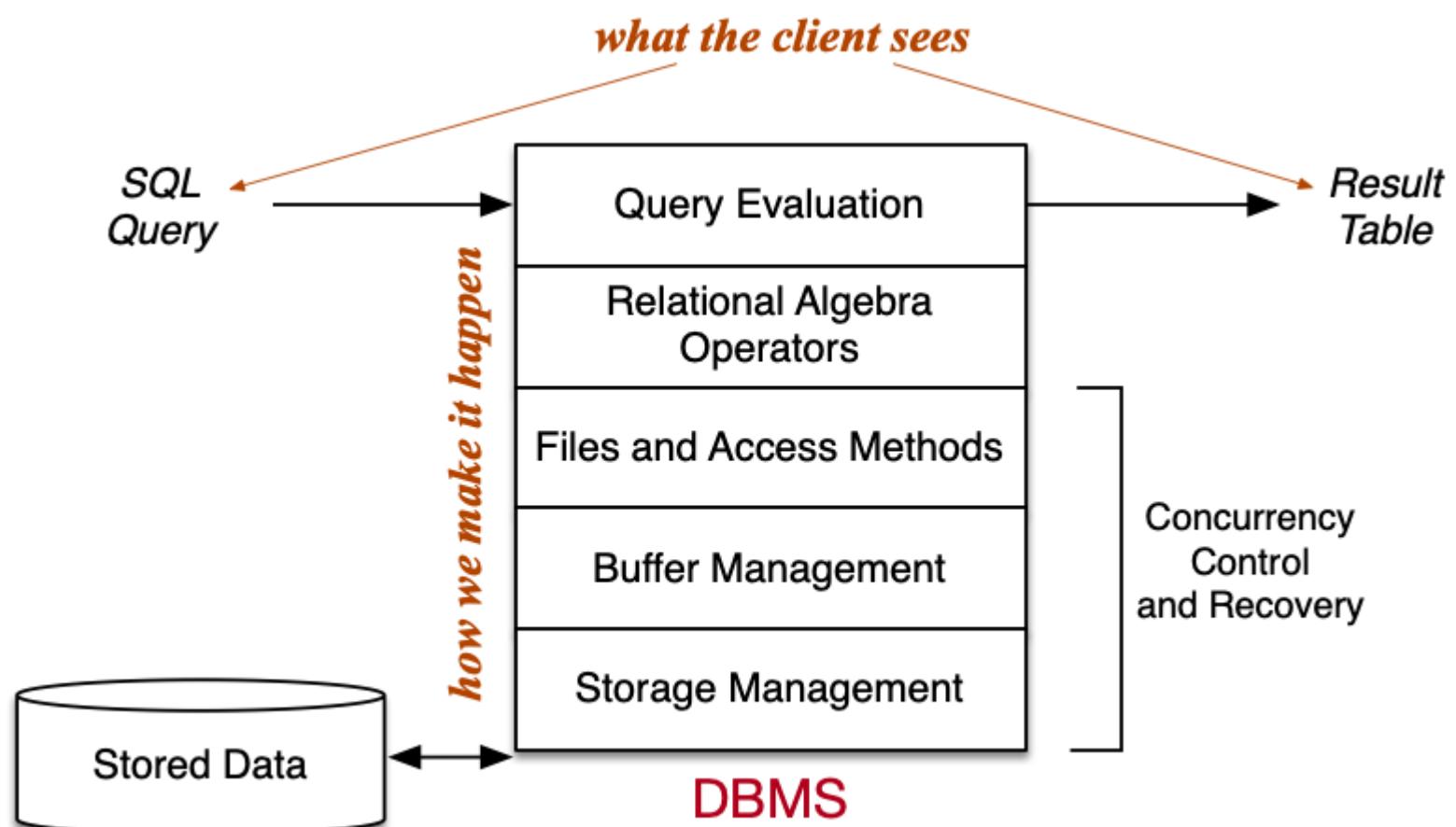
❖ DBMS Architecture

Most RDBMSs are client-server systems:



❖ DBMS Architecture (cont)

Layers within the DBMS server:



PostgreSQL Overview

- [PostgreSQL](#)
- [PostgreSQL Online](#)
- [User View of PostgreSQL](#)
- [PostgreSQL Functionality](#)
- [PostgreSQL Architecture](#)

❖ PostgreSQL

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

- provides a relational engine with:
 - efficient implementation of relational operations
 - transaction processing (concurrent access)
 - backup/recovery (from application/system failure)
 - novel query optimisation (based on genetic algorithm)
 - 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

❖ PostgreSQL Online

Web site: www.postgresql.org

Key developers: Tom Lane, Andres Freund, Bruce Momjian, ...

Full list of developers: postgresql.org/contributors/

Source code on CSE server: [/web/cs9315/23T1/postgresql/postgresql-15.1.tar.bz2](http://web/cs9315/23T1/postgresql/postgresql-15.1.tar.bz2)

Documentation: postgresql.org/docs/15/index.html

❖ User View of PostgreSQL

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

```
$ psql webcms
psql (15.1)
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 access the system
- **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 (cont)

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

- attributes containing arrays of atomic values

```
create table R ( id integer, values integer[] );
insert into R values ( 123, '{5,4,3,2,1}' );
```

- table-valued functions

```
create function f(integer) returns setof TupleType;
```

- multiple languages available for functions

- PLpgsql, Python, Perl, Java, R, Tcl, ...

- function bodies are strings in whatever language

❖ PostgreSQL Functionality (cont)

Other variations in PostgreSQL's **CREATE TABLE**

- **TEMPORARY** tables
- **PARTITION'd** tables
- **GENERATED** attribute values (derived attributes)
- **FOREIGN TABLE** (data stored outside PostgreSQL)
- table type inheritance

```
create table R ( a integer, b text);
create table S ( x float, y float);
create table T inherits ( R, S );
```

❖ PostgreSQL Functionality (cont)

PostgreSQL stored procedures differ from SQL standard:

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

```
create function ( Args ) returns ResultType
as $$ ...
... body of function definition ...
$$ language FunctionBodyLanguage;
```

- where each **Arg** has a *Name* and *Type*

❖ PostgreSQL Functionality (cont)

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 (cont)

Uses 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 (until **vacuum'd**)
- need to check "visibility" of every tuple fetched

PostgreSQL also provides locking to enforce critical concurrency.

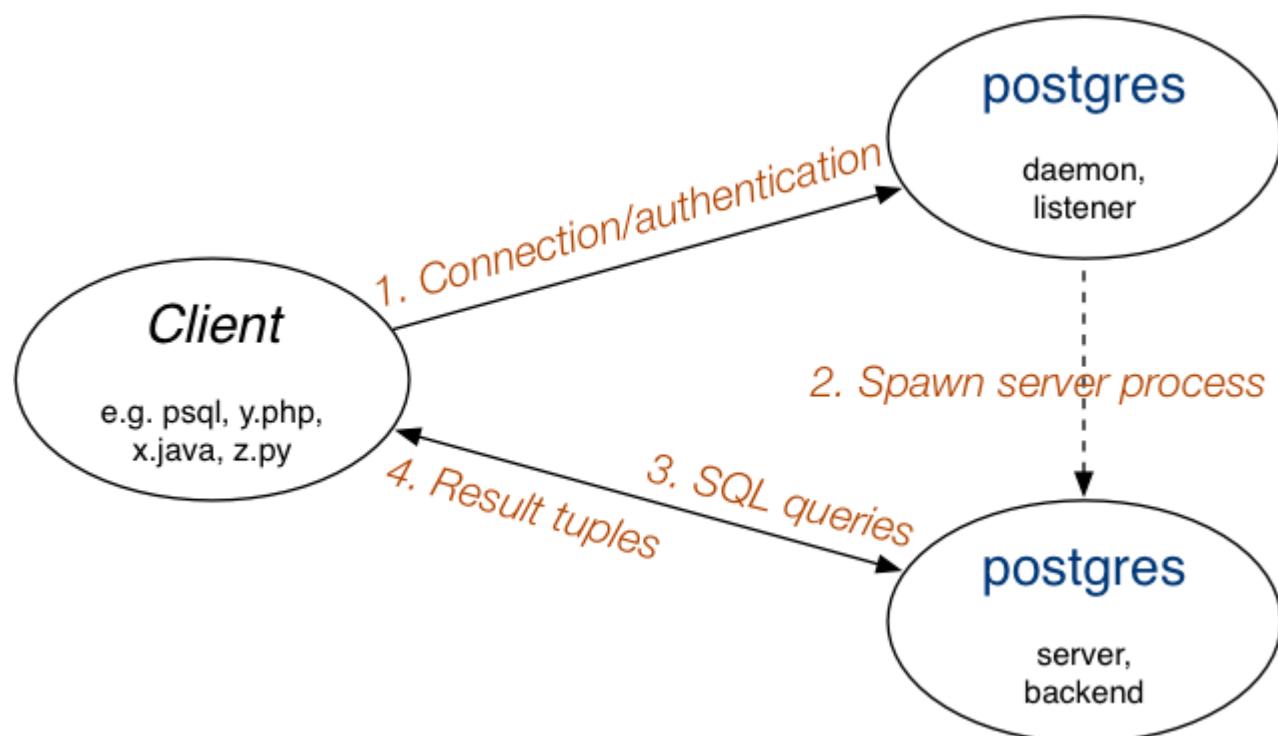
❖ PostgreSQL Functionality (cont)

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 engine
- can add new data types, operators, aggregates, indexes
 - 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 Architecture

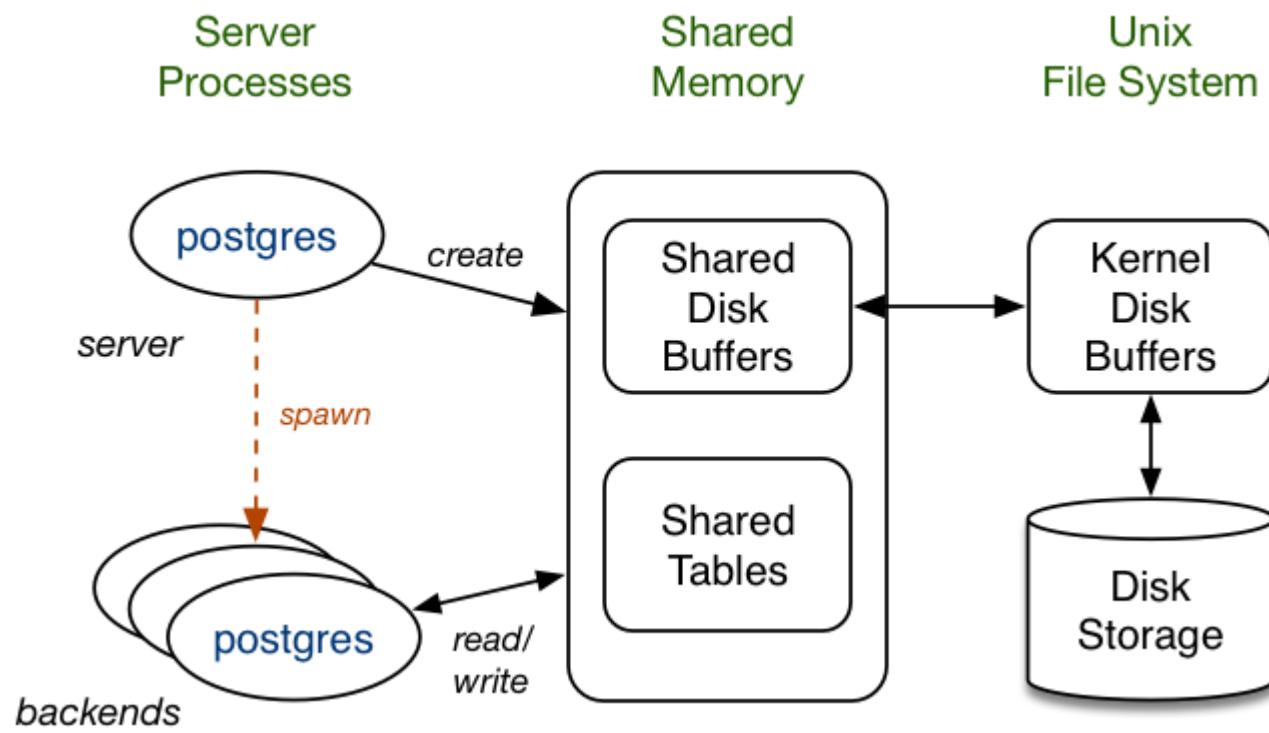
Client/server process architecture:



The listener process is sometimes called **postmaster**

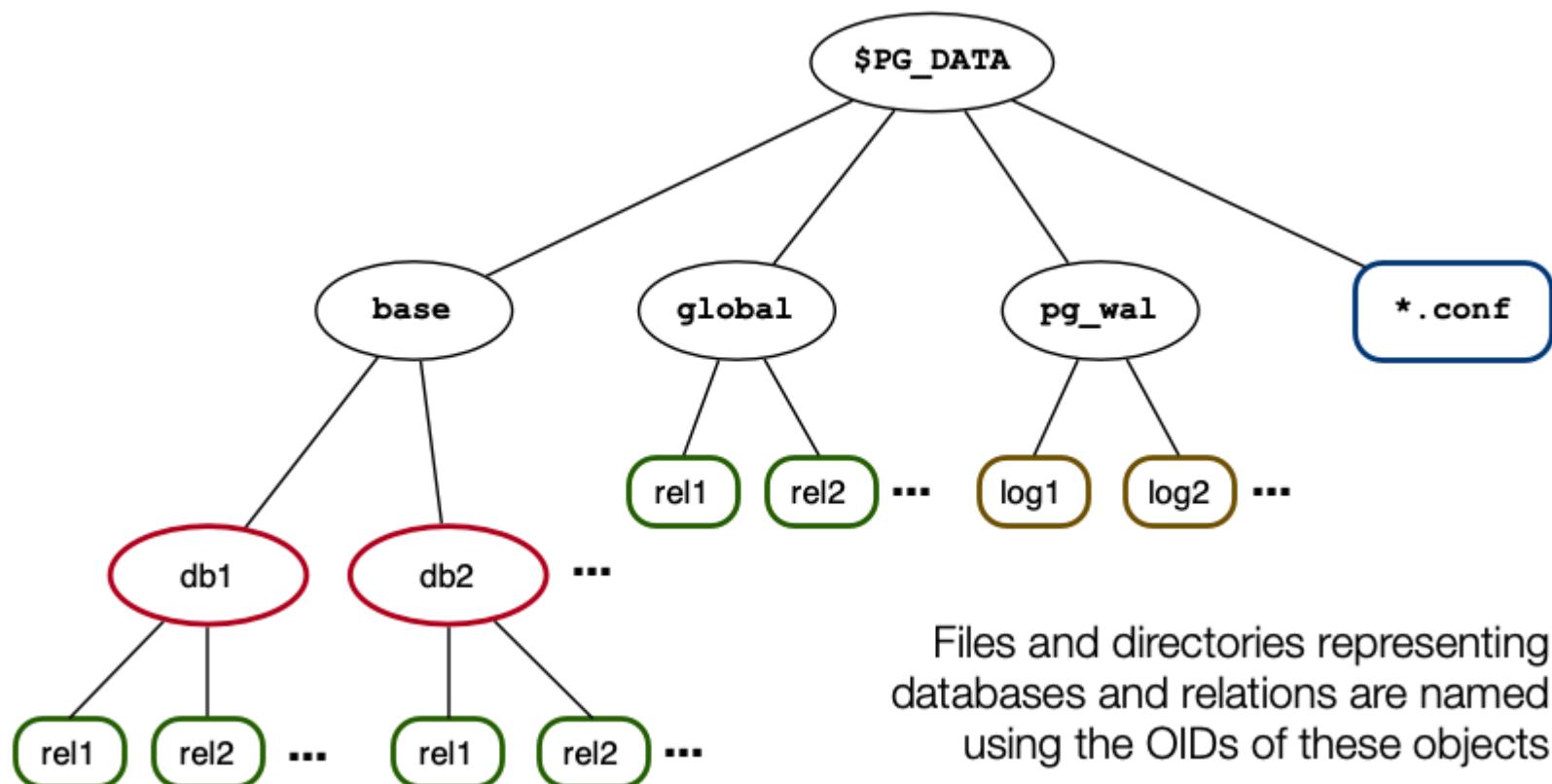
❖ PostgreSQL Architecture (cont)

Memory/storage architecture:



❖ PostgreSQL Architecture (cont)

File-system architecture:



PostgreSQL Install

- [Installing/Using PostgreSQL](#)
- [Using PostgreSQL for Assignments](#)
- [Installing PostgreSQL](#)
- [Before Installing ...](#)

❖ Installing/Using PostgreSQL

Environment setup for running PostgreSQL in COMP9315:

- **nw-syd-vxdb** a CSE server for database studies.
- See Prac Exercise 1 for details

Some hints/tips:

- Read instructions ... **read log/error messages** ... debug
- Give complete context when asking for help (OS, settings, etc)
- Don't work on other CSE servers
 - **ssh zID@nw-syd-vxdb.cse.unsw.edu.au**
 - binaries compiled on vxdb only run on vxdb
 - other incompatible versions of PostgreSQL exist
try **which psql** ... if not the version under **/localstorage/zID ...**
- Stop the PostgreSQL server after playing with it

❖ Using PostgreSQL for Assignments

If changes don't modify storage structures ...

```
$ edit source code
$ pg_ctl stop
$ make
$ make install
$ pg_ctl start -l $PGDATA/log
  # run tests, analyse results, ...
$ pg_ctl stop
```

In this case, existing databases will continue to work ok.

❖ Using PostgreSQL for Assignments (cont)

If changes modify storage structures ...

```
$ edit source code
$ save a copy of postgresql.conf
$ pg_dump -O -x testdb > testdb.dump
$ pg_ctl stop
$ make
$ make install
$ rm -fr $PGDATA
$ initdb
$ restore postgresql.conf
$ pg_ctl start -l $PGDATA/log
$ createdb testdb
$ psql testdb -f testdb.dump
# run tests and analyse results
```

Old databases will not work with the new server.

❖ Using PostgreSQL for Assignments (cont)

Troubleshooting ...

- read the **\$PGDATA/log** file
- which socket file are you trying to connect to?
- check the **\$PGDATA** directory for socket files
- remove **postmaster.pid** if sure no server running
- ...

Prac Exercise P01 has useful tips down the bottom

❖ Installing PostgreSQL

PostgreSQL is also available via the COMP9315 web site.

<https://www.cse.unsw.edu.au/~cs9315/23T1/postgresql/postgresql-15.1.tar.bz2>

File: **src.tar.bz2** is ~23MB **

Unpacked, source code + binaries is ~130MB **

Path on the CSE server:

/web/cs9315/23T1/postgresql/postgresql-15.1.tar.bz2

If using on CSE, do not put it under your home directory

Place it under **/localstorage/\$USER/** of the **vxdb** server which has large size quota

❖ Before Installing ...

If you have databases from previous DB courses

- the databases may not work under v12.5
- to preserve them, use dump/restore

E.g.

```
... login to the server or your machine ...
... run your old server for the last time ...
$ pg_dump -O -x myFavDB > /srvr/YOU/myFavDB.dump
... stop your old server for the last time ...
... remove data from your old server ...
$ rm -fr /srvr/YOU/pgsql
... install and run your new PostgreSQL 12.5 server ...
$ createdb myFavDB
$ psql myFavDB -f /srvr/YOU/myFavDB.dump
... your old database is restored under 12.5 ...
```


Catalogs

- [Database Objects](#)
- [Catalogs](#)
- [Representing Databases](#)
- [Representing Tables](#)

❖ Database Objects

RDBMSs manage different kinds of objects

- databases, schemas, tablespaces
- relations/tables, attributes, tuples/records
- constraints, assertions
- views, stored procedures, triggers, rules

Many objects have names (and, in PostgreSQL, some have OIDs).

How are the different types of objects represented?

How do we go from a name (or OID) to bytes stored on disk?

❖ Catalogs

Consider what information the RDBMS needs about relations:

- name, owner, primary key of each relation
- name, data type, constraints for each attribute
- authorisation for operations on each relation

Similarly for other DBMS objects (e.g. views, functions, triggers, ...)

This information is stored in the **system catalog tables**

Standard for catalogs in SQL:2003: **INFORMATION_SCHEMA**

❖ Catalogs (cont)

The catalog is affected by several types of SQL operations:

- **create Object as Definition**
- **drop Object ...**
- **alter Object Changes**
- **grant Privilege on Object**

where *Object* is one of table, view, function, trigger, schema, ...

E.g. **drop table Groups;** produces something like

```
delete from Tables  
where schema = 'public' and name = 'groups';
```

❖ Catalogs (cont)

In PostgreSQL, the system catalog is available to users via:

- special commands in the **psql** shell (e.g. **\d**)
- SQL standard **information_schema**

e.g. **select * from information_schema.tables;**

The low-level representation is available to sysadmins via:

- a global schema called **pg_catalog**
- a set of tables/views in that schema (e.g. **pg_tables**)

❖ Catalogs (cont)

You can explore the PostgreSQL catalog via `psql` commands

- `\d` gives a list of all tables and views
- `\d Table` gives a schema for `Table`
- `\df` gives a list of user-defined functions
- `\df+ Function` gives details of `Function`
- `\ef Function` allows you to edit `Function`
- `\dv` gives a list of user-defined views
- `\d+ View` gives definition of `View`

You can also explore via SQL on the catalog tables

❖ Catalogs (cont)

A PostgreSQL installation (cluster) typically has many DBs

Some catalog information is global, e.g.

- catalog tables defining: databases, users, ...
- one copy of each such table for the whole PostgreSQL installation
- shared by all databases in the cluster (in **PGDATA/pg_global**)

Other catalog information is local to each database, e.g

- schemas, tables, attributes, functions, types, ...
- separate copy of each "local" table in each database
- a copy of many "global" tables is made on database creation

❖ Catalogs (cont)

Side-note: PostgreSQL tuples contain

- owner-specified attributes (from `create table`)
- system-defined attributes

oid unique identifying number for tuple (optional)

tableoid which table this tuple belongs to

xmin/xmax which transaction created/deleted tuple (for MVCC)

OIDs are used as primary keys in many of the catalog tables.

❖ Representing Databases

Above the level of individual DB schemata, we have:

- **databases** ... represented by **pg_database**
- **schemas** ... represented by **pg_namespace**
- **table spaces** ... represented by **pg_tablespace**

These tables are global to each PostgreSQL cluster.

Keys are names (strings) and must be unique within cluster.

❖ Representing Databases (cont)

pg_database contains information about databases:

- **oid, datname, datdba, datacl[], encoding, ...**

pg_namespace contains information about schemata:

- **oid, nspname, nspowner, nspacl[]**

pg_tablespace contains information about tablespaces:

- **oid, spcname, spcowner, spcacl[]**

PostgreSQL represents access via array of access items:

Role=Privileges/Grantor

where Privileges is a string enumerating privileges, e.g.

Jake=rwad/Simon

❖ Representing Tables

Representing one table needs tuples in several catalog tables.

Due to O-O heritage, base table for tables is called **pg_class**.

The **pg_class** table also handles other "table-like" objects:

- views ... represents attributes/domains of view
- composite (tuple) types ... from **CREATE TYPE AS**
- sequences, indexes (top-level defn), other "special" objects

All tuples in **pg_class** have an OID, used as primary key.

Some fields from the **pg_class** table:

- **oid, relname, relnamespace, reltype, relowner**
- **relkind, reltuples, relnatts, relhaspkey, relacl, ...**

❖ Representing Tables (cont)

Details of catalog tables representing database tables

pg_class holds core information about tables

- **relname, relnamespace, reltype, relowner, ...**
- **relkind, relnatts, relhaspkey, relacl[], ...**

pg_attribute contains information about attributes

- **attrelid, attname, atttypid, attnum, ...**

pg_type contains information about types

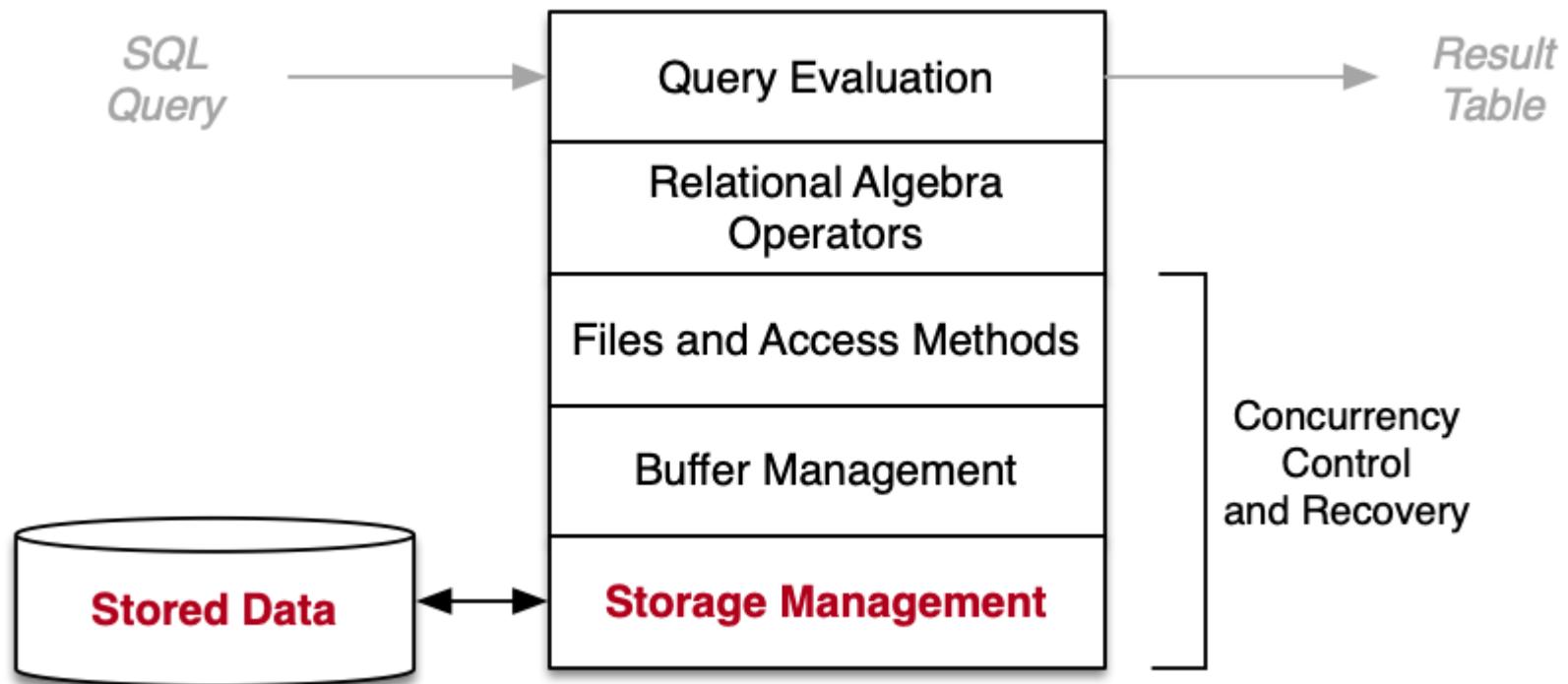
- **typname, typnamespace, typowner, typlen, ...**
- **typtype, typrelid, typinput, typoutput, ...**

Storage Management

- [Storage Management](#)
- [Storage Technology](#)
- [Views of Data in Query Evaluation](#)
- [Storage Management](#)
- [Cost Models](#)

❖ Storage Management

Lowest levels of DBMS related to storage management:



❖ Storage Technology

Persistent storage is

- large, cheap, relatively slow, accessed in blocks
- used for long-term storage of data

Computational storage is

- small, expensive, fast, accessed by byte/word
- used for all analysis of data

Access cost HDD:RAM \approx 100000:1, e.g.

- 10ms to read block containing two tuples
- 1 μ s to compare fields in two tuples

❖ Storage Technology (cont)

Hard disk drives (HDD) are well-established, cheap, high-volume, ...

- spinning magnetic medium
- access requires moving r/w head to position
- transfers blocks of data (e.g. 1KB)

Latency: move to track + spin to block = ~10ms (avg)

Volume: one HDD can store up to 20TB (typically 4TB/8TB/...)

Summary: very large, persistent, slow, block-based transfer

❖ Storage Technology (cont)

Solid state drives (SSD) are modern, high-volume devices ...

- faster than HDDs, no latency
- can read single items
- update requires block erase then write
- over time, writes "wear out" blocks
- require controllers that spread write load

Volume: one SSD can store up to 8TB (typically 1TB/2TB/...)

Summary: large, persistent, fast, (partly) block-based transfer

❖ Storage Technology (cont)

Comparison of storage device properties:

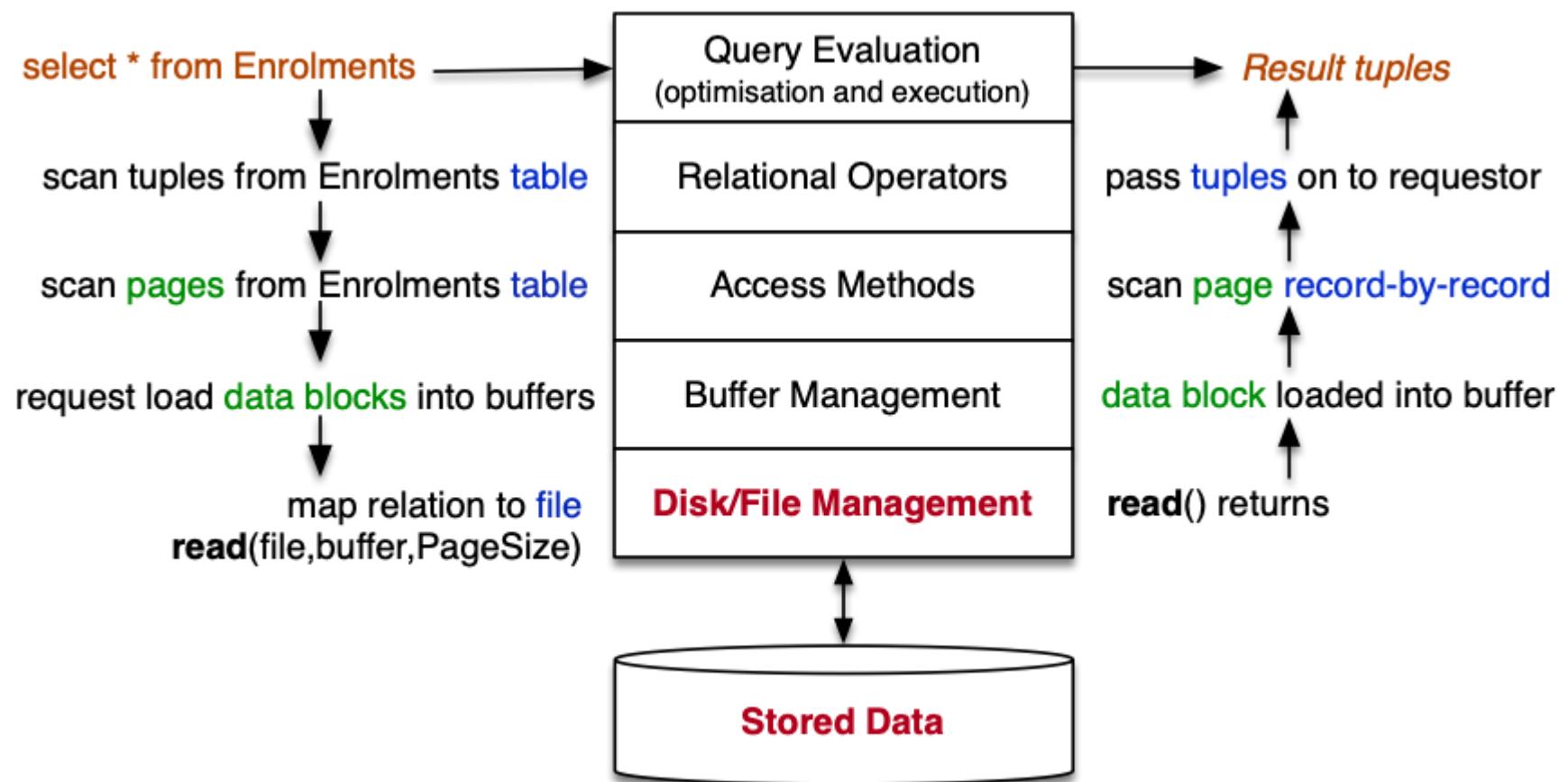
	RAM	HDD	SDD
Capacity	~ 32GB	~ 8TB	~ 2TB
Cost/byte	~ \$10 / GB	~ \$40 / TB	~ \$200 / TB
Read latency	~ 1µs	~ 10ms	~ 50µs
Write latency	~ 1µs	~ 10ms	~ 900µs
Read unit	byte	block (e.g. 1KB)	byte
Writing	byte	write a block	write on empty block

❖ Storage Technology (cont)

Aims of storage management in DBMS:

- provide view of data as collection of pages/tuples
- map from database objects (e.g. tables) to disk files
- manage transfer of data to/from disk storage
- use buffers to minimise disk/memory transfers
- interpret loaded data as tuples/records
- basis for file structures used by access methods

❖ Views of Data in Query Evaluation



❖ Views of Data in Query Evaluation (cont)

Representing database objects during query execution:

- **DB** (handle on an authorised/opened database)
- **Rel** (handle on an opened relation)
- **Page** (memory buffer to hold contents of disk block)
- **Tuple** (memory holding data values from one tuple)

Addressing in DBMSs:

- **PageID = FileID+Offset** ... identifies a block of data
 - where **Offset** gives location of block within file
- **TupleID = PageID+Index** ... identifies a single tuple
 - where **Index** gives location of tuple within page

❖ Storage Management

Topics in storage management ...

- Disks and Files
 - performance issues and organisation of disk files
- Buffer Management
 - using caching to improve DBMS system throughput
- Tuple/Page Management
 - how tuples are represented within disk pages
- DB Object Management (Catalog)
 - how tables/views/functions/types, etc. are represented

❖ Cost Models

Throughout this course, we compare costs of DB operations

Important aspects in determining cost:

- data is always transferred to/from disk as whole blocks (pages)
- cost of manipulating tuples in memory is negligible
- overall cost determined primarily by #data-blocks read/written

Complicating factors in determining costs:

- not all page accesses require disk access (buffer pool)
- tuples typically have variable size (tuples/page ?)

More details later ...

Week 1 Exercises

- [Exercise 1: Unix File I/O \(revision\)](#)
- [Exercise 2: PostgreSQL files](#)
- [PostgreSQL servers](#)
- [Exercise 3: Table Statistics](#)
- [Exercise 4: Extracting a Schema](#)
- [Exercise 5: Enumerated Types](#)
- [Relational Algebra \(RA\)](#)
- [Exercise 6: Relational Algebra \(RA\)](#).

❖ Exercise 1: Unix File I/O (revision)

Write a C program that reads a file, block-by-block.

Command-line parameters:

- block size in bytes
- name of input file

Use low-level C operations: **open**, **read**.

Count and display how many blocks/bytes read.

❖ Exercise 2: PostgreSQL files

Explore the PostgreSQL **pg_catalog** to determine ...

How many tables are in the catalog?

What users? namespaces? tablespaces? are there

What information is stored about tables, attributes, types?

How many tables are in the **public** schema?

Which directory contains the database X ?

Which files in that directory correspond to table Y ?

❖ PostgreSQL servers

Script to create your `/localStorage/$USER` is now working

PostgreSQL-14.1 has removed the `Ready to install.` message.

`Leaving directory` is not an error.

What the `Makefile` does ...

- changes into a directory
- runs the `Makefile` in that directory
- leaves that directory
- repeat until all directories are done

❖ PostgreSQL servers (cont)

Some people have been asking (paraphrasing)

- can I just use a pre-compiled binary version of PostgreSQL

That would be ok if we were just **using** PostgreSQL.

COMP9315 requires you to **modify** PostgreSQL.

You can't do this if you start from a binary; you need the source code.

❖ Exercise 3: Table Statistics

Using the PostgreSQL catalog, write a PLpgsql function

- to return table name and #tuples in table
- for all tables in the **public** schema

```
create type TableInfo as
  (tabname text, ntuples int);

create function pop()
  returns setof TableInfo
as $$
...
$$ language plpgsql;
```

Hint: you will need to use dynamically-generated queries.

❖ Exercise 4: Extracting a Schema

Write a PLpgsql function:

- **function schema() returns setof text**
- giving a list of table schemas in the **public** schema

It should behave as follows:

```
db=# select * from schema();
      tables
-----
table1(x, y, z)
table2(a, b)
table3(id, name, address)
...
```

❖ Exercise 5: Enumerated Types

PostgreSQL allows you to define enumerated types, e.g.

```
create type Mood as enum ('sad', 'happy');
```

Creates a type with two ordered values '**sad**' < '**happy**'

What is created in the catalog for the above definition?

Hint:

```
pg_type(oid, typename, typelen, typetype, ...)  
pg_enum(oid, enumtypid, enumlabel)
```

❖ Relational Algebra (RA)

Relational algebra is the "machine language" of RDBMSs

SQL is translated to RA before being executed

Reminder (I hope) ...

- Select ... selects a subset of tuples based on a condition
- Project ... selects a subset of fields based on a list
- Join ... "merges" two tables according to a condition

❖ Exercise 6: Relational Algebra (RA)

Translate each of the following SQL statements to RA

- **select * from R**
- **select a,b from R**
- **select * from R where a > 5**
- **select * from R join S on R.a = S.y**

Assume a schema: **R(a,b,c), S(x,y)**

Indicate: the fields and # tuples in the result

File Management

- [File Management](#)
- [DBMS File Organisation](#)
- [Single-file DBMS](#)
- [Single-file Storage Manager](#)
- [Example: Scanning a Relation](#)
- [Single-File Storage Manager](#)
- [Multiple-file Disk Manager](#)
- [DBMS File Parameters](#)

❖ File Management

Aims of file management subsystem:

- organise layout of data within the filesystem
- handle mapping from database ID to file address
- transfer blocks of data between buffer pool and filesystem
- also attempts to handle file access error problems (retry)

Builds higher-level operations on top of OS file operations.

❖ File Management (cont)

Typical file operations provided by the operating system:

```
fd = open(fileName, mode)
    // open a named file for reading/writing/appending
close(fd)
    // close an open file, via its descriptor
nread = read(fd, buf, nbytes)
    // attempt to read data from file into buffer
nwritten = write(fd, buf, nbytes)
    // attempt to write data from buffer to file
lseek(fd, offset, seek_type)
    // move file pointer to relative/absolute file offset
fsync(fd)
    // flush contents of file buffers to disk
```

❖ DBMS File Organisation

How is data for DB objects arranged in the file system?

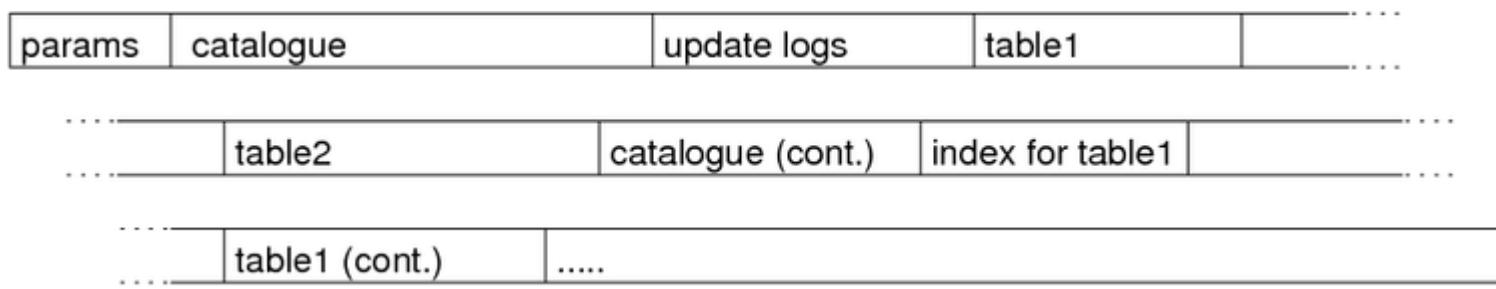
Different DBMSs make different choices, e.g.

- by-pass the file system and use a raw disk partition
- have a single very large file containing all DB data
- have several large files, with tables spread across them
- have multiple data files, one for each table
- have multiple files for each table
- etc.

❖ Single-file DBMS

Consider a single file for the entire database (e.g. SQLite)

Objects are allocated to regions (segments) of the file.



If an object grows too large for allocated segment, allocate an extension.

What happens to allocated space when objects are removed?

❖ Single-file DBMS (cont)

Allocating space in Unix files is easy:

- simply seek to the place you want and write the data
- if nothing there already, data is appended to the file
- if something there already, it gets overwritten

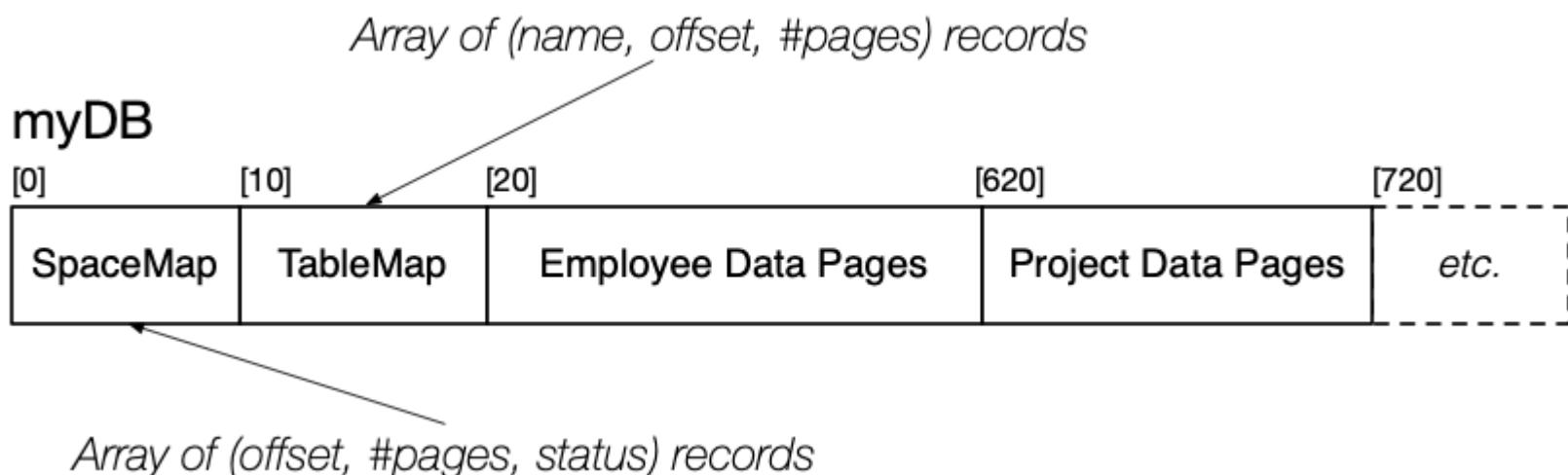
If the seek goes way beyond the end of the file:

- Unix does not (yet) allocate disk space for the "hole"
- allocates disk storage only when data is written there

With the above, a disk/file manager is easy to implement.

❖ Single-file Storage Manager

Consider the following simple single-file DBMS layout:



E.g.

SpaceMap = [(0,10,U), (10,10,U), (20,600,U), (620,100,U), (720,20,F)]

TableMap = [("employee",20,500), ("project",620,40)]

❖ Single-file Storage Manager (cont)

Each file segment consists of a number fixed-size blocks

The following data/constant definitions are useful

```
#define PAGESIZE 2048      // bytes per page

typedef long PageId;      // PageId is block index
                          // pageOffset=PageId*PAGESIZE

typedef char *Page;        // pointer to page/block buffer
```

Typical **PAGESIZE** values: 1024, 2048, 4096, 8192

❖ Single-file Storage Manager (cont)

Possible storage manager data structures for opened DBs & Tables

```
typedef struct DBrec {
    char *dbname;          // copy of database name
    int fd;                // the database file
    SpaceMap map;          // map of free/used areas
    TableMap names;        // map names to areas + sizes
} *DB;

typedef struct Relrec {
    char *relname;         // copy of table name
    int start;             // page index of start of table data
    int npages;            // number of pages of table data
    ...
} *Rel;
```

❖ Example: Scanning a Relation

With the above disk manager, a query like

```
select name from Employee
```

might be implemented as

```
DB db = openDatabase("myDB");
Rel r = openRelation(db, "Employee");
Page buffer = malloc(PAGESIZE*sizeof(char));
for (int i = 0; i < r->npages; i++) {
    PageId pid = r->start+i;
    get_page(db, pid, buffer);
    for each tuple in buffer {
        get tuple data and extract name
        add (name) to result tuples
    }
}
```

❖ Single-File Storage Manager

```
// start using DB, buffer meta-data
DB openDatabase(char *name) {
    DB db = new(struct DBrec);
    db->dbname = strdup(name);
    db->fd = open(name,O_RDWR);
    db->map = readSpaceTable(db->fd);
    db->names = readNameTable(db->fd);
    return db;
}
// stop using DB and update all meta-data
void closeDatabase(DB db) {
    writeSpaceTable(db->fd,db->map);
    writeNameTable(db->fd,db->map);
    fsync(db->fd);
    close(db->fd);
    free(db->dbname);
    free(db);
}
```

❖ Single-File Storage Manager (cont)

```
// set up struct describing relation
Rel openRelation(DB db, char *rname) {
    Rel r = new(struct Relrec);
    r->relname = strdup(rname);
    // get relation data from map tables
    r->start = ...;
    r->npages = ...;
    return r;
}

// stop using a relation
void closeRelation(Rel r) {
    free(r->relname);
    free(r);
}
```

❖ Single-File Storage Manager (cont)

```
// assume that Page = byte[PageSize]
// assume that PageId = block number in file

// read page from file into memory buffer
void get_page(DB db, PageId p, Page buf) {
    lseek(db->fd, p*PAGESIZE, SEEK_SET);
    read(db->fd, buf, PAGESIZE);
}

// write page from memory buffer to file
void put_page(Db db, PageId p, Page buf) {
    lseek(db->fd, p*PAGESIZE, SEEK_SET);
    write(db->fd, buf, PAGESIZE);
}
```

❖ Single-File Storage Manager (cont)

Managing contents of space mapping table can be complex:

```
// assume an array of (offset,length,status) records

// allocate n new pages
PageId allocate_pages(int n) {
    if (no existing free chunks are large enough) {
        int endfile = lseek(db->fd, 0, SEEK_END);
        addNewEntry(db->map, endfile, n);
    } else {
        grab "worst fit" chunk
        split off unused section as new chunk
    }
    // note that file itself is not changed
}
```

❖ Single-File Storage Manager (cont)

Similar complexity for freeing chunks

```
// drop n pages starting from p
void deallocate_pages(PageId p, int n) {
    if (no adjacent free chunks) {
        markUnused(db->map, p, n);
    } else {
        merge adjacent free chunks
        compress mapping table
    }
    // note that file itself is not changed
}
```

Changes take effect when **closeDatabase()** executed.

❖ Multiple-file Disk Manager

Most DBMSs don't use a single large file for all data.

They typically provide:

- multiple files partitioned physically or logically
- mapping from DB-level objects to files (e.g. via catalog meta-data)

Precise file structure varies between individual DBMSs.

Using multiple files (one file per relation) can be easier, e.g.

- adding a new relation
- extending the size of a relation
- computing page offsets within a relation

❖ Multiple-file Disk Manager (cont)

Example of single-file vs multiple-file:

Single file database



$Page[i]$ offset = ??

Multi file database

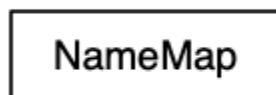
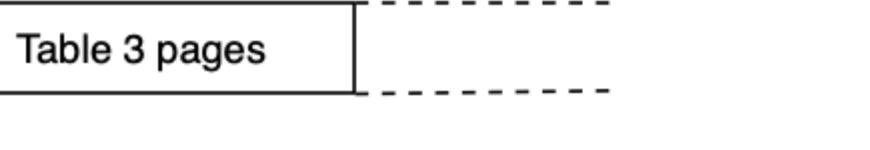
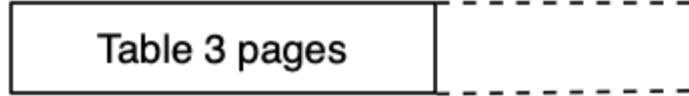


Table 1 pages

$Page[i]$ offset = $i * PageSize$

Table 2 pages



Consider how you would compute file offset of page[i] in table[1] ...

❖ Multiple-file Disk Manager (cont)

Structure of **PageId** for data pages in such systems ...

If system uses one file per table, **PageId** contains:

- relation identifier (which can be mapped to filename)
- page number (to identify page within the file)

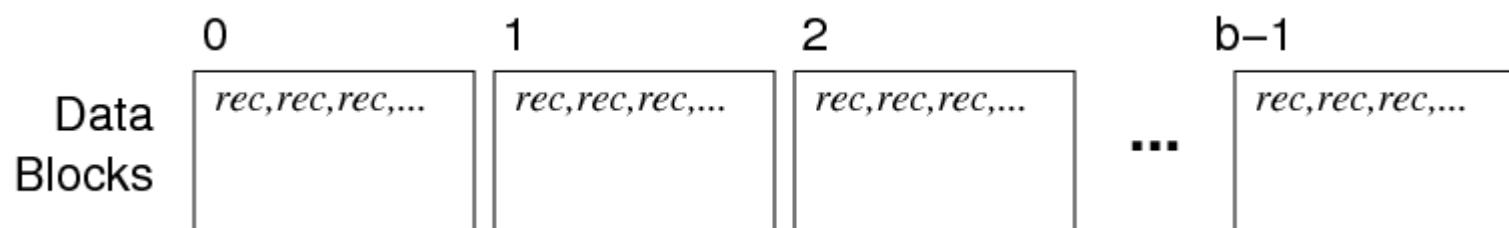
If system uses several files per table, **PageId** contains:

- relation identifier
- file identifier (combined with relid, gives filename)
- page number (to identify page within the file)

❖ DBMS File Parameters

Our view of relations in DBMSs:

- a relation is a set of r tuples, with average size R bytes
- the tuples are stored in b data pages on disk
- each page has size B bytes and contains up to c tuples
- data is transferred disk \leftrightarrow memory in whole pages
- cost of disk \leftrightarrow memory transfer T_r, T_w dominates other costs



❖ DBMS File Parameters (cont)

Typical DBMS/table parameter values:

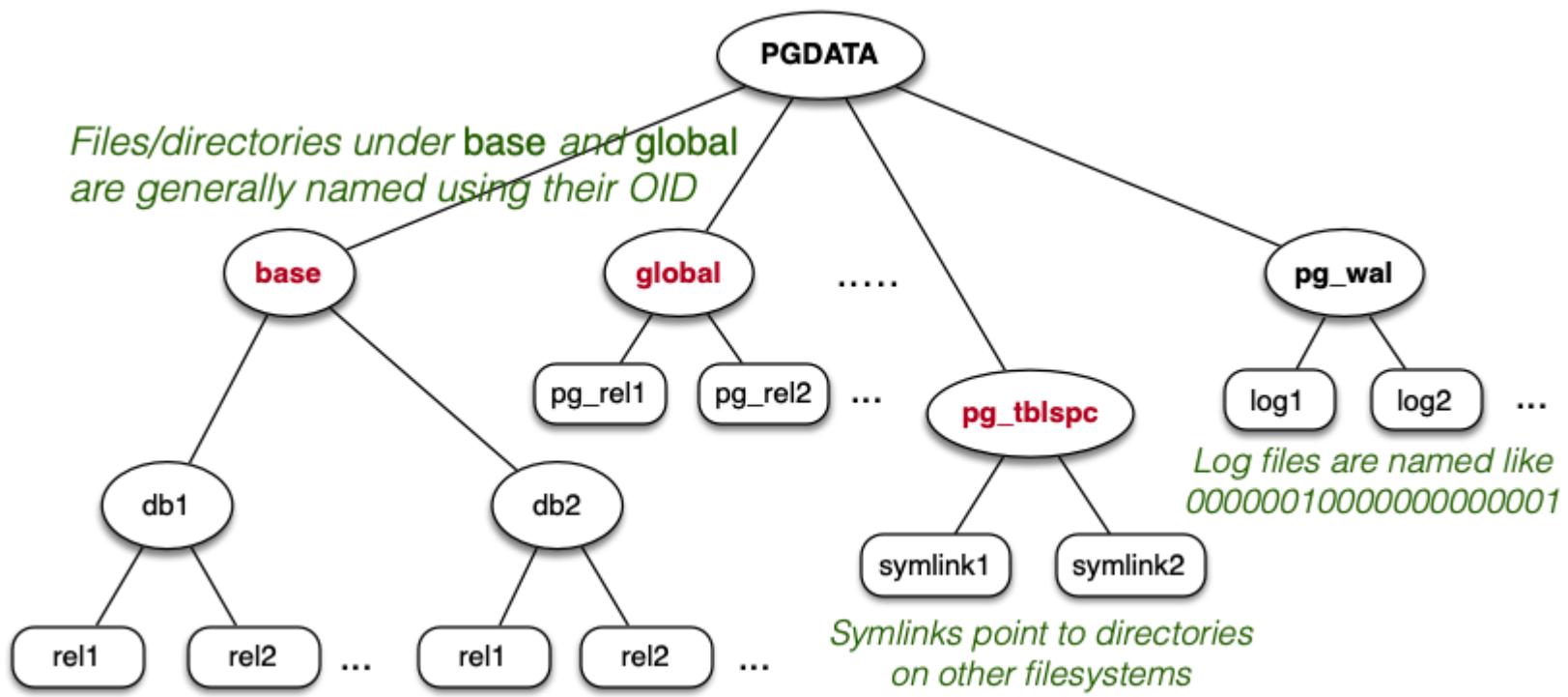
Quantity	Symbol	E.g. Value
total # tuples	r	10^6
record size	R	128 bytes
total # pages	b	10^5
page size	B	8192 bytes
# tuples per page	c	60
page read/write time	T_r, T_w	10 msec
cost to process one page in memory	-	≈ 0

PostgreSQL File Manager

- [PostgreSQL File Manager](#)
- [Relations as Files](#)
- [File Descriptor Pool](#)
- [File Manager](#)

❖ PostgreSQL File Manager

PostgreSQL uses the following file organisation ...



❖ PostgreSQL File Manager (cont)

Components of storage subsystem:

- mapping from relations to files (**RelFileNode**)
- abstraction for open relation pool (**storage/smgr**)
- functions for managing files (**storage/smgr/md.c**)
- file-descriptor pool (**storage/file**)

PostgreSQL has two basic kinds of files:

- heap files containing data (tuples)
- index files containing index entries

Note: **smgr** designed for many storage devices; only disk handler provided

❖ Relations as Files

PostgreSQL identifies relation files via their OIDs.

The core data structure for this is **RelFileNode**:

```
// include/storage/relfilenode.h
typedef struct RelFileNode {
    Oid    spcNode;    // tablespace
    Oid    dbNode;     // database
    Oid    relNode;    // relation
} RelFileNode;
```

Global (shared) tables (e.g. **pg_database**) have

- **spcNode == GLOBALTABLESPACE_OID**
- **dbNode == 0**

❖ Relations as Files (cont)

The **relpath** function maps **RelFileNode** to file:

```
// include/common/relpath.h
// common/relpath.c
char *relpath(RelFileNode r) // simplified
{
    char *path = malloc(ENOUGH_SPACE);

    if (r.spcNode == GLOBALTABLESPACE_OID) {
        /* Shared system relations live in PGDATA/global */
        Assert(r.dbNode == 0);
        sprintf(path, "%s/global/%u",
                DataDir, r.relNode);
    }
    else if (r.spcNode == DEFAULTTABLESPACE_OID) {
        /* The default tablespace is PGDATA/base */
        sprintf(path, "%s/base/%u/%u",
                DataDir, r.dbNode, r.relNode);
    }
    else {
        /* All other tablespaces accessed via symlinks */
        sprintf(path, "%s/pg_tblspc/%u/%u/%u",
                DataDir
                r.spcNode, r.dbNode, r.relNode);
    }
    return path;
}
```

❖ File Descriptor Pool

Unix has limits on the number of concurrently open files.

PostgreSQL maintains a pool of open file descriptors:

- to hide this limitation from higher level functions
- to minimise expensive `open()` operations

File names are simply strings: `typedef char *FileName`

Open files are referenced via: `typedef int File`

A `File` is an index into a table of "virtual file descriptors".

❖ File Descriptor Pool (cont)

Interface to file descriptor (pool):

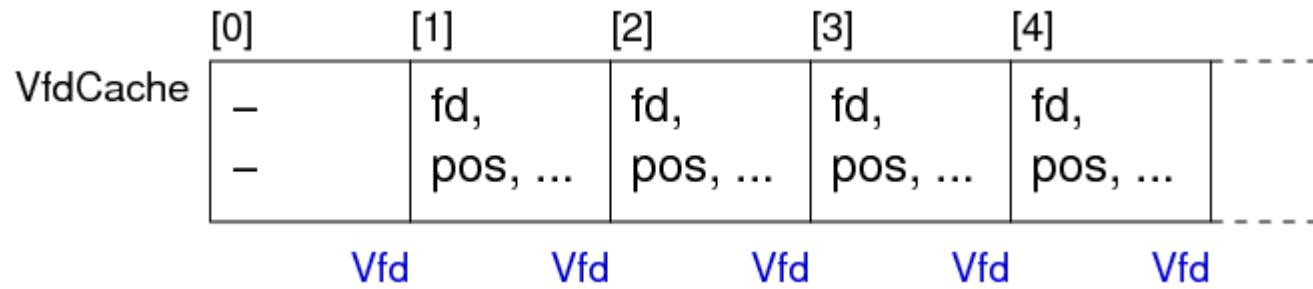
```
backend/storage/file/fd.c
File FileNameOpenFile(FileName fileName,
                      int fileFlags, int fileMode);
    // open a file in the database directory ($PGDATA/base/...)
File OpenTemporaryFile(bool interXact);
    // open temp file; flag: close at end of transaction?
void FileClose(File file);
int FileRead(File file, char *buffer, int amount);
int FileWrite(File file, char *buffer, int amount);
int FileSync(File file);
long FileSeek(File file, long offset, int whence);
int FileTruncate(File file, long offset);
```

Analogous to Unix syscalls **open()**, **close()**, **read()**, **write()**, **lseek()**, ...

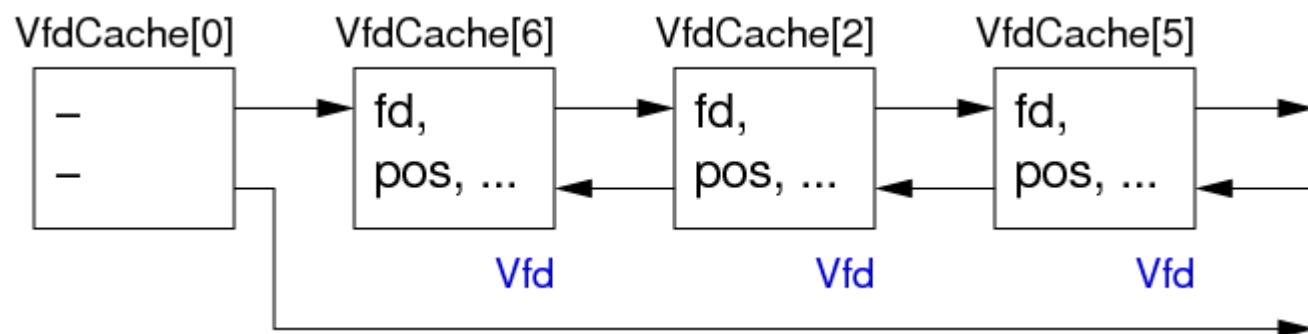
❖ File Descriptor Pool (cont)

Virtual file descriptors (**vfd**)

- physically stored in dynamically-allocated array



- also arranged into list by recency-of-use



VfdCache[0] holds list head/tail pointers.

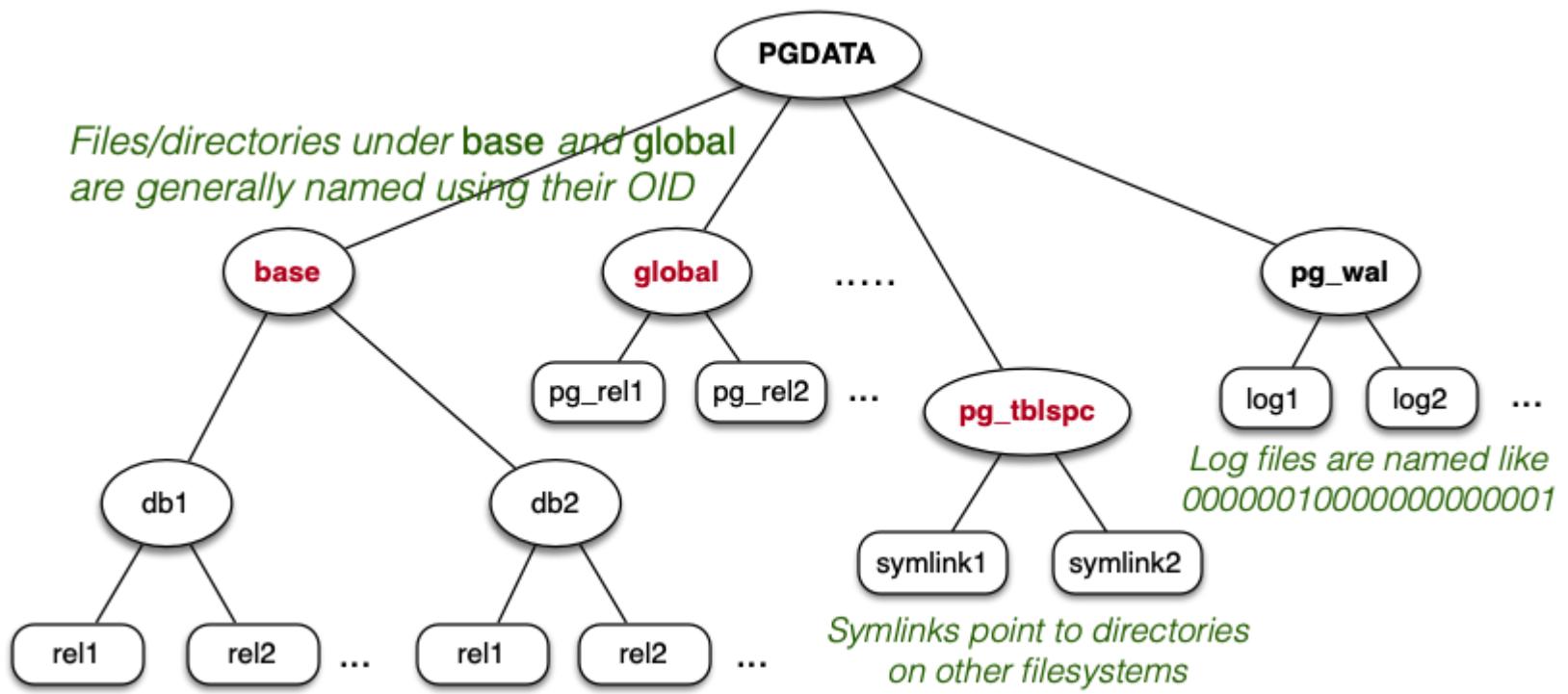
❖ File Descriptor Pool (cont)

Virtual file descriptor records (simplified):

```
backend/storage/file/fd.c
typedef struct vfd
{
    s_short    fd;           // current FD, or VFD_CLOSED if none
    u_short    fdstate;      // bitflags for VFD's state
    File       nextFree;     // link to next free VFD, if in freelist
    File       lruMoreRecently; // doubly linked recency-of-use list
    File       lruLessRecently;
    long       seekPos;      // current logical file position
    char       *fileName;    // name of file, or NULL for unused VFD
    // NB: fileName is malloc'd, and must be free'd when closing the VFD
    int        fileFlags;    // open(2) flags for (re)opening the file
    int        fileMode;     // mode to pass to open(2)
} Vfd;
```

❖ File Manager

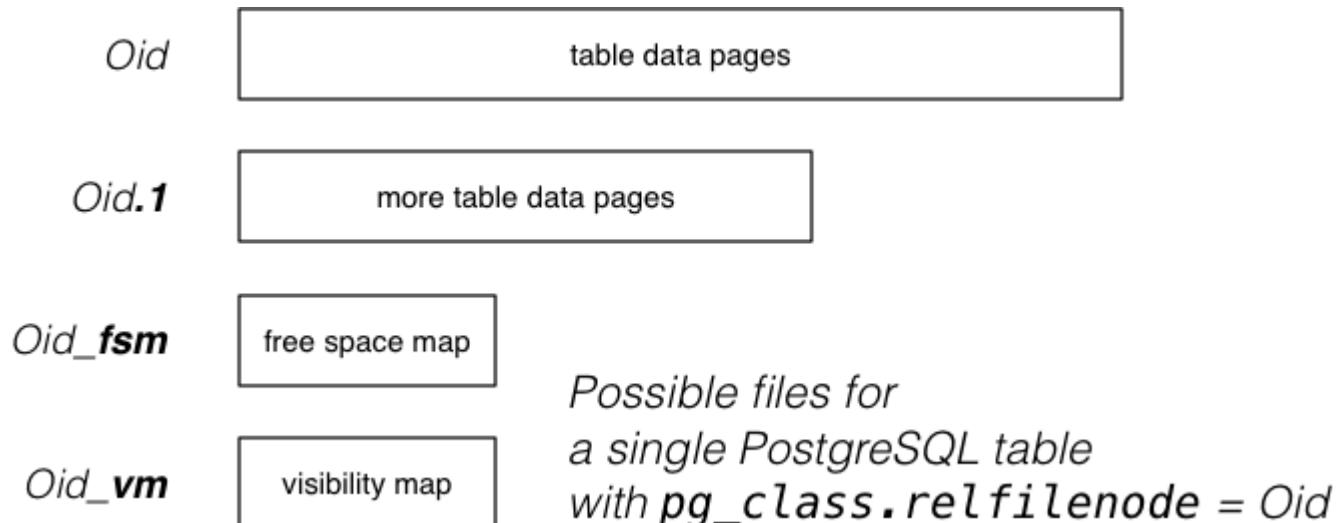
Reminder: PostgreSQL file organisation



❖ File Manager (cont)

PostgreSQL stores each table

- in the directory **PGDATA/pg_database.oid**
- often in multiple files (aka *forks*)



❖ File Manager (cont)

Data files ($Oid, Oid.1, \dots$):

- sequence of fixed-size blocks/pages (typically 8KB)
- each page contains tuple data and admin data (see later)
- max size of data files 1GB (Unix limitation)

	Page 0	Page 1	Page 2	Page 3	Page 4	Page 5
Oid	tuples...	tuples...	tuples...	tuples...	tuples...	tuples...

PostgreSQL Data File (Heap)

❖ File Manager (cont)

Free space map (*Oid_fsm*):

- indicates where free space is in data pages
- "free" space is only free after **VACUUM**
(**DELETE** simply marks tuples as no longer in use **xmax**)

Visibility map (*Oid_vm*):

- indicates pages where all tuples are "visible"
(*visible* = accessible to all currently active transactions)
- such pages can be ignored by **VACUUM**

❖ File Manager (cont)

The "magnetic disk storage manager" (**storage/smgr/md.c**)

- manages its own pool of open file descriptors (Vfd's)
- may use several Vfd's to access data, if several forks
- manages mapping from **PageID** to file+offset.

PostgreSQL **PageID** values are structured:

```
include/storage/buf_internals.h
typedef struct
{
    RelFileNode rnode;      // which relation/file
    ForkNumber forkNum;    // which fork (of reln)
    BlockNumber blockNum;  // which page/block
} BufferTag;
```

❖ File Manager (cont)

Access to a block of data proceeds (roughly) as follows:

```
// pageID set from pg_catalog tables
// buffer obtained from Buffer pool
getBlock(BufferTag pageID, Buffer buf)
{
    Vfd vf; off_t offset;
    (vf, offset) = findBlock(pageID)
    lseek(vf.fd, offset, SEEK_SET)
    vf.seekPos = offset;
    nread = read(vf.fd, buf, BLOCKSIZE)
    if (nread < BLOCKSIZE) ... we have a problem
}
```

BLOCKSIZE is a global configurable constant (default: 8192)

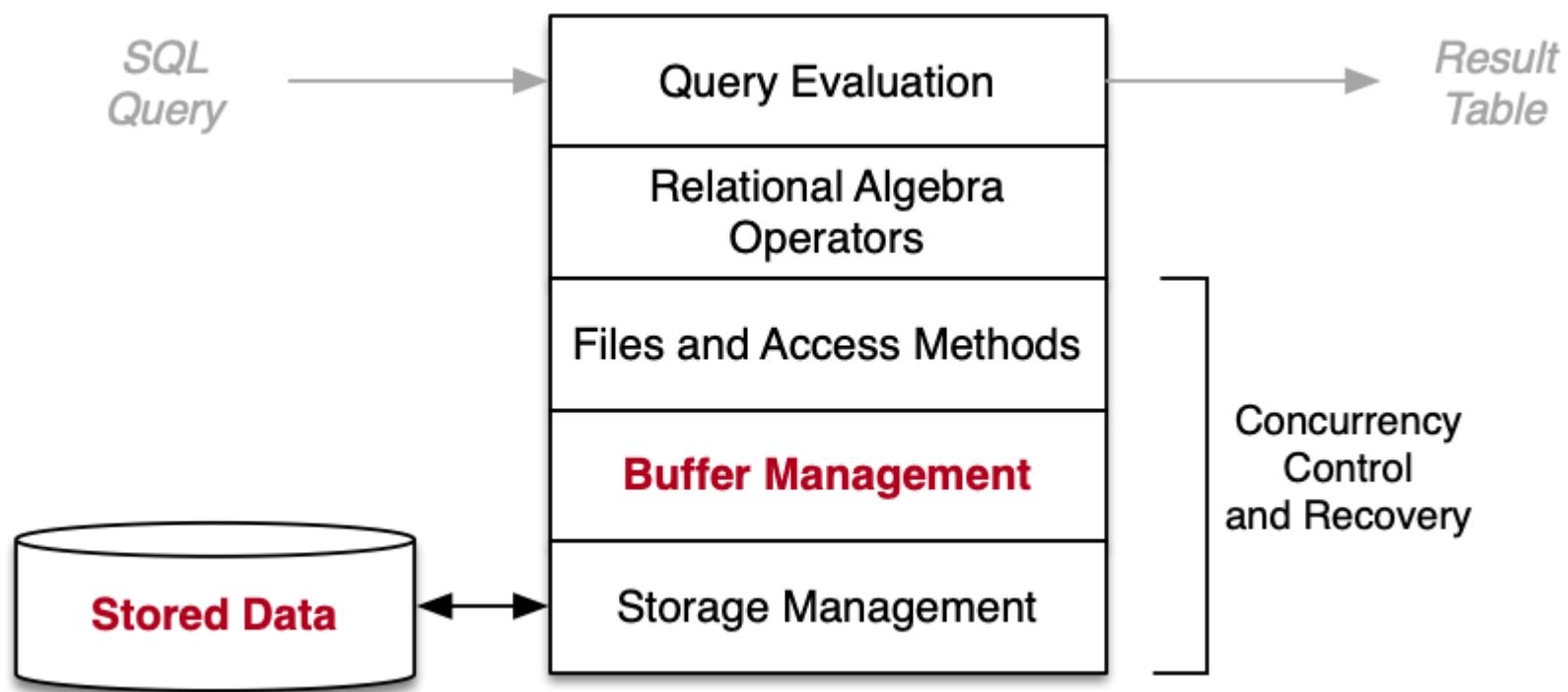
❖ File Manager (cont)

```
findBlock(BufferTag pageID) returns (Vfd, off_t)
{
    offset = pageID.blockNum * BLOCKSIZE
    fileName = relpath(pageID.rnode)
    if (pageID.forkNum > 0)
        fileName = fileName+"."+pageID.forkNum
    if (fileName is not in Vfd pool)
        fd = allocate new Vfd for fileName
    else
        fd = use Vfd from pool
    if (pageID.forkNum > 0) {
        offset = offset - (pageID.forkNum*MAXFILESIZE)
    }
    return (fd, offset)
}
```


Buffer Pool

- [Buffer Pool](#)
- [Page Replacement Policies](#)
- [Effect of Buffer Management](#)

❖ Buffer Pool



❖ Buffer Pool (cont)

Aim of buffer pool:

- hold pages read from database files, for possible re-use

Used by:

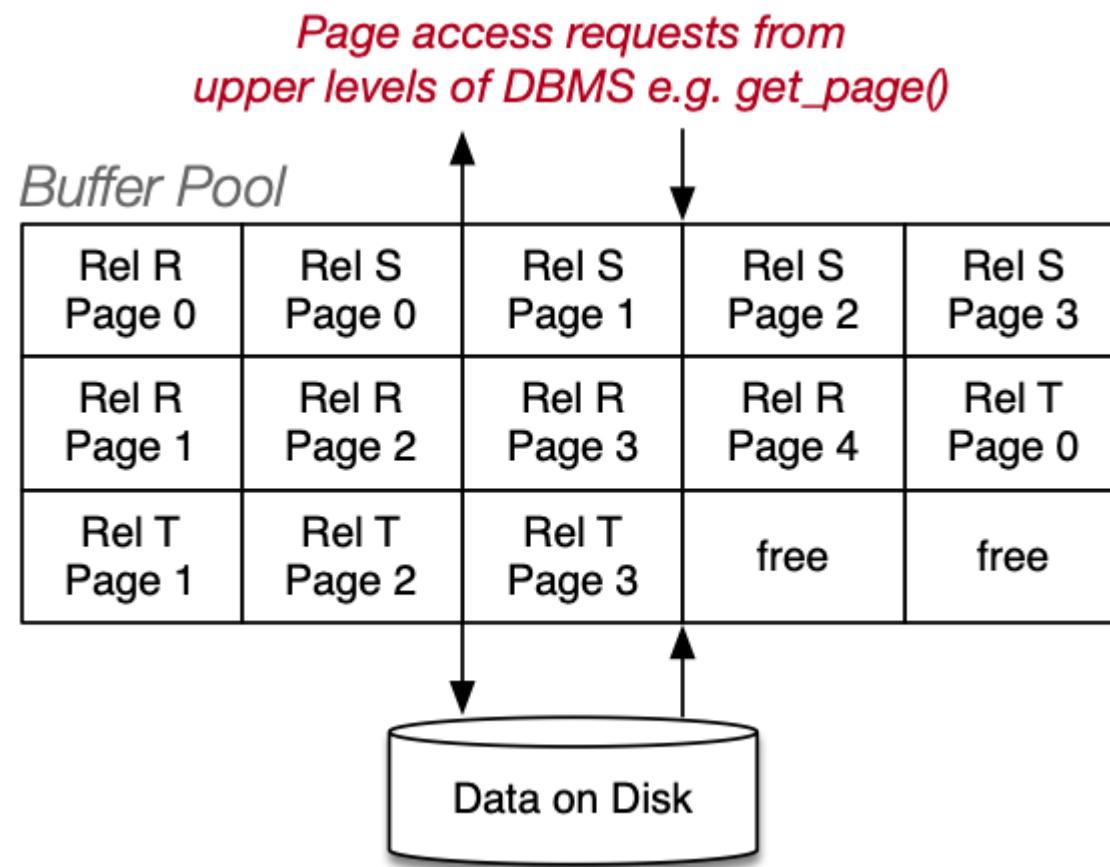
- **access methods** which read/write data pages
- e.g. sequential scan, indexed retrieval, hashing

Uses:

- file manager functions to access data files

Note: we use the terms **page** and **block** interchangably

❖ Buffer Pool (cont)



❖ Buffer Pool (cont)

Buffer pool operations: (both take single **PageID** argument)

- **request_page(pid)**, **release_page(pid)**, ...

To some extent ...

- **request_page()** replaces **getBlock()**
- **release_page()** replaces **putBlock()**

Buffer pool data structures:

- **Page frames [NBUFS]**
- **FrameData directory [NBUFS]**
- **Page is byte [BUFSIZE]**

❖ Buffer Pool (cont)

directory

[0] [1] [2]

Info about frame 0	Info about frame 1	Info about frame 2	Info about frame 3	Info about NBUFS-1
--------------------	--------------------	--------------------	--------------------	-------	--------------------

[0] [1] [2]

data or empty	data or empty	data or empty	[NBUFS-1]
---------------	---------------	---------------	-------	-----------

frames

❖ Buffer Pool (cont)

For each frame, we need to know: (**FrameData**)

- which Page it contains, or whether empty/free
- whether it has been modified since loading (**dirty bit**)
- how many transactions are currently using it (**pin count**)
- time-stamp for most recent access (assists with replacement)

Pages are referenced by PageID ...

- PageID = BufferTag = (rnode, forkNum, blockNum)

❖ Buffer Pool (cont)

How scans are performed without Buffer Pool:

```
Buffer buf;
int N = numberofBlocks(Rel);
for (i = 0; i < N; i++) {
    pageID = makePageID(db, Rel, i);
    getBlock(pageID, buf);
    for (j = 0; j < nTuples(buf); j++)
        process(buf, j)
}
```

Requires **N** page reads.

If we read it again, **N** page reads.

❖ Buffer Pool (cont)

How scans are performed with Buffer Pool:

```
Buffer buf;
int N = numberofBlocks(Rel);
for (i = 0; i < N; i++) {
    pageID = makePageID(db, Rel, i);
    bufID = request_page(pageID);
    buf = frames[bufID]
    for (j = 0; j < nTuples(buf); j++)
        process(buf, j)
    release_page(pageID);
}
```

Requires **N** page reads on the first pass.

If we read it again, $0 \leq$ page reads $\leq N$

❖ Buffer Pool (cont)

Implementation of **request_page()**

```
int request_page(PageID pid)
{
    if (pid in Pool)
        bufID = index for pid in Pool
    else {
        if (no free frames in Pool)
            evict a page (free a frame)
        bufID = allocate free frame
        directory[bufID].page = pid
        directory[bufID].pin_count = 0
        directory[bufID].dirty_bit = 0
    }
    directory[bufID].pin_count++
    return bufID
}
```

❖ Buffer Pool (cont)

The **release_page(pid)** operation:

- Decrement pin count for specified page

Note: no effect on disk or buffer contents until replacement required

The **mark_page(pid)** operation:

- Set dirty bit on for specified page

Note: doesn't actually write to disk; indicates that page changed

The **flush_page(pid)** operation:

- Write the specified page to disk (using **write_page**)

Note: not generally used by higher levels of DBMS

❖ Buffer Pool (cont)

Evicting a page ...

- find frame(s) *preferably* satisfying
 - pin count = 0 (i.e. nobody using it)
 - dirty bit = 0 (not modified)
- if selected frame was modified, flush frame to disk
- flag directory entry as "frame empty"

If multiple frames can potentially be released

- need a policy to decide which is best choice

❖ Page Replacement Policies

Several schemes are commonly in use:

- Least Recently Used (LRU)
- Most Recently Used (MRU)
- First in First Out (FIFO)
- Random

LRU / MRU require knowledge of when pages were last accessed

- how to keep track of "last access" time?
- base on request/release ops or on *real* page usage?

❖ Page Replacement Policies (cont)

Cost benefit from buffer pool (with n frames) is determined by:

- number of available frames (more \Rightarrow better)
- replacement strategy vs page access pattern

Example (a): sequential scan, LRU or MRU, $n \geq b$

First scan costs b reads; subsequent scans are "free".

Example (b): sequential scan, MRU, $n < b$

First scan costs b reads; subsequent scans cost $b - n$ reads.

Example (c): sequential scan, LRU, $n < b$

All scans cost b reads; known as **sequential flooding**.

❖ Effect of Buffer Management

Consider a query to find customers who are also employees:

```
select c.name  
from Customer c, Employee e  
where c.ssn = e.ssn;
```

This might be implemented inside the DBMS via nested loops:

```
for each tuple t1 in Customer {  
    for each tuple t2 in Employee {  
        if (t1.ssn == t2.ssn)  
            append (t1.name) to result set  
    }  
}
```

❖ Effect of Buffer Management (cont)

In terms of page-level operations, the algorithm looks like:

```
Rel rC = openRelation("Customer");
Rel rE = openRelation("Employee");
for (int i = 0; i < nPages(rC); i++) {
    PageID pid1 = makePageID(db,rC,i);
    Page p1 = request_page(pid1);
    for (int j = 0; j < nPages(rE); j++) {
        PageID pid2 = makePageID(db,rE,j);
        Page p2 = request_page(pid2);
        // compare all pairs of tuples from p1,p2
        // construct solution set from matching pairs
        release_page(pid2);
    }
    release_page(pid1);
}
```

❖ Effect of Buffer Management (cont)

Costs depend on relative size of tables, #buffers (n), replacement strategy

Requests: each rC page requested once, each rE page requested rC times

If $n\text{Pages}(rC) + n\text{Pages}(rE) \leq n$

- read each page exactly once, holding all pages in buffer pool

If $n\text{Pages}(rE) \leq n-1$, and LRU replacement

- sequential flooding (see earlier slide)

If $n == 2$ (worst case)

- read each page every time it's requested

PostgreSQL Buffer Manager

- [PostgreSQL Buffer Manager](#)
- [Clock-sweep Replacement Strategy](#)

❖ PostgreSQL Buffer Manager

PostgreSQL buffer manager:

- provides a shared pool of memory buffers for all backends
- all access methods get data from disk via buffer manager

Buffers are located in a large region of shared memory.

Definitions: `src/include/storage/buf*.h`

Functions: `src/backend/storage/buffer/*.c`

Buffer code is also used by backends who want a private buffer pool

❖ PostgreSQL Buffer Manager (cont)

Buffer pool consists of:

BufferDescriptors

- shared fixed array (size **NBuffers**) of **BufferDesc**

BufferBlocks

- shared fixed array (size **NBuffers**) of 8KB frames

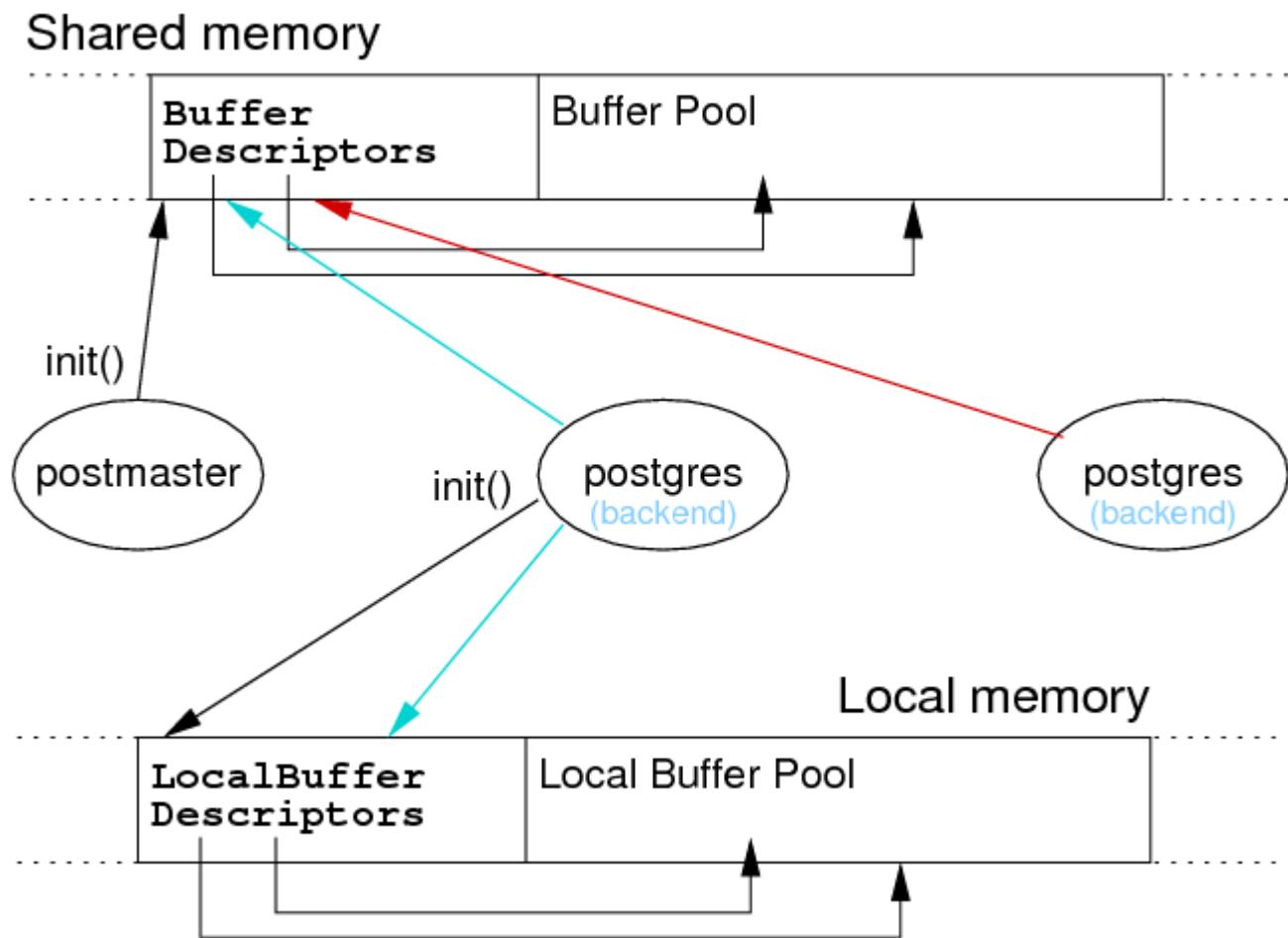
Buffer = index values in above arrays

- indexes: global buffers **1 .. NBuffers**; local buffers negative

Size of buffer pool is set in **postgresql.conf**, e.g.

```
shared_buffers = 16MB      # min 128KB, 16*8KB buffers
```

❖ PostgreSQL Buffer Manager (cont)



❖ PostgreSQL Buffer Manager (cont)

`include/storage/buf.h`

- basic buffer manager data types (e.g. `Buffer`)

`include/storage/bufmgr.h`

- definitions for buffer manager function interface
(i.e. functions that other parts of the system call to use buffer manager)

`include/storage/buf_internals.h`

- definitions for buffer manager internals (e.g. `BufferDesc`)

Code: `backend/storage/buffer/*.c`

Commentary: `backend/storage/buffer/README`

❖ PostgreSQL Buffer Manager (cont)

Definition of buffer descriptors (simplified):

```
include/storage/buf_internals.h
typedef struct BufferDesc
{
    BufferTag    tag;        // ID of page contained in buffer
    int          buf_id;    // buffer's index number (from 0)

    // state, containing flags, refcount and usagecount
    pg_atomic_uint32 state;

    int          freeNext;   // link in freelist chain
    ...
} BufferDesc;
```

❖ Clock-sweep Replacement Strategy

PostgreSQL page replacement strategy: **clock-sweep**

- treat buffer pool as circular list of buffer slots
- **NextVictimBuffer** (NVB) holds index of next possible evictee
- if **Buf[NVB]** page is pinned or "popular", leave it
 - **usage_count** implements "popularity/recency" measure
 - incremented on each access to buffer (up to small limit)
 - decremented each time considered for eviction
- else if **pin_count = 0** and **usage_count = 0** then grab this buffer
- increment **NextVictimBuffer** and try again (wrap at end)

❖ Clock-sweep Replacement Strategy (cont)

Action of clock-sweep:

NVB					
	[0]	[1]	[2]	[3]	[4]
Before Clock Sweep	pin: 1 use: 3	pin: 0 use: 1	pin: 0 use: 2	pin: 0 use: 0	pin: 1 use: 1
					pin: 1 use: 2

NVB					
	[0]	[1]	[2]	[3]	[4]
After Clock Sweep	pin: 1 use: 2	pin: 0 use: 0	pin: 0 use: 1	pin: 1 use: 0	pin: 1 use: 1
					pin: 1 use: 2

❖ Clock-sweep Replacement Strategy (cont)

For specialised kinds of access (e.g. sequential scan),

- clock-sweep is not the best replacement strategy
- can allocate a private "buffer ring"
- use this buffer ring with alternative replacement strategy

Page Internals

- [Pages](#)
- [Page Formats](#)
- [Page Formats](#)
- [Storage Utilisation](#)
- [Overflows](#)

❖ Pages

Database applications view data as:

- a collection of records (tuples)
- records can be accessed via a **TupleId/RecordId/RID**
- **$\text{TupleId} = (\text{PageID} + \text{TupIndex})$**

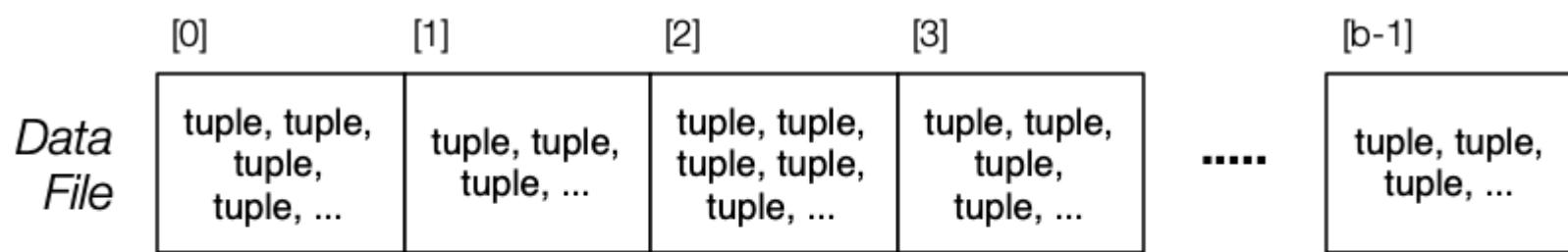
The disk and buffer manager provide the following view:

- data is a sequence of fixed-size pages (aka "blocks")
- pages can be (random) accessed via a **PageID**
- each page contains zero or more tuple values

Page format = how space/tuples are organised within a page

❖ Pages (cont)

Data files consist of pages containing tuples:



*r tuples contained in b pages
each page can hold up to c tuples*

Each data file (in PostgreSQL) is related to one table.

❖ Page Formats

Ultimately, a **Page** is simply an array of bytes (**byte[]**).

We want to interpret/manipulate it as a collection of **Records** (tuples).

Tuples are addressed by a record ID (**rid = (PageId, TupIndex)**)

Typical operations on **Pages**:

- **request_page(pid)** ... get page via its **PageId**
- **get_record(rid)** ... get record via its **TupleId**
- **rid = insert_record(pid, rec)** ... add new record
- **update_record(rid, rec)** ... update value of record
- **delete_record(rid)** ... remove record from page

❖ Page Formats (cont)

Page format = tuples + data structures allowing tuples to be found

Characteristics of **Page** formats:

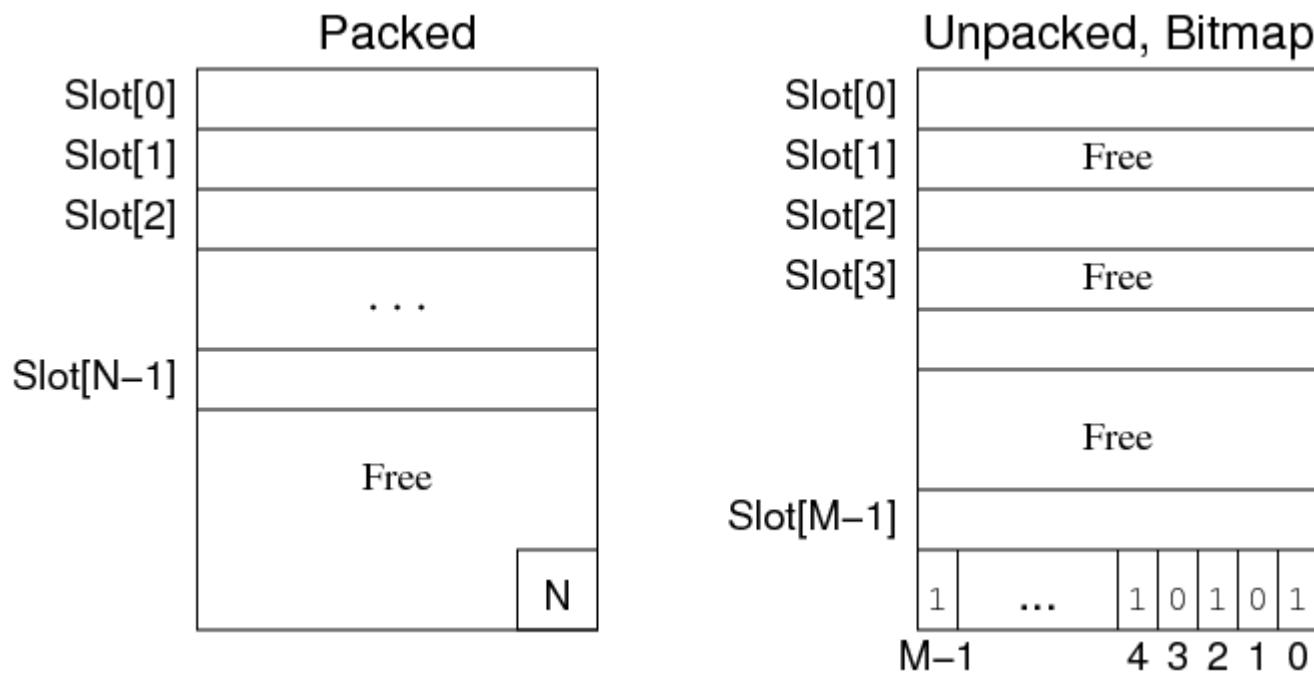
- record size variability (fixed, variable)
- how free space within **Page** is managed
- whether some data is stored outside **Page**
 - does **Page** have an associated overflow chain?
 - are large data values stored elsewhere? (e.g. TOAST)
 - can one tuple span multiple **Pages**?

Implementation of **Page** operations critically depends on format.

❖ Page Formats (cont)

For fixed-length records, use **record slots**.

- **insert**: place new record in first available slot
- **delete**: two possibilities for handling free record slots:



❖ Page Formats

For variable-length records, must use **slot directory**.

Possibilities for handling free-space within block:

- compacted (one region of free space)
- fragmented (distributed free space)

In practice, a combination is useful:

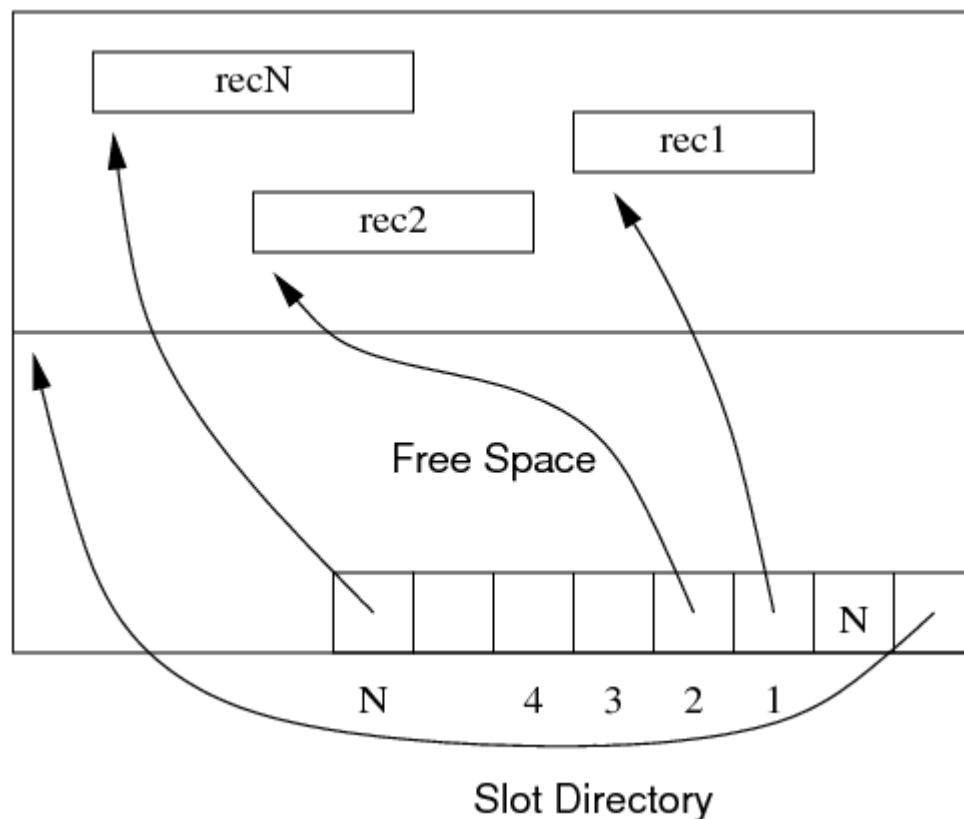
- normally fragmented (cheap to maintain)
- compacted when needed (e.g. record won't fit)

Important aspect of using slot directory

- location of tuple within page can change, tuple index does not change

❖ Page Formats (cont)

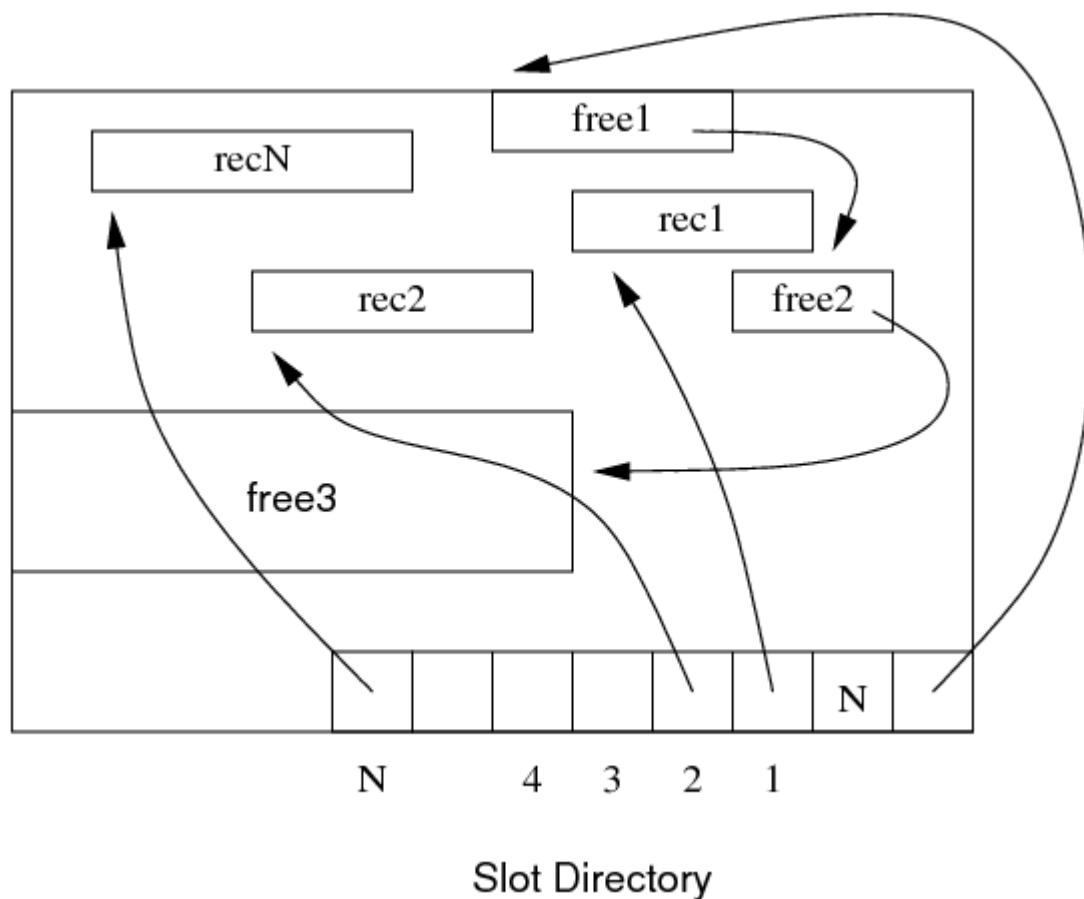
Compacted free space:



Note: "pointers" are implemented as word offsets within block.

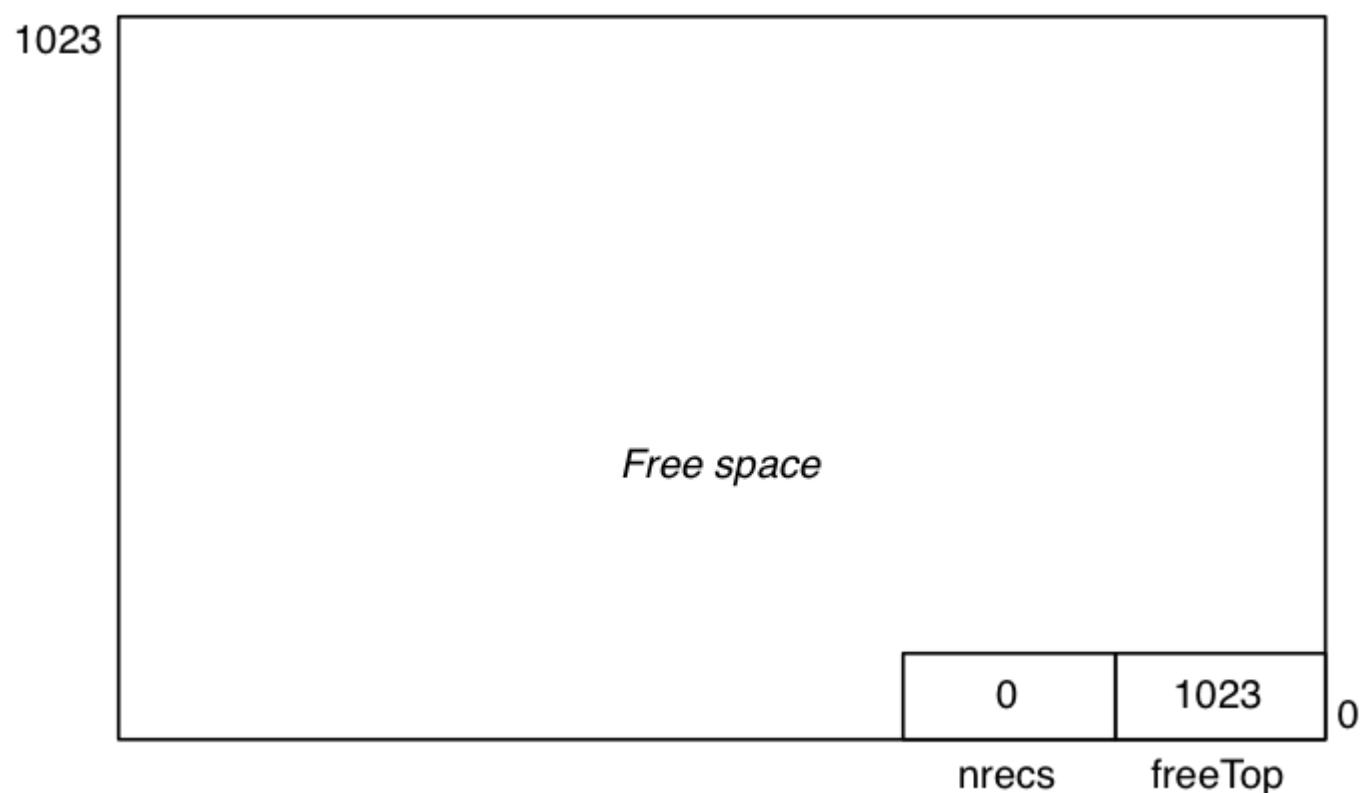
❖ Page Formats (cont)

Fragmented free space:



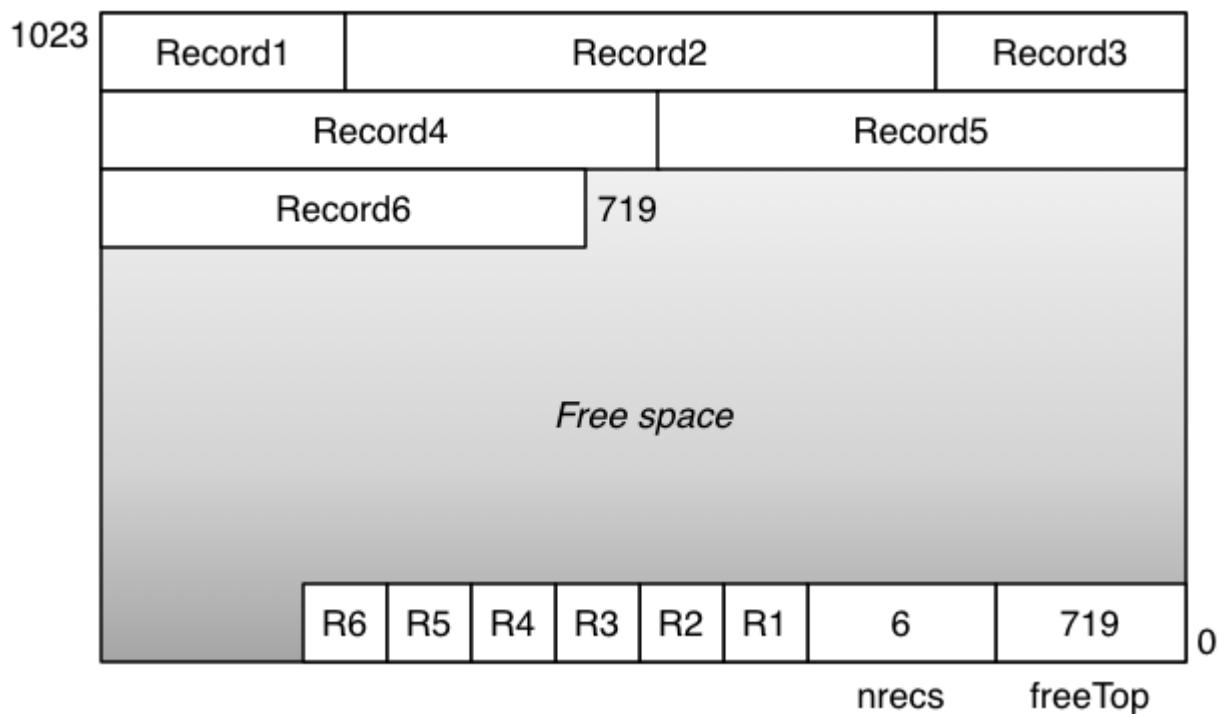
❖ Page Formats (cont)

Initial page state (compacted free space) ...



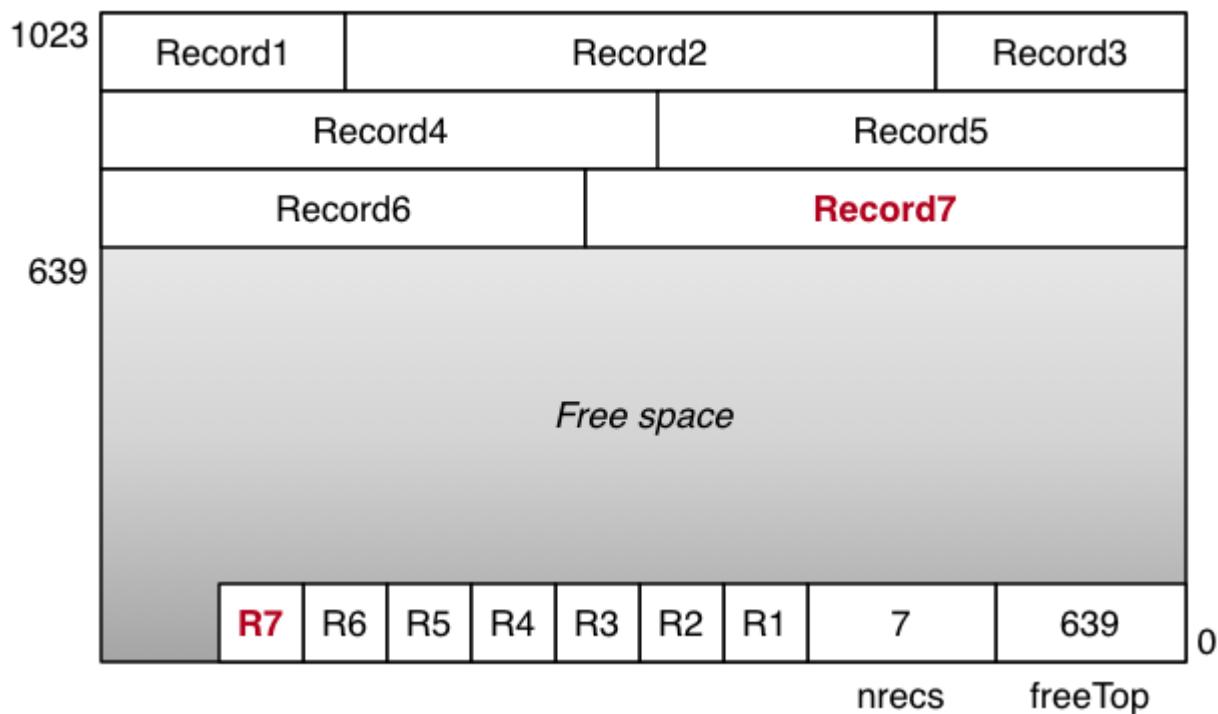
❖ Page Formats (cont)

Before inserting record 7 (compacted free space) ...



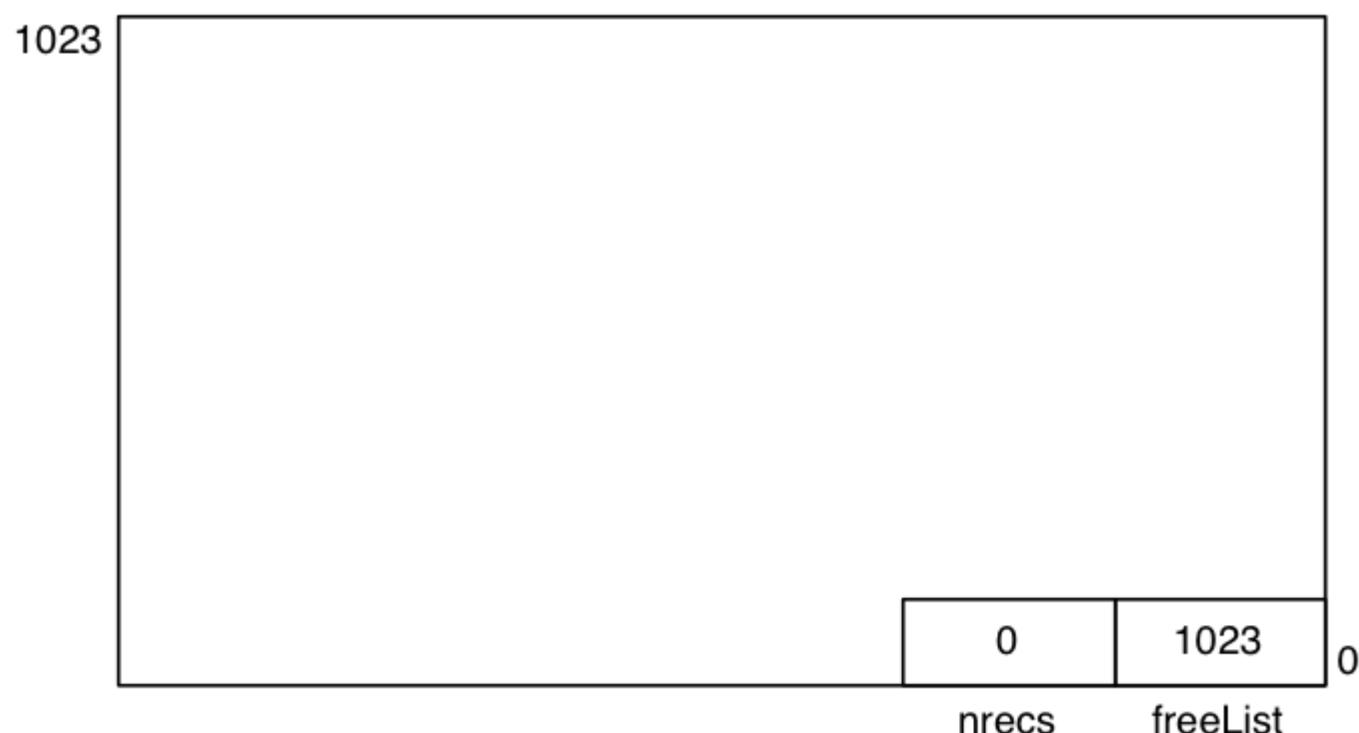
❖ Page Formats (cont)

After inserting record 7 (80 bytes) ...



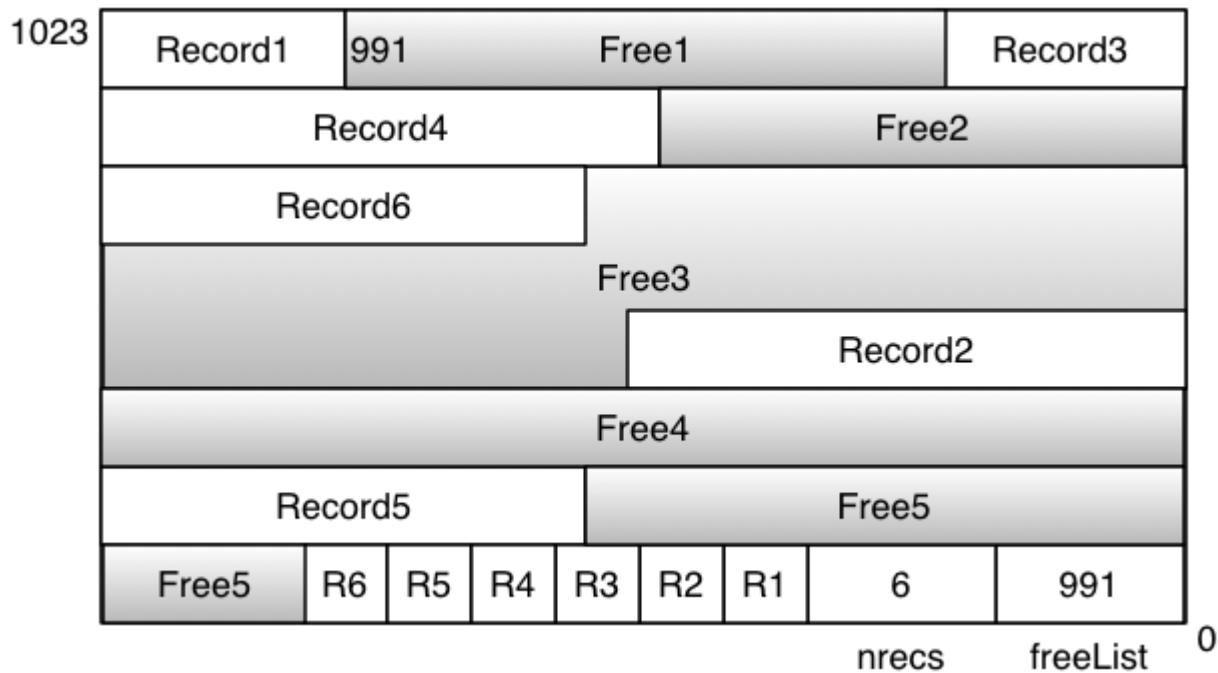
❖ Page Formats (cont)

Initial page state (fragmented free space) ...



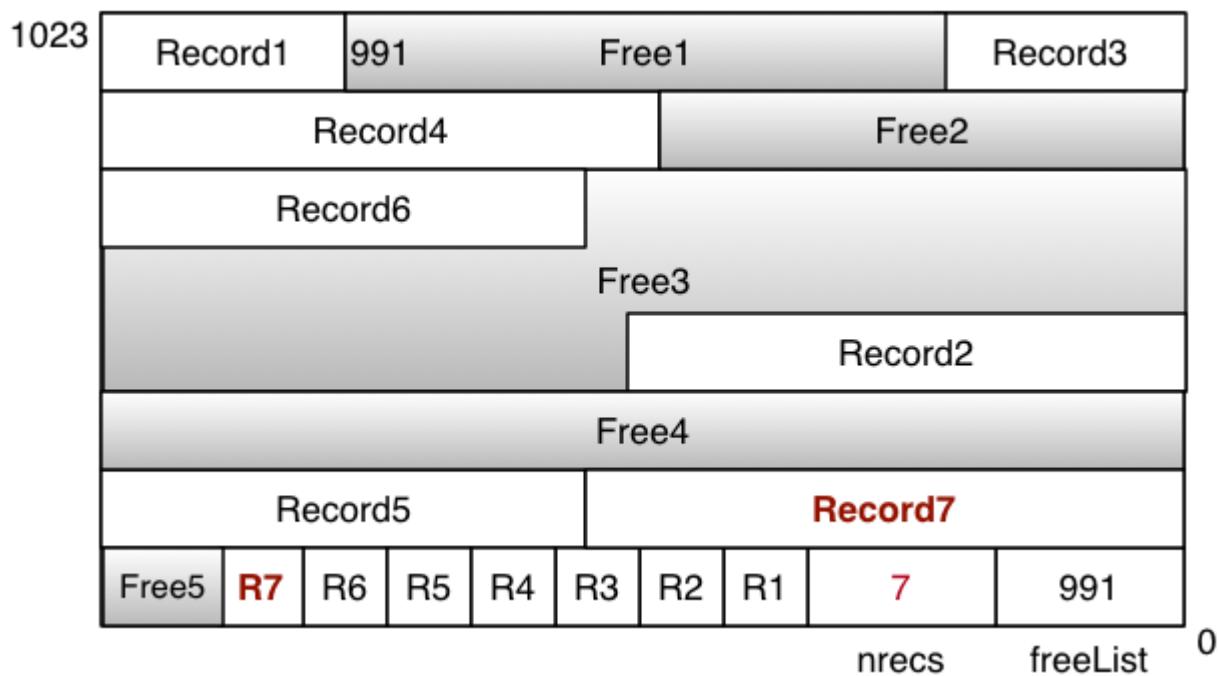
❖ Page Formats (cont)

Before inserting record 7 (fragmented free space) ...



❖ Page Formats (cont)

After inserting record 7 (80 bytes) ...



❖ Storage Utilisation

How many records can fit in a page? (denoted C = capacity)

Depends on:

- page size ... typical values: 1KB, 2KB, 4KB, 8KB
- record size ... typical values: 64B, 200B, app-dependent
- page header data ... typically: 4B - 32B
- slot directory ... depends on how many records

We typically consider average record size (R)

Given C , $\text{HeaderSize} + C*\text{SlotSize} + C*R \leq \text{PageSize}$

❖ Overflows

Sometimes, it may not be possible to insert a record into a page:

1. no free-space fragment large enough
2. overall free-space is not large enough
3. the record is larger than the page
4. no more free directory slots in page

For case (1), can first try to compact free-space within the page.

If still insufficient space, we need an alternative solution ...

❖ Overflows (cont)

File organisation determines how cases (2)..(4) are handled.

If records may be inserted anywhere that there is free space

- cases (2) and (4) can be handled by making a new page
- case (3) requires either spanned records or "overflow file"

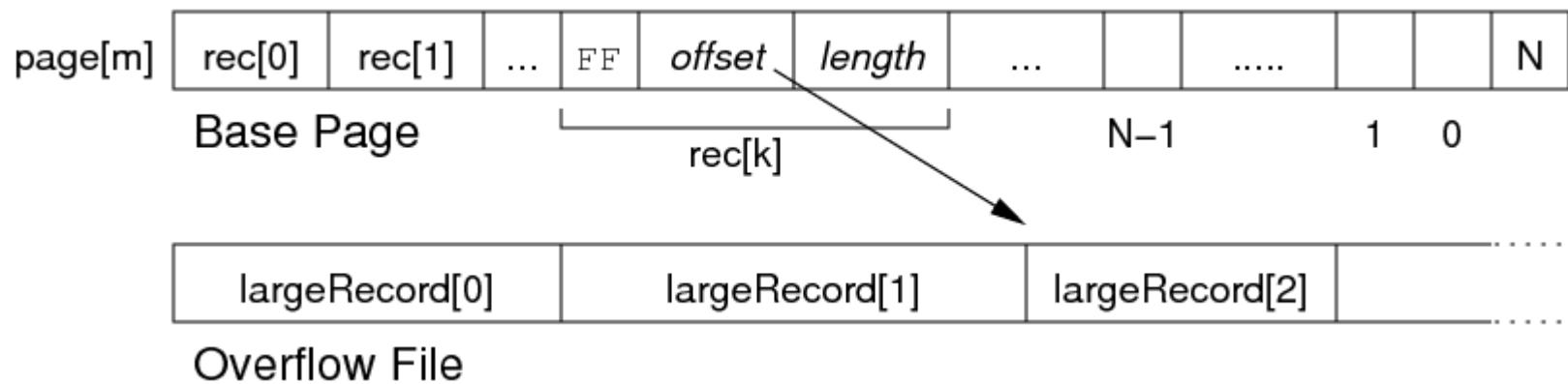
If file organisation determines record placement (e.g. hashed file)

- cases (2) and (4) require an "overflow page"
- case (3) requires an "overflow file"

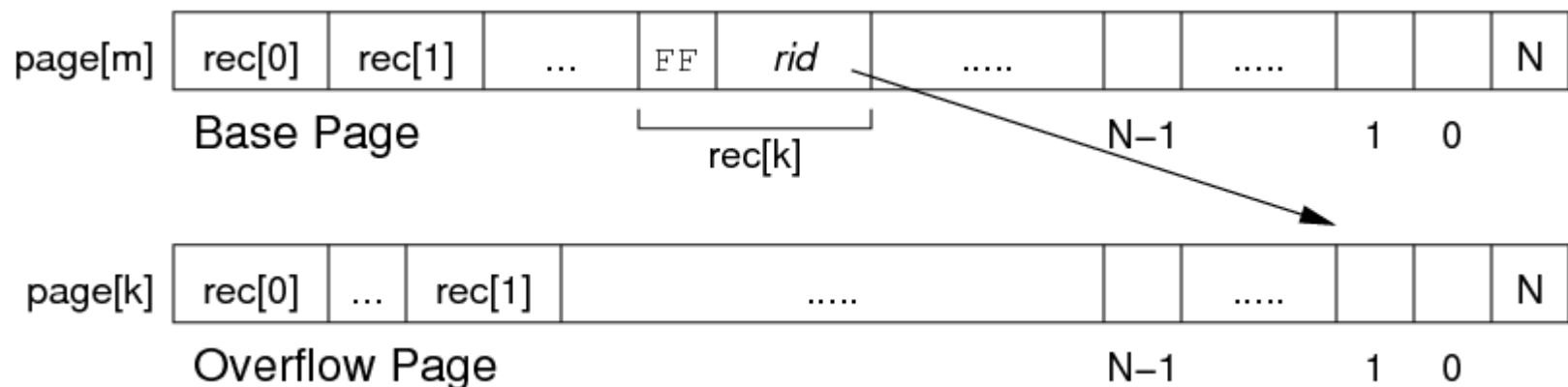
With overflow pages, *rid* structure may need modifying (*rel,page,ovfl,rec*)

❖ Overflows (cont)

Overflow files for very large records and BLOBs:



Record-based handling of overflows:



We discuss overflow pages in more detail when covering Hash Files.

PostgreSQL Page Internals

- [PostgreSQL Page Representation](#)
- [TOAST'ing](#)

❖ PostgreSQL Page Representation

Functions: **src/backend/storage/page/* .c**

Definitions: **src/include/storage/bufpage.h**

Each page is 8KB (default **BLCKSZ**) and contains:

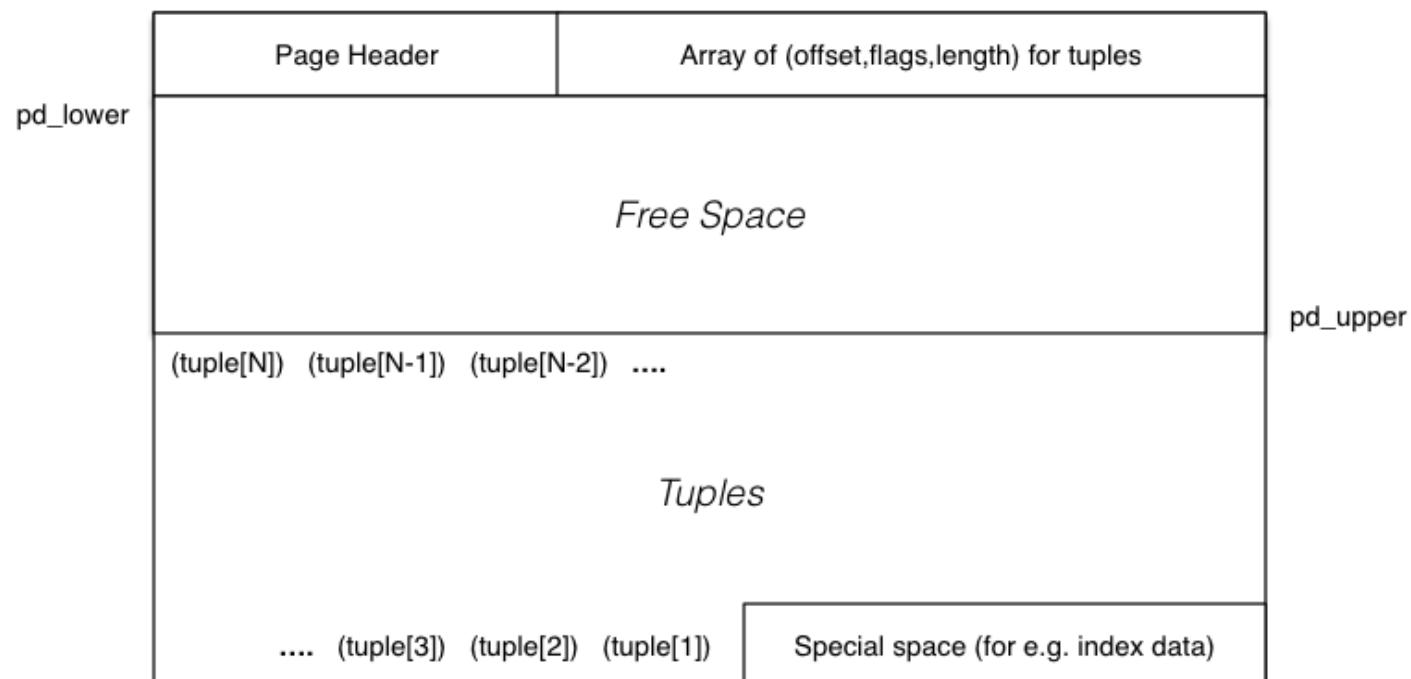
- header (free space pointers, flags, xact data)
- array of (offset,length) pairs for tuples in page
- free space region (between array and tuple data)
- actual tuples themselves (inserted from end towards start)
- (optionally) region for special data (e.g. index data)

Large data items are stored in separate (TOAST) files (implicit)

Also supports ~SQL-standard BLOBs (explicit large data items)

❖ PostgreSQL Page Representation (cont)

PostgreSQL page layout:



❖ PostgreSQL Page Representation (cont)

Page-related data types:

```
// a Page is simply a pointer to start of buffer
typedef Pointer Page;

// indexes into the tuple directory
typedef uint16 LocationIndex;

// entries in tuple directory (line pointer array)
typedef struct ItemIdData
{
    unsigned lp_off:15,      // tuple offset from start of page
        lp_flags:2,        // unused,normal,redirect,dead
        lp_len:15;         // length of tuple (bytes)
} ItemIdData;
```

❖ PostgreSQL Page Representation (cont)

Page-related data types: (cont)

```
include/storage/bufpage.h
typedef struct PageHeaderData
{
    XLogRecPtr    pd_lsn;          // xact log record for last change
    uint16        pd_tli;          // xact log reference information
    uint16        pd_flags;        // flag bits (e.g. free, full, ...)
    LocationIndex pd_lower;       // offset to start of free space
    LocationIndex pd_upper;       // offset to end of free space
    LocationIndex pd_special;    // offset to start of special space
    uint16        pd_pagesize_version;
    TransactionId pd_prune_xid;  // is pruning useful in data page?
    ItemIdData    pd_linp[];       // beginning of line pointer array, FLEXIBLE_ARRAY_MEMBER
} PageHeaderData;

typedef PageHeaderData *PageHeader;
```

❖ PostgreSQL Page Representation (cont)

Operations on **Pages**:

void PageInit(Page page, Size pageSize, ...)

- initialize a **Page** buffer to empty page
- in particular, sets **pd_lower** and **pd_upper**

OffsetNumber PageAddItem(Page page,
Item item, Size size, ...)

- insert one tuple (or index entry) into a **Page**
- fails if: not enough free space, too many tuples

void PageRepairFragmentation(Page page)

- compact tuple storage to give one large free space region

❖ PostgreSQL Page Representation (cont)

PostgreSQL has two kinds of pages:

- **heap pages** which contain tuples
- **index pages** which contain index entries

Both kinds of page have the same page layout.

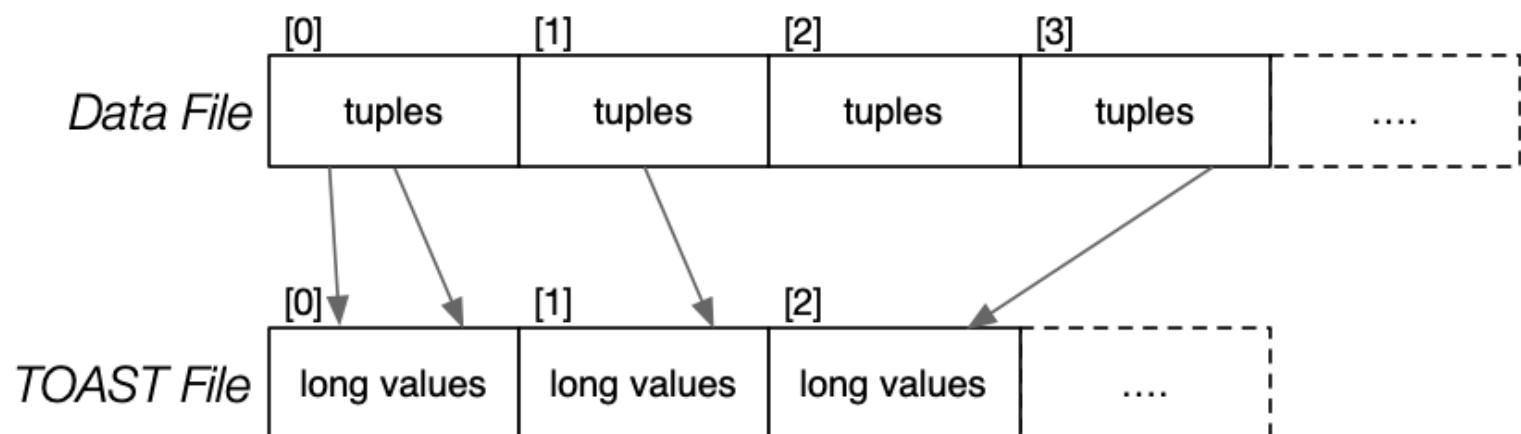
One important difference:

- index entries tend be a smaller than tuples
- can typically fit more index entries per page

❖ TOAST'ing

TOAST = The Oversized-Attribute Storage Technique

- handles storage of large attribute values (> 2KB) (e.g. long `text`)



❖ TOAST'ing (cont)

Large attribute values are stored out-of-line (i.e. in separate file)

- "value" of attribute in tuple is a reference to TOAST data
- TOAST'd values may be compressed
- TOAST'd values are stored in 2K chunks

Strategies for storing TOAST-able columns ...

- **PLAIN** ... allows no compression or out-of-line storage
- **EXTENDED** ... allows both compression and out-of-line storage
- **EXTERNAL** ... allows out-of-line storage but not compression
- **MAIN** ... allows compression but not out-of-line storage

Week 2 Exercises

- [Exercise 1: PostgreSQL Files](#)
- [Exercise 2: Relation Scan Cost](#)
- [Exercise 3: Buffer Cost Benefit \(i\)](#)
- [Exercise 4 : Buffer Cost Benefit \(ii\)](#)
- [Exercise 5: Clock-sweep Page Replacement](#)
- [Clock-sweep Replacement Strategy](#)
- [Exercise 6: Fixed-length Records](#)
- [Exercise 7: Inserting/Deleting Fixed-length Records](#)
- [Exercise 8: Inserting Variable-length Records](#)
- [Exercise 9: PostgreSQL Pages](#)

❖ Exercise 1: PostgreSQL Files

In your PostgreSQL server

- examine the content of the **\$PGDATA/base** directory
- find the directory containing the **uni** database (from P03)
- find the file in this directory for the **People** table
- examine the contents of the **People** file
- what are the other files in the directory?
- are there **forks** in any of your database?

❖ Exercise 2: Relation Scan Cost

Consider a table $R(x,y,z)$ with 10^4 tuples, implemented as

- number of tuples $r = 10,000$
- average size of tuples $R = 200$ bytes
- size of data pages $B = 4096$ bytes
- time to read one data page $T_r = 10\text{msec}$
- time to check one tuple 1 usec
- time to form one result tuple 1 usec
- time to write one result page $T_w = 10\text{msec}$

Calculate the total time-cost for answering the query:

```
insert into S select * from R where x > 10;
```

if 50% of the tuples satisfy the condition.

❖ Exercise 3: Buffer Cost Benefit (i)

Consider two relations being joined

```
select * from Customer join Employee
```

To evaluate $C \text{ join } E$ (in pseudo Python notation)

```
for each page  $P_C$  of  $C$ :  
    for each page  $P_E$  of  $E$ :  
        for each tuple  $T_C$  in  $P_C$ :  
            for each tuple  $T_E$  in  $P_E$ :  
                if ( $T_C$  matches  $T_E$ ):  
                    add  $T_C \cdot T_E$  to  $P_{out}$   
                if ( $P_{out}$  full):  
                    write and clear  $P_{out}$ 
```

❖ Exercise 3: Buffer Cost Benefit (i) (cont)

Assume that:

- the **Customer** relation has b_C pages (e.g. 10)
- the **Employee** relation has b_E pages (e.g. 4)

Compute how many page reads occur for $C \text{ join } E \dots$

- if we have only 3 buffers (i.e. effectively no buffer pool)
- if we have 20 buffers
- when a buffer pool with MRU replacement strategy is used
- when a buffer pool with LRU replacement strategy is used

For the last two, buffer pool has $n=3$ slots ($n < b_C$ and $n < b_E$) + output buffer

❖ Exercise 4 : Buffer Cost Benefit (ii)

If the tables were larger, the above analysis would be tedious.

Write a C program to simulate buffer pool usage

- assuming a nested loop join as above
- **argv[1]** gives number of pages in "outer" table
- **argv[2]** gives number of pages in "inner" table
- **argv[3]** gives number of slots in buffer pool
- **argv[4]** gives replacement strategy (LRU,MRU,FIFO-Q)

❖ Exercise 5: Clock-sweep Page Replacement

Using the following data type for buffer frame descriptors:

```
struct FrameDesc {  
    Tag pid;      // ID of page in frame e.g. "R0"  
    int pin;       // number tx's using this page  
    int usage;     // clock-sweep usage counter  
}
```

Show how the buffer pool changes for

- $n = 4$, $b_R = 3$, $b_S = 4$, $b_T = 6$
- when executing **select * from T** via sequential scan
- when executing **select * from R join S** using nested-loop join

❖ Clock-sweep Replacement Strategy

PostgreSQL page replacement strategy: **clock-sweep**

- treat buffer pool as circular list of buffer slots
- **NextVictimBuffer** (NVB) holds index of next possible evictee
- if **Buf[NVB]** page is pinned or "popular", leave it
 - **usage_count** implements "popularity/recency" measure
 - incremented on each access to buffer (up to small limit)
 - decremented each time considered for eviction
- else if **pin_count = 0** and **usage_count = 0** then grab this buffer
- increment **NextVictimBuffer** and try again (wrap at end)

❖ Exercise 6: Fixed-length Records

Give examples of table definitions

- which result in fixed-length records
- which result in variable-length records

```
create table R ( ... );
```

What are the common features of each type of table?

❖ Exercise 7: Inserting/Deleting Fixed-length Records

For each of the following Page formats:

- compacted/packed free space
- unpacked free space (with bitmap)

Implement

- a suitable data structure to represent a **Page**
- a function to insert a new record
- a function to delete a record

❖ Exercise 8: Inserting Variable-length Records

For both of the following page formats

1. variable-length records, with compacted free space
2. variable-length records, with fragmented free space

implement the **insert()** function.

Use the above page format, but also assume:

- page size is 1024 bytes
- tuples start on 4-byte boundaries
- references into page are all 8-bits (1 byte) long
- a function **recSize(r)** gives size in bytes

❖ Exercise 9: PostgreSQL Pages

Draw diagrams of a PostgreSQL heap page

- when it is initially empty
- after three tuples have been inserted with lengths of 60, 80, and 70 bytes
- after the 80 byte tuple is deleted (but before vacuuming)
- after a new 50 byte tuple is added

Show the values in the tuple header.

Assume that there is no special space in the page.

Week 3

- [Week 03](#)

❖ Week 03

Things to Note ...

- Assignment 1 due before 9pm Friday 17 March
- Quiz 2 coming next week ... on Moodle
- Prac Exercise 4 released
- Theory Exercise 3 released

This Week ...

- Storage: representing tuples
- Relational Algebra Operations
- RelOps: Scan, Sort

Coming Up ...

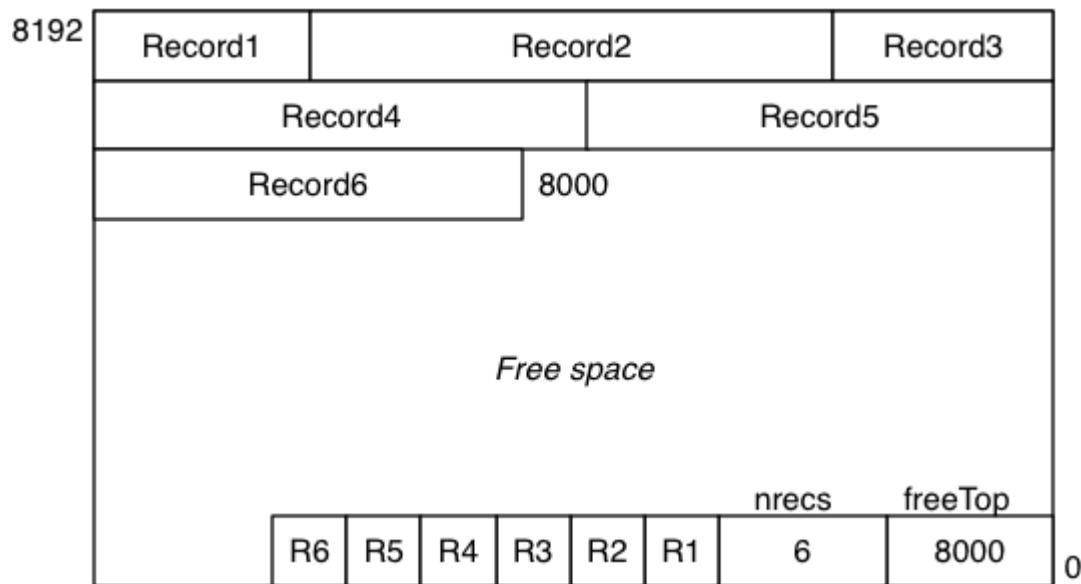
- Relops: Projection, 1d Selection, Hashing, Indexing, Nd Selection

Tuple Representation

- [Tuples](#)
- [Records vs Tuples](#)
- [Converting Records to Tuples](#)
- [Operations on Records](#)
- [Operations on Tuples](#)
- [Fixed-length Records](#)
- [Variable-length Records](#)
- [Data Types](#)
- [Field Descriptors](#)

❖ Tuples

Each page contains a collection of tuples



What do tuples contain? How are they structured internally?

❖ Records vs Tuples

A **table** is defined by a **schema**, e.g.

```
create table Employee (
    id    integer primary key,
    name  varchar(20) not null,
    job   varchar(10),
    dept  smallint references Dept(id)
);
```

where a schema is a collection of attributes (name,type,constraints)

Reminder: schema information (meta-data) is also stored, in the DB catalog

❖ Records vs Tuples (cont)

Tuple = collection of attribute values based on a schema, e.g.

(33357462, 'Neil Young', 'Musician', 277)

iid:integer

name:varchar(20)

job:varchar(10)

dept: smallint

Record = sequence of bytes, containing data for one tuple, e.g.

01101001	11001100	01010101	00111100	10100011	01011111	01011010	
----------	----------	----------	----------	----------	----------	----------	--

Bytes need to be interpreted relative to schema to get tuple

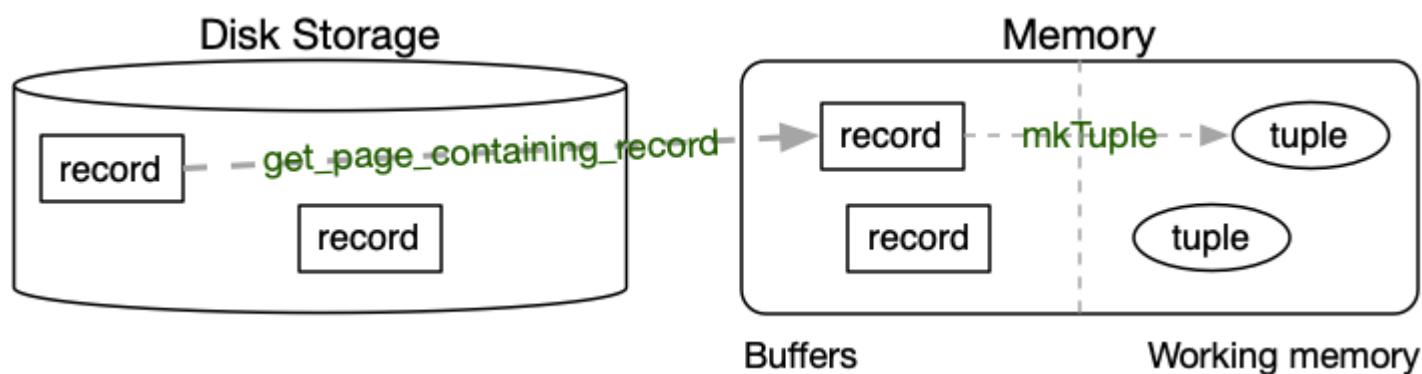
❖ Converting Records to Tuples

A **Record** is an array of bytes (**byte[]**)

- representing the data values from a typed **Tuple**
- stored on disk (persistent) or in a memory buffer

A **Tuple** is a collection of named, typed values (cf. C **struct**)

- to manipulate the values, need an "interpretable" structure
- stored in working memory, and temporary



❖ Converting Records to Tuples (cont)

Information on how to interpret bytes in a record ...

- may be contained in schema data in DBMS catalog
- may be stored in the page directory
- may be stored in the record (in a record header)
- may be stored partly in the record and partly in the schema

For variable-length records, some formatting info ...

- must be stored in the record or in the page directory
- at the least, need to know how many bytes in each varlen value

❖ Operations on Records

Common operation on records ... access record via **RecordId**:

```
Record get_record(Relation rel, RecordId rid) {  
    (pid,tid) = rid;  
    Page buf = get_page(rel, pid);  
    return get_bytes(rel, buf, tid);  
}
```

Cannot use a **Record** directly; need a **Tuple**:

```
Relation rel = ... // relation schema  
Record rec = get_record(rel, rid)  
Tuple t = mkTuple(rel, rec)
```

Once we have a **Tuple**, we can access individual attributes/fields

❖ Operations on Tuples

Once we have a record, we need to interpret it as a tuple ...

Tuple t = mkTuple(rel, rec)

- convert record to tuple data structure for relation **rel**

Once we have a tuple, we want to examines its contents ...

Typ getTypField(Tuple t, int i)

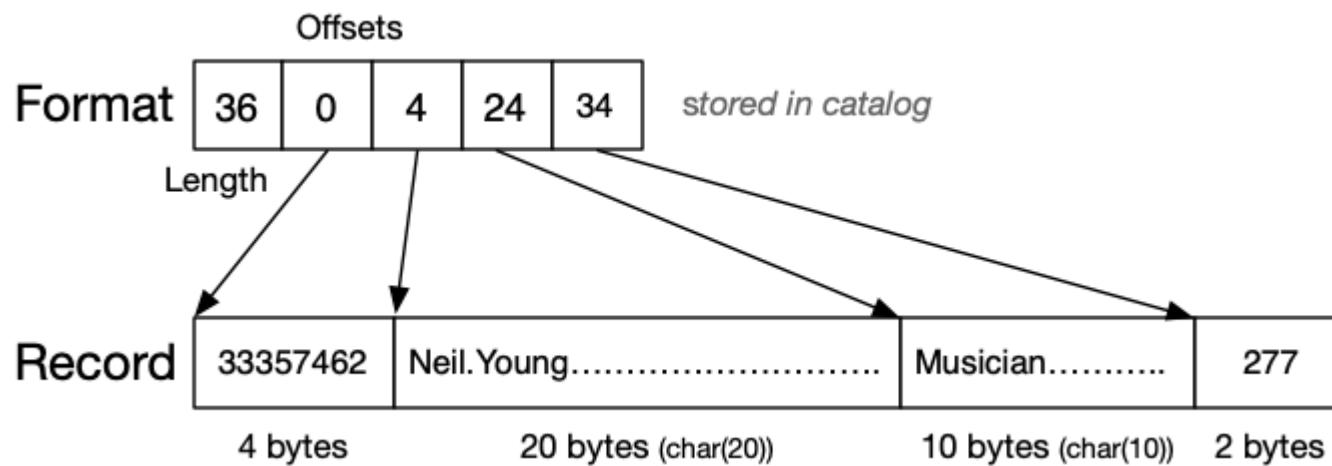
- extract the **i**'th field from a **Tuple** as a value of type **Typ**

E.g. **int x = getIntField(t,1), char *s = getStrField(t,2)**

❖ Fixed-length Records

A possible encoding scheme for fixed-length records:

- record format (length + offsets) stored in catalog
- data values stored in fixed-size slots in data pages



Since record format is frequently used at query time, cache in memory.

❖ Variable-length Records

Possible encoding schemes for variable-length records:

- Prefix each field by length

4	33357462	10	Neil Young	8	Musician	2	277
---	----------	----	------------	---	----------	---	-----

- Terminate fields by delimiter

33357462	X	Neil Young	X	Musician	X	277	X
----------	---	------------	---	----------	---	-----	---

- Array of offsets

len — offsets[] ——|———— data ——————|

34	10	14	24	32	33357462	Neil Young	Musician	277
----	----	----	----	----	----------	------------	----------	-----

❖ Data Types

DBMSs typically define a fixed set of base types, e.g.

DATE, FLOAT, INTEGER, NUMBER(*n*), VARCHAR(*n*), ...

This determines implementation-level data types for field values:

DATE	time_t
FLOAT	float, double
INTEGER	int, long
NUMBER(<i>n</i>)	int[] (?)
VARCHAR(<i>n</i>)	char[]

PostgreSQL allows new base types to be added

❖ Field Descriptors

A **Tuple** could be implemented as

- a list of field descriptors for a record instance
(where a **FieldDesc** gives (offset,length,type) information)
- along with a reference to the **Record** data

```
typedef struct {
    ushort      nfields;      // number of fields/attrs
    ushort      data_off;     // offset in struct for data
    FieldDesc   fields[];    // field descriptions
    Record      data;        // pointer to record in buffer
} Tuple;
```

Fields are derived from relation descriptor + record instance data.

❖ Field Descriptors (cont)

Tuple **data** could be

- a pointer to bytes stored elsewhere in memory

nfields,data_off,fields,datap



e.g.

nfields	data_off	fields	datap
4	16	(0,4,int) (6,10,char) (18,8,char) (28,2,int)	0

4	33357462	10	Neil Young	8	Musician	2	277
[0]	[2]	[6]	[8]	[18]	[20]	[28]	[30]

Note that the **offset** refers to the length field at the start of each attribute.

❖ Field Descriptors (cont)

Or, tuple **data** could be ...

- appended to **Tuple struct** (used widely in PostgreSQL)

nfields,data_off,fields,data



e.g.

nfields data_off fields

4	16	(0,4,int)	(6,10,char)	(18,8,char)	(28,2,int)	...
---	----	-----------	-------------	-------------	------------	-----

...	4	33357462	10	Neil Young	8	Musician	2	277
-----	---	----------	----	------------	---	----------	---	-----

PostgreSQL Tuples

- [PostgreSQL Tuples](#)
- [PostgreSQL Attribute Values](#)

❖ PostgreSQL Tuples

Definitions: `include/postgres.h`, `include/access/*tup*.h`

Functions: `backend/access/common/*tup*.c` e.g.

- `HeapTuple heap_form_tuple(desc, values[], isnull[])`
- `heap_deform_tuple(tuple, desc, values[], isnull[])`

PostgreSQL implements tuples via:

- a contiguous chunk of memory
- starting with a header giving e.g. #fields, nulls
- followed by data values (as a sequence of `Datum`)

❖ PostgreSQL Tuples (cont)

HeapTupleData contains information about a stored tuple

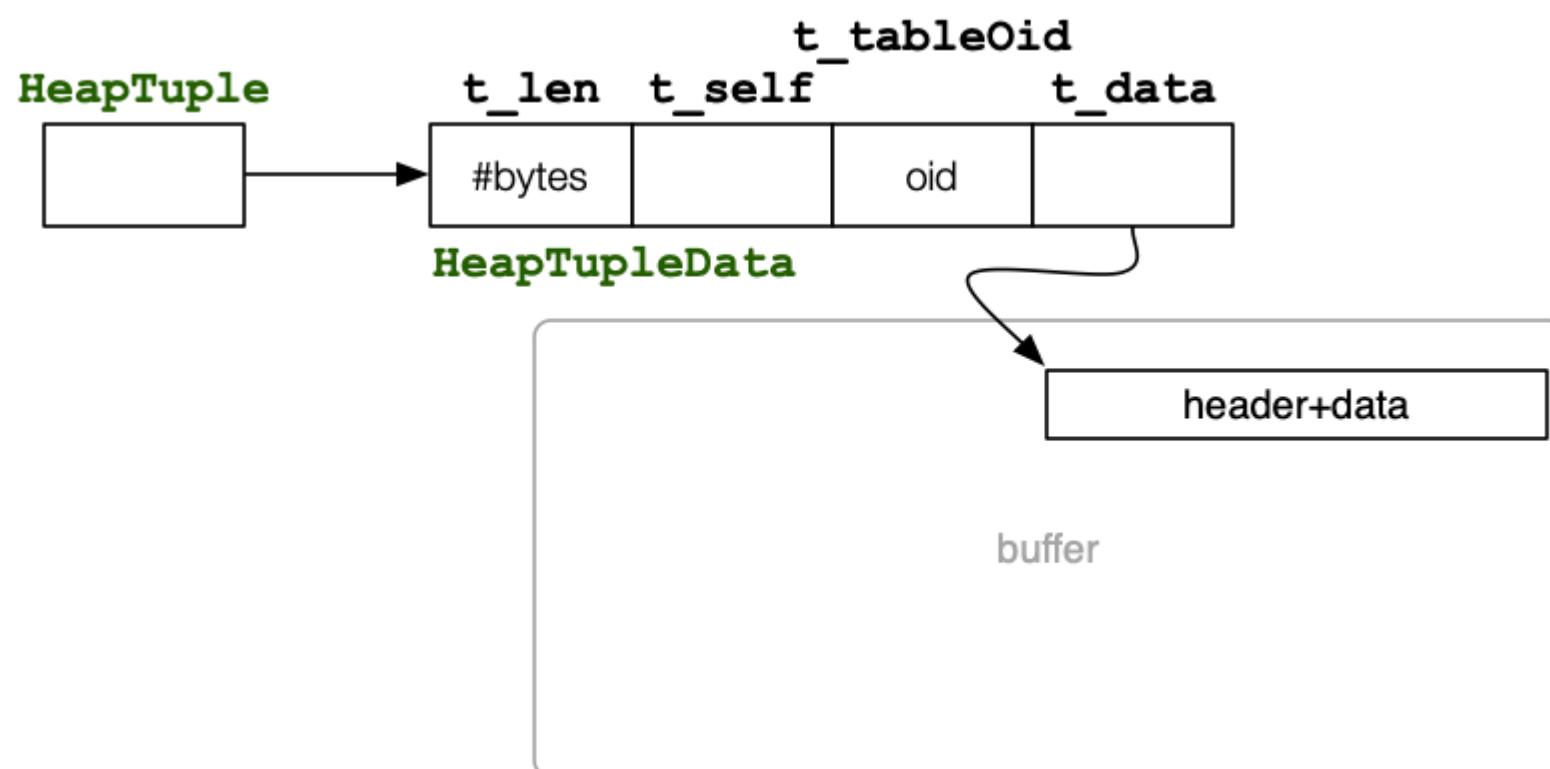
```
include/access/htup.h
typedef HeapTupleData *HeapTuple;

typedef struct HeapTupleData
{
    uint32          t_len;    // length of *t_data
    ItemPointerData t_self;   // SelfItemPointer
    Oid             t_tableOid; // table the tuple came from
    HeapTupleHeader t_data;   // -> tuple header and data
} HeapTupleData;
```

HeapTupleHeader is a pointer to a location in a buffer

❖ PostgreSQL Tuples (cont)

Structure of **HeapTuple**:



❖ PostgreSQL Tuples (cont)

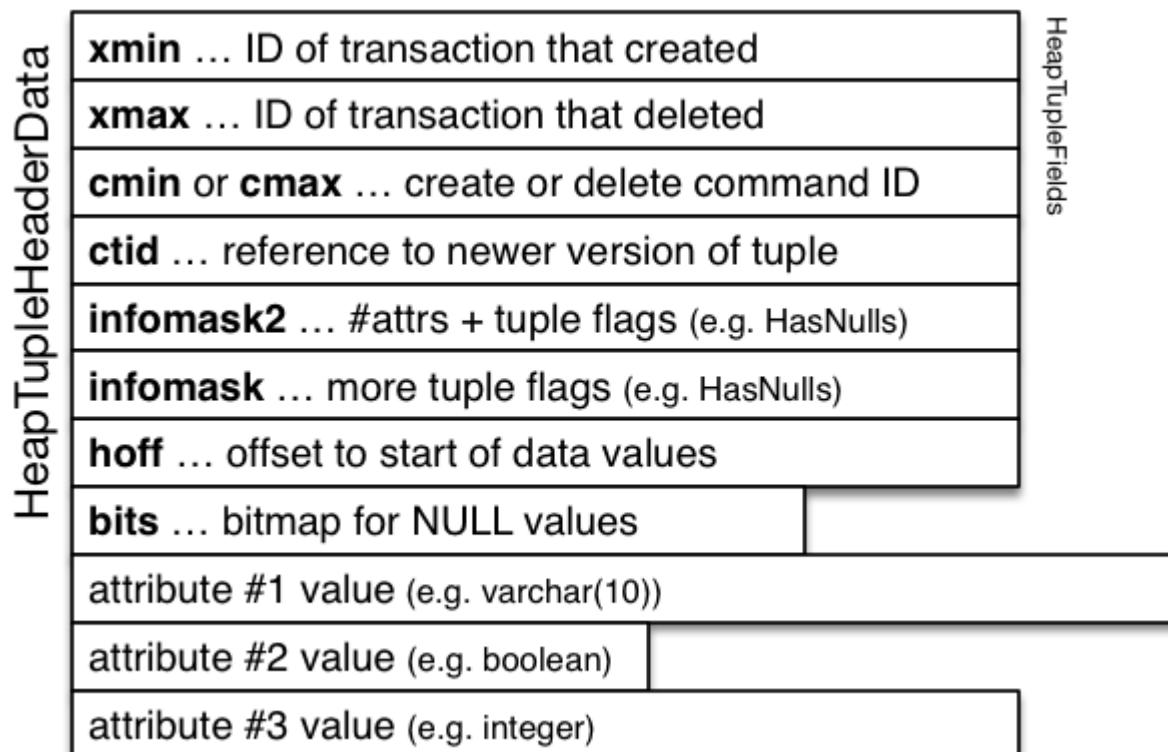
PostgreSQL stores each record as tuple header, followed by data:

```
include/access/htup_details.h
typedef HeapTupleHeaderData *HeapTupleHeader;

typedef struct HeapTupleHeaderData // simplified
{
    HeapTupleFields t_heap;
    ItemPointerData t_ctid;           // TID of newer version
    uint16          t_infomask2;      // #attributes + flags
    uint16          t_infomask;       // flags e.g. has_null
    uint8           t_hoff;           // sizeof header incl. t_bits
    // above is fixed size (23 bytes) for all heap tuples
    bits8          t_bits[1];        // bitmap of NULLs, var.len.
    // OID goes here if HEAP_HASOID is set in t_infomask
    // actual data follows at end of struct
} HeapTupleHeaderData;
```

❖ PostgreSQL Tuples (cont)

Tuple structure:



❖ PostgreSQL Tuples (cont)

Some of the bits in **t_infomask** ..

```
#define HEAP_HASNULL      0x0001
     /* has null attribute(s) */
#define HEAP_HASVARWIDTH   0x0002
     /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL   0x0004
     /* has external stored attribute(s) */
#define HEAP_HASOID_OLD    0x0008
     /* has an object-id field */
```

Location of **NULLs** is stored in **t_bits[]** array

❖ PostgreSQL Tuples (cont)

Tuple-related data types: (cont)

```
typedef struct HeapTupleFields // simplified
{
    TransactionId t_xmin; // inserting xact ID
    TransactionId t_xmax; // deleting or locking xact ID
    union {
        CommandId t_cid; // inserting or deleting command ID
        TransactionId t_xvac; // old-style VACUUM FULL xact ID
    } t_field3;
} HeapTupleFields;
```

Note that not all system fields from stored tuple appear

- **oid** is stored after the tuple header, if used
- both **xmin/xmax** are stored, but only one of **cmin/cmax**

❖ PostgreSQL Tuples (cont)

Tuple-related data types: (cont)

```
include/access/tupdesc.h
// TupleDesc: schema-related information for HeapTuples

typedef struct tupleDesc
{
    int             natts;          // # attributes in tuple
    Oid             tdtypeid;       // composite type ID for tuple type
    int32           tdtypmod;       // typmod for tuple type
    bool            tdhhasoid;      // does tuple have oid attribute?
    int             tdrefcount;     // reference count (-1 if not counting)
    TupleConstr   *constr;         // constraints, or NULL if none
    FormData_pg_attribute attrs[];
    // attrs[N] is a pointer to description of attribute N+1
} *TupleDesc;
```

❖ PostgreSQL Tuples (cont)

Tuple-related data types: (cont)

```
// FormData_pg_attribute:  
// schema-related information for one attribute  
  
typedef struct FormData_pg_attribute  
{  
    Oid        attrelid;      // OID of reln containing attr  
    NameData  attname;       // name of attribute  
    Oid        atttypid;     // OID of attribute's data type  
    int16      attlen;        // attribute length  
    int32      attndims;     // # dimensions if array type  
    bool       attnotnull;   // can attribute have NULL value  
    ....          // and many other fields  
} FormData_pg_attribute;
```

For details, see **include/catalog/pg_attribute.h**

❖ PostgreSQL Attribute Values

Attribute values in PostgreSQL tuples are packaged as **Datums**

```
// representation of a data value
typedef uintptr_t Datum;
```

The actual data value:

- may be stored in the **Datum** (e.g. **int**)
- may have a header with length (for varlen attributes)
- may be stored in a TOAST file (if large value)

❖ PostgreSQL Attribute Values (cont)

Attribute values can be extracted as **Datum** from **HeapTuples**

```
Datum heap_getattr(  
    HeapTuple tup,           // tuple (in memory)  
    int attnum,             // which attribute  
    TupleDesc tupDesc,       // field descriptors  
    bool *isnull            // flag to record NULL  
)
```

isnull is set to true if value of field is **NULL**

attnum can be negative ... to access system attributes (e.g. OID)

For details, see **include/access/htup_details.h**

❖ PostgreSQL Attribute Values (cont)

Values of **Datum** objects can be manipulated via e.g.

```
// DatumGetBool:  
//   Returns boolean value of a Datum.  
  
#define DatumGetBool(X) ((bool) ((X) != 0))  
  
// BoolGetDatum:  
//   Returns Datum representation for a boolean.  
  
#define BoolGetDatum(X) ((Datum) ((X) ? 1 : 0))
```

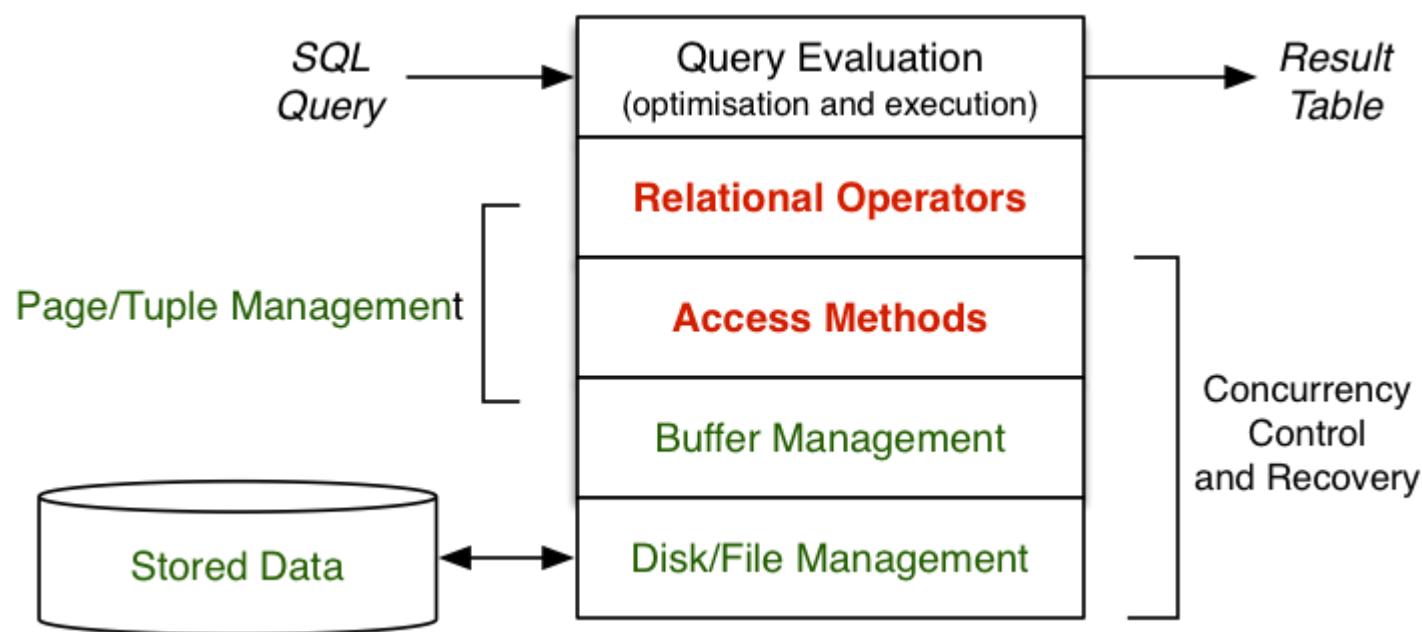
For details, see **include/postgres.h**

Relational Operations

- [DBMS Architecture \(revisited\)](#)
- [Relational Operations](#)
- [Cost Models](#)
- [Query Types](#)

❖ DBMS Architecture (revisited)

Implementation of relational operations in DBMS:



❖ Relational Operations

DBMS core = relational engine, with implementations of

- selection, projection, join, set operations
- scanning, sorting, grouping, aggregation, ...

In this part of the course:

- examine methods for implementing each operation
- develop cost models for each implementation
- characterise when each method is most effective

Terminology reminder:

- tuple = collection of data values under some schema ≈ record
- page = block = collection of tuples + management data = i/o unit
- relation = table ≈ file = collection of tuples

❖ Relational Operations (cont)

In order to implement relational operations the low-levels of the system provides:

- **Relation openRel(db, name)**
 - get handle on relation **name** in database **db**
- **Page request_page(rel, pid)**
 - get page **pid** from relation **rel**, return buffer containing page
- **Record get_record(buf, tid)**
 - return record **tid** from page **buf**
- **Tuple mkTuple(rel, rec)**
 - convert record **rec** to a tuple, based on **rel** schema

❖ Relational Operations (cont)

Example of using low-level functions

```
// scan a relation Emps
Page p; // current page
Tuple t; // current tuple
Relation r = relOpen(db, "Emps");
for (int i = 0; i < nPages(r); i++) {
    p = request_page(rel, i);
    for (int j = 0; j < nRecs(p); j++)
        t = mkTuple(r, get_record(p, j));
    ... process tuple t ...
}
}
```

❖ Relational Operations (cont)

Two "dimensions of variation":

- which relational operation (e.g. Sel, Proj, Join, Sort, ...)
- which access-method (e.g. file struct: heap, indexed, hashed, ...)

Each **query method** involves an operator and a file structure:

- e.g. primary-key selection on hashed file
- e.g. primary-key selection on indexed file
- e.g. join on ordered heap files (sort-merge join)
- e.g. join on hashed files (hash join) which table to hash?
- e.g. two-dimensional range query on R-tree indexed file

We are interested in *cost* of query methods (and insert/delete operations)

❖ Relational Operations (cont)

SQL vs DBMS engine

- **select ... from R where C**
 - find relevant tuples (satisfying C) in file(s) of R
- **insert into R values(...)**
 - place new tuple in some page of a file of R
- **delete from R where C**
 - find relevant tuples and "remove" from file(s) of R
- **update R set ... where C**
 - find relevant tuples in file(s) of R and "change" them

❖ Cost Models

An important aspect of this course is

- analysis of cost of various query methods

Cost can be measured in terms of

- *Time Cost*: total time taken to execute method, or
- *Page Cost*: number of pages read and/or written

Primary assumptions in our cost models:

- memory (RAM) is "small", fast, byte-at-a-time
- disk storage is very large, slow, page-at-a-time

❖ Cost Models (cont)

Since *time cost* is affected by many factors

- speed of i/o devices (fast/slow disk, SSD)
- load on machine

we do not consider time cost in our analyses.

For comparing methods, *page cost* is better

- identifies workload imposed by method
- BUT is clearly affected by buffering

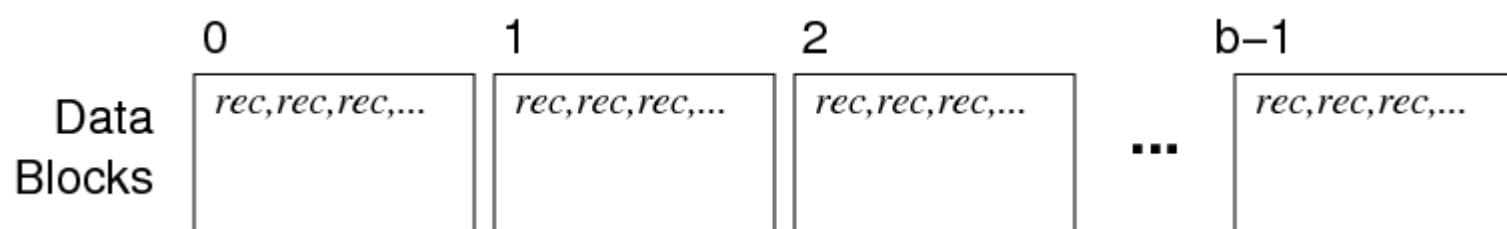
Estimating costs with multiple concurrent ops *and* buffering is difficult!!

Additional assumption: every page request leads to some i/o

❖ Cost Models (cont)

In developing cost models, we also assume:

- a relation is a set of r tuples, with average tuple size R bytes
- the tuples are stored in b data pages on disk
- each page has size B bytes and contains up to c tuples
- the tuples which answer query q are contained in b_q pages
- data is transferred disk \leftrightarrow memory in whole pages
- cost of disk \leftrightarrow memory transfer $T_{r/w}$ is very high



❖ Cost Models (cont)

Our cost models are "rough" (based on assumptions)

But do give an $O(x)$ feel for how expensive operations are.

Example "rough" estimation: how many piano tuners in Sydney?

- Sydney has $\approx 4\ 000\ 000$ people
- Average household size $\approx 3 \therefore 1\ 300\ 000$ households
- Let's say that 1 in 10 households owns a piano
- Therefore there are $\approx 130\ 000$ pianos
- Say people get their piano tuned every 2 years (on average)
- Say a tuner can do 2/day, 250 working-days/year
- Therefore 1 tuner can do 500 pianos per year
- Therefore Sydney would need $\approx 130000/2/500 = 130$ tuners

Actual number of tuners in Yellow Pages = 120

Example borrowed from Alan Fekete at Sydney University.

❖ Query Types

Type	SQL	RelAlg	a.k.a.
Scan	<code>select * from R</code>	R	-
Proj	<code>select x,y from R</code>	$Proj[x,y]R$	-
Sort	<code>select * from R order by x</code>	$Sort[x]R$	<i>ord</i>
Sel_1	<code>select * from R where id = k</code>	$Sel[id=k]R$	<i>one</i>
Sel_n	<code>select * from R where a = k</code>	$Sel[a=k]R$	-
Join	<code>select * from R,S where R.id = S.r</code>	$R \text{Join}[id=r] S$	-

Different query classes exhibit different query processing behaviours.

Scanning

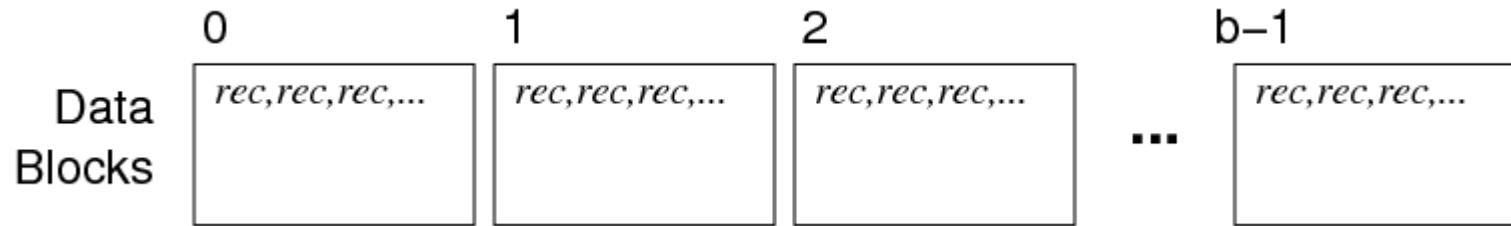
- [Scanning](#)
- [Selection via Scanning](#)
- [Iterators](#)
- [Example Query](#)
- [next_tuple\(.\) Function](#)
- [Relation Copying](#)
- [Scanning in PostgreSQL](#)
- [Scanning in other File Structures](#)

❖ Scanning

Consider executing the query:

```
select * from Rel;
```

where the relation has a file structure like:



This would be done by a simple scan of all records/tuples.

❖ Scanning (cont)

Abstract view of how the scan might be implemented:

```
for each tuple T in relation Rel {  
    add tuple T to result set  
}
```

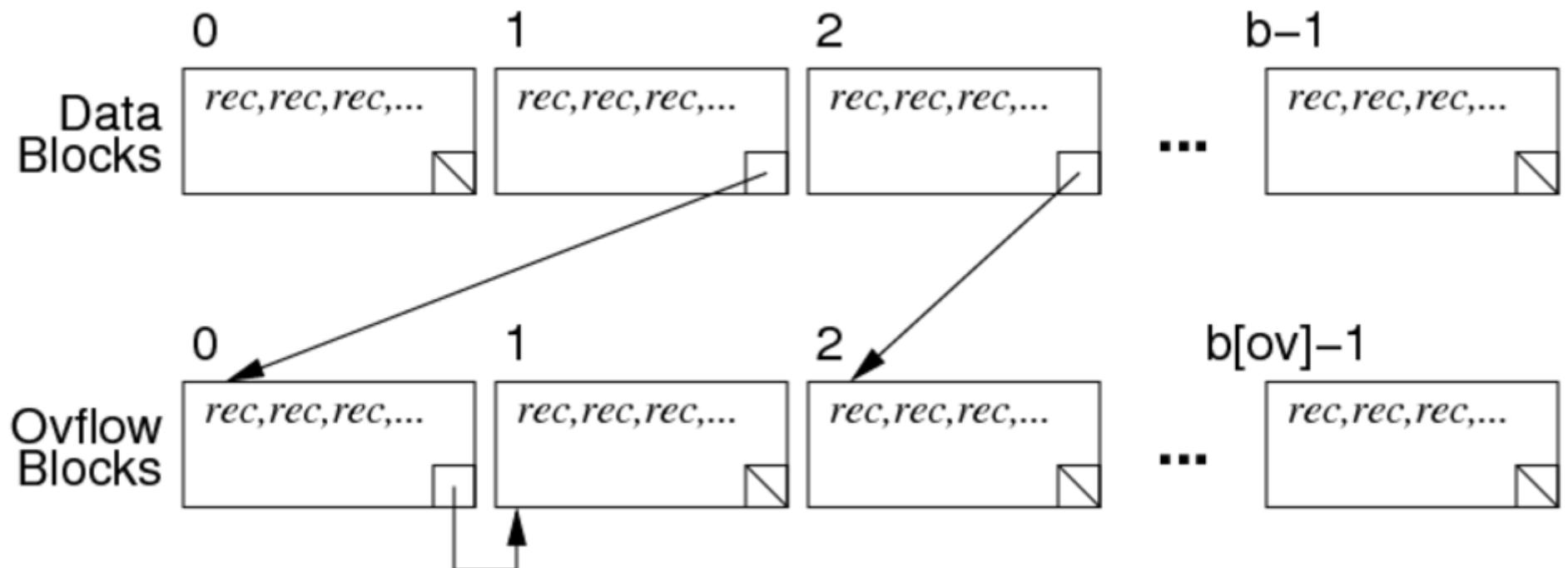
Operational view:

```
for each page P in file of relation Rel {  
    for each tuple T in page P {  
        add tuple T to result set  
    }  
}
```

Cost = read every data page once = b

❖ Scanning (cont)

Consider a file with overflow pages, e.g.



❖ Scanning (cont)

In this case, the implementation changes to:

```
for each page P in data file of relation Rel {  
    for each tuple t in page P {  
        add tuple t to result set  
    }  
    for each overflow page V of page P {  
        for each tuple t in page V {  
            add tuple t to result set  
    }    }    }
```

Cost: read each data page and each overflow page once

$$\text{Cost} = b + b_{OV}$$

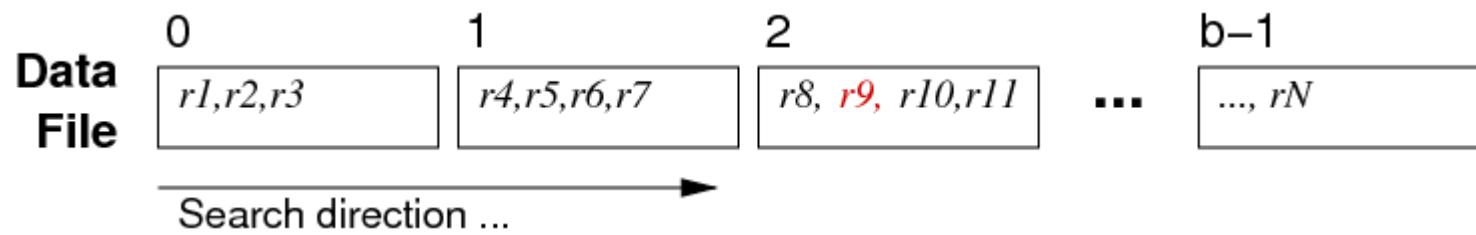
where b_{OV} = total number of overflow pages

❖ Selection via Scanning

Consider a one query like:

```
select * from Employee where id = 762288;
```

In an unordered file, search for matching tuple requires:



Guaranteed at most one answer; but could be in any page.

❖ Selection via Scanning (cont)

Overview of scan process:

```
for each page P in relation Employee {  
    for each tuple t in page P {  
        if (t.id == 762288) return t  
    } }
```

Cost analysis for one searching in unordered file

- best case: read one page, find tuple
- worst case: read all b pages, find in last (or don't find)
- average case: read half of the pages ($b/2$)

Page Costs: $\text{Cost}_{avg} = b/2$ $\text{Cost}_{min} = 1$ $\text{Cost}_{max} = b$

❖ Iterators

Access methods typically involve **iterators**, e.g.

Scan s = start_scan(Relation r, ...)

- commence a scan of relation **r**
- **Scan** may include condition to implement **WHERE**-clause
- **Scan** holds data on progress through file (e.g. current page)

Tuple next_tuple(Scan s)

- return **Tuple** immediately following last accessed one
- returns **NULL** if no more **Tuples** left in the relation

❖ Example Query

Example: simple scan of a table ...

```
select name from Employee
```

implemented as:

```
DB db = openDatabase("myDB");
Relation r = openRelation(db, "Employee", READ);
Scan s = start_scan(r);
Tuple t; // current tuple
while ((t = next_tuple(s)) != NULL) {
    char *name = getStrField(t, 2);
    printf("%s\n", name);
}
```

❖ next_tuple() Function

Consider the following possible **Scan** data structure

```
typedef ScanData *Scan;

typedef struct {
    Relation rel;
    Page    *page;      // Page buffer
    int      curPID;   // current pid
    int      curTID;   // current tid
} ScanData;
```

Assume tuples are indexed 0..**nTuples(p)-1**

Assume pages are indexed 0..**nPages(rel)-1**

❖ **next_tuple()** Function (cont)

Implementation of **Tuple next_tuple(Scan s)** function

```
Tuple next_tuple(Scan s)
{
    if (s->curTID >= nTuples(s->page)-1) {
        // get a new page; exhausted current page
        s->curPID++;
        if (s->curPID >= nPages(s->rel))
            return NULL;
        else {
            s->page = get_page(s->rel, s->curPID);
            s->curTID = -1;
        }
    }
    s->curTID++;
    return get_tuple(s->rel, s->page, s->curTID);
}
```

❖ Relation Copying

Consider an SQL statement like:

```
create table T as (select * from S);
```

Effectively, copies data from one table to a new table.

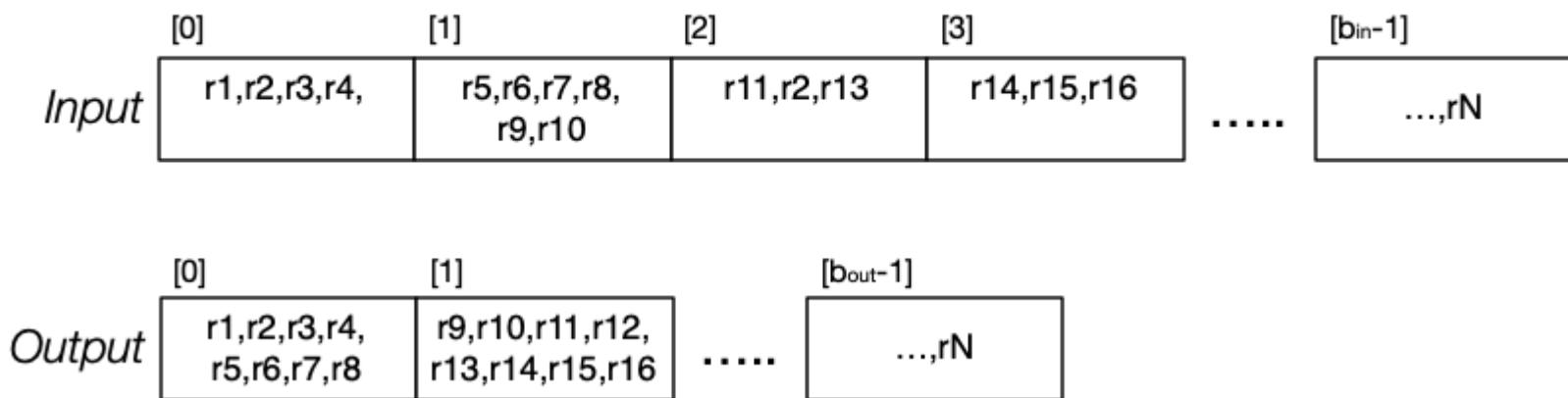
Process:

```
make empty relation T
s = start scan of S
while (t = next_tuple(s)) {
    insert tuple t into relation T
}
```

❖ Relation Copying (cont)

It is possible that \mathbf{T} is smaller than \mathbf{S}

- may be unused free space in \mathbf{S} where tuples were removed
- if \mathbf{T} is built by simple append, will be compact



❖ Relation Copying (cont)

In terms of existing relation/page/tuple operations:

```

Relation in;          // relation handle (incl. files)
Relation out;         // relation handle (incl. files)
int ipid,opid,tid;   // page and record indexes
Record rec;          // current record (tuple)
Page ibuf,obuf;       // input/output file buffers

in = openRelation("S", READ);
out = openRelation("T", NEW|WRITE);
clear(obuf); opid = 0;
for (ipid = 0; ipid < nPages(in); ipid++) {
    ibuf = get_page(in, ipid);
    for (tid = 0; tid < nTuples(ibuf); tid++) {
        rec = get_record(ibuf, tid);
        if (!hasSpace(obuf,rec)) {
            put_page(out, opid++, obuf);
            clear(obuf);
        }
        insert_record(obuf,rec);
    }
}
if (nTuples(obuf) > 0) put_page(out, opid, obuf);

```

❖ Scanning in PostgreSQL

Scanning defined in: [backend/access/heap/heapam.c](#)

Implements iterator data/operations:

- **HeapScanDesc** ... struct containing iteration state
- **scan = heap_beginscan(rel, ..., nkeys, keys)**
- **tup = heap_getnext(scan, direction)**
- **heap_endscan(scan)** ... frees up **scan** struct
- **res = HeapKeyTest(tuple, ..., nkeys, keys)**
... performs **ScanKeys** tests on tuple ... is it a result tuple?

❖ Scanning in PostgreSQL (cont)

```
typedef HeapScanDescData *HeapScanDesc;

typedef struct HeapScanDescData
{
    // scan parameters
    Relation      rs_rd;          // heap relation descriptor
    Snapshot      rs_snapshot;    // snapshot ... tuple visibility
    int           rs_nkeys;       // number of scan keys
    ScanKey       rs_key;         // array of scan key descriptors
    ...
    // state set up at initscan time
    PageNumber    rs_npaged;      // number of pages to scan
    PageNumber    rs_startpage;   // page # to start at
    ...
    // scan current state, initially set to invalid
    HeapTupleData rs_ctup;        // current tuple in scan
    PageNumber    rs_cpage;       // current page # in scan
    Buffer        rs_cbuf;        // current buffer in scan
    ...
} HeapScanDescData;
```

❖ Scanning in other File Structures

Above examples are for **heap** files

- simple, unordered, maybe indexed, no hashing

Other access file structures in PostgreSQL:

- **btree, hash, gist, gin**
- each implements:
 - startscan, getnext, endscan
 - insert, delete (update=delete+insert)
 - other file-specific operators

Sorting

- [The Sort Operation](#)
- [Two-way Merge Sort](#)
- [Comparison for Sorting](#)
- [Cost of Two-way Merge Sort](#)
- [n-Way Merge Sort](#)
- [Cost of n-Way Merge Sort](#)
- [Sorting in PostgreSQL](#)

❖ The Sort Operation

Sorting is explicit in queries only in the **order by** clause

```
select * from Students order by name;
```

Sorting is used internally in other operations:

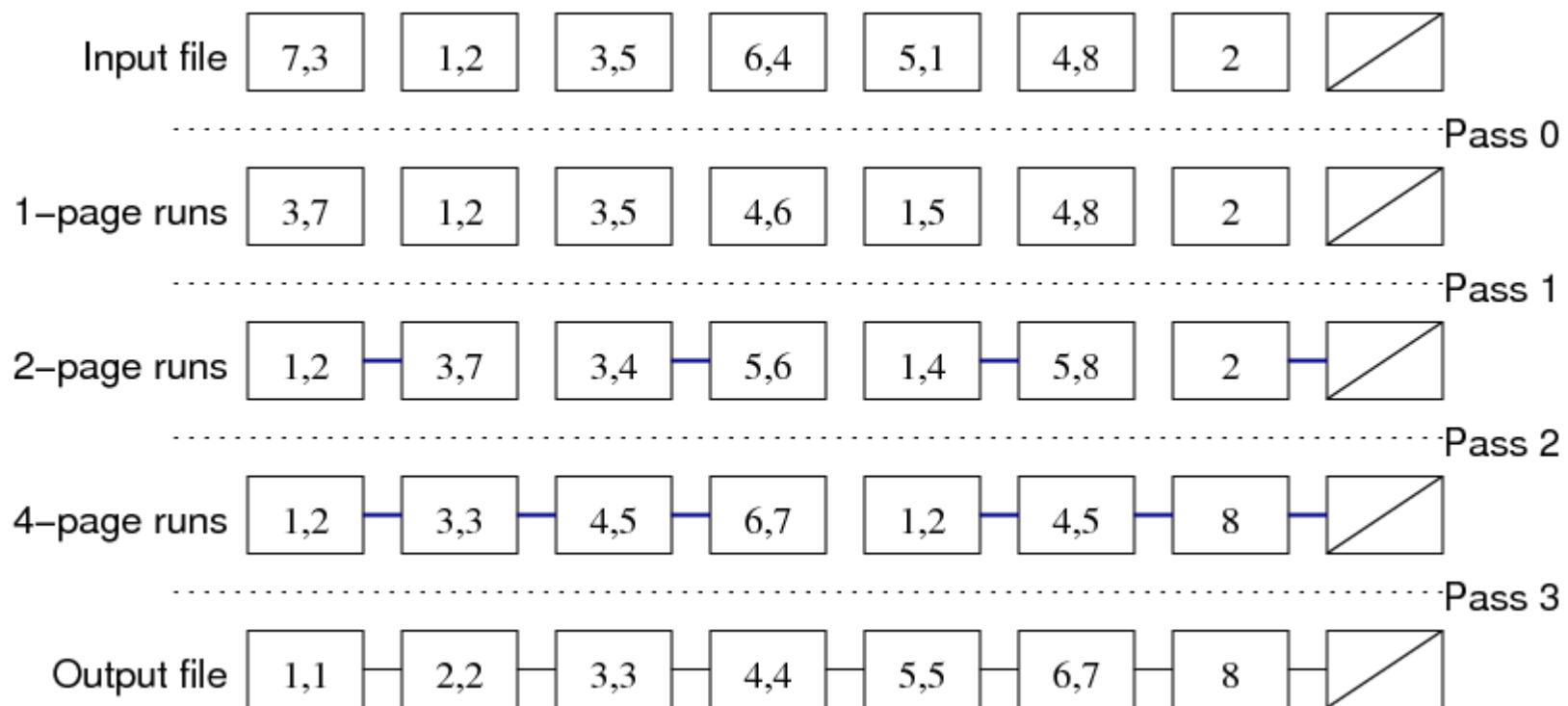
- eliminating duplicate tuples for projection
- ordering files to enhance select efficiency
- implementing various styles of join
- forming tuple groups in **group by**

Sort methods such as quicksort are designed for in-memory data.

For large data on disks, need external sorts such as **merge sort**.

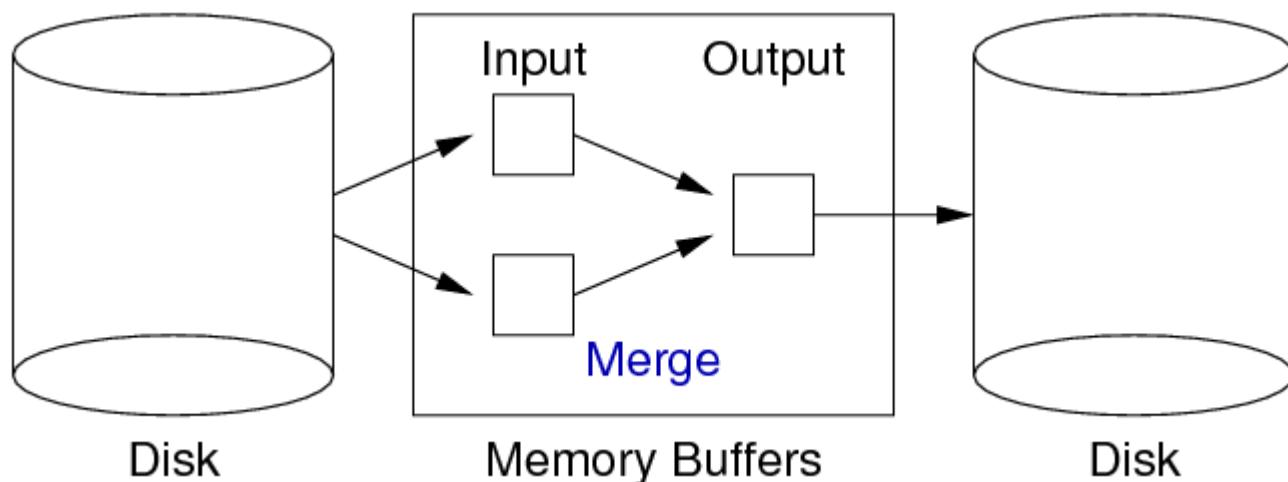
❖ Two-way Merge Sort

Example:



❖ Two-way Merge Sort (cont)

Requires at least three in-memory buffers:



Assumption: cost of Merge operation on two in-memory buffers ≈ 0 .

❖ Comparison for Sorting

Above assumes that we have a function to compare tuples.

Needs to understand ordering on different data types.

Need a function **tupCompare(r1, r2, f)** (cf. C's **strcmp**)

```
int tupCompare(r1,r2,f)
{
    if (r1.f < r2.f) return -1;
    if (r1.f > r2.f) return 1;
    return 0;
}
```

Assume =, <, > are available for all attribute types.

❖ Comparison for Sorting (cont)

In reality, need to sort on multiple attributes and ASC/DESC, e.g.

```
-- example multi-attribute sort
select * from Students
order by age desc, year_enrolled
```

Sketch of multi-attribute sorting function

```
int tupCompare(r1,r2,criteria)
{
    foreach (f,ord) in criteria {
        if (ord == ASC) {
            if (r1.f < r2.f) return -1;
            if (r1.f > r2.f) return 1;
        }
        else {
            if (r1.f > r2.f) return -1;
            if (r1.f < r2.f) return 1;
        }
    }
    return 0;
}
```

❖ Cost of Two-way Merge Sort

For a file containing b data pages:

- require $\text{ceil}(\log_2 b)$ passes to sort,
- each pass requires b page reads, b page writes

Gives total cost: $2.b.\text{ceil}(\log_2 b)$

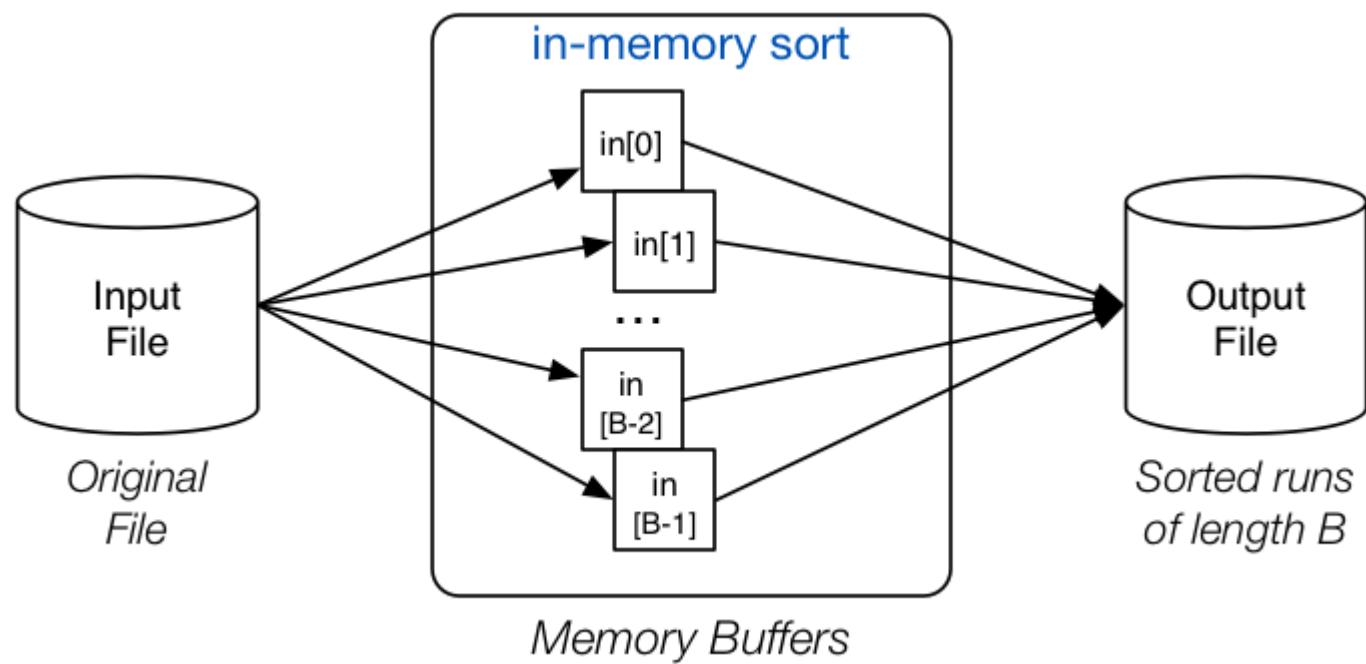
Example: Relation with $r=10^5$ and $c=50 \Rightarrow b=2000$ pages.

Number of passes for sort: $\text{ceil}(\log_2 2000) = 11$

Reads/writes entire file 11 times! Can we do better?

❖ n-Way Merge Sort

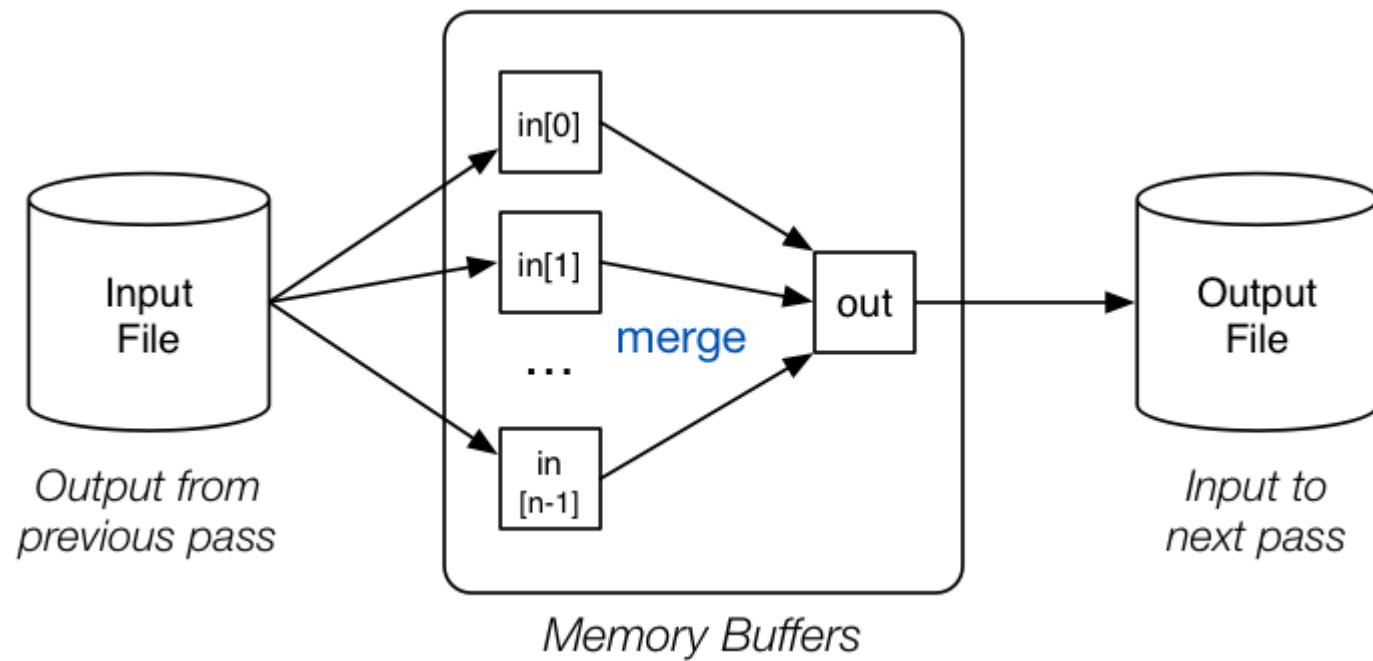
Initial pass uses: B total buffers



Reads B pages at a time, sorts in memory, writes out in order

❖ n-Way Merge Sort (cont)

Merge passes use: $n = B-1$ input buffers, 1 output buffer



❖ n-Way Merge Sort (cont)

Method:

```
// Produce B-page-long runs
for each group of B pages in Rel {
    read B pages into memory buffers
    sort group in memory
    write B pages out to Temp
}
// Merge runs until everything sorted
numberOfRuns = ceil(b/B)
while (numberOfRuns > 1) {
    // n-way merge, where n=B-1
    for each group of n runs in Temp {
        merge into a single run via input buffers
        write run to newTemp via output buffer
    }
    numberOfRuns = ceil(numberOfRuns/n)
    Temp = newTemp // swap input/output files
}
```

❖ Cost of n-Way Merge Sort

Consider file where $b = 4096$, $B = 16$ total buffers:

- pass 0 produces 256×16 -page sorted runs
- pass 1
 - performs 15-way merge of groups of 16-page sorted runs
 - produces 18×240 -page sorted runs (17 full runs, 1 short run)
- pass 2
 - performs 15-way merge of groups of 240-page sorted runs
 - produces 2×3600 -page sorted runs (1 full run, 1 short run)
- pass 3
 - performs 15-way merge of groups of 3600-page sorted runs
 - produces 1×4096 -page sorted runs

(cf. two-way merge sort which needs 11 passes)

❖ Cost of n-Way Merge Sort (cont)

Generalising from previous example ...

For b data pages and B buffers

- first pass: read/writes b pages, gives $b_0 = \text{ceil}(b/B)$ runs
- then need $\text{ceil}(\log_n b_0)$ passes until sorted, where $n = B-1$
- each pass reads and writes b pages (i.e. $2.b$ page accesses)

Cost = $2.b.(1 + \text{ceil}(\log_n b_0))$, where b_0 and n are defined above

❖ Sorting in PostgreSQL

Sort uses a merge-sort (from Knuth) similar to above:

- `backend/utils/sort/tuplesort.c`
- `include/utils/sortsupport.h`

Tuples are mapped to **SortTuple** structs for sorting:

- containing pointer to tuple and sort key
- no need to reference actual Tuples during sort
- unless multiple attributes used in sort

If all data fits into memory, sort using **qsort()**.

If memory fills while reading, form "runs" and do disk-based sort.

❖ Sorting in PostgreSQL (cont)

Disk-based sort has phases:

- divide input into sorted runs using HeapSort
- merge using N buffers, one output buffer
- $N =$ as many buffers as **workMem** allows

Described in terms of "tapes" ("tape" \cong sorted run)

Implementation of "tapes": [backend/utils/sort/logtape.c](#)

❖ Sorting in PostgreSQL (cont)

Sorting comparison operators are obtained via catalog:

```
// backend/utils/sort/tuplesort.c
// gets pointer to function via pg_operator
struct Tuplesortstate { ... SortTupleComparator ... };

// include/utils/sortsupport.h
// returns negative, zero, positive
ApplySortComparator(Datum datum1, bool isnull1,
                     Datum datum2, bool isnull2,
                     SortSupport sort_helper);
```

Flags in **SortSupport** indicate: ascending/descending, nulls-first/last.

ApplySortComparator() is PostgreSQL's version of **tupCompare()**

COMP9315 Week 3 Exercises

- [Assignment 1](#)
- [Variable-length Structs](#)
- [Exercise 1: Building Tuples in PostgreSQL](#)
- [Exercise 2: How big is a **FieldDesc**?](#)
- [Exercise 3: Cost of Relation Scan](#)
- [Exercise 4: Cost of Search in Sorted File](#)
- [Exercise 5: Cost of Search in Hashed File](#)
- [Exercise 6: Update Operation Costs](#)
- [Exercise 7: Cost of n-Way Merge Sort](#)
- [Exercise 8: Cost of Relation Copy](#)
- [Exercise 9: PostgreSQL Sort](#)

❖ Assignment 1

Creating a new base type requires

- telling the SQL front-end about it
- building C functions to manipulate values of the type
- setting up ordering to allow indexing

At the SQL level (**gcoord.source**)...

```
create type GeoCoord ( type info and function links )
```

Also useful to define comparison operators on the type (e.g. $< > =$)

❖ Assignment 1 (cont)

Once created, the type can be used in client SQL programs ...

e.g. a new base type GeoCoord in schemas ...

```
create table StoreInfo (id integer primary key, location GeoCoord, ...);
```

e.g. inserting data ...

```
insert into StoreInfo values (1, 'Sydney,33.86°S,151.21°E');
```

e.g. retrieving ...

```
select * from StoreInfo  
where location = 'Melbourne,37.84°S,144.95°E';
```

```
select * from StoreInfo
```

```
where location > 'Sydney,33.86°S,151.21°E';
```

❖ Assignment 1 (cont)

At the C level (**gcoord.c**) ...

```
PG_FUNCTION_INFO_V1(gcoord_in);

Datum
gcoord_in(PG_FUNCTION_ARGS)
{
    // parse input string
    // convert to internal representation
    // return value as Datum
}
```

Link between C and SQL (**gcoord.source**) ...

```
CREATE FUNCTION gcoord_in(cstring)
RETURNS GeoCoord
AS '_OBJWD_/gcoord'
LANGUAGE C IMMUTABLE STRICT;
```

❖ Assignment 1 (cont)

Required functions for type T

- **$T_in()$** ... invoked when PG receives value of type T
- **$T_out()$** ... convert value of type T to printable

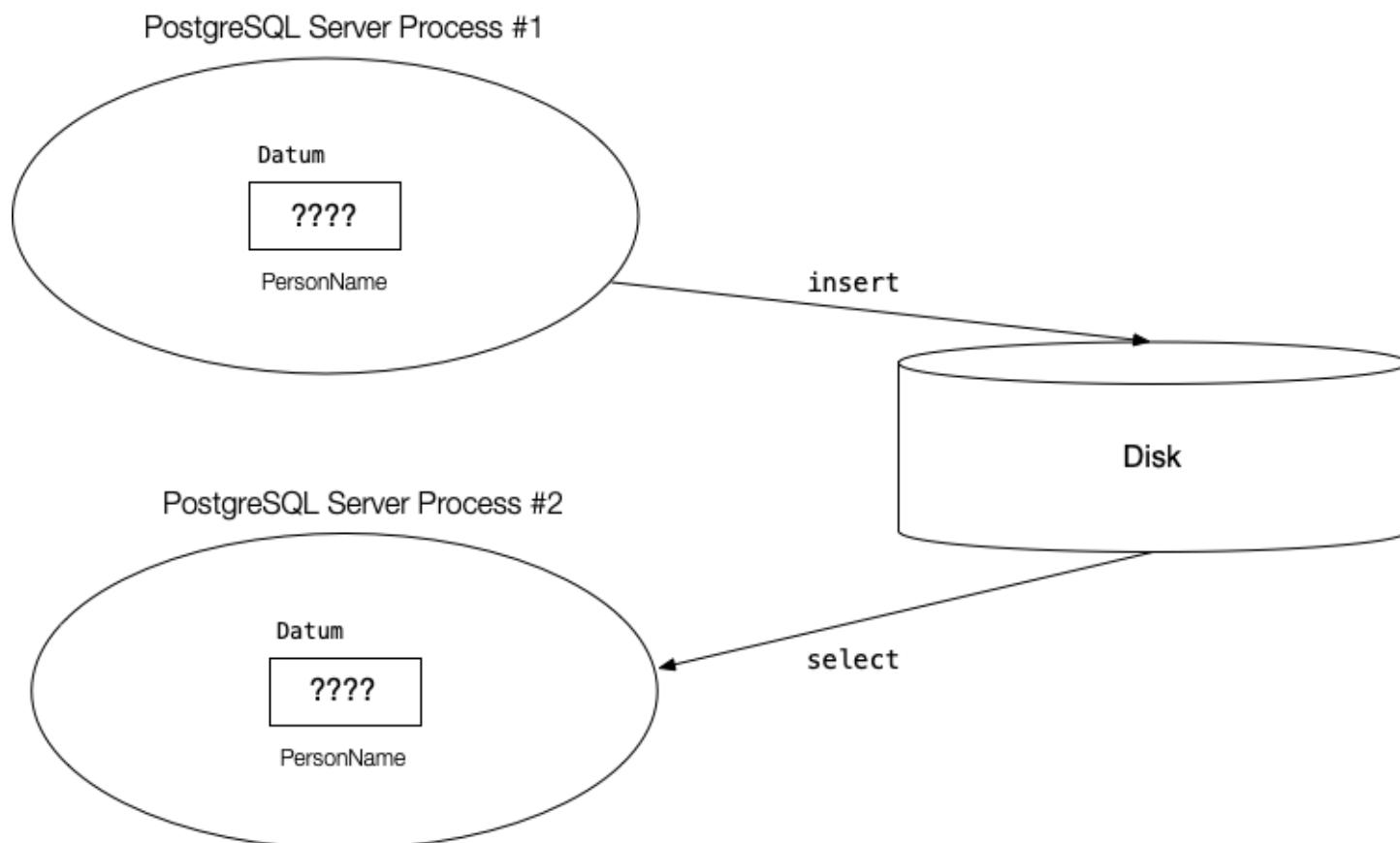
Useful options for new type

- **INTERNALLENGTH = nbytes|VARIABLE**
- **ALIGNMENT = char|int2|int4|double**

See PG Manual **CREATE TYPE** under SQL Commands for more details.

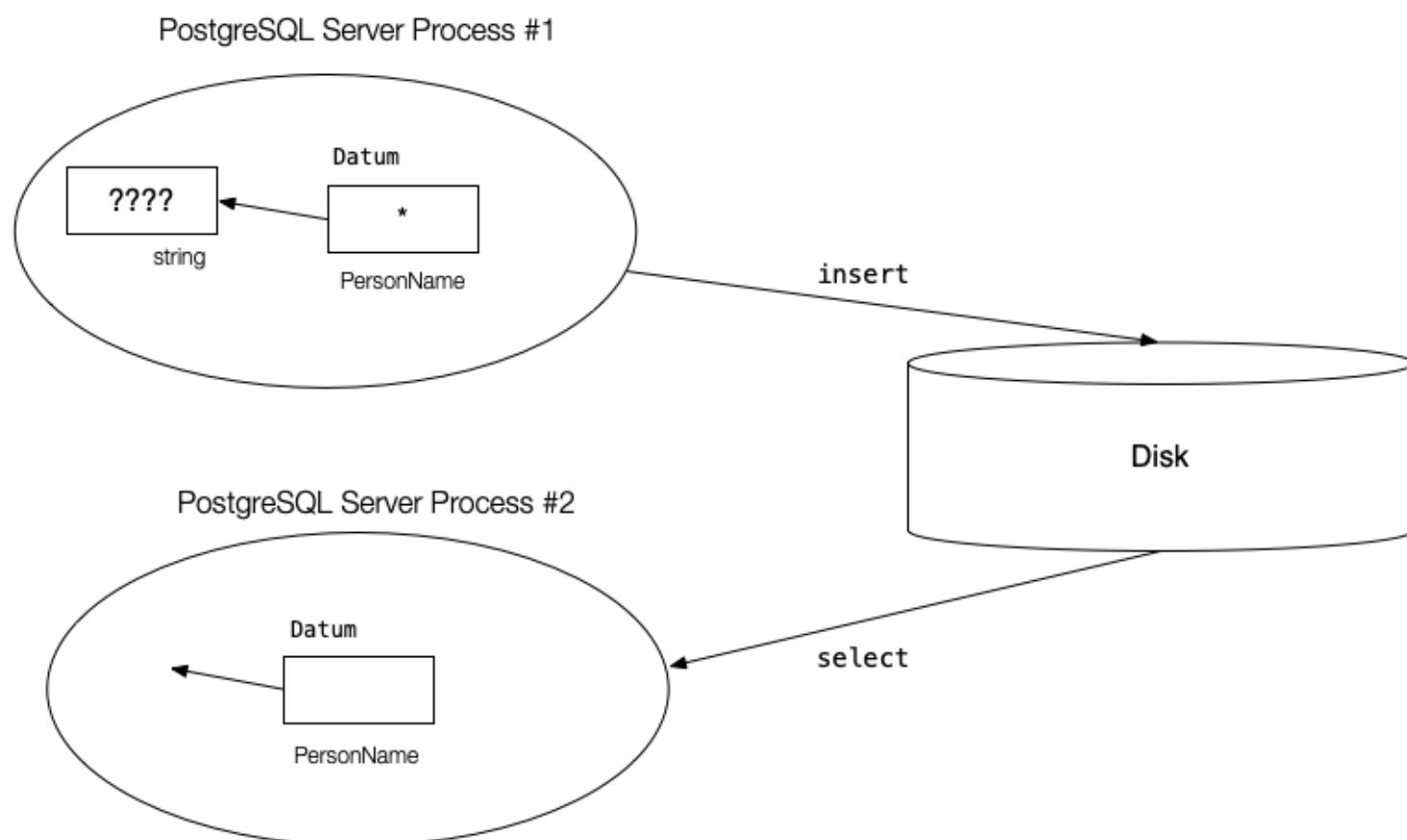
❖ Assignment 1 (cont)

Data transfers that *should* happen ...



❖ Assignment 1 (cont)

Data transfers that students sometimes implement ...



❖ Assignment 1 (cont)

Copy **complex.c** and **complex.source** ... BUT

- **Complex** is a fixed-length type (two **ints**)
- do not make **GeoCoord** fixed-length (how long?)
- change **all** references to **Complex** to **GeoCoord**
- make sure **_OBJWD_** references the correct directory

This assignment requires some prior reading (code + doco)

- do **not** leave it to the last minute

❖ Assignment 1 (cont)

Debugging the assignment ...

- can't easily use GDB or vscode; need debugging print's
- can use **ereport()** and **elog()**; see PG Docs 56.2

Implementing **GeoCoord** ...

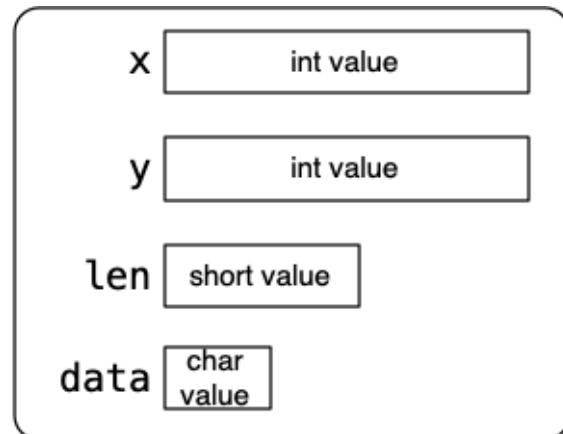
- three data usages: readable, computable, storeable
 - readable → storeable in **gcoord_in()**
 - storeable → readable in **gcoord_out()**
 - storeable → computable in e.g. **gcoord_cmp()**
- reminder: pointers to **malloc**'d memory structures are not storeable

Rebuilding PostgreSQL will *not* solve problems in **gcoord.***

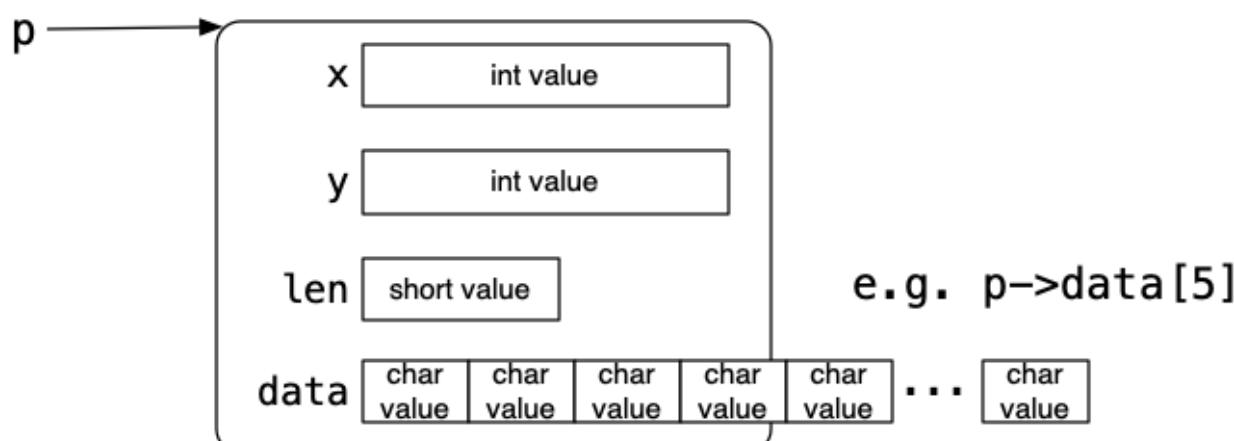
❖ Variable-length Structs

PostgreSQL uses the following trick:

```
struct vlen {
    int x;
    int y;
    short len;
    char data[1];
}
```

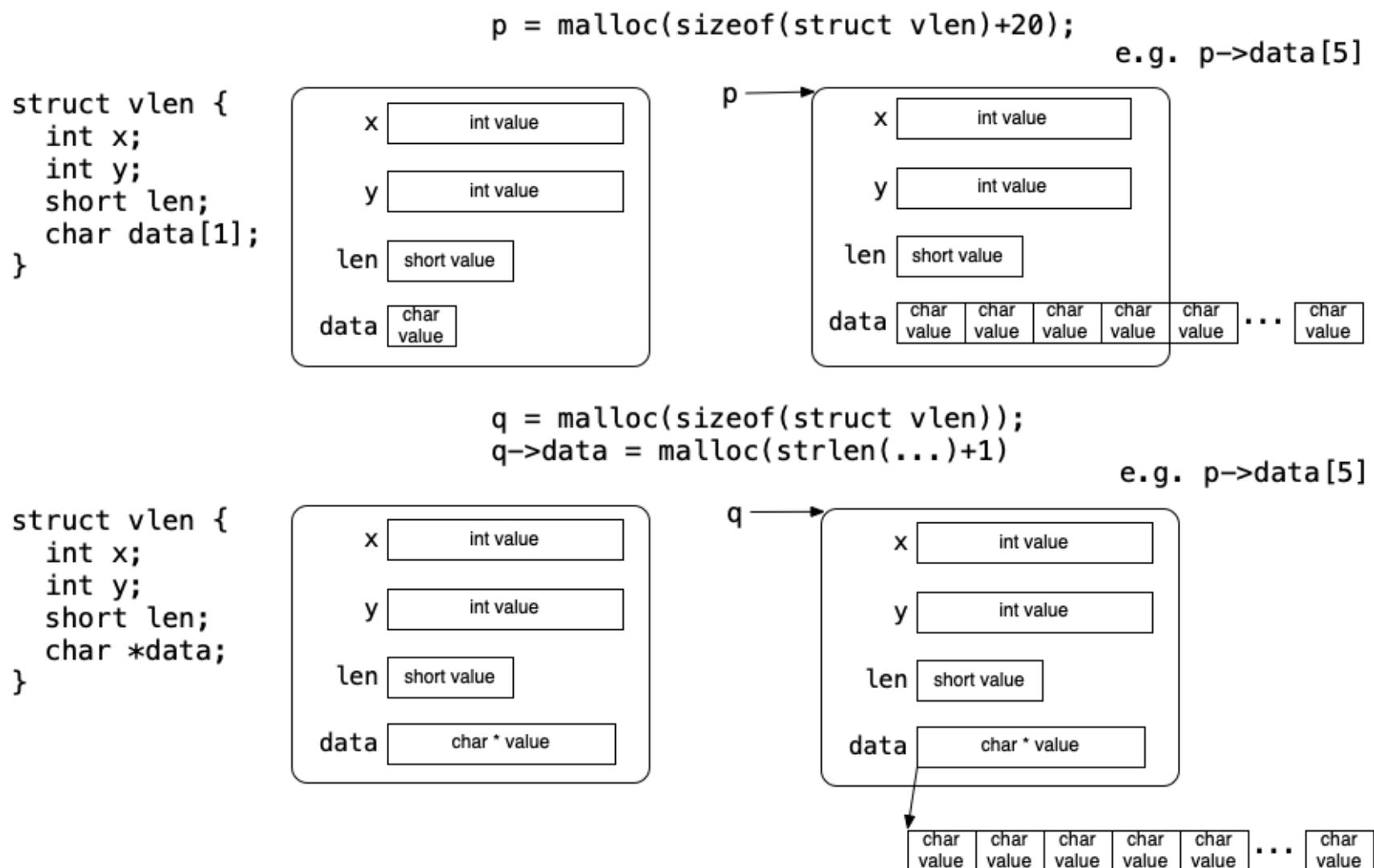


```
p = malloc(sizeof(struct vlen)+20)
```



❖ Variable-length Structs (cont)

Difference between in-struct and ex-struct:



❖ Exercise 1: Building Tuples in PostgreSQL

Examine the code for

```
HeapTuple heap_form_tuple(desc,values[],isnull[])
```

and determine how a PostgreSQL tuple is built

```
HeapTuple = -> HeapTupleData
HeapTupleData =
  (t_len,t_self,t_tableOid,t_data->HeapTupleHeaderData)
HeapTupleHeaderData =
  (t_heap,t_ctid,t_infomask2,t_infomask,t_off,t_bits[],...)
HeapTupleFields =
  (t_xmin,t_xmax,(t_cid|t_xvac))
TupleDesc =
  (natts,tdtypeid,tdtypmod,tdhasoid,constraints[],atts[])
FormData_pg_attribute =
  (attrelid,attname,atttypid,attlen,attndims,attnotnull,...)
```

❖ Exercise 2: How big is a FieldDesc?

FieldDesc = (offset,length,type), where

- offset = offset of field within record data
- length = length (in bytes) of field
- type = data type of field

If pages are 8KB in size, how many bits are needed for each?

E.g.

nfields	data_off	fields = FieldDesc[4]
4	16	(0,4,int) (6,10,char) (18,8,char) (28,2,int)

❖ Exercise 3: Cost of Relation Scan

How much does it cost to scan all tuples in a relation?

E.g. 10000 tuples, 100 tuples per page, 10ms to read a page

Give examples of when this might be needed?

❖ Exercise 4: Cost of Search in Sorted File

How much does it cost to find a record in a relation?

- if the file is sorted on the search field
- each page includes min/max values for each field

Example query: `select * from R where x = 42;`

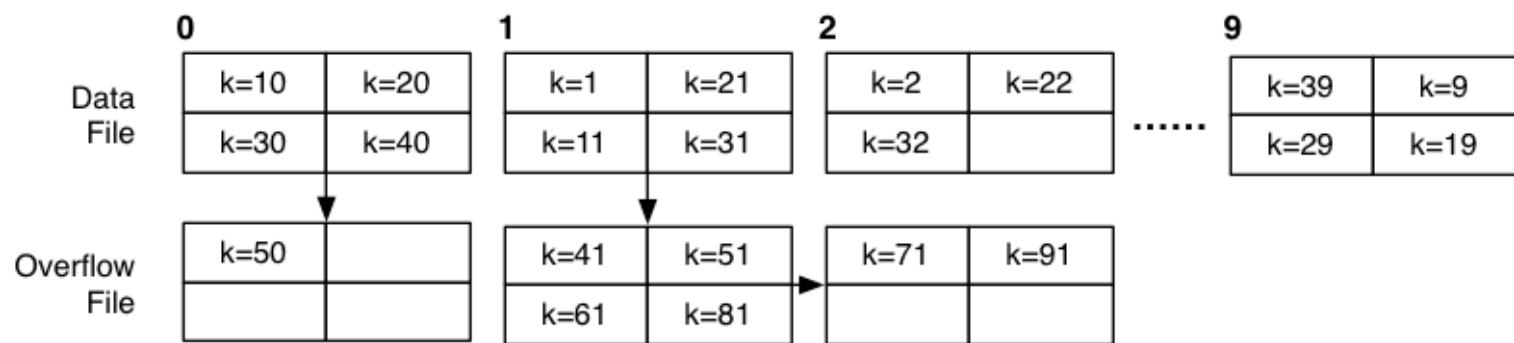
Assume: $B = 8192$, record is located in page 10

Give a pseudo-code algorithm for the search process

Give upper and lower bounds for cost.

❖ Exercise 5: Cost of Search in Hashed File

Consider the hashed file structure $b = 10, c = 4, h(k) = k \% 10$



Describe how the following queries

```
select * from R where k = 51;
select * from R where k > 50;
```

might be solved in a file structure like the above ($h(k) = k \% b$).

Estimate the minimum and maximum cost (as #pages read)

❖ Exercise 6: Update Operation Costs

For each of the following file structures

- heap file, sorted file, hash file

Determine #page-reads + #page-writes for insert and delete

You can assume the existence of a file header containing

- values for r, R, b, B, c
- index of first page with free space (and a free list)

Assume also

- each page contains a header and directory as well as tuples
- no buffering (worst case scenario)
- sorted and hash files use overflow pages

❖ Exercise 7: Cost of n-Way Merge Sort

How many reads+writes to sort the following:

- $r = 1048576$ tuples (2^{20})
- $R = 62$ bytes per tuple (fixed-size)
- $B = 4096$ bytes per page
- $H = 96$ bytes of header data per page
- all pages are full

Consider for the cases:

- 9 total buffers, 8 input buffers, 1 output buffer
- 33 total buffers, 32 input buffers, 1 output buffer
- 257 total buffers, 256 input buffers, 1 output buffer

❖ Exercise 8: Cost of Relation Copy

Analyse cost for relation copying:

1. if both input and output are heap files
2. if input is sorted and output is heap file
3. if input is heap file and output is sorted

Assume ...

- r records in input file, c records/page
- b_{in} = number of pages in input file
- some pages in input file are *not* full
- all pages in output file are full (except the last)

Give cost in terms of #pages read + #pages written

❖ Exercise 9: PostgreSQL Sort

Explore the PostgreSQL code for sorting

`src/backend/utils/sort/...`

`include/utils/sortsupport.h`

Week 4

- [Week 04](#)

❖ Week 04

Things to Note ...

- Quiz 2 due before Friday 10 March, 9pm
- Quizzes are *fortnightly* (weeks 2,4,6,8,10)

This Week ...

- Projection
- Selection (heaps, sorted files)
- 1d Selection (hashing)

Coming Up ...

- 1d Selection (indexing), Nd Selection

Produced: 23 Feb 2023

Implementing Projection

- [The Projection Operation](#)
- [Sort-based Projection](#)
- [Cost of Sort-based Projection](#)
- [Hash-based Projection](#)
- [Cost of Hash-based Projection](#)
- [Projection on Primary Key](#)
- [Index-only Projection](#)
- [Comparison of Projection Methods](#)
- [Projection in PostgreSQL](#)

❖ The Projection Operation

Consider the query:

```
select distinct name, age from Employee;
```

If the **Employee** relation has four tuples such as:

```
(94002, John, Sales, Manager, 32)
(95212, Jane, Admin, Manager, 39)
(96341, John, Admin, Secretary, 32)
(91234, Jane, Admin, Secretary, 21)
```

then the result of the projection is:

```
(Jane, 21)    (Jane, 39)    (John, 32)
```

Note that duplicate tuples (e.g. **(John, 32)**) are eliminated.

❖ The Projection Operation (cont)

Relies on function **Tuple projTuple(AttrList, Tuple)**

- first arg is list of attributes
- second arg is a tuple containing those attributes (and more)
- return value is a new tuple containing only those attributes

Examples, using tuples of type **(id:int, name:text, degree:int)**

```
projTuple([id], (1234, 'John', 3778))  
returns (id=1234)
```

```
projTuple([name,degree]), (1234, 'John', 3778))  
returns (name='John', degree=3778)
```

❖ The Projection Operation (cont)

Without **distinct**, projection is straightforward

```
// attrs = [attr1, attr2, ...]
bR = nPages(Rel)
for i in 0 .. bR-1 {
    P = read page i
    for j in 0 .. nTuples(P)-1 {
        T = getTuple(P, j)
        T' = projTuple(attrs, T)
        if (outBuf is full) write and clear
            append T' to outBuf
    }
}
if (nTuples(outBuf) > 0) write
```

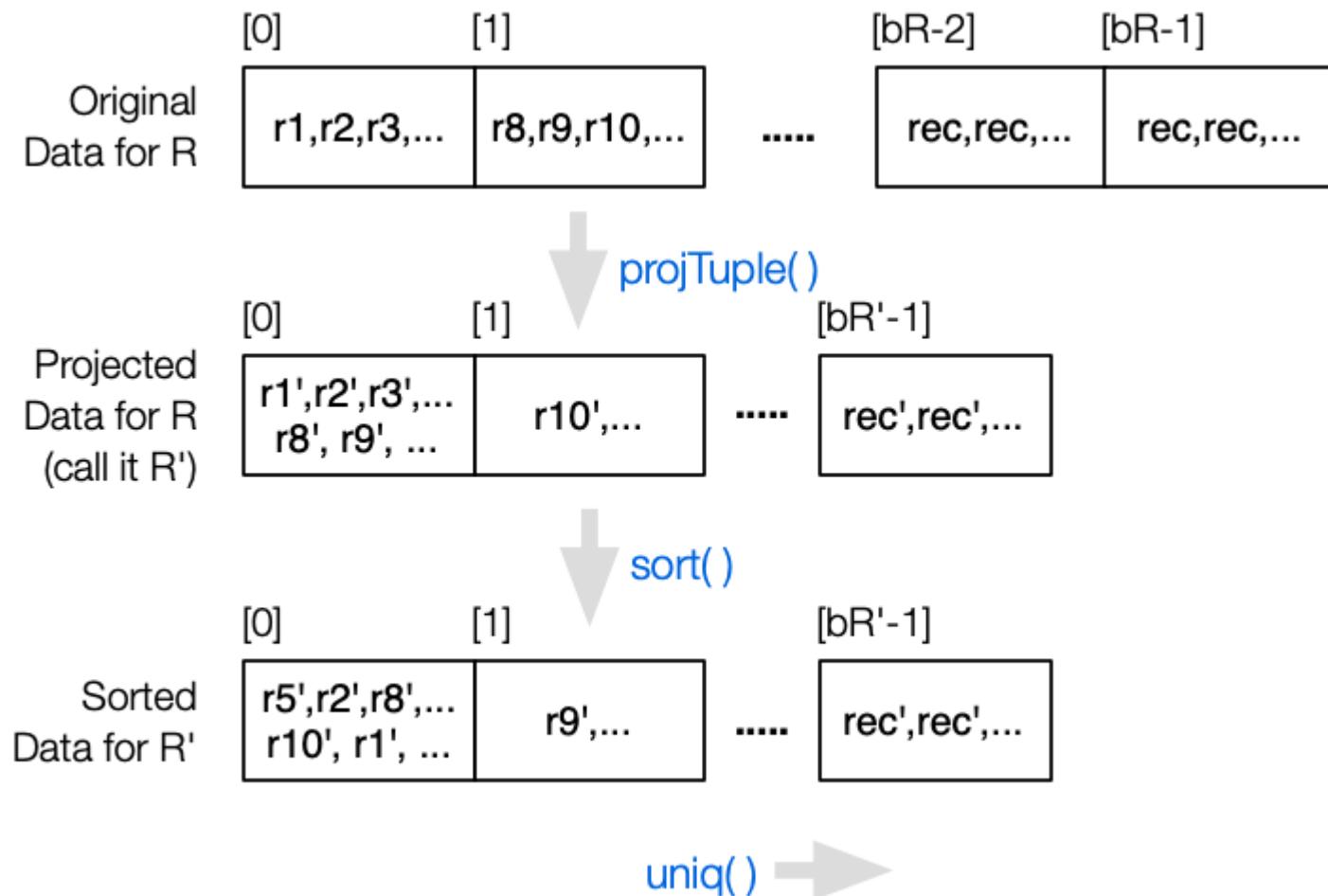
Typically, $b_{\text{OutFile}} < b_{\text{InFile}}$ (same number of tuples, but tuples are smaller)

❖ The Projection Operation (cont)

With **distinct**, the projection operation needs to:

1. scan the entire relation as input
 - already seen how to do scanning
2. create output tuples containing only requested attributes
 - implementation depends on tuple internal structure
 - essentially, make a new tuple with fewer attributes and where the values may be computed from existing attributes
3. eliminate any duplicates produced
 - two approaches: sorting or hashing

❖ Sort-based Projection



❖ Sort-based Projection (cont)

Requires a temporary file/relation .

```
for each tuple T in RelFile {  
    T' = projTuple([attr1,attr2,...],T)  
    add T' to TempFile  
}  
  
sort TempFile on [ attrs ]  
  
for each tuple T in TempFile {  
    if (T == Prev) continue  
    write T to Result  
    Prev = T  
}
```

Reminder: "**for each tuple**" means page-by-page, tuple-by-tuple

❖ Cost of Sort-based Projection

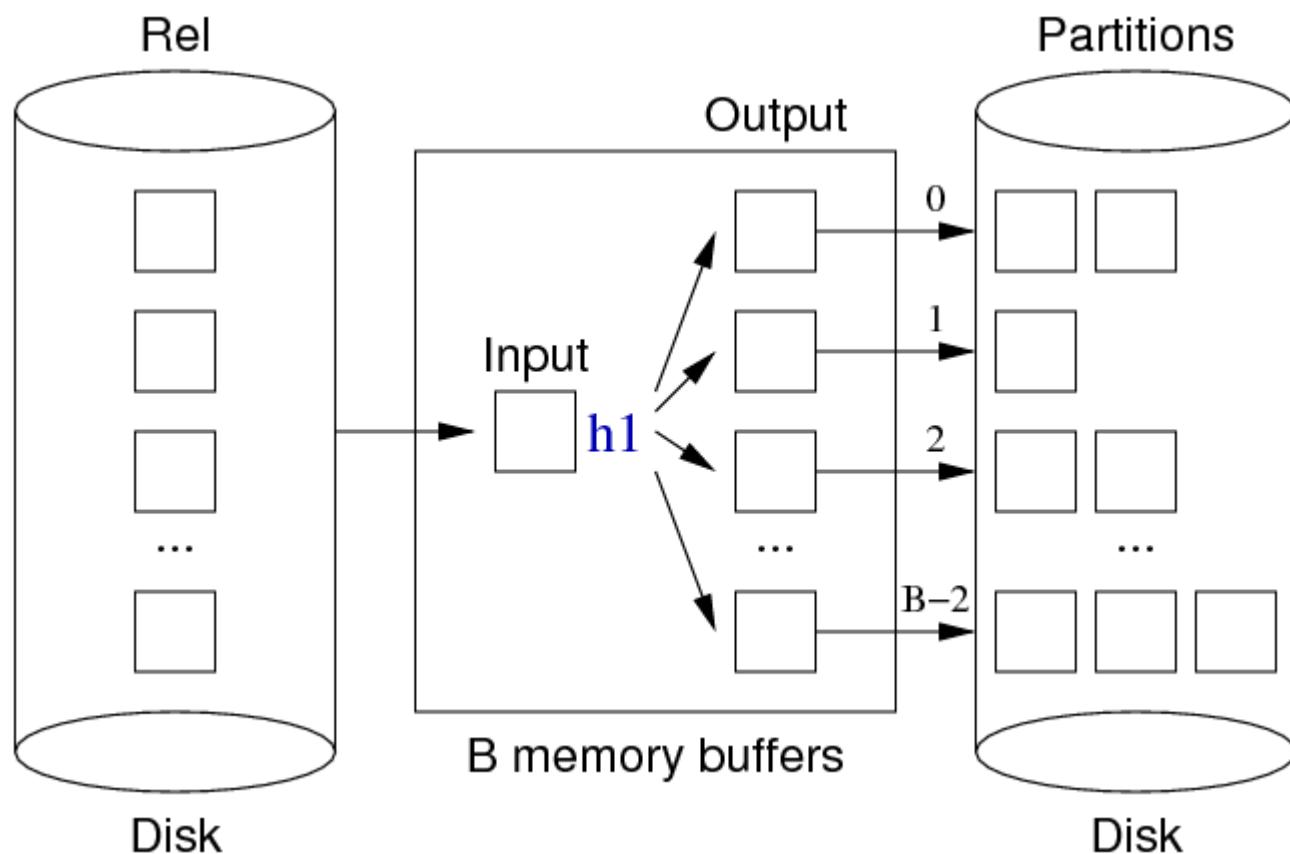
The costs involved are (assuming $B=n+1$ buffers for sort):

- scanning original relation **Rel**: b_R (with c_R)
- writing **Temp** relation: b_T (smaller tuples, $c_T > c_R$, sorted)
- sorting **Temp** relation:
 $2.b_T.\text{ceil}(\log_n b_0)$ where $b_0 = \text{ceil}(b_T/B)$
- scanning **Temp**, removing duplicates: b_T
- writing the result relation: b_{Out} (maybe less tuples)

Cost = sum of above = $b_R + b_T + 2.b_T.\text{ceil}(\log_n b_0) + b_T + b_{Out}$

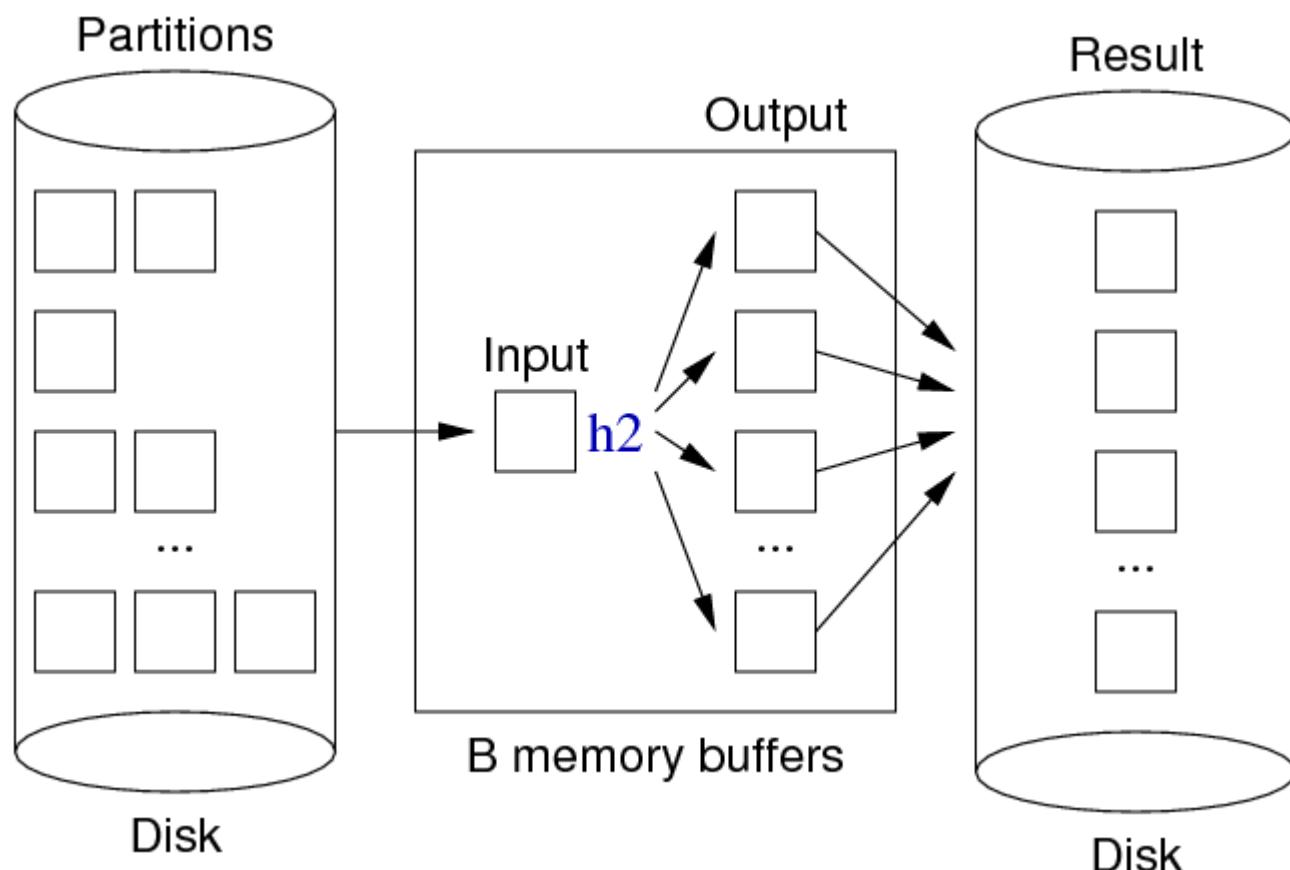
❖ Hash-based Projection

Partitioning phase:



❖ Hash-based Projection (cont)

Duplicate elimination phase:



❖ Hash-based Projection (cont)

Algorithm for both phases:

```
for each tuple T in relation Rel {  
    T' = mkTuple(attrs, T)  
    H = h1(T', n)  
    B = buffer for partition[H]  
    if (B full) write and clear B  
    insert T' into B  
}  
for each partition P in 0..n-1 {  
    for each tuple T in partition P {  
        H = h2(T, n)  
        B = buffer for hash value H  
        if (T not in B) insert T into B  
        // assumes B never gets full  
    }  
    write and clear all buffers  
}
```

Reminder: "**for each tuple**" means page-by-page, tuple-by-tuple

❖ Cost of Hash-based Projection

The total cost is the sum of the following:

- scanning original relation R : b_R
- writing partitions: b_P (b_R vs b_P ?)
- re-reading partitions: b_P
- writing the result relation: b_{Out}

$$\text{Cost} = b_R + 2b_P + b_{Out}$$

To ensure that n is larger than the largest partition ...

- use hash functions (h1,h2) with uniform spread
- allocate at least $\sqrt{b_R}+1$ buffers
- if insufficient buffers, significant re-reading overhead

❖ Projection on Primary Key

No duplicates, so simple approach from above works:

```
bR = nPages(Rel)
for i in 0 .. bR-1 {
    P = read page i
    for j in 0 .. nTuples(P) {
        T = getTuple(P, j)
        T' = projTuple([pk], T)
        if (outBuf is full) write and clear
        append T' to outBuf
    }
}
if (nTuples(outBuf) > 0) write
```

❖ Index-only Projection

Can do projection without accessing data file iff ...

- relation is indexed on (A_1, A_2, \dots, A_n) (indexes described later)
- projected attributes are a prefix of (A_1, A_2, \dots, A_n)

Basic idea:

- scan through index file (which is already sorted on attributes)
- duplicates are already adjacent in index, so easy to skip

Cost analysis ...

- index has b_i pages (where $b_i < b_R$)
- Cost = b_i reads + b_{Out} writes

❖ Comparison of Projection Methods

Difficult to compare, since they make different assumptions:

- index-only: needs an appropriate index
- hash-based: needs buffers and good hash functions
- sort-based: needs only buffers \Rightarrow use as default

Best case scenario for each (assuming $n+1$ in-memory buffers):

- index-only: $b_i + b_{Out} < b_R + b_{Out}$
- hash-based: $b_R + 2.b_P + b_{Out}$
- sort-based: $b_R + b_T + 2.b_T.\text{ceil}(\log_n b_0) + b_T + b_{Out}$

We normally omit b_{Out} ... each method produces the same result

❖ Projection in PostgreSQL

Code for projection forms part of execution iterators:

- include/nodes/execnodes.h
- backend/executor/execQual.c

Types:

- **ProjectionInfo** { **type**, **pi_state**, **pi_exprContext** }
- **ExprState** { **tag**, **flags**, **resnull**, **resvalue**, ... }

Functions:

- **ExecProject(projInfo, ...)** ... extracts projected data
- **check_sql_fn_retval(...)** ... evaluates attribute value

Selection Overview

- [Varieties of Selection](#)
- [Implementing Select Efficiently](#)

❖ Varieties of Selection

Selection: `select * from R where C`

- filters a subset of tuples from one relation **R**
- based on a condition **C** on the attribute values

We consider three distinct styles of selection:

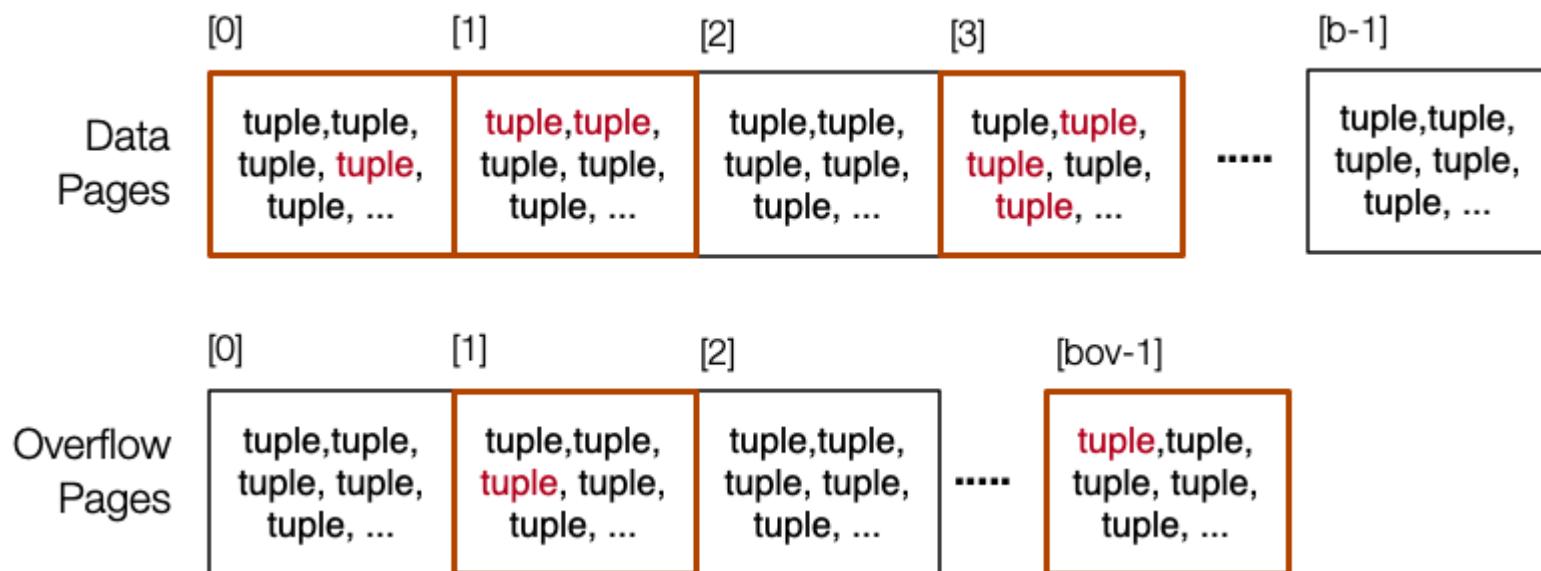
- 1-d (one dimensional) (condition uses only 1 attribute)
- n -d (multi-dimensional) (condition uses >1 attribute)
- similarity (approximate matching, with ranking)

Each style has several possible file-structures/techniques.

❖ Varieties of Selection (cont)

Selection returns a subset of tuples from a table

- r_q = number of tuples that match query q
- b_q = number of pages containing tuples that match query q



In the diagram, $r_q = 8$, $b_q = 5$

❖ Varieties of Selection (cont)

Different categories of selection queries:

one ... queries with at most 1 result ... $0 \leq r_q \leq 1, 0 \leq b_q \leq 1$

- typically, equality test on primary key attribute, e.g.
- **select * from R where id = 1234**

pmr ... partial match retrieval ... $0 \leq r_q \leq r, 0 \leq b_q \leq b+b_{ov}$

- conjunction of equality tests on multiple attributes, e.g.
- **select * from R where age=65** (1-d)
- **select * from R where age=65 and gender='m'** (n-d)

❖ Varieties of Selection (cont)

More categories of selection queries:

rng ... range queries ... $0 \leq r_q \leq r, 0 \leq b_q \leq b+b_{ov}$

- conjunction of inequalities, on one or more attributes, e.g.
- **select * from R where age≥18 and age≤21** (1-d)
- **select * from R where 18≤age≤21 and 160≤height≤190** (n-d)

pat ... pattern-based queries ... $0 \leq r_q \leq r, 0 \leq b_q \leq b+b_{ov}$

- string-based matching using **like** or regular expressions
- **select * from R where name like '%oo%**
- **select * from R where name ~ '^Smi'**

❖ Varieties of Selection (cont)

More categories of selection queries:

sim ... similarity matching ... in theory, $r_q = r$... everything matches to some degree

- uses "similarity" measure ($0 \leq sim \leq 1$, 0=different, 1=identical)
- **select * from Images where similar to SampleImage**
- results are ranked by sim value, from most to least similar
- can become a filter via
 - threshold ... only items where $sim \geq$ min similarity
 - top-k ... k items with highest similarities

We focus on **one**, **pmr** and **rng** queries, but will discuss others

❖ Implementing Select Efficiently

Two basic approaches:

- physical arrangement of tuples
 - sorting (search strategy)
 - hashing (static, dynamic, n -dimensional)
- additional indexing information
 - index files (primary, secondary, trees)
 - signatures (superimposed, disjoint)

Our analysis assumes 1 input buffer available for each relation.

If more buffers are available, most methods benefit.

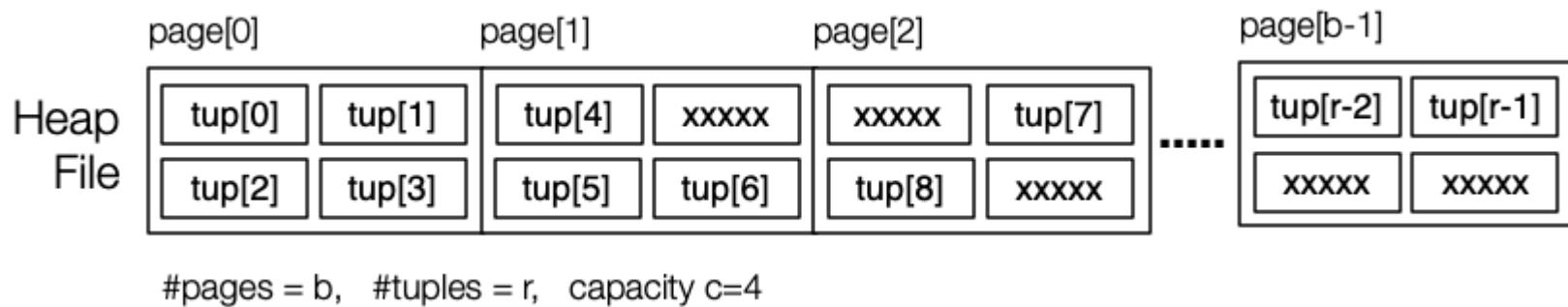
Heap Files

- [Heap Files](#)
- [Selection in Heaps](#)
- [Insertion in Heaps](#)
- [Deletion in Heaps](#)
- [Updates in Heaps](#)
- [Heaps in PostgreSQL](#)

❖ Heap Files

Heap files

- sequence of pages containing tuples
- no inherent ordering of tuples (added in next free slot)
- pages may contain free space from deleted tuples
- does not generally involve overflow pages



Note: this is **not** "heap" as in the top-to-bottom ordered tree.

❖ Selection in Heaps

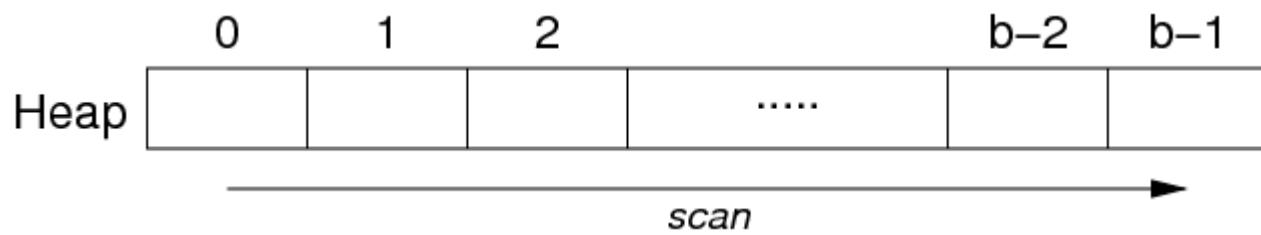
For all selection queries, the only possible strategy is:

```
// select * from R where C
rel = openRelation("R", READ);
for (p = 0; p < nPages(rel); p++) {
    get_page(rel, p, buf);
    for (i = 0; i < nTuples(buf); i++) {
        T = get_tuple(buf, i);
        if (T satisfies C)
            add tuple T to result set
    }
}
```

i.e. linear scan through file searching for matching tuples

❖ Selection in Heaps (cont)

The heap is scanned from the first to the last page:



$$Cost_{range} = Cost_{pmr} = b$$

If we know that only one tuple matches the query (one query), a simple optimisation is to stop the scan once that tuple is found.

$$Cost_{one} : \text{ Best} = 1 \quad \text{Average} = b/2 \quad \text{Worst} = b$$

❖ Insertion in Heaps

Insertion: new tuple is appended to file (in last page).

```
rel = openRelation("R", READ | WRITE);  
pid = nPages(rel)-1;  
get_page(rel, pid, buf);  
if (size(newTup) > size(buf))  
    { deal with oversize tuple }  
else {  
    if (!hasSpace(buf, newTup))  
        { pid++; nPages(rel)++; clear(buf); }  
    insert_record(buf, newTup);  
    put_page(rel, pid, buf);  
}
```

$$Cost_{insert} = 1_r + 1_w$$

❖ Insertion in Heaps (cont)

Alternative strategy:

- find any page from **R** with enough space
- preferably a page already loaded into memory buffer

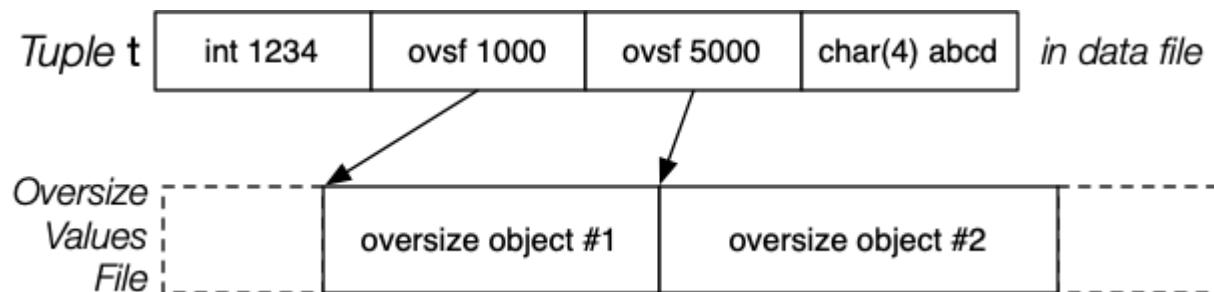
PostgreSQL's strategy:

- use last updated page of **R** in buffer pool
- otherwise, search buffer pool for page with enough space
- assisted by free space map (FSM) associated with each table
- for details: **backend/access/heap/ {heapam.c, hio.c}**

❖ Insertion in Heaps (cont)

Dealing with oversize tuple **t**:

```
for i in 1 .. nAttr(t) {
    if (t[i] not oversized) continue
    off = appendToFile.ovf, t[i])
    t[i] = (OVERSIZE, off)
}
insert into buf as before
```



❖ Insertion in Heaps (cont)

PostgreSQL's tuple insertion:

```
heap_insert(Relation relation,          // relation desc
            HeapTuple newtuple,        // new tuple data
            CommandId cid, ...)     // SQL statement
```

- finds page which has enough free space for **newtuple**
- ensures page loaded into buffer pool and locked
- copies tuple data into page buffer, sets **xmin**, etc.
- marks buffer as dirty
- writes details of insertion into transaction log
- returns OID of new tuple if relation has OIDs

❖ Deletion in Heaps

SQL: **delete from R where Condition**

Implementation of deletion:

```
rel = openRelation("R",READ|WRITE);
for (p = 0; p < nPages(rel); p++) {
    get_page(rel, p, buf);
    ndels = 0;
    for (i = 0; i < nTuples(buf); i++) {
        tup = get_tuple(buf,i);
        if (tup satisfies Condition)
            { ndels++; delete_record(buf,i); }
    }
    if (ndels > 0) put_page(rel, p, buf);
    if (ndels > 0 && unique) break;
}
```

❖ Deletion in Heaps (cont)

PostgreSQL tuple deletion:

```
heap_delete(Relation relation,      // relation desc
             ItemPointer tid, ..., // tupleID
             CommandId cid, ...) // SQL statement
```

- gets page containing tuple **tid** into buffer pool and locks it
- sets flags, commandID and **xmax** in tuple; dirties buffer
- writes indication of deletion to transaction log

Vacuuming eventually compacts space in each page.

❖ Updates in Heaps

SQL: **update R set F = val where Condition**

Analysis for updates is similar to that for deletion

- scan all pages
- replace any updated tuples (within each page)
- write affected pages to disk

$$Cost_{update} = b_r + b_{qw}$$

Complication: new tuple larger than old version (too big for page)

Solution: delete, re-organise free space, then insert

❖ Updates in Heaps (cont)

PostgreSQL tuple update:

```
heap_update(Relation relation,          // relation desc
             ItemPointer otid,        // old tupleID
             HeapTuple newtuple, ..., // new tuple data
             CommandId cid, ...)    // SQL statement
```

- essentially does **delete(otid)**, then **insert(newtuple)**
- also, sets old tuple's **ctid** field to reference new tuple
- can also update-in-place if no referencing transactions

❖ Heaps in PostgreSQL

PostgreSQL stores all table data in heap files (by default).

Typically there are also associated index files.

If a file is more useful in some other form:

- PostgreSQL may make a transformed copy during query execution
- programmer can set it via **create index...using hash**

Heap file implementation: [src/backend/access/heap](#)

❖ Heaps in PostgreSQL (cont)

PostgreSQL "heap file" may use multiple physical files

- files are named after the OID of the corresponding table
- first data file is called simply **OID**
- if size exceeds 1GB, create a **fork** called **OID.1**
- add more forks as data size grows (one fork for each 1GB)
- other files:
 - free space map (**OID_fsm**), visibility map (**OID_vm**)
 - optionally, TOAST file (if table has large varlen attributes)
- for details: Chapter 72 in PostgreSQL v15 documentation

Sorted Files

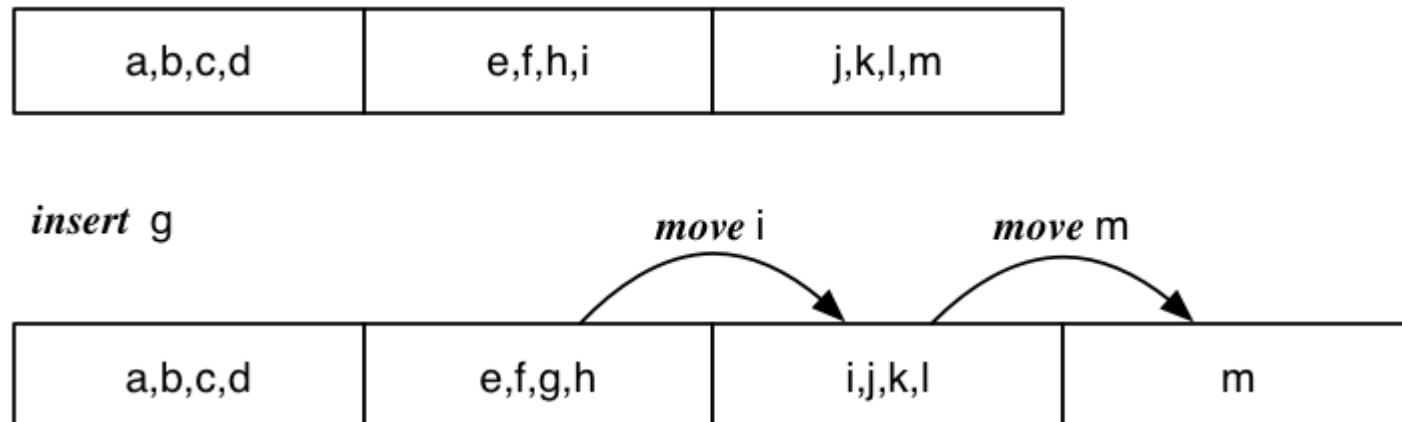
- [Sorted Files](#)
- [Selection in Sorted Files](#)
- [Insertion into Sorted Files](#)
- [Deletion from Sorted Files](#)

❖ Sorted Files

Records stored in file in order of some field **k** (the sort key).

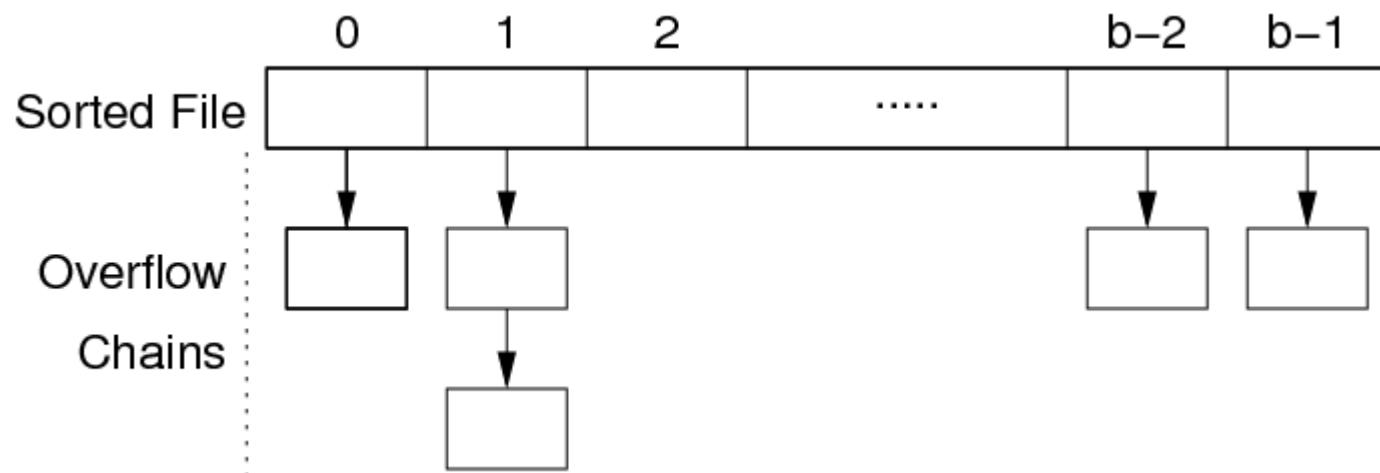
Makes searching more efficient; makes insertion less efficient

E.g. assume $c = 4$



❖ Sorted Files (cont)

In order to mitigate insertion costs, use overflow pages.



Total number of overflow pages = b_{ov} .

Average overflow chain length = $Ov = b_{ov} / b$.

Bucket = data page + its overflow page(s)

❖ Selection in Sorted Files

For one queries on sort key, use binary search.

```
// select * from R where k = val  (sorted on R.k)
lo = 0; hi = nPages(rel)-1
while (lo <= hi) {
    mid = (lo+hi) / 2; // int division with truncation
    (tup,loVal,hiVal) = searchBucket(rel,mid,x,val);
    if (tup != NULL) return tup;
    else if (val < loVal) hi = mid - 1;
    else if (val > hiVal) lo = mid + 1;
    else return NOT_FOUND;
}
return NOT_FOUND;
```

where **rel** is relation handle, **mid,lo,hi** are page indexes,
k is a field/attr, **val,loVal,hiVal** are values for **k**

❖ Selection in Sorted Files (cont)

Search a page and its overflow chain for a key value

```
searchBucket(rel,p,k,val)
{
    get_page(rel,p,buf);
    (tup,min,max) = searchPage(buf,k,val,+INF,-INF)
    if (tup != NULL) return(tup,min,max);
    ovf = openOvFile(f);
    ovp = overflow(buf);
    while (tup == NULL && ovp != NO_PAGE) {
        get_page(ovf,ovp,buf);
        (tup,min,max) = searchPage(buf,k,val,min,max)
        ovp = overflow(buf);
    }
    return (tup,min,max);
}
```

Assumes each page contains index of next page in Ov chain

❖ Selection in Sorted Files (cont)

Search within a page for key; also find min/max key values

```
searchPage(buf, k, val, min, max)
{
    res = NULL;
    for (i = 0; i < nTuples(buf); i++) {
        tup = get_tuple(buf, i);
        if (tup.k == val) res = tup;
        if (tup.k < min) min = tup.k;
        if (tup.k > max) max = tup.k;
    }
    return (res,min,max);
}
```

❖ Selection in Sorted Files (cont)

The above method treats each bucket like a single large page.

Cases:

- best: find tuple in first data page we read
- worst: full binary search, and not found
 - examine $\log_2 b$ data pages
 - plus examine all of their overflow pages
- average: examine some data pages + their overflow pages

$Cost_{one}$: Best = 1 Worst = $\log_2 b + b_{ov}$

Average case cost analysis needs assumptions (e.g. data distribution)

❖ Selection in Sorted Files (cont)

For pmr query, on non-unique attribute k , where file is sorted on k

- tuples containing k may span several pages

E.g. **select * from R where k = 2**

[0]	[1]	[2]	[3]	[4]
1,1,2,2	2,2,2,2	2,3,3,4	4,5,6,7	7,8,9,9

+-----|
binary search lands here

Begin by locating a page p containing $\mathbf{k=val}$ (as for one query).

Scan backwards and forwards from p to find matches.

Thus, $Cost_{pmr} = Cost_{one} + (b_q - 1) \cdot (1 + Ov)$

❖ Selection in Sorted Files (cont)

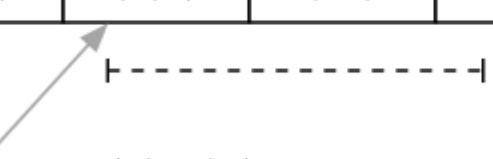
For range queries on unique sort key (e.g. primary key):

- use binary search to find lower bound
- read sequentially until reach upper bound

E.g. **select * from R where k >= 5 and k <= 13**

[0]	[1]	[2]	[3]	[4]
1,2,3,4	5,7,8,9	10,11,12	13,14,15	16,18,19

binary search lands here



$$Cost_{range} = Cost_{one} + (b_q - 1) \cdot (1 + O_v)$$

❖ Selection in Sorted Files (cont)

For range queries on non-unique sort key, similar method to *pmr*.

- binary search to find lower bound
- then go backwards to start of run
- then go forwards to last occurrence of upper-bound

E.g. **select * from R where k >= 2 and k <= 6**

[0]	[1]	[2]	[3]	[4]
1,1,2,2	2,2,2,2	2,3,3,4	4,5,6,7	7,8,9,9

+-----+
binary search lands here

$$Cost_{range} = Cost_{one} + (b_q - 1) \cdot (1 + O_v)$$

❖ Selection in Sorted Files (cont)

So far, have assumed query condition involves sort key k .

But what about `select * from R where j = 100.0 ?`

If condition contains attribute j , not the sort key

- file is unlikely to be sorted by j as well
- sortedness gives no searching benefits

Cost_{one} , $\text{Cost}_{\text{range}}$, Cost_{pmr} as for heap files

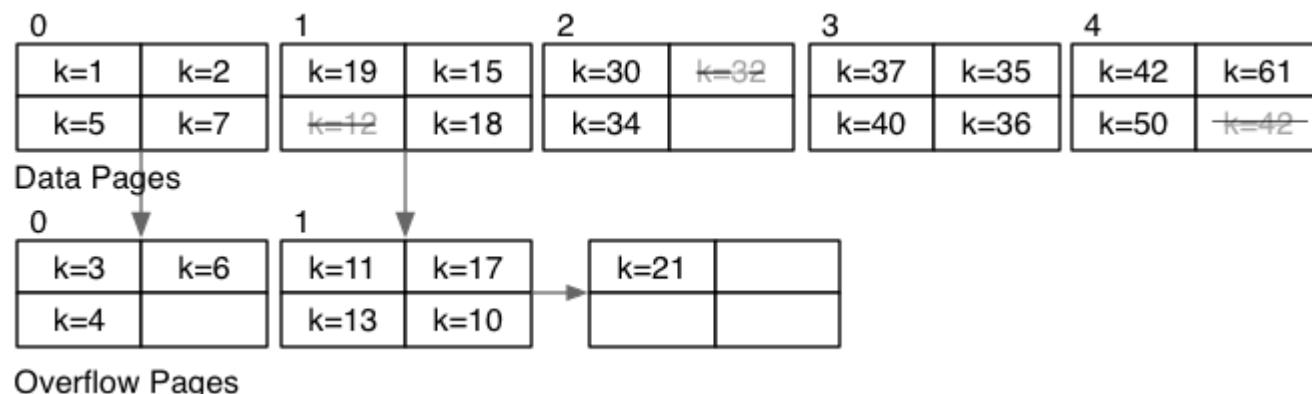
❖ Insertion into Sorted Files

Insertion approach:

- find appropriate page for tuple (via binary search)
 - if page not full, insert into page
 - otherwise, insert into next overflow page with space

Thus, $\text{Cost}_{\text{insert}} = \text{Cost}_{\text{one}} + \delta_w$ (where $\delta_w = 1$ or 2)

Consider insertions of $k=33$, $k=25$, $k=99$ into:



❖ Deletion from Sorted Files

E.g. `delete from R where k = 2`

Deletion strategy:

- find matching tuple(s)
- mark them as deleted

Cost depends on **selectivity** of selection condition

Recall: selectivity determines b_q (# pages with matches)

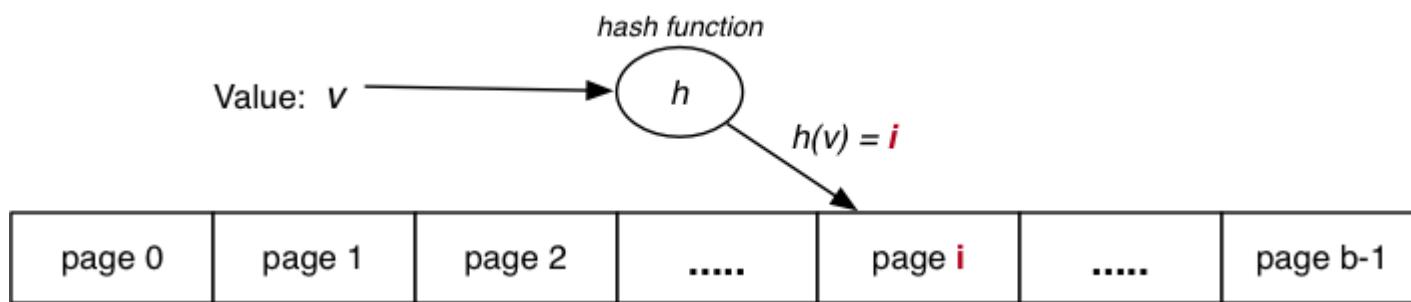
Thus, $\text{Cost}_{\text{delete}} = \text{Cost}_{\text{select}} + b_{qw}$

Hashed Files

- [Hashing](#)
- [Hashing Performance](#)
- [Selection with Hashing](#)
- [Insertion with Hashing](#)
- [Deletion with Hashing](#)
- [Problem with Hashing...](#)
- [Flexible Hashing](#)

❖ Hashing

Basic idea: use key value to compute page address of tuple.



e.g. tuple with key = v is stored in page i

Requires: hash function $h(v)$ that maps $\text{KeyDomain} \rightarrow [0..b-1]$.

- hashing converts key value (any type) into integer value
- integer value is then mapped to page index
- note: can view integer value as a bit-string

❖ Hashing (cont)

PostgreSQL hash function (simplified):

```
Datum hash_any(unsigned char *k, int keylen)
{
    uint32 a, b, c, len, *ka = (uint32 *)k;
    /* Set up the internal state */
    len = keylen;
    a = b = c = 0x9e3779b9+len+3923095;
    /* handle most of the key */
    while (len >= 12) {
        a += ka[0]; b += ka[1]; c += ka[2];
        mix(a, b, c);
        ka += 3; len -= 12;
    }
    ... collect data from remaining bytes into a,b,c ...
    mix(a, b, c);
    return UInt32GetDatum(c);
}
```

See **backend/access/hash/hashfunc.c** for details (incl **mix()**)

❖ Hashing (cont)

`hash_any()` gives hash value as 32-bit quantity (`uint32`).

Two ways to map raw hash value into a page address:

- if $b = 2^k$, bitwise AND with k low-order bits set to one

```
uint32 hashToPageNum(uint32 hval) {  
    uint32 mask = 0xFFFFFFFF;  
    return (hval & (mask >> (32-k)));  
}
```

- otherwise, use mod to produce value in range $0..b-1$

```
uint32 hashToPageNum(uint32 hval) {  
    return (hval % b);  
}
```

❖ Hashing Performance

Aims:

- distribute tuples evenly amongst buckets
- have most buckets nearly full (attempt to minimise wasted space)

Note: if data distribution not uniform, address distribution can't be uniform.

Best case: every bucket contains same number of tuples.

Worst case: every tuple hashes to same bucket.

Average case: some buckets have more tuples than others.

Use overflow pages to handle "overfull" buckets (cf. sorted files)

All tuples in each bucket must have same hash value.

❖ Hashing Performance (cont)

Two important measures for hash files:

- load factor: $L = r / bc$
- average overflow chain length: $Ov = b_{ov} / b$

Three cases for distribution of tuples in a hashed file:

Case	L	Ov
Best	≈ 1	0
Worst	$>> 1$	**
Average	< 1	$0 < Ov < 1$

(** performance is same as Heap File)

To achieve average case, aim for $0.75 \leq L \leq 0.9$.

❖ Selection with Hashing

Select via hashing on unique key k (one)

```
// select * from R where k = val
getPageViaHash(R, val, P)
for each tuple t in page P {
    if (t.k == val) return t
}
for each overflow page Q of P {
    for each tuple t in page Q {
        if (t.k == val) return t
    }
}
```

$Cost_{one}$: Best = 1, Avg = $1+Ov/2$ Worst = $1+\max(OvLen)$

❖ Selection with Hashing (cont)

Working out which page, given a key ...

```
getPageViaHash(Reln R, Value key, Page p)
{
    uint32 h = hash_any(key, len(key));
    PageID pid = h % nPages(R);
    get_page(R, pid, buf);
}
```

❖ Selection with Hashing (cont)

Select via hashing on non-unique hash key nk (pmr)

```
// select * from R where nk = val
getPageViaHash(R, val, P)
for each tuple t in page P {
    if (t.nk == val) add t to results
}
for each overflow page Q of P {
    for each tuple t in page Q {
        if (t.nk == val) add t to results
    }
}
return results
```

$$Cost_{pmr} = 1 + Ov$$

If Ov is small (e.g. 0 or 1), very good retrieval cost

❖ Selection with Hashing (cont)

Hashing does not help with *range* queries? ** ... (integer values vs float values)

$$Cost_{range} = b + b_{ov}$$

Selection on attribute j which is not hash key ...

$$Cost_{one}, \ Cost_{range}, \ Cost_{pmr} = b + b_{ov}$$

** what if the hash function is order-preserving (and most aren't)

❖ Insertion with Hashing

Insertion uses similar process to *one* queries.

```
// insert tuple t with key=val into rel R
getPageViaHash(R, val, P)
if room in page P {
    insert t into P; return
}
for each overflow page Q of P {
    if room in page Q {
        insert t into Q; return
    }
}
add new overflow page Q
link Q to previous page
insert t into Q
```

$\text{Cost}_{\text{insert}}$: Best: $1_r + 1_w$ Worst: $1 + \max(\text{OvLen})_r + 2_w$

❖ Deletion with Hashing

Similar performance to select on non-unique key:

```
// delete from R where k = val
// f = data file ... ovf = overflow file
getPageViaHash(R, val, P)
ndel = delTuples(P,k,val)
if (ndel > 0) put_page(dataFile(R),P.pid,P)
for each overflow page Q of P {
    ndel = delTuples(Q,k,val)
    if (ndel > 0) put_page(ovFile(R),Q.pid,Q)
}
```

Extra cost over select is cost of writing back modified pages.

Method works for both unique and non-unique hash keys.

❖ Problem with Hashing...

So far, discussion of hashing has assumed a fixed file size (b).

What size file to use?

- the size we need right now (performance degrades as file overflows)
- the maximum size we might ever need (significant waste of space)

Problem: change file size \Rightarrow change hash function \Rightarrow rebuild file

Methods for hashing with files whose size changes:

- extendible hashing, dynamic hashing (need a directory, no overflows)
- **linear hashing** (expands file "systematically", no directory, has overflows)

❖ Flexible Hashing

All flexible hashing methods ...

- treat hash as 32-bit bit-string
- adjust hashing by using more/less bits

Start with hash function to convert value to bit-string:

```
uint32 hash(unsigned char *val)
```

Require a function to extract d bits from bit-string:

```
unit32 bits(int d, uint32 val)
```

Use result of **bits()** as page address.

❖ Flexible Hashing (cont)

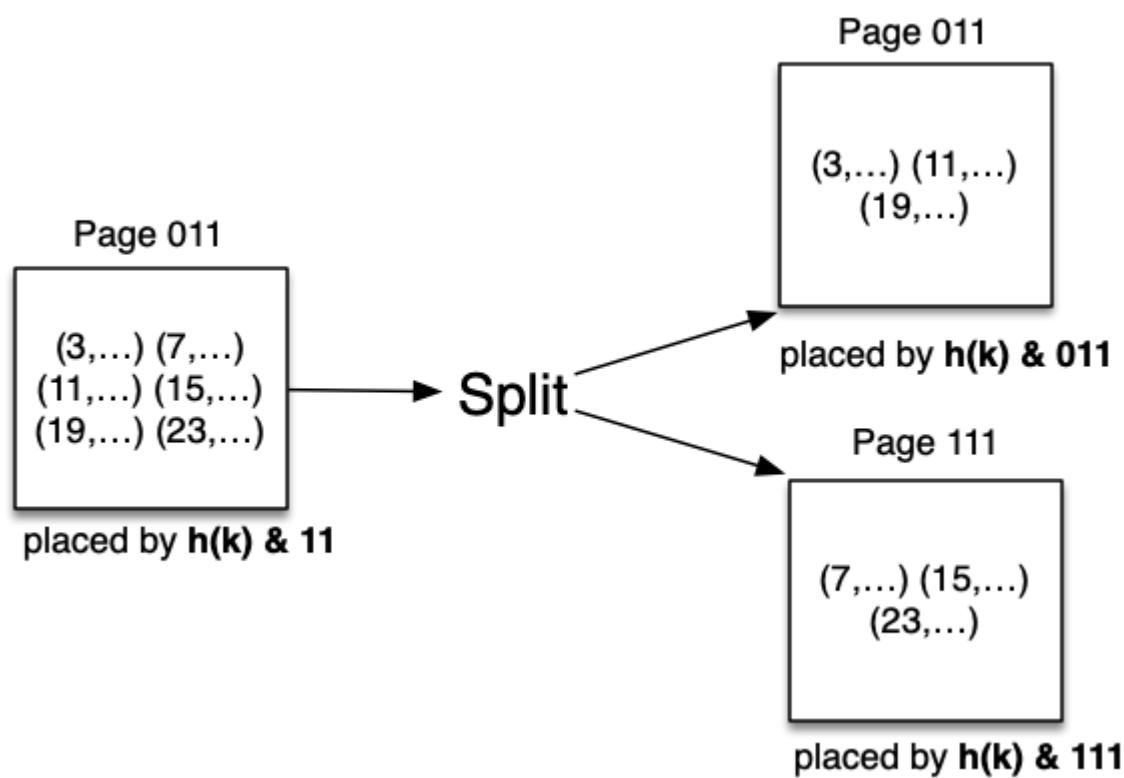
Important concept for flexible hashing: **splitting**

- consider one page (all tuples have same hash value)
- recompute page numbers by considering one extra bit
- if current page is **101**, new pages have hashes **0101** and **1101**
- some tuples stay in page **0101** (was **101**)
- some tuples move to page **1101** (new page)
- also, rehash any tuples in overflow pages of page **101**

Result: expandable data file, never requiring a complete file rebuild

❖ Flexible Hashing (cont)

Example of splitting:



Tuples only show key value; assume $h(val) = val$

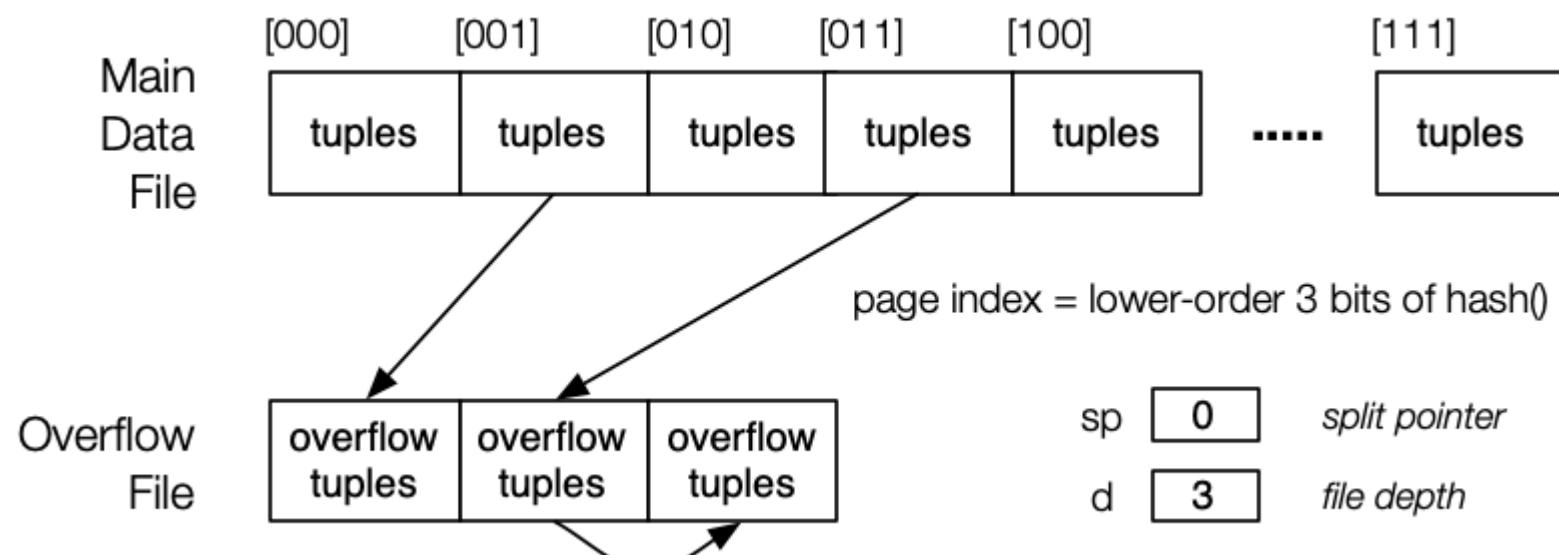
Linear Hashing

- [Linear Hashing](#)
- [Selection with Lin.Hashing](#)
- [File Expansion with Lin.Hashing](#)
- [Insertion with Lin.Hashing](#)
- [Splitting](#)
- [Insertion Cost](#)
- [Deletion with Lin.Hashing](#)

◆ Linear Hashing

File organisation:

- file of primary data pages
- file of overflow data pages
- registers called *split pointer* (sp) and *depth* (d)



❖ Linear Hashing (cont)

Linear Hashing uses a systematic method of growing data file ...

- hash function "adapts" to changing address range (via sp and d)
- systematic splitting controls length of overflow chains

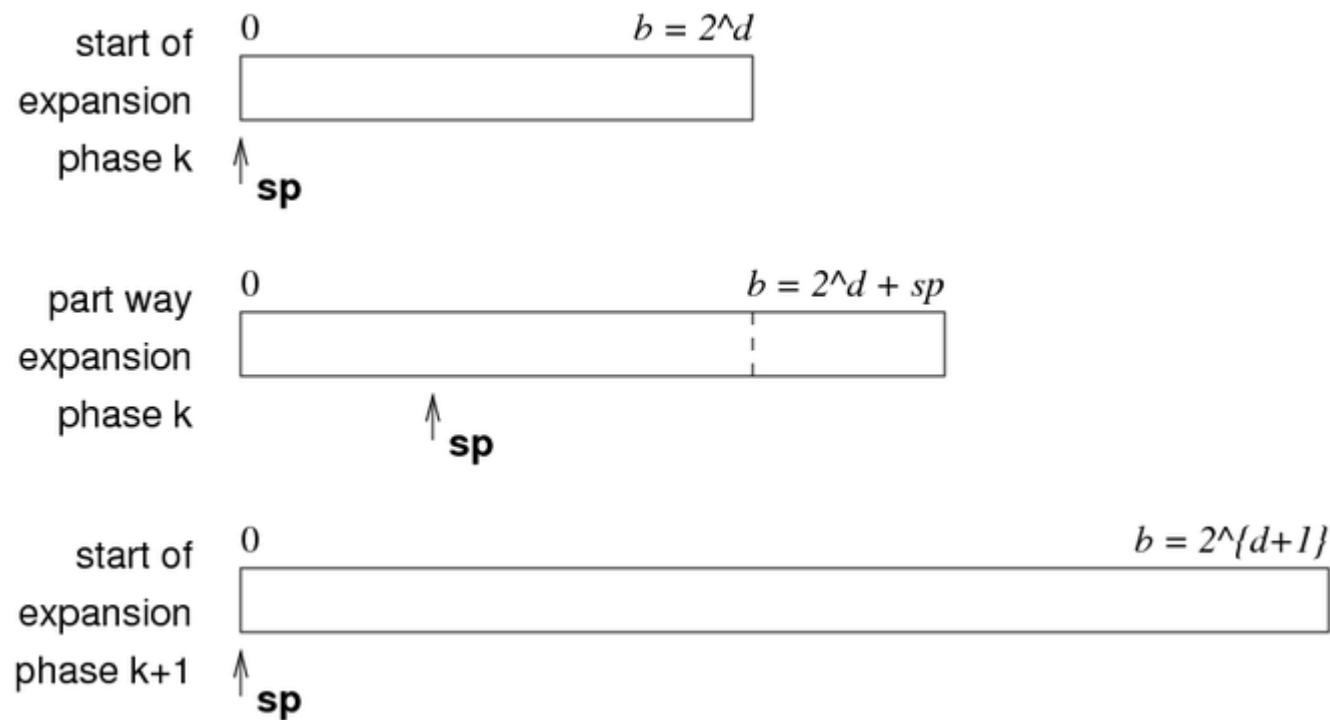
Advantage: does **not** require auxiliary storage for a directory

Disadvantage: requires overflow pages (don't split on full pages)

❖ Linear Hashing (cont)

File grows linearly (one page at a time, at regular intervals).

Has "phases" of expansion; over each phase, b doubles.



❖ Selection with Lin.Hashing

If $b=2^d$, the file behaves exactly like standard hashing.

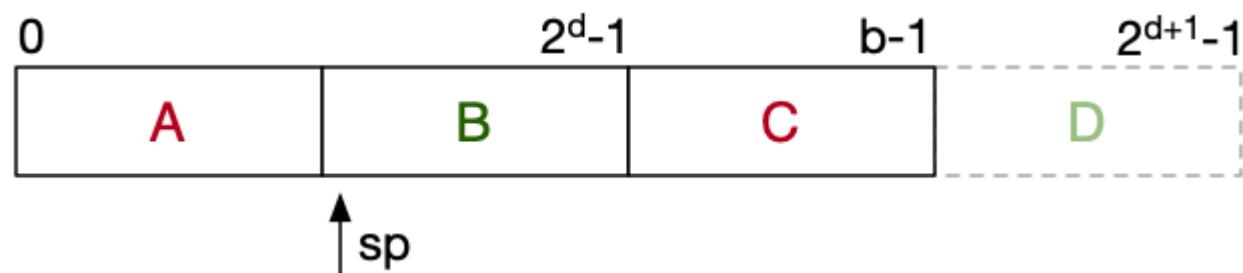
Use d bits of hash to compute page address.

```
// select * from R where k = val
h = hash(val);
P = bits(d,h); // lower-order bits
for each tuple t in page P
    and its overflow pages {
        if (t.k == val) return t;
    }
```

Average Cost_{one} = 1+Ov

❖ Selection with Lin.Hashing (cont)

If $b \neq 2^d$, treat different parts of the file differently.



Parts A and C are treated as if part of a file of size 2^{d+1} .

Part B is treated as if part of a file of size 2^d .

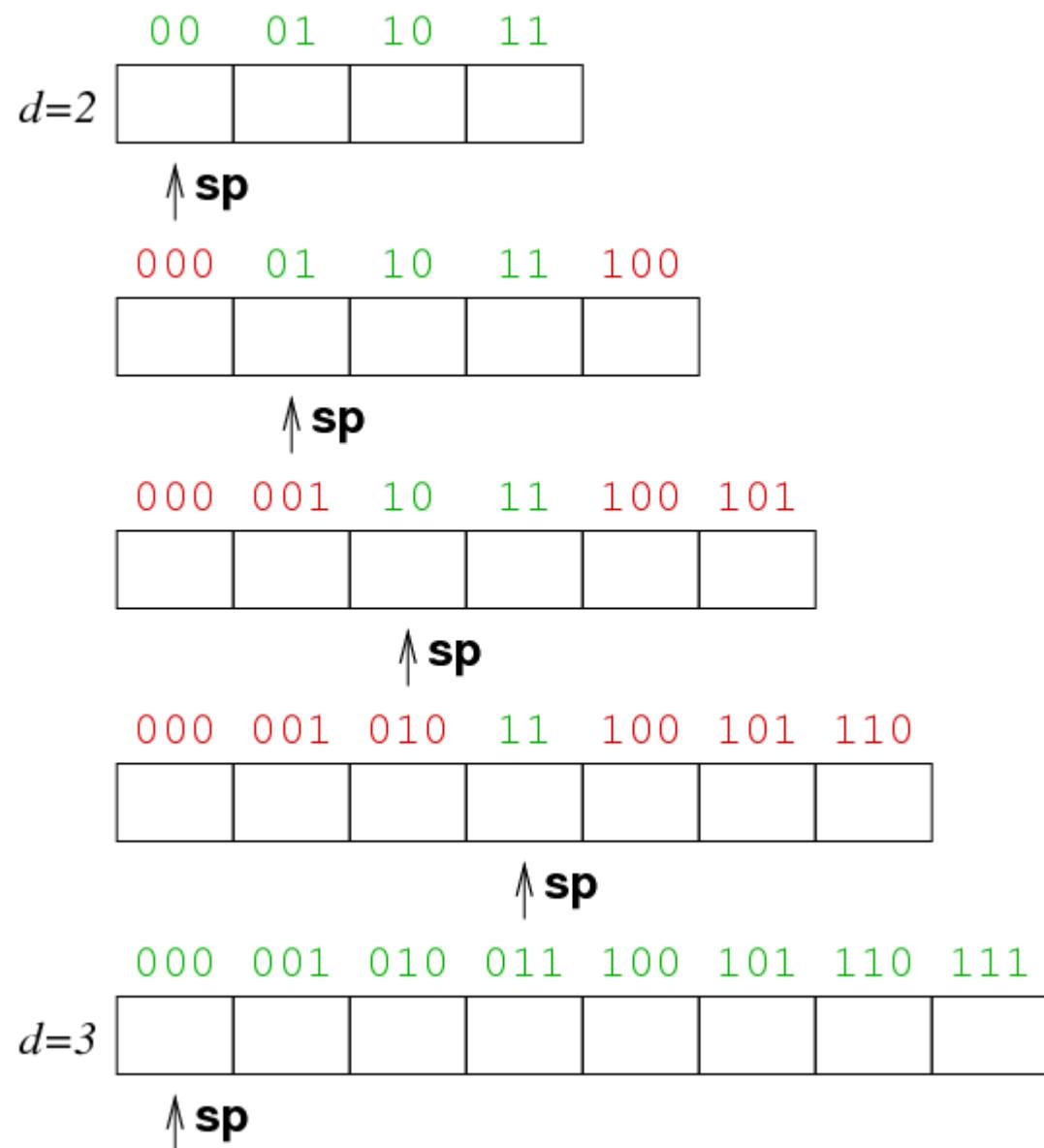
Part D does not yet exist (tuples in B may eventually move into it).

❖ Selection with Lin.Hashing (cont)

Modified search algorithm:

```
// select * from R where k = val
h = hash(val);
pid = bits(d,h);
if (pid < sp) { pid = bits(d+1,h); }
P = getPage(f, pid)
for each tuple t in page P
    and its overflow pages {
        if (t.k == val) return R;
    }
```

❖ File Expansion with Lin.Hashing



❖ Insertion with Lin.Hashing

Abstract view:

```
pid = bits(d, hash(val));
if (pid < sp) pid = bits(d+1,hash(val));
// bucket P = page P + its overflow pages
P = getPage(f,pid)
for each page Q in bucket P {
    if (space in Q) {
        insert tuple into Q
        break
    }
}
if (no insertion) {
    add new ovflow page to bucket P
    insert tuple into new page
}
if (need to split) {
    partition tuples from bucket sp
        into buckets sp and sp+2^d
    sp++;
    if (sp == 2^d) { d++; sp = 0; }
}
```

❖ Splitting

How to decide that we "need to split"?

Two approaches to triggering a split:

- split every time a tuple is inserted into full page
- split when load factor reaches threshold (every k inserts)

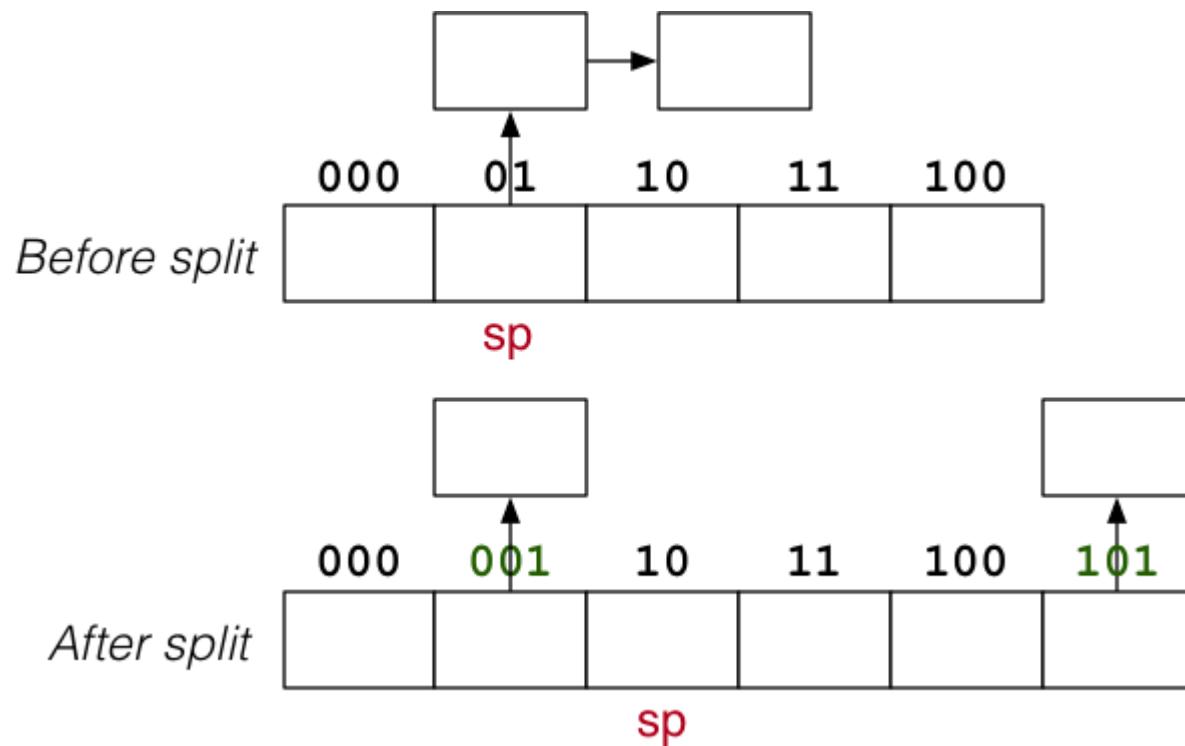
Note: always split page sp , even if not full or "current"

Systematic splitting like this ...

- eventually reduces length of every overflow chain
- helps to maintain short average overflow chain length

❖ Splitting (cont)

Splitting process for page $sp=01$:



❖ Splitting (cont)

Splitting algorithm:

```
// partition tuples between two buckets
newp = sp + 2^d; oldp = sp;
for all tuples t in P[oldp] and its overflows {
    p = bits(d+1,hash(t.k));
    if (p == newp)
        add tuple t to bucket[newp]
    else
        add tuple t to bucket[oldp]
}
sp++;
if (sp == 2^d) { d++; sp = 0; }
```

❖ Insertion Cost

If no split required, cost same as for standard hashing:

$\text{Cost}_{\text{insert}}$: Best: $1_r + 1_w$, Avg: $(1+0v)_r + 1_w$, Worst: $(1+\max(0v))_r + 2_w$

If split occurs, incur $\text{Cost}_{\text{insert}}$ plus cost of splitting:

- read page sp (plus all of its overflow pages)
- write page sp (and its new overflow pages)
- write page $sp+2^d$ (and its new overflow pages)

On average, $\text{Cost}_{\text{split}} = (1+0v)_r + (2+0v)_w$

❖ Deletion with Lin.Hashing

Deletion is similar to ordinary static hash file.

But might wish to contract file when enough tuples removed.

Rationale: r shrinks, b stays large \Rightarrow wasted space.

Method:

- remove last bucket in data file (contracts linearly).
- merge tuples from bucket with its buddy page (using $d-1$ hash bits)

Hashing in PostgreSQL

- [Hashing in PostgreSQL](#)
- [PostgreSQL Hash Function](#)
- [Hash Files in PostgreSQL](#)

❖ Hashing in PostgreSQL

PostgreSQL uses linear hashing on tables which have been:

```
create index Ix on R using hash (k);
```

Hash file implementation: **backend/access/hash**

- **hashfunc.c** ... a family of hash functions
- **hashinsert.c** ... insert, with overflows
- **hashpage.c** ... utilities + splitting
- **hashsearch.c** ... iterator for hash files

Detailed info in **src/backend/access/hash/README**

Based on "A New Hashing Package for Unix", Margo Seltzer, Winter Usenix 1991

❖ PostgreSQL Hash Function

PostgreSQL generic hash function (simplified):

```
Datum hash_any(unsigned char *k, int keylen)
{
    uint32 a, b, c, len, *ka = (uint32 *)k;
    /* Set up the internal state */
    len = keylen;
    a = b = c = 0x9e3779b9+len+3923095;
    /* handle most of the key */
    while (len >= 12) {
        a += ka[0]; b += ka[1]; c += ka[2];
        mix(a, b, c);
        ka += 3; len -= 12;
    }
    ... collect data from remaining bytes into a,b,c ...
    mix(a, b, c);
    return UInt32GetDatum(c);
}
```

See **backend/access/hash/hashfunc.c** for details (incl **mix()**)

❖ PostgreSQL Hash Function (cont)

hash_any() gives hash value as 32-bit quantity (**uint32**).

Typically invoked from a type-specific function, e.g.

```
Datum  
hashint4(PG_FUNCTION_ARGS)  
{  
    return hash_uint32(PG_GETARG_INT32(0));  
}
```

where **hash_uint32()** is a faster version of **hash_any()**

Hash value is "wrapped" as a **Datum**

❖ PostgreSQL Hash Function (cont)

Implementation of hash → page ID

```
Bucket
_hash_hashkey2bucket(uint32 hashkey, uint32 maxbucket,
                      uint32 highmask, uint32 lowmask)
{
    Bucket      bucket;

    bucket = hashkey & highmask;
    if (bucket > maxbucket)
        bucket = bucket & lowmask;

    return bucket;
}
```

❖ Hash Files in PostgreSQL

PostgreSQL uses different file organisation ...

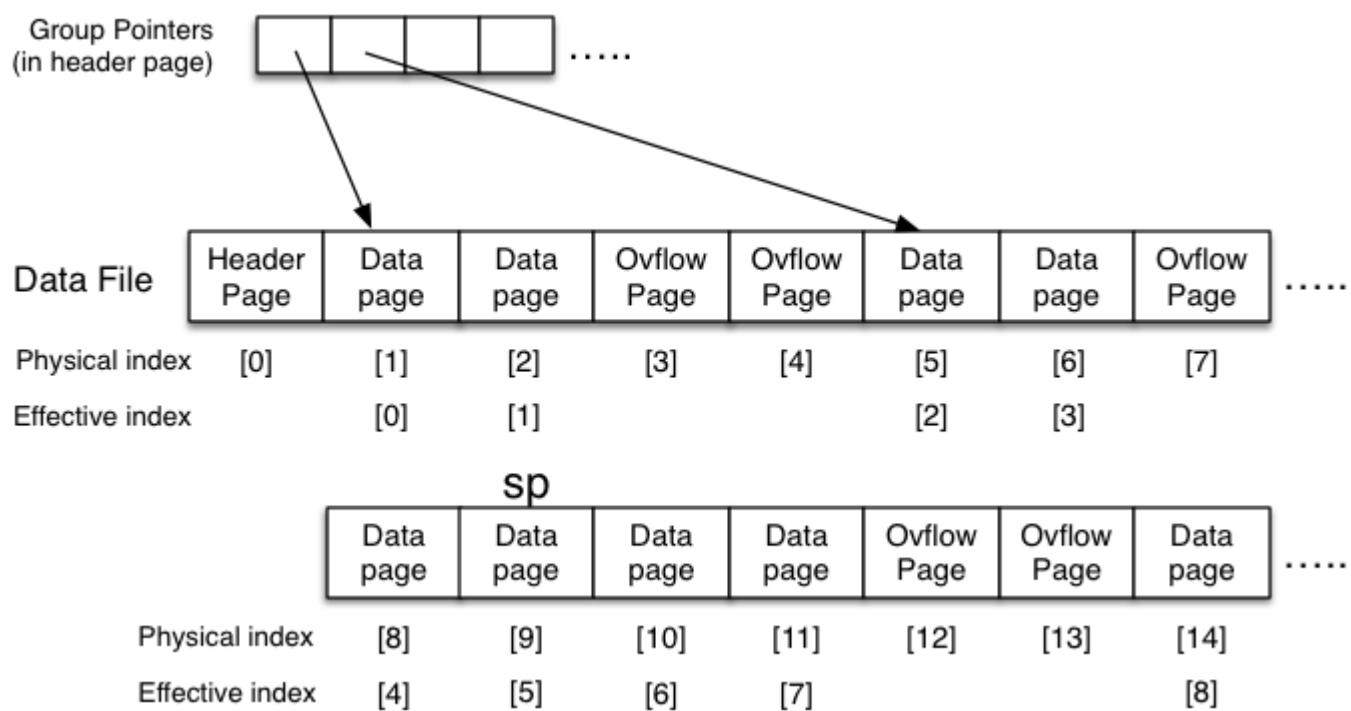
- has a single file containing header, main and overflow pages
- has groups of main pages of size 2^n
- in between groups, arbitrary number of overflow pages
- maintains collection of group pointers in header page
- each group pointer indicates start of main page group

If overflow pages become empty, add to free list and re-use.

Confusingly, PostgreSQL calls "group pointers" as "split pointers"

❖ Hash Files in PostgreSQL (cont)

PostgreSQL hash file structure:



❖ Hash Files in PostgreSQL (cont)

Approximate method for converting bucket # to page address:

```
// which page is primary page of bucket
uint bucket_to_page(headerp, B) {
    uint *splits = headerp->hashm_spares;
    uint chunk, base, offset, lg2(uint);
    chunk = (B<2) ? 0 : lg2(B+1)-1;
    base = splits[chunk];
    offset = (B<2) ? B : B-(1<<chunk);
    return (base + offset);
}
// returns ceil(log_2(n))
int lg2(uint n) {
    int i, v;
    for (i = 0, v = 1; v < n; v <<= 1) i++;
    return i;
}
```


Week 4 Exercises

- [Exercise 1: File Merging](#)
- [Exercise 2: Sort-based Projection](#)
- [Exercise 3: Hash-based Projection](#)
- [Exercise 4: Query Types](#)
- [Exercise 5: Cost of Deletion in Heaps](#)
- [Exercise 6.1: Searching in Sorted File](#)
- [Exercise 6.2: Optimising Sorted-file Search](#)
- [Exercise 7: Insertion into Static Hashed File](#)
- [Hash Values and Bit-strings](#)
- [Exercise 8: Bit Manipulation](#)
- [Exercise 9: Insertion into Linear Hashed File](#)

❖ Exercise 1: File Merging

Implement a merging algorithm

- for two sorted files, using 3 buffers, with $b_1=5, b_2=3$
- for one unsorted file, using 3 buffers, with $b = 12$
- for one unsorted file, using 5 buffers, with $b = 27$

Assume that we have functions

- **get_page(*rel*, *pid*, *buf*)** ... read specified page into buffer
- **put_page(*rel*, *pid*, *buf*)** ... write a page to disk, at position *pid*
- **clear_page(*rel*, *buf*)** ... make page have zero tuples
- **sort_page(*buf*)** ... in-memory sort of tuples in page
- **nPages(*rel*)**, **nTuples(*buf*)**, **get_tuple(*buf*, *tid*)**

❖ Exercise 2: Sort-based Projection

Consider a table $R(x,y,z)$ with tuples:

Page 0:	(1,1,'a')	(11,2,'a')	(3,3,'c')
Page 1:	(13,5,'c')	(2,6,'b')	(9,4,'a')
Page 2:	(6,2,'a')	(17,7,'a')	(7,3,'b')
Page 3:	(14,6,'a')	(8,4,'c')	(5,2,'b')
Page 4:	(10,1,'b')	(15,5,'b')	(12,6,'b')
Page 5:	(4,2,'a')	(16,9,'c')	(18,8,'c')

SQL: **create T as (select distinct y from R)**

Assuming:

- 3 memory buffers, 2 for input, one for output
- pages/buffers hold 3 \mathbf{R} tuples (i.e. $c_R=3$), 6 \mathbf{T} tuples (i.e. $c_T=6$)

Show how sort-based projection would execute this statement.

❖ Exercise 3: Hash-based Projection

Consider a table $R(x,y,z)$ with tuples:

```

Page 0:  (1,1,'a')    (11,2,'a')   (3,3,'c')
Page 1:  (13,5,'c')   (2,6,'b')    (9,4,'a')
Page 2:  (6,2,'a')    (17,7,'a')   (7,3,'b')
Page 3:  (14,6,'a')   (8,4,'c')    (5,2,'b')
Page 4:  (10,1,'b')   (15,5,'b')   (12,6,'b')
Page 5:  (4,2,'a')    (16,9,'c')   (18,8,'c')
-- and then the same tuples repeated for pages 6-11

```

SQL: `create T as (select distinct y from R)`

Assuming:

- 4 memory buffers, one for input, 3 for partitioning
- pages/buffers hold 3 \mathbf{R} tuples (i.e. $c_R=3$), 4 \mathbf{T} tuples (i.e. $c_T=4$)
- hash functions: $h1(x) = x \% 3$, $h2(x) = (x \% 4) \% 3$

Show how hash-based projection would execute this statement.

❖ Exercise 4: Query Types

Using the relation:

```
create table Courses (
    id      integer primary key,
    code    char(8),   -- e.g. 'COMP9315'
    title   text,      -- e.g. 'Computing 1'
    year    integer,   -- e.g. 2000..2016
    convenor integer references Staff(id),
    constraint once_per_year unique (code,year)
);
```

give examples of each of the following query types:

1. a 1-d *one* query, an n-d *one* query
2. a 1-d *pmr* query, an n-d *pmr* query
3. a 1-d *range* query, an n-d *range* query

Suggest how many solutions each might produce ...

❖ Exercise 5: Cost of Deletion in Heaps

Consider the following queries ...

```
delete from Employees where id = 12345 -- one  
delete from Employees where dept = 'Marketing' -- pmr  
delete from Employees where 40 <= age and age < 50 -- range
```

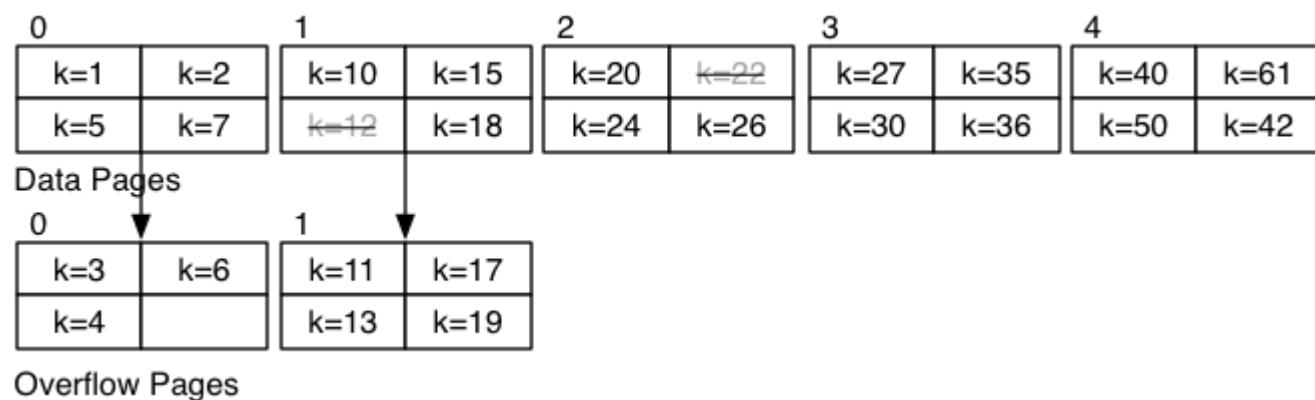
Show how each will be executed and estimate the cost, assuming:

- $b = 100, b_{q2} = 3, b_{q3} = 20$

State any other assumptions.

❖ Exercise 6.1: Searching in Sorted File

Consider this sorted file with overflows ($b=5$, $c=4$):



Compute the cost for answering each of the following:

- **select * from R where k = 24**
- **select * from R where k = 3**
- **select * from R where k = 14**
- **select max(k) from R**

❖ Exercise 6.2: Optimising Sorted-file Search

The **searchBucket(f, p, k, val)** function requires:

- read the p^{th} page from data file
- scan it to find a match and min/max k values in page
- while no match, repeat the above for each overflow page
- if we find a match in any page, return it
- otherwise, remember min/max over all pages in bucket

Suggest an optimisation that would improve **searchBucket()** performance for most buckets.

❖ Exercise 7: Insertion into Static Hashed File

Consider a file with $b=4$, $c=3$, $d=2$, $h(x) = \text{bits}(d, \text{hash}(x))$

Insert tuples in alpha order with the following keys and hashes:

k	$\text{hash}(k)$	k	$\text{hash}(k)$	k	$\text{hash}(k)$	k	$\text{hash}(k)$
a	10001	g	00000	m	11001	s	01110
b	11010	h	00000	n	01000	t	10011
c	01111	i	10010	o	00110	u	00010
d	01111	j	10110	p	11101	v	11111
e	01100	k	00101	q	00010	w	10000
f	00010	l	00101	r	00000	x	00111

The hash values are the 5 lower-order bits from the full 32-bit hash.

❖ Hash Values and Bit-strings

Hashing requires $h(k) :: \text{KeyVal} \rightarrow \text{HashVal}$

HashVal is typically a 32-bit integer, which is mapped to $0 .. b-1$

For arbitrary b , mapping done via **PageID** = $h(k) \% b$

If $b = 2^d$, mapping can be done via bitwise AND

E.g. $b == 8$, **PageID** = $h(k) \& 0b0111$

For any d , use a mask with lower-order d bits set to 1

❖ Exercise 8: Bit Manipulation

1. Write a function to display **uint32** values as **01010110...**

```
char *showBits(uint32 val, char *buf);
```

(assumes supplied buffer is large enough, like **gets()**)

2. Write a function to extract the d bits of a **uint32**

```
uint32 bits(int d, uint32 val);
```

If $d > 0$, gives low-order bits; if $d < 0$, gives high-order bits

❖ Exercise 9: Insertion into Linear Hashed File

Consider a file with $b=4$, $c=3$, $d=2$, $sp=0$, $hash(x)$ as below

Insert tuples in alpha order with the following keys and hashes:

k	$hash(k)$	k	$hash(k)$	k	$hash(k)$	k	$hash(k)$
a	10001	g	00000	m	11001	s	01110
b	11010	h	00000	n	01000	t	10011
c	01111	i	10010	o	00110	u	00010
d	01111	j	10110	p	11101	v	11111
e	01100	k	00101	q	00010	w	10000
f	00010	l	00101	r	00000	x	00111

The hash values are the 5 lower-order bits from the full 32-bit hash.

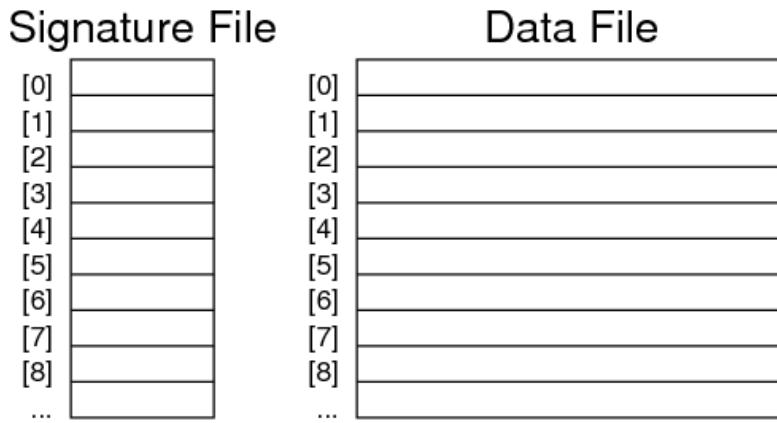
Split before every sixth insert.

SIMC Indexing

- [Signature-based indexing](#)
- [Superimposed Codewords \(SIMC\)](#)
- [SIMC Example](#)
- [SIMC Queries](#)
- [Example SIMC Query](#)
- [SIMC Parameters](#)
- [Query Cost for SIMC](#)
- [Page-level SIMC](#)
- [Bit-sliced SIMC](#)
- [Comparison of Approaches](#)
- [Signature-based Indexing in PostgreSQL](#)

❖ Signature-based indexing

Reminder: file organisation for signature indexing (two files)



One signature slot per tuple slot; unused signature slots are zeroed.

❖ Superimposed Codewords (SIMC)

In a superimposed codewords (simc) indexing scheme

- a tuple descriptor is formed by overlaying attribute codewords
- each codeword is m bits long and has k bits set to 1

A tuple descriptor $\text{desc}(t)$ is

- a bit-string, m bits long, where $j \leq nk$ bits are set to 1
- $\text{desc}(t) = \text{cw}(A_1) \text{ OR } \text{cw}(A_2) \text{ OR } \dots \text{ OR } \text{cw}(A_n)$

Method (assuming all n attributes are used in descriptor):

```
bits desc = 0
for (i = 1; i <= n; i++) {
    bits cw = codeword(A[i],m,k)
    desc = desc | cw
}
```

❖ SIMC Example

Consider the following tuple (from bank deposit database)

Branch	AcctNo	Name	Amount
Perryridge	102	Hayes	400

It has the following codewords/descriptor (for $m = 12$, $k = 2$)

$$A_i \quad cw(A_i)$$

$$\text{Perryridge } \mathbf{010000000001}$$

$$102 \quad \mathbf{000000000011}$$

$$\text{Hayes } \mathbf{000001000100}$$

$$400 \quad \mathbf{000010000100}$$

$$desc(t) \quad \mathbf{010011000111}$$

❖ SIMC Queries

To answer query q in SIMC

- first generate $\text{desc}(q)$ by OR-ing codewords for known attributes
- then attempt to match $\text{desc}(q)$ against all signatures in sig file

E.g. consider the query (**Perryridge**, ?, ?, ?).

A_i	$cw(A_i)$
Perryridge	010000000001
?	000000000000
?	000000000000
?	000000000000
$\text{desc}(q)$	010000000001

❖ SIMC Queries (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}  
// scan r signatures  
for each descriptor D[i] in signature file {  
    if (matches(D[i],desc(q))) {  
        pid = pageOf(tupleID(i))  
        pagesToCheck = pagesToCheck ∪ pid  
    }  
}  
// then scan bSQ = bQ + δ pages to check for matches
```

Matching can be implemented efficiently ...

```
#define matches(sig,qdesc) ((sig & qdesc) == qdesc)
```

❖ Example SIMC Query

Consider the query and the example database:

Signature	Deposit Record
01000000001	(Perryridge,?,?,?)
100101001001	(Brighton,217,Green,750)
010011000111	(Perryridge,102,Hayes,400)
101001001001	(Downtown,101,Johnshon,512)
10110000011	(Mianus,215,Smith,700)
010101010101	(Clearview,117,Throggs,295)
100101010011	(Redwood,222,Lindsay,695)

Gives two matches: one **true match**, one **false match**.

❖ SIMC Parameters

False match probability p_F = likelihood of a false match

How to reduce likelihood of false matches?

- use different hash function for each attribute (h_i for A_i)
- increase descriptor size (m)
- choose k so that \approx half of bits are set

Larger m means larger signature file \Rightarrow read more signature data.

Having k too high \Rightarrow increased overlapping.

Having k too low \Rightarrow increased hash collisions.

❖ SIMC Parameters (cont)

How to determine "optimal" m and k ?

1. start by choosing acceptable p_F
(e.g. $p_F \leq 10^{-4}$ i.e. one false match in 10,000)
2. then choose m and k to achieve no more than this p_F .

Formulae to derive m and k given p_F and n :

$$k = 1/\log_e 2 \cdot \log_e (1/p_F)$$

$$m = (1/\log_e 2)^2 \cdot n \cdot \log_e (1/p_F)$$

Formula from Bloom (1970), "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM, 13 (7): 422–426

❖ Query Cost for SIMC

Cost to answer pmr query: $Cost_{pmr} = b_D + b_{sq}$

- read r descriptors on b_D descriptor pages
- then read b_{sq} data pages and check for matches

$b_D = \lceil r/c_D \rceil$ and $c_D = \lfloor B/\lceil m/8 \rceil \rfloor$

E.g. $m=64$, $B=8192$, $r=10^4 \Rightarrow c_D = 1024$, $b_D=10$

b_{sq} includes pages with r_q matching tuples and r_F false matches

Expected false matches = $r_F = (r - r_q) \cdot p_F \cong r \cdot p_F$ if r_q is way smaller than r

E.g. Worst $b_{sq} = r_q + r_F$, Best $b_{sq} = 1$

❖ Page-level SIMC

SIMC has one descriptor per tuple ... potentially inefficient.

Alternative approach: one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

Size of page descriptor (PD) (clearly larger than tuple descriptor):

- use above formulae but with $c.n$ "attributes"

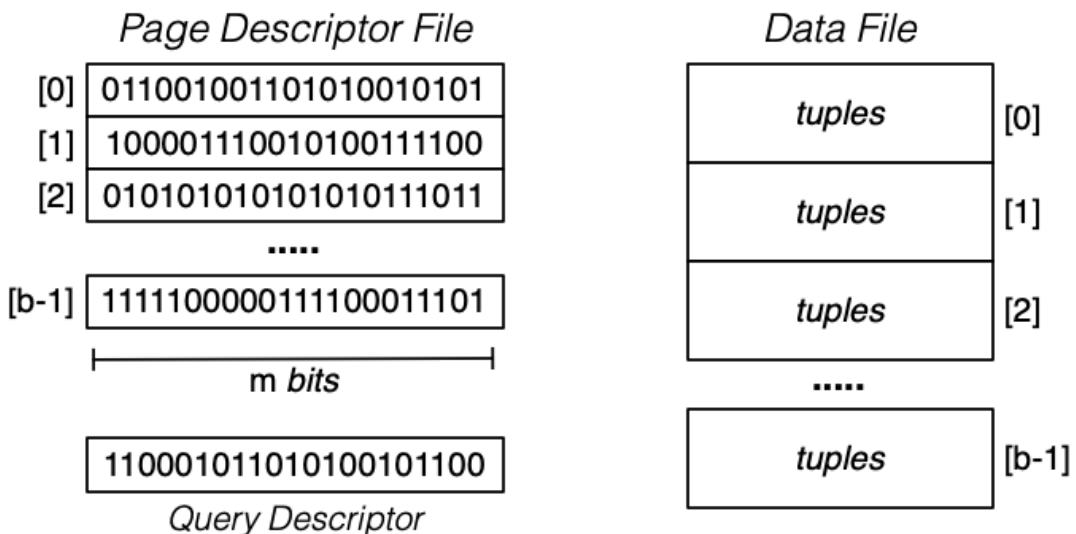
E.g. $n = 4$, $c = 64$, $p_F = 10^{-3}$ $\Rightarrow m_p \cong 3680\text{bits} \cong 460\text{bytes}$

Typically, pages are 1..8KB \Rightarrow 8..64 PD/page (c_{PD}).

E.g. $m_p \cong 460$, $B = 8192$, $c_{PD} \cong 17$

❖ Page-level SIMC (cont)

File organisation for page-level superimposed codeword index



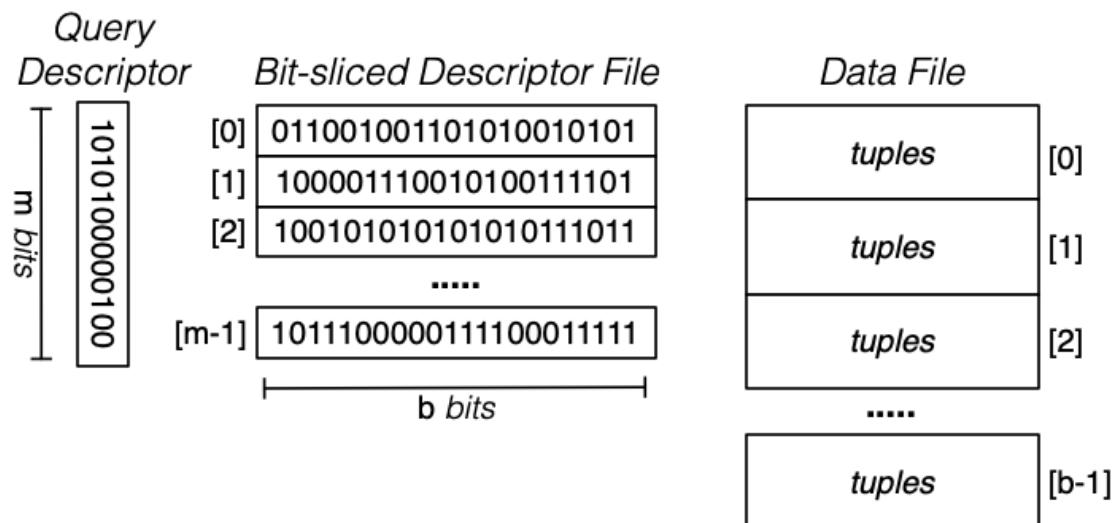
❖ Page-level SIMC (cont)

Algorithm for evaluating *pmr* query using page descriptors

```
pagesToCheck = {}  
// scan  $b$   $m_p$ -bit page descriptors  
for each descriptor D[i] in signature file {  
    if (matches(D[i],desc(q))) {  
        pid = i  
        pagesToCheck = pagesToCheck ∪ pid  
    }  
}  
// read and scan  $b_{sq}$  data pages  
for each pid in pagesToCheck {  
    Buf = getPage(dataFile,pid)  
    check tuples in Buf for answers  
}
```

❖ Bit-sliced SIMC

Improvement: store b m -bit page descriptors as m b -bit "bit-slices"



❖ Bit-sliced SIMC (cont)

Algorithm for evaluating *pmr* query using bit-sliced descriptors

```
matches = ~0 //all ones
// scan m r-bit slices
for each bit i set to 1 in desc(q) {
    slice = fetch bit-slice i
    matches = matches & slice
}
for each bit i set to 1 in matches {
    fetch page i
    scan page for matching records
}
```

Effective because $desc(q)$ typically has less than half bits set to 1

❖ Comparison of Approaches

Tuple-based

- r signatures, m -bit signatures, k bits/attribute
- read all pages of signature file in filtering for a query

Page-based

- b signatures, m_p -bit signatures, k bits/attribute
- read all pages of signature file in filtering for a query

Bit-sliced

- m signatures, b -bit slices, k bits/attribute
- read less than half of the signature file in filtering for a query

All signature files are roughly the same size, for a given p_F

❖ Signature-based Indexing in PostgreSQL

PostgreSQL supports signature based indexing via the **bloom** module

(Signature-based indexes like this are often called **Bloom filters**)

Creating a Bloom index

```
create index Idx on R using bloom (a1,a2,a3)
with    (length=64, col1=3, col2=4, col3=2);
```

Example: 10000 tuples, query = select * from R where a1=55 and a2=42, no matching tuples, random numeric values for attributes

No indexes ... execution time 15ms
B-tree index on all attributes ... execution time 12ms
Bloom index ... execution time 0.4ms

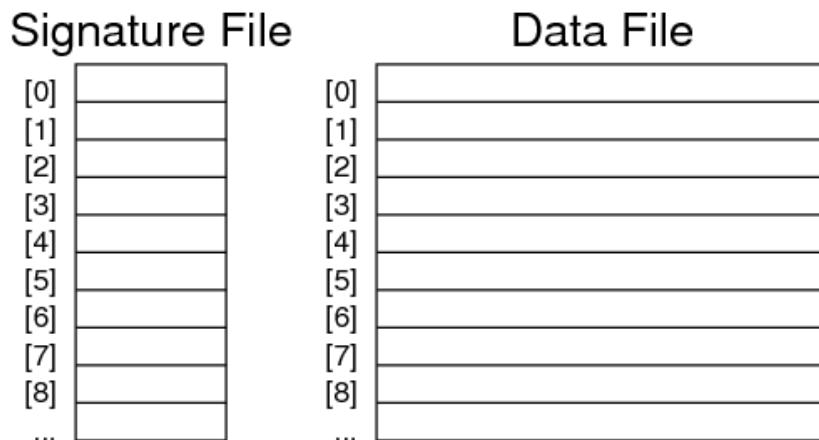
For more details, see PostgreSQL doc, Appendix F.7

CATC Indexing

- [Signature-based indexing](#)
- [Concatenated Codewords \(CATC\)](#)
- [CATC Example](#)
- [CATC Queries](#)
- [Example CATC Query](#)
- [CATC Parameters](#)
- [Query Cost for CATC](#)
- [Variations on CATC](#)
- [Comparison with SIMC](#)

❖ Signature-based indexing

Reminder: file organisation for signature indexing (two files)



One signature slot per tuple slot; unused signature slots are zeroed.

We use the terms "signature" and "descriptor" interchangeably

❖ Concatenated Codewords (CATC)

In a concatenated codewords (**catc**) indexing scheme

- a tuple signature is formed by concatenating attribute codewords
- the signature is m bits long, with $\approx m/2$ bits set to 1
- codeword for $attr_i$ is u_i bits long and has $\approx u_i/2$ bits set to 1
- each codeword could be different length, but always $\sum_{1..n} u_i = m$

A tuple descriptor (signature) $desc(t)$ is

- $desc(t) = cw(A_n) + cw(A_{n-1}) \dots + cw(A_+) + cw(A_1)$
- where "+" represents bit-string concatenation

The order that the concatenated codewords appears doesn't matter, as long as it's done consistently

❖ CATC Example

Consider the following tuple (from bank deposit database)

Branch	AcctNo	Name	Amount
Perryridge	102	Hayes	400

It has the following codewords/descriptor (for $m = 16$, $u_i = 4$)

A_i	$cw(A_i)$
Perryridge	0101
102	1001
Hayes	1010
400	1100
$desc(t)$	1100101010010101

❖ CATC Queries

To answer query q in CATC

- first generate $\text{desc}(q)$ by combining codewords for all attributes
- for known A_i , use $\text{cw}(A_i)$; for "unknown" A_i , use $\text{cw}(A_i) = 0$

E.g. consider the query (**Perryridge**, ?, **Hayes**, ?).

A_i	$\text{cw}(A_i)$
Perryridge	0101
?	0000
Hayes	1010
?	0000
$\text{desc}(q)$	0000101000000101

❖ CATC Queries (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}  
// scan r signatures  
for each descriptor D[i] in signature file {  
    if (matches(D[i],desc(q))) {  
        pid = pageOf(tupleID(i))  
        pagesToCheck = pagesToCheck ∪ pid  
    }  
}  
// then scan b_sq = b_q + δ pages to check for matches
```

Matching can be implemented efficiently ...

```
#define matches(sig,qdesc) ((sig & qdesc) == qdesc)
```

❖ Example CATC Query

Consider the query and the example database:

Signature	Deposit Record
000010100000101	(Perryridge,?,Hayes,?)
1010100101101001	(Brighton,217,Green,750)
1100101010010101	(Perryridge,102,Hayes,400)
1010011010010110	(Downtown,101,Johnshon,512)
0110101001010011	(Mianus,215,Smith,700)
1010101011000101	(Clearview,117,Throggs,295)
1001010100111001	(Redwood,222,Lindsay,695)

Gives two matches: one **true match**, one **false match**.

❖ CATC Parameters

False match probability p_F = likelihood of a false match

How to reduce likelihood of false matches?

- increase descriptor size (m)

Larger m means larger signature file \Rightarrow read more signature data.

Since u_i 's are relatively small, hash collisions may be a serious issue

But making u_i 's means larger signatures \Rightarrow optimisation problem

❖ CATC Parameters (cont)

How to determine "optimal" m and u ?

1. start by choosing acceptable p_F
(e.g. $p_F \leq 10^{-4}$ i.e. one false match in 10,000)
2. then choose m to achieve no more than this p_F .

Formulae to derive "good" m : $m = (1/\log_e 2)^2 \cdot n \cdot \log_e (1/p_F)$

Choice of u_i values

- each A_i has same u_i , or
- allocate u_i based on size of attribute domains

❖ Query Cost for CATC

Cost to answer *pmr* query: $\text{Cost}_{\text{pmr}} = b_D + b_{sq}$

- read r descriptors on b_D descriptor pages
- then read b_{sq} data pages and check for matches

$b_D = \text{ceil}(r/c_D)$ and $c_D = \text{floor}(B/\text{ceil}(m/8))$

E.g. $m=64$, $B=8192$, $r=10^4 \Rightarrow c_D = 1024$, $b_D=10$

b_{sq} includes pages with r_q matching tuples and r_F false matches

Expected false matches = $r_F = (r - r_q).p_F \approx r.p_F$ if r_q is way smaller than r

E.g. Worst $b_{sq} = r_q + r_F$, Best $b_{sq} = 1$

❖ Variations on CATC

CATC has one descriptor per tuple ... potentially inefficient.

Alternative approach: one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

Size of page descriptor $m_p = (1/\log_e 2)^2 \cdot c.n \cdot \log_e (1/p_F)$

Size of codewords is proportionally larger (unless attribute domain small)

E.g. $n = 4, c = 64, p_F = 10^{-3} \Rightarrow m_p \approx 3680 \text{ bits} \approx 460 \text{ bytes}$

Typically, pages are 1..8KB $\Rightarrow 8..64 \text{ PD/page } (c_{PD})$.

E.g. $m_p \approx 460, B = 8192, c_{PD} \approx 17$

❖ Variations on CATC (cont)

Improvement: store $r \times m$ -bit descriptors as $m \times r$ -bit "bit-slices"

If $r = 2^X$ then uses same storage as tuple descriptors

Query cost: scan $u_i / 2$ bit-slices for each known attribute

If k is set of known attribute values, #slices = $\sum_{i \in k} u_i / 2$

E.g. $r = 128, m = 64, n = 4, u_i = 16$

(a, ?, c, ?) requires scan of 2×8 128-bit (16-byte) slices

compared to scan of 128 tuple descriptors, where each descriptor is 64-bits (8-bytes)

❖ Comparison with SIMC

Assume same m , p_F , n for each method ...

CATC has u_i -bit codewords, each has $\cong u_i / 2$ bits set to 1

SIMC has m -bit codewords, each has k bits set to 1

Signatures for both have m bits, with $\cong m / 2$ bits set to 1

CATC has flexibility in u_i , but small(er) codewords so more hash collisions

SIMC has less hash collisions, but has errors from "unfortunate" overlays

Implementing Join

- [Join](#)
- [Join Example](#)
- [Implementing Join](#)
- [Join Summary](#)
- [Join in PostgreSQL](#)

❖ Join

DBMSs are engines to **store**, **combine** and **filter** information.

Join (\bowtie) is the primary means of **combining** information.

Join is important and potentially expensive

Most common join condition: equijoin, e.g. **(R.pk = S.fk)**

Join varieties (natural, inner, outer, semi, anti) all behave similarly.

We consider three strategies for implementing join

- **nested loop** ... simple, widely applicable, inefficient without buffering
- **sort-merge** ... works best if tables are sorted on join attributes
- **hash-based** ... requires good hash function and sufficient buffering

❖ Join Example

Consider a university database with the schema:

```
create table Student(  
    id      integer primary key,  
    name    text, ...  
);  
create table Enrolled(  
    stude   integer references Student(id),  
    subj    text references Subject(code), ...  
);  
create table Subject(  
    code    text primary key,  
    title   text, ...  
);
```

We use this example for each join implementation, by way of comparison

❖ Join Example (cont)

Goal: *List names of students in all subjects, arranged by subject.*

SQL query to provide this information:

```
select E.subj, S.name  
from Student S  
      join Enrolled E on (S.id = E.student_id)  
order by E.subj, S.name;
```

And its relational algebra equivalent:

Sort[subj] (Project[subj, name] (Join[id=student_id] (Student, Enrolled)))

To simplify formulae, we denote **Student** by **S** and **Enrolled** by **E**

❖ Join Example (cont)

Some database statistics:

Sym	Meaning	Value
r_S	# student records	20,000
r_E	# enrollment records	80,000
c_S	Student records/page	20
c_E	Enrolled records/page	40
b_S	# data pages in Student	1,000
b_E	# data pages in Enrolled	2,000

Also, in cost analyses later, N = number of memory buffers.

❖ Join Example (cont)

Relation statistics for $\text{Out} = \text{Student} \bowtie \text{Enrolled}$

Sym	Meaning	Value
r_{Out}	# tuples in result	80,000
C_{Out}	result records/page	80
b_{Out}	# data pages in result	1,000

Notes:

- r_{Out} ... one result tuple for each **Enrolled** tuple
- C_{Out} ... result tuples have only **subj** and **name**
- in analyses, ignore cost of writing result ... same in all methods

❖ Implementing Join

A naive join implementation strategy

```
for each tuple  $T_S$  in Students {  
    for each tuple  $T_E$  in Enrolled {  
        if (testJoinCondition( $C, T_S, T_E$ )) {  
             $T_1 = \text{concat}(T_S, T_E)$   
             $T_2 = \text{project}([\text{subj}, \text{name}], T_1)$   
            ResultSet = ResultSet  $\cup \{T_2\}$   
        } } }
```

Problems:

- join condition is tested $r_E \cdot r_S = 16 \times 10^8$ times
- tuples scanned = $r_S + r_S \cdot r_E = 20000 + 20000 \cdot 80000 = 1600020000$

❖ Implementing Join (cont)

An alternative naive join implementation strategy

```
for each tuple  $T_E$  in Enrolled {  
    for each tuple  $T_S$  in Students {  
        if (testJoinCondition( $C, T_S, T_E$ )) {  
             $T_1 = \text{concat}(T_S, T_E)$   
             $T_2 = \text{project}([\text{subj}, \text{name}], T_1)$   
            ResultSet = ResultSet  $\cup \{T_2\}$   
        } } }
```

Relatively minor performance difference ...

- tuples scanned = $r_E + r_E \cdot r_S = 80000 + 80000 \cdot 20000 = 1600080000$

Terminology: relation in outer loop is **outer**; other relation is **inner**

❖ Join Summary

None of nested-loop/[sort-merge](#)/[hash](#) join is superior in some overall sense.

Which strategy is best for a given query depends on:

- sizes of relations being joined, size of buffer pool
- any indexing on relations, whether relations are sorted
- which attributes and operations are used in the query
- number of tuples in S matching each tuple in R
- distribution of data values (uniform, skew, ...)

Given query Q , choosing the "best" join strategy is critical;

the cost difference between best and worst case can be very large.

E.g. $\text{Join}_{[id=\text{student}]}(Student, Enrolled)$: 3,000 ...
2,000,000 page reads

❖ Join in PostgreSQL

Join implementations are under:

src/backend/executor

PostgreSQL supports the three join methods that we discuss:

- nested loop join (**nodeNestloop.c**)
- sort-merge join (**nodeMergejoin.c**)
- hash join (**nodeHashjoin.c**) (hybrid hash join)

The query optimiser chooses appropriate join, by considering

- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)

Nested-loop Join

- [Join Example](#)
- [Nested Loop Join](#)
- [Block Nested Loop Join](#)
- [Cost on Example Query](#)
- [Block Nested Loop Join](#)
- [Index Nested Loop Join](#)

❖ Join Example

SQL query on student/enrolment database:

```
select E.subj, S.name  
from Student S join Enrolled E on (S.id = E.stude)  
order by E.subj
```

And its relational algebra equivalent:

*Sort[subj] (Project[subj,name] (Join[id=stude]
(Student,Enrolled)))*

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$,
 $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with N buffers

❖ Nested Loop Join

Basic strategy ($R.a \bowtie S.b$):

```
Result = {}
for each page i in R {
    pageR = getPage(R,i)
    for each page j in S {
        pageS = getPage(S,j)
        for each pair of tuples  $t_R, t_S$ 
            from pageR,pageS {
                if ( $t_R.a == t_S.b$ )
                    Result = Result  $\cup$  ( $t_R:t_S$ )
    } } }
```

Needs input buffers for R and S, output buffer for "joined" tuples

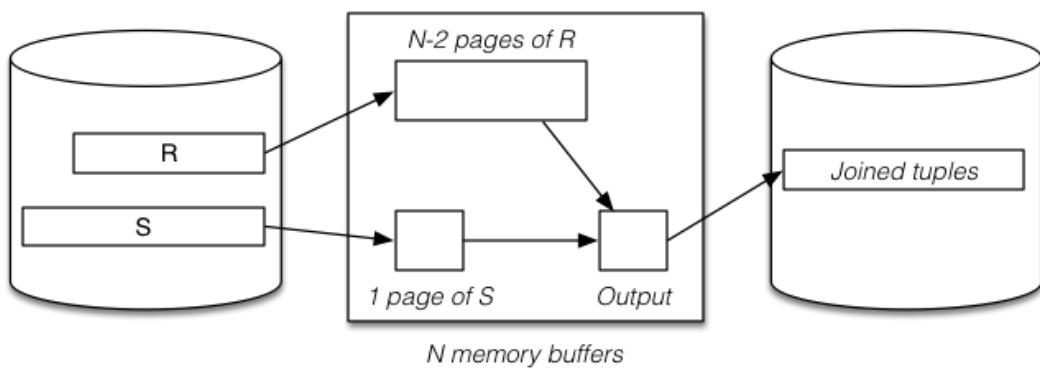
Terminology: R is outer relation, S is inner relation

Cost = $b_R \cdot b_S \dots$ ouch!

❖ Block Nested Loop Join

Method (for N memory buffers):

- read $N-2$ -page chunk of R into memory buffers
- for each S page
check join condition on all (t_R, t_S) pairs in buffers
- repeat for all $N-2$ -page chunks of R



❖ Block Nested Loop Join (cont)

Best-case scenario: $b_R \leq N-2$

- read b_R pages of relation R into buffers
- while whole R is buffered, read b_S pages of S

$$\text{Cost} = b_R + b_S$$

Typical-case scenario: $b_R > N-2$

- read $\lceil b_R/(N-2) \rceil$ chunks of pages from R
- for each chunk, read b_S pages of S

$$\text{Cost} = b_R + b_S \cdot \lceil b_R/N-2 \rceil$$

Note: always requires $r_R \cdot r_S$ checks of the join condition

❖ Cost on Example Query

With $N = 12$ buffers, and S as outer and E as inner

- Cost = $b_S + b_E \cdot \text{ceil}(b_S/(N-2)) = 1000 + 2000 \cdot \text{ceil}(1000/10) = 201000$

With $N = 12$ buffers, and E as outer and S as inner

- Cost = $b_E + b_S \cdot \text{ceil}(b_E/(N-2)) = 2000 + 1000 \cdot \text{ceil}(2000/10) = 202000$

With $N = 102$ buffers, and S as outer and E as inner

- Cost = $b_S + b_E \cdot \text{ceil}(b_S/(N-2)) = 1000 + 2000 \cdot \text{ceil}(1000/100) = 21000$

With $N = 102$ buffers, and E as outer and S as inner

- Cost = $b_E + b_S \cdot \text{ceil}(b_E/(N-2)) = 2000 + 1000 \cdot \text{ceil}(2000/100) = 22000$

❖ Block Nested Loop Join

Why block nested loop join is actually useful in practice ...

Many queries have the form

```
select *
from   R join S on (R.i = S.j)
where   R.x = K
```

This would typically be evaluated as

```
Tmp = Sel[x=K](R)
Res = Join[i=j](Tmp, S)
```

If **Tmp** is small \Rightarrow may fit in memory (in small #buffers)

❖ Index Nested Loop Join

A problem with nested-loop join:

- needs repeated scans of *entire* inner relation S

If there is an index on S , we can avoid such repeated scanning.

Consider $\text{Join}_{[i=j]}(R, S)$:

```
for each tuple r in relation R {  
    use index to select tuples  
        from S where s.j = r.i  
    for each selected tuple s from S {  
        add (r,s) to result  
    }    }
```

❖ Index Nested Loop Join (cont)

This method requires:

- one scan of R relation (b_R)
 - only one buffer needed, since we use R tuple-at-a-time
- for each **tuple** in R (r_R), one index lookup on S
 - cost depends on type of index and number of results
 - best case is when each $R.i$ matches few S tuples

Cost = $b_R + r_R \cdot Sel_S$ (Sel_S is the cost of performing a select on S).

Typical $Sel_S = 1-2$ (hashing) .. b_q (unclustered index)

Trade-off: $r_R \cdot Sel_S$ vs $b_R \cdot b_S$, where $b_R < r_R$ and $Sel_S < b_S$

Sort-merge Join

- [Sort-Merge Join](#)
- [Sort-Merge Join on Example](#)

❖ Sort-Merge Join

Basic approach:

- sort both relations on join attribute (reminder: *Join [i=j]* (R, S))
- scan together using **merge** to form result (r, s) tuples

Advantages:

- no need to deal with "entire" S relation for each r tuple
- deal with runs of matching R and S tuples

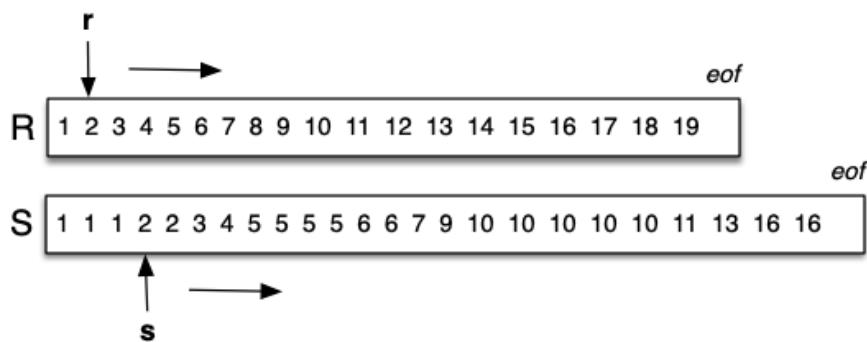
Disadvantages:

- cost of sorting both relations (already sorted on join key?)
- some rescanning required when long runs of S tuples

❖ Sort-Merge Join (cont)

Standard merging requires two cursors:

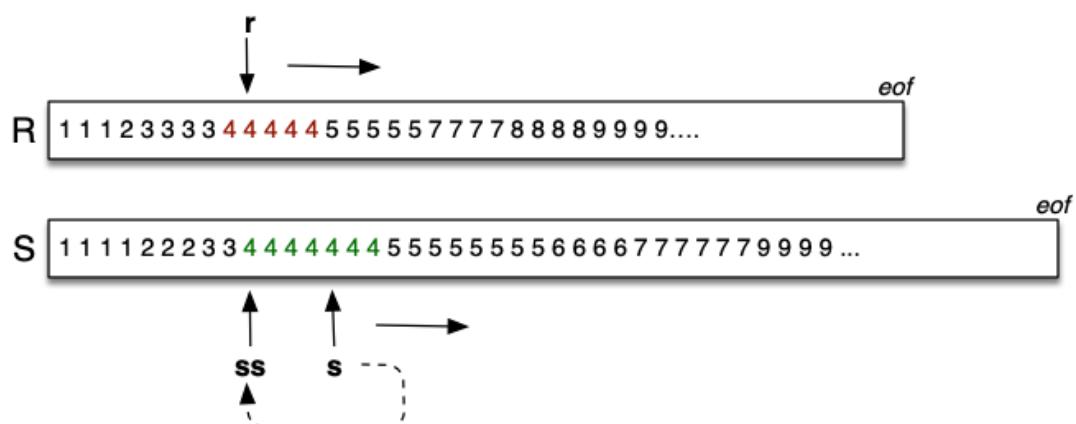
```
while (r != eof && s != eof) {  
    if (r.val ≤ s.val) { output(r.val); next(r); }  
    else { output(s.val); next(s); }  
}  
while (r != eof) { output(r.val); next(r); }  
while (s != eof) { output(s.val); next(s); }
```



❖ Sort-Merge Join (cont)

Merging for join requires 3 cursors to scan sorted relations:

- r = current record in R relation
- s = current record in S relation
- ss = start of current run in S relation



❖ Sort-Merge Join (cont)

Algorithm using query iterators/scanners:

```
Query ri, si;  Tuple r,s;  
  
ri = startScan("SortedR");  
si = startScan("SortedS");  
r=nextTuple(ri)  
s=nextTuple(si)  
while (True) {  
    // align cursors to start of next common run  
    while(r != NULL && s != NULL){  
        if ( r.i < s.j ) r = nextTuple(r.i);  
        else if ( r.i > s.j ) s=nextTuple(s.j);  
        else break;  
    }  
    if (r == NULL || s == NULL) break;  
    // must have (r.i == s.j) here  
    ...
```

❖ Sort-Merge Join (cont)

```
...
    // remember start of current run in S
    TupleID startRun = scanCurrent(si)
    // scan common run, generating result tuples
    while (r != NULL && r.i == s.j) {
        while (s != NULL and s.j == r.i) {
            addTuple(outbuf, combine(r,s));
            if (isFull(outbuf)) {
                writePage(outf, outp++, outbuf);
                clearBuf(outbuf);
            }
            s = nextTuple(si);
        }
        r = nextTuple(ri);
        setScan(si, startRun);
    }
}
```

❖ Sort-Merge Join (cont)

Buffer requirements:

- for sort phase:
 - as many as possible (remembering that cost is $O(\log N)$)
 - if insufficient buffers, sorting cost can dominate
- for merge phase:
 - one output buffer for result
 - one input buffer for relation R
 - (preferably) enough buffers for longest run in S

❖ Sort-Merge Join (cont)

Cost of sort-merge join.

Step 1: sort each relation (if not already sorted):

- Cost = $2.b_R (1 + \lceil \log_{N-1}(b_R / N) \rceil) + 2.b_S (1 + \lceil \log_{N-1}(b_S / N) \rceil)$
(where N = number of memory buffers)

Step 2: merge sorted relations:

- if every run of values in S fits completely in buffers, merge requires single scan, Cost = $b_R + b_S$
- if some runs in of values in S are larger than buffers, need to re-scan run for each corresponding value from R

❖ Sort-Merge Join on Example

SQL query on student/enrolment database:

```
select E.subj, S.name  
from Student S join Enrolled E on (S.id = E.stude)  
order by E.subj
```

And its relational algebra equivalent:

*Sort[subj] (Project[subj,name] (Join[id=stude]
(Student,Enrolled)))*

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$,
 $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with N buffers

❖ Sort-Merge Join on Example (cont)

Case 1: $\text{Join}_{[id=\text{student}]}(\text{Student}, \text{Enrolled})$

- relations are not sorted on $id\#$
- memory buffers $N=32$; all runs are of length < 30

$$\begin{aligned}\text{Cost} &= \text{sort}(S) + \text{sort}(E) + b_S + b_E \\ &= 2b_S(1 + \log_{31}(b_S/32)) + 2b_E(1 + \log_{31}(b_E/32)) + b_S \\ &\quad + b_E \\ &= 2 \times 1000 \times (1+2) + 2 \times 2000 \times (1+2) + 1000 + 2000 \\ &= 6000 + 12000 + 1000 + 2000 \\ &= 21,000\end{aligned}$$

❖ Sort-Merge Join on Example (cont)

Case 2: $\text{Join}_{[id=\text{student}]}(\text{Student}, \text{Enrolled})$

- *Student* and *Enrolled* already sorted on *id#*
- memory buffers $N=4$ (*S* input, $2 \times E$ input, output)
- 5% of the "runs" in *E* span two pages
- there are no "runs" in *S*, since *id#* is a primary key

For the above, no re-scans of *E* runs are ever needed

$\text{Cost} = 2,000 + 1,000 = 3,000$ (regardless of which relation is outer)

Hash Join

- [Hash Join](#)
- [Simple Hash Join](#)
- [Grace Hash Join](#)
- [Hybrid Hash Join](#)
- [Costs for Join Example](#)
- [Join in PostgreSQL](#)

◆ Hash Join

Basic idea:

- use hashing as a technique to partition relations
- to avoid having to consider all pairs of tuples

Requires sufficient memory buffers

- to hold substantial portions of partitions
- (preferably) to hold largest partition of outer relation

Other issues:

- works only for equijoin $R.i=S.j$ (but this is a common case)
- susceptible to data skew (or poor hash function)

Variations: *simple, grace, hybrid.*

❖ Simple Hash Join

Basic approach:

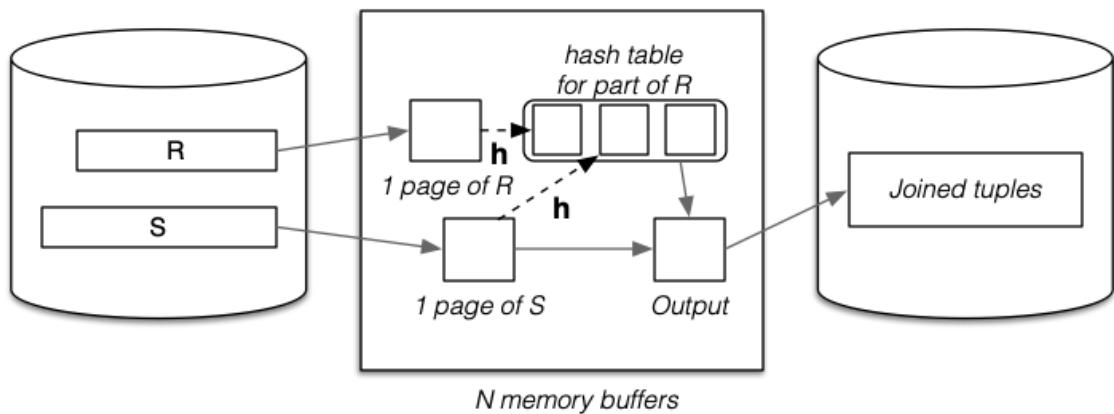
- hash part of outer relation R into memory buffers (build)
- scan inner relation S , using hash to search (probe)
 - if $R.i=S.j$, then $h(R.i)=h(S.j)$ (hash to same buffer)
 - only need to check one memory buffer for each S tuple
- repeat until whole of R has been processed

No overflows allowed in in-memory hash table

- works best with uniform hash function
- can be adversely affected by data/hash skew

❖ Simple Hash Join (cont)

Data flow in hash join:



❖ Simple Hash Join (cont)

Algorithm for simple hash join $\text{Join}[R.i=S.j](R, S)$:

```

for each tuple r in relation R {
    insert r into buffer[h(R.i)]
    if (buffer[h(R.i)]) is full or
        r is the last tuple in R) {
        for each tuple s in relation S {
            for each tuple rr in buffer[h(S.j)] {
                if ((rr,s) satisfies join condition) {
                    add (rr,s) to result
                }
            }
        clear all hash table buffers
    }
}

```

Best case: # join tests $\leq r_S.c_R$ (cf. nested-loop $r_S.r_R$)

❖ Simple Hash Join (cont)

Cost for simple hash join ...

Best case: all tuples of R fit in the hash table

- Cost = $b_R + b_S$
- Same page reads as block nested loop, but less join tests

Good case: refill hash table m times (where $m \geq \text{ceil}(b_R / (N-3))$)

- Cost = $b_R + m.b_S$
- More page reads than block nested loop, but less join tests

Worst case: everything hashes to same page

- Cost = $b_R + b_R.b_S$

❖ Grace Hash Join

Basic approach (for $R \bowtie S$):

- partition both relations on join attribute using hashing (h_1)
- load each partition of R into $N-3$ *buffer hash table (h_2)
- scan through corresponding partition of S to form results
- repeat until all partitions exhausted

For best-case cost ($O(b_R + b_S)$):

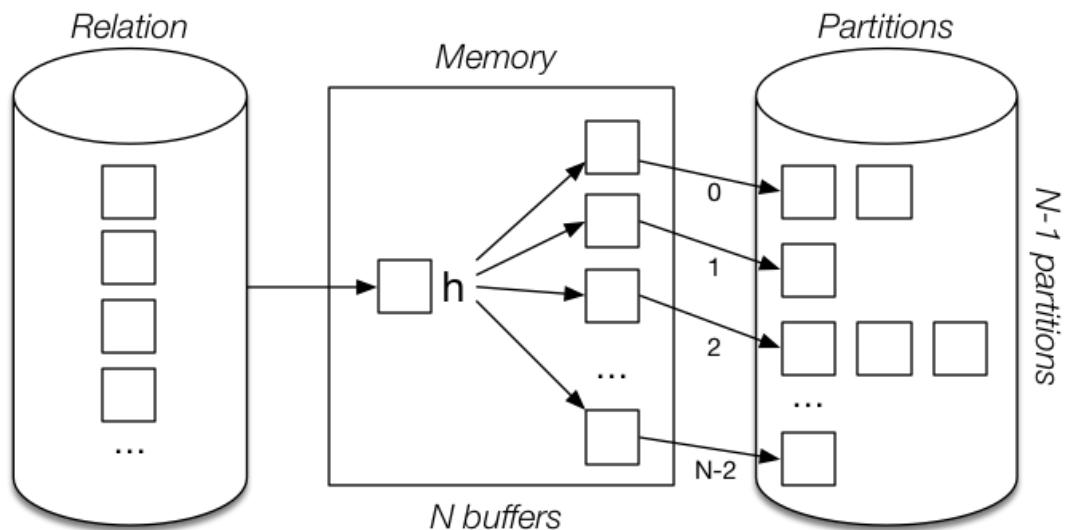
- need $\geq \sqrt{b_R}$ buffers to hold largest partition of outer relation

If $< \sqrt{b_R}$ buffers or poor hash distribution

- need to scan some partitions of S multiple times

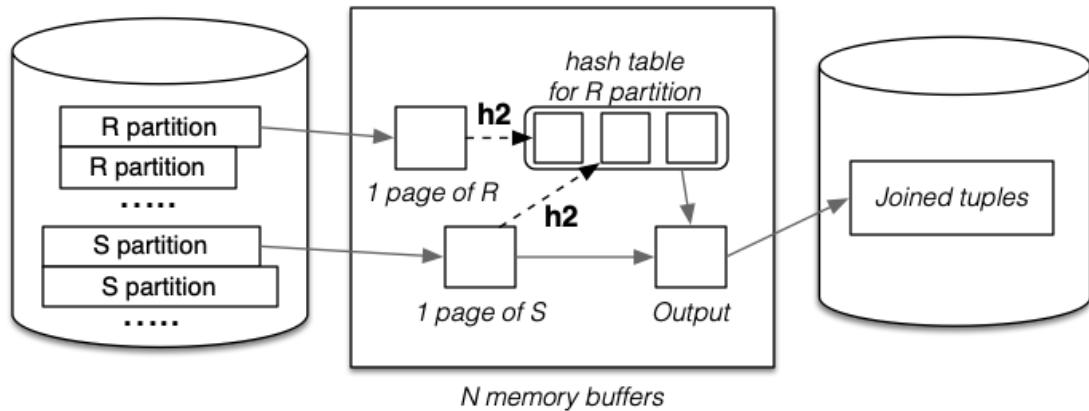
❖ Grace Hash Join (cont)

Partition phase (applied to both R and S):



❖ Grace Hash Join (cont)

Probe/join phase:



The second hash function ($h2$) simply speeds up the matching process. Without it, would need to scan entire R partition for each record in S partition.

❖ Grace Hash Join (cont)

Cost of grace hash join:

- #pages in all partition files of $Rel \approx b_{Rel}$ (maybe slightly more)
- partition relation R ... Cost = $read(b_R) + write(\approx b_R) = 2b_R$
- partition relation S ... Cost = $read(b_S) + write(\approx b_S) = 2b_S$
- probe/join requires one scan of each (partitioned) relation
Cost = $b_R + b_S$
- all hashing and comparison occurs in memory \Rightarrow tiny cost

$$\text{Total Cost} = 2b_R + 2b_S + b_R + b_S = 3(b_R + b_S)$$

❖ Hybrid Hash Join

A variant of grace hash join if we have $\sqrt{b_R} < N < b_R + 2$

- create $k < N$ partitions, 1 in memory, $k-1$ on disk
- buffers: 1 input, $k-1$ output, $p = N-k-2$ for in-memory partition

When we come to scan and partition S relation

- any tuple with hash 0 can be resolved (using in-memory partition)
- other tuples are written to one of k partition files for S

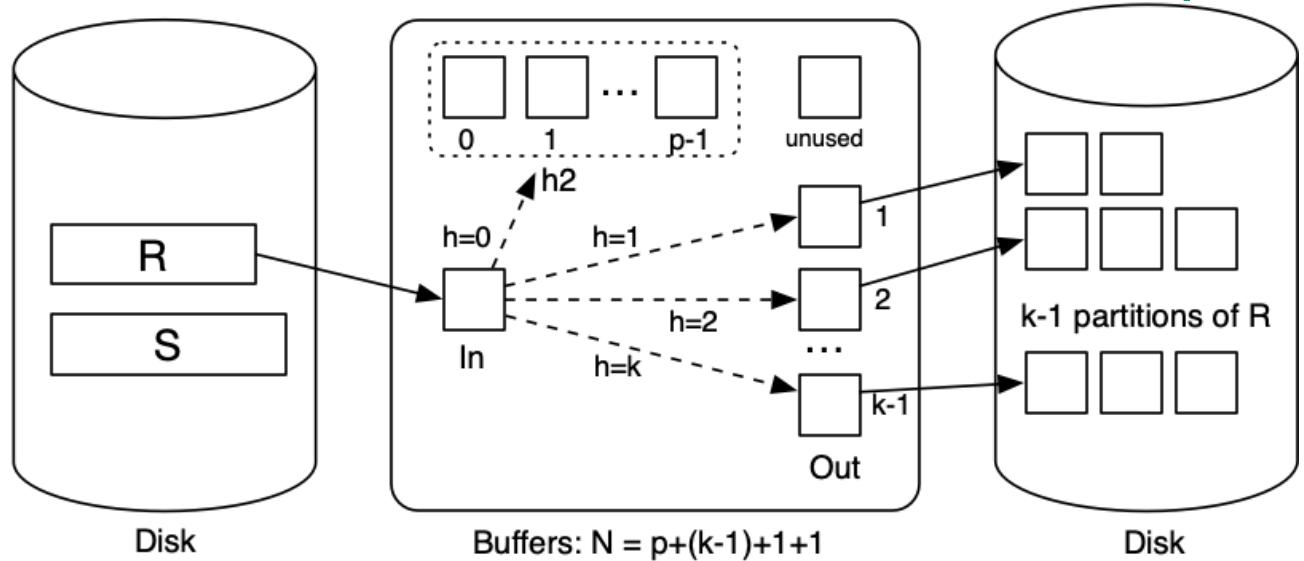
Final phase is same as grace join, but with only $k-1$ partitions.

Comparison:

- grace hash join creates $N-1$ partitions on disk
- hybrid hash join creates 1 (memory) + $k-1$ (disk) partitions

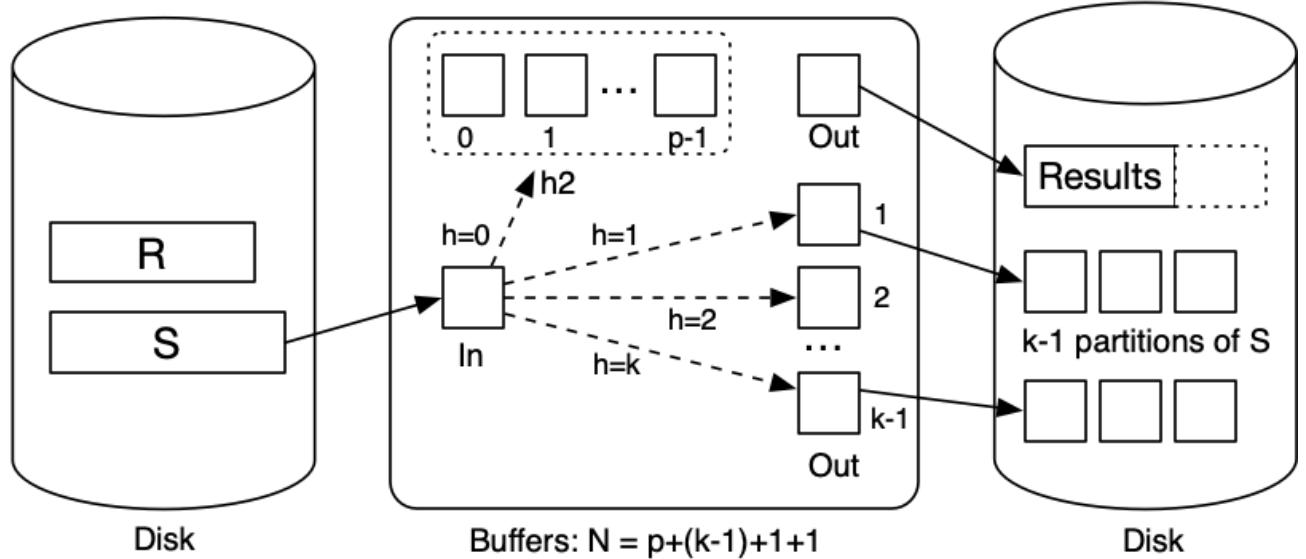
◆ Hybrid Hash Join (cont)

First phase of hybrid hash join (partitioning R):



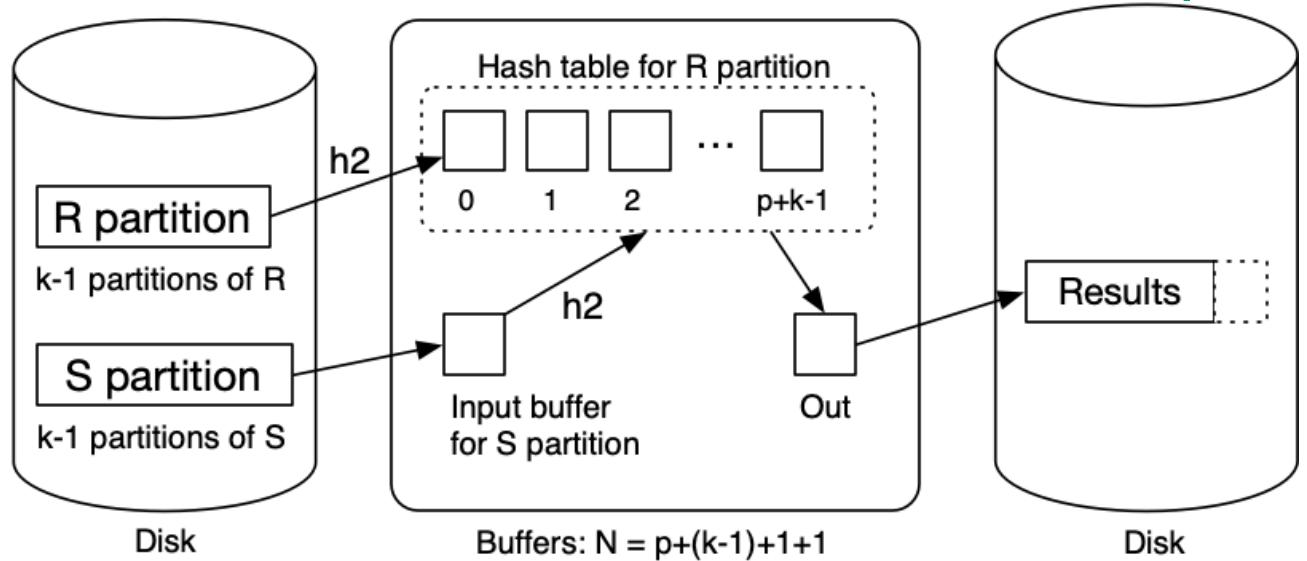
◆ Hybrid Hash Join (cont)

Next phase of hybrid hash join (partitioning S):



◆ Hybrid Hash Join (cont)

Final phase of hybrid hash join (finishing join):



❖ Hybrid Hash Join (cont)

Some observations:

- with k partitions, each partition has expected size $\text{ceil}(b_R/k)$
- holding 1 partition in memory needs $\text{ceil}(b_R/k)$ buffers
- trade-off between in-memory partition space and #partitions

Other notes:

- if $N = b_R + 2$, using block nested loop join is simpler
- cost depends on N (but less than grace hash join)

For k partitions, Cost = $(3 - 2/k) \cdot (b_R + b_S)$

❖ Costs for Join Example

SQL query on student/enrolment database:

```
select E.subj, S.name  
from Student S join Enrolled E on (S.id = E.stude)  
order by E.subj
```

And its relational algebra equivalent:

```
Sort[subj] ( Project[subj, name] ( Join[id=stude] ( Student, Enrolled )  
))
```

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$,
 $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with N buffers

❖ Costs for Join Example (cont)

Costs for hash join variants on example ($N=103$):

Hash Join Method	Cost Analysis	Cost
Hybrid Hash Join	$(3-2/k).(b_S+b_E) = 2.8((1000+2000)$ assuming $k = 10 \dots$ and one partition fits in 91 pages	8700
Grace Hash Join	$3(b_S+b_E) = 3(1000+2000)$	9000
Simple Hash Join	$b_S + b_E \cdot \text{ceil}(b_R/(N-3)) = 1000 + \text{ceil}(1000/100) \cdot 2000 = 1000 + 10 \cdot 2000$	21000
Sort-merge Join	$\text{sort}(S) + \text{sort}(E) + b_S + b_E = 2 \cdot 1000 \cdot 2 + 2 \cdot 2000 \cdot 2 + 1000 + 2000$	11000
Nested-loop Join	$b_S + b_E \cdot \text{ceil}(b_S/(N-2)) = 1000 + 2000 \cdot \text{ceil}(1000/101) = 1000 + 10 \cdot 2000$	21000

❖ Join in PostgreSQL

Join implementations are under: `src/backend/executor`

PostgreSQL supports three kinds of join:

- nested loop join (`nodeNestloop.c`)
- sort-merge join (`nodeMergejoin.c`)
- hash join (`nodeHashjoin.c`) (hybrid hash join)

Query optimiser chooses appropriate join, by considering

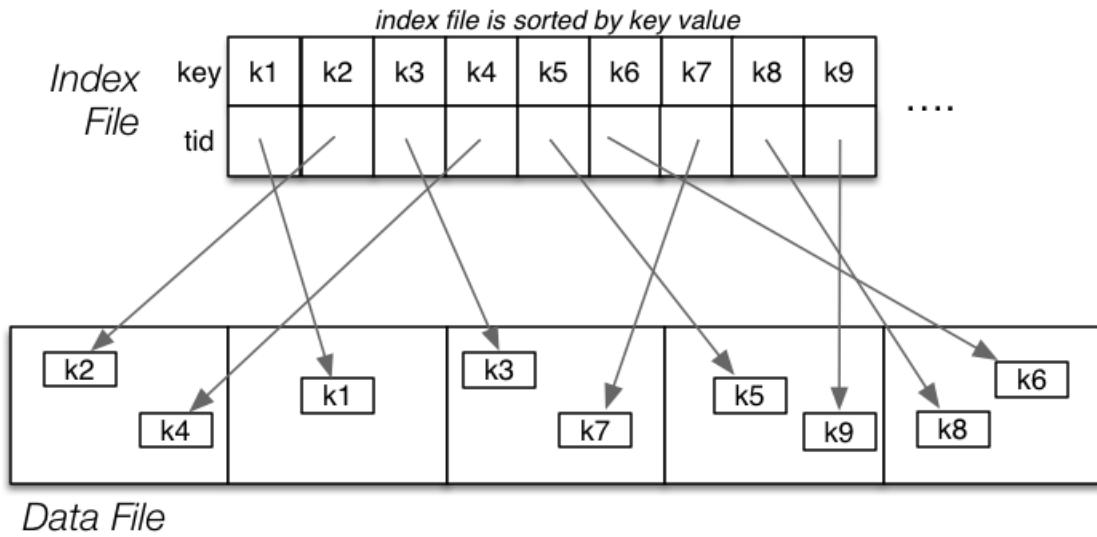
- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)

Indexing

- [Indexing](#)
- [Indexes](#)
- [Dense Primary Index](#)
- [Sparse Primary Index](#)
- [Selection with Primary Index](#)
- [Insertion with Primary Index](#)
- [Deletion with Primary Index](#)
- [Clustering Index](#)
- [Secondary Index](#)
- [Multi-level Indexes](#)
- [Select with Multi-level Index](#)

◆ Indexing

An index is a file of (keyVal,tupleID) pairs, e.g.



❖ Indexes

A 1-d **index** is based on the value of a single attribute A .

Some possible properties of A :

- may be used to sort data file (or may be sorted on some other field)
- values may be unique (or there may be multiple instances)

Taxonomy of index types, based on properties of index attribute:

primary	index on unique field, may be sorted on A
clustering	index on non-unique field, file sorted on A
secondary	file <i>not</i> sorted on A

A given table may have indexes on several attributes.

❖ Indexes (cont)

Indexes themselves may be structured in several ways:

- dense every tuple is referenced by an entry in the index file
- sparse only some tuples are referenced by index file entries
- single-level tuples are accessed directly from the index file
- multi-level may need to access several index pages to reach tuple

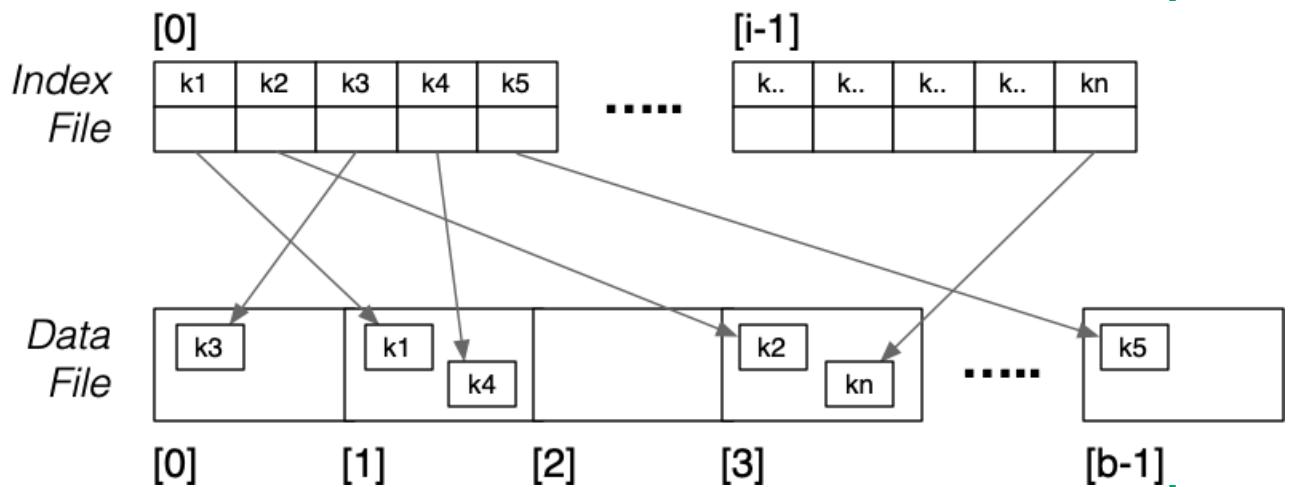
Index file has total i pages (where typically $i \ll b$)

Index file has page capacity c_i (where typically $c_i \gg c$)

Dense index: $i = \lceil r/c_i \rceil$ Sparse index: $i = \lceil b/c_i \rceil$

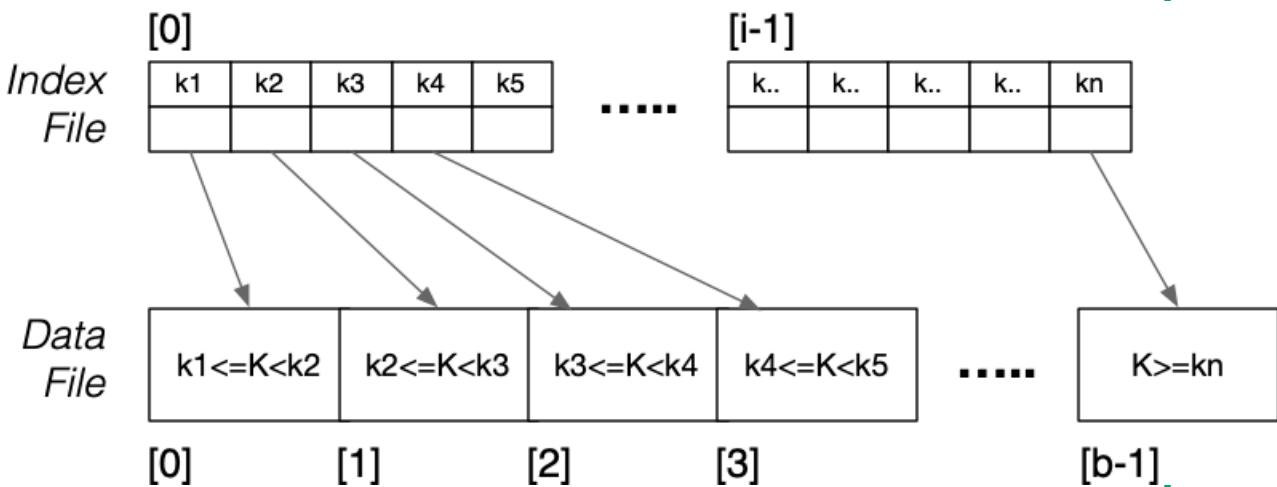
◆ Dense Primary Index

Data file unsorted; one index entry for each tuple



❖ Sparse Primary Index

Data file sorted; one index entry for each page



❖ Selection with Primary Index

For *one* queries:

```
ix = binary search index for entry with key K
if nothing found { return NotFound }
b = getPage(pageOf(ix.tid))
t = getTuple(b,offsetOf(ix.tid))
    -- may require reading overflow pages
return t
```

Worst case: read $\log_2 i$ index pages + read 1+Ov data pages.

Thus, $Cost_{one,prim} = \log_2 i + 1 + Ov$

Assume: index pages are same size as data pages \Rightarrow same reading cost

❖ Selection with Primary Index (cont)

For *range* queries on primary key:

- use index search to find lower bound
- read index sequentially until reach upper bound
- accumulate set of buckets to be examined
- examine each bucket in turn to check for matches

For *pmr* queries involving primary key:

- search as if performing *one* query.

For queries not involving primary key, index gives no help.

❖ Selection with Primary Index (cont)

Method for range queries (when data file is not sorted)

```
// e.g. select * from R where a between lo and hi
pages = {}    results = {}
ixPage = findIndexPage(R.ixf,lo)
while (ixTup = getNextIndexTuple(R.ixf)) {
    if (ixTup.key > hi) break;
    pages = pages ∪ page0f(ixTup.tid)
}
foreach pid in pages {
    // scan data page plus overflow chain
    while (buf = getPage(R.datf,pid)) {
        foreach tuple T in buf {
            if (lo<=T.a && T.a<=hi)
                results = results ∪ T
    }  }  }
```

❖ Insertion with Primary Index

Overview:

tid = insert tuple into page P at position p
find location for new entry in index file
insert new index entry (k,tid) into index file

Problem: order of index entries must be maintained

- need to avoid overflow pages in index (but see later)
- so, reorganise index file by moving entries up

Reorganisation requires, on average, read/write half of index file:

$$Cost_{insert,prim} = (\log_2 i)_r + i/2 \cdot (1_r + 1_w) + (1+Ov)_r + (1+\delta)_w$$

❖ Deletion with Primary Index

Overview:

find tuple using index
mark tuple as deleted
delete index entry for tuple

If we delete index entries by marking ...

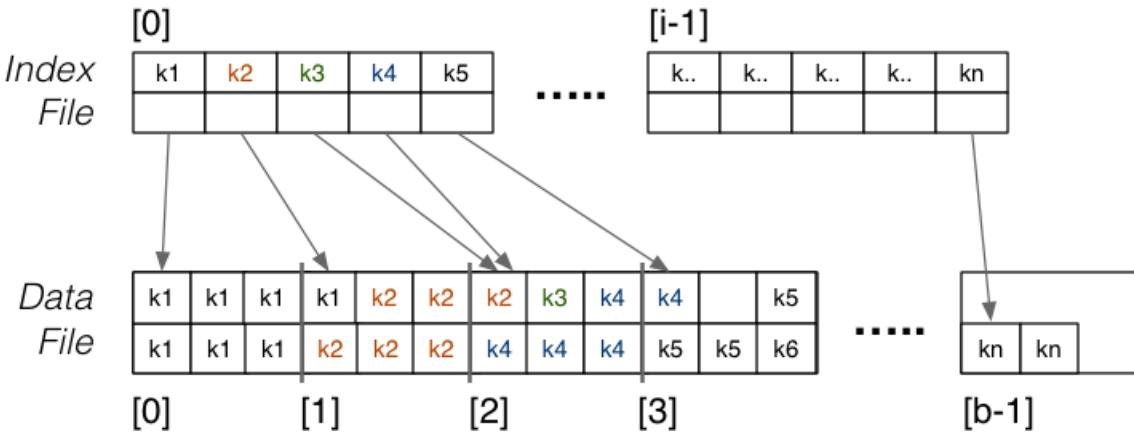
- $Cost_{delete,prim} = (\log_2 i)_r + (1 + Ov)_r + 1_w + 1_w$

If we delete index entry by index file reorganisation ...

- $Cost_{delete,prim} = (\log_2 i)_r + (1 + Ov)_r + i/2.(1_r+1_w) + 1_w$

❖ Clustering Index

Data file sorted; one index entry for each key value

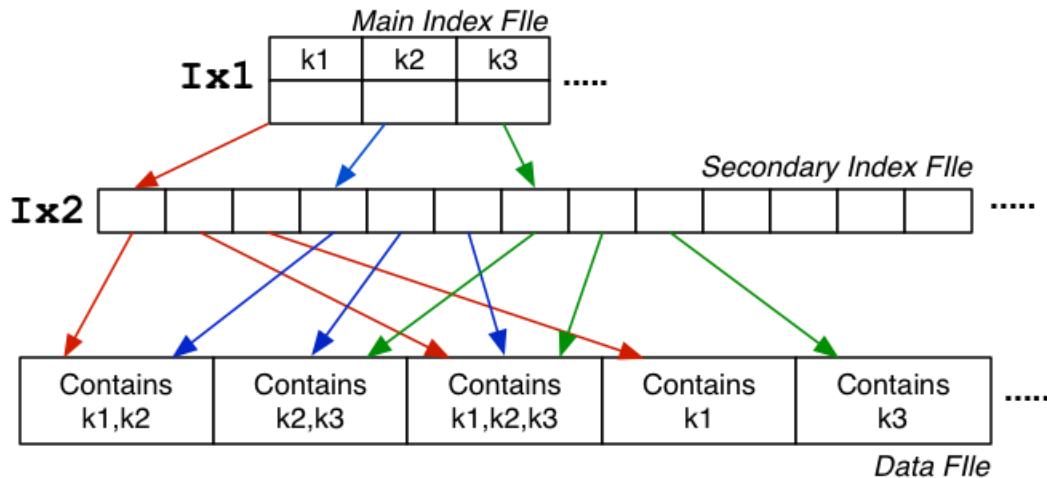


Cost penalty: maintaining both index and data file as sorted

(Note: can't mark index entry for value X until all X tuples are deleted)

❖ Secondary Index

Data file not sorted; want one index entry for each key value



$$Cost_{pmr} = (\log_2 i_{ix1} + a_{ix2} + b_q \cdot (1 + Ov))$$

❖ Multi-level Indexes

Secondary Index used two index files to speed up search

- by keeping the initial index search relatively quick
- **Ix1** small (depends on number of unique key values)
- **Ix2** larger (depends on amount of repetition of keys)
- typically, $b_{Ix1} \ll b_{Ix2} \ll b$

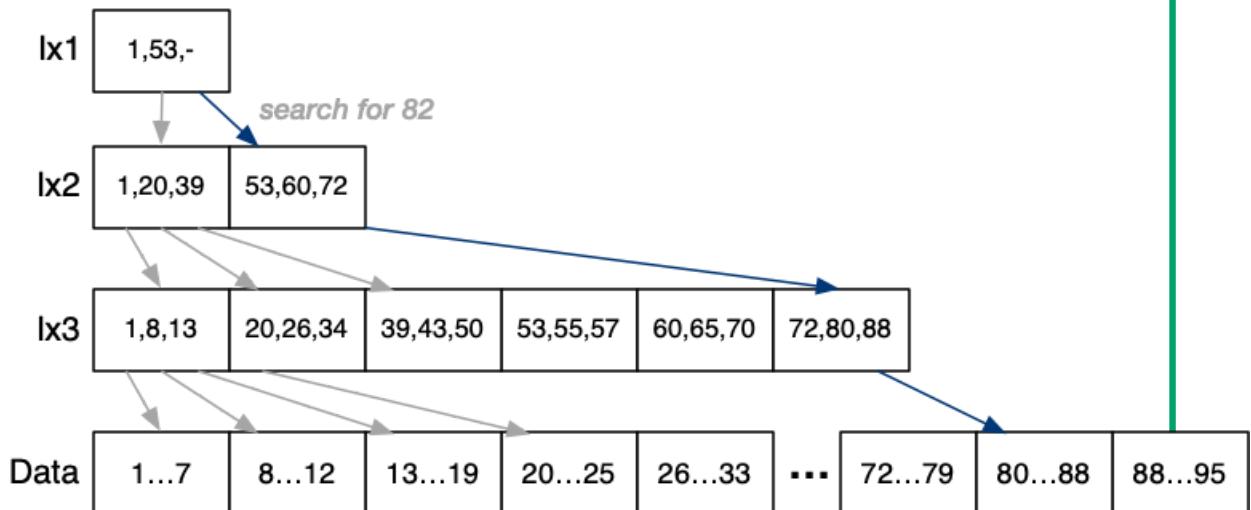
Could improve further by

- making **Ix1** sparse, since **Ix2** is guaranteed to be ordered
- in this case, $b_{Ix1} = \lceil b_{Ix2} / c_i \rceil$
- if **Ix1** becomes too large, add **Ix3** and make **Ix2** sparse
- if data file ordered on key, could make **Ix3** sparse

Ultimately, reduce top-level of index hierarchy to one page.

❖ Multi-level Indexes (cont)

Example data file with three-levels of index:



Assume: not primary key, $c = 20$, $c_i = 3$

In reality, more likely $c = 100$, $c_i = 1000$

❖ Select with Multi-level Index

For *one* query on indexed key field:

```

xpid = top level index page
for level = 1 to d {
    read index entry xpid
    search index page for J'th entry
        where index[J].key <= K < index[J+1].key
    if (J == -1) { return NotFound }
    xpid = index[J].page
}
pid = xpid // pid is data page index
search page pid and its overflow pages

```

$$Cost_{one,mli} = (d + 1 + Ov)_r$$

(Note that $d = \lceil \log_{c_i} r \rceil$ and c_i is large because index entries are small)

B-trees

- [B-Trees](#)
- [B-Tree Depth](#)
- [Selection with B-Trees](#)
- [Insertion into B-Trees](#)
- [Example: B-tree Insertion](#)
- [B-Tree Insertion Cost](#)
- [B-trees in PostgreSQL](#)

❖ B-Trees

B-trees are multi-way search trees with the properties:

- they are updated so as to remain balanced
- each node has at least $(n-1)/2$ entries in it
- each tree node occupies an entire disk page

B-tree insertion and deletion methods

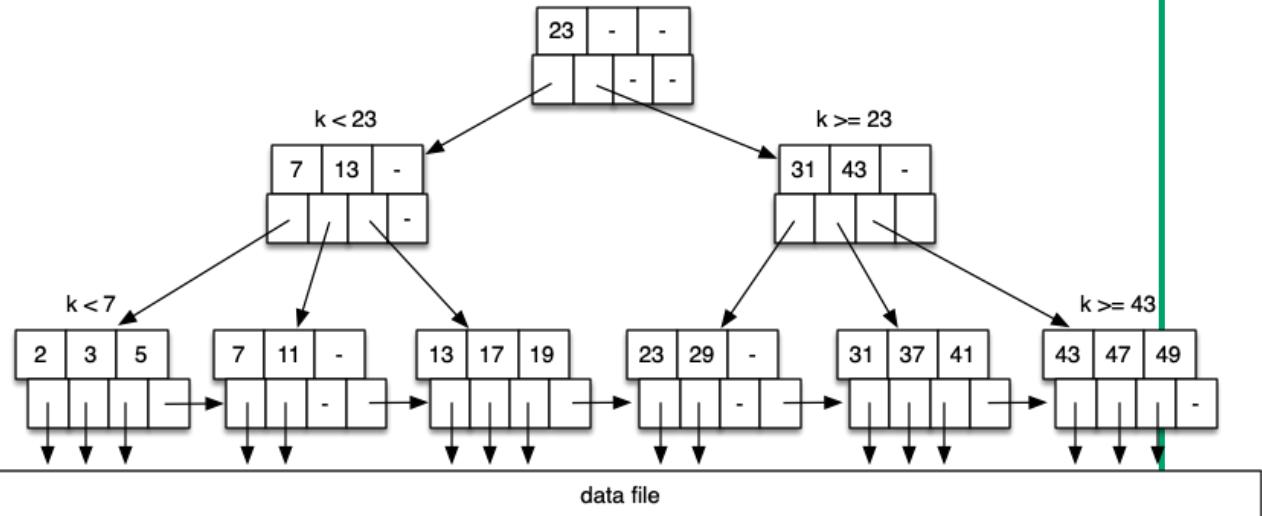
- are moderately complicated to describe
- can be implemented very efficiently

Advantages of B-trees over general multi-way search trees:

- better storage utilisation (around 2/3 full)
- better worst case performance (shallower)

❖ B-Trees (cont)

Example B-tree (depth=3, n=3) (actually B+ tree)



(Note: in DBs, nodes are pages \Rightarrow large branching factor, e.g. $n=500$)

❖ B-Tree Depth

Depth depends on effective branching factor (i.e. how full nodes are).

Simulation studies show typical B-tree nodes are 69% full.

Gives load $L_i = 0.69 \times c_i$ and depth of tree $\sim \text{ceil}(\log_{L_i} r)$.

Example: $c_i=128$, $L_i=88$

Level	#nodes	#keys
root	1	88
1	89	7832
2	7921	697048
3	704969	62037272

Note: c_i is generally larger than 128 for a real B-tree.

❖ Selection with B-Trees

For *one* queries:

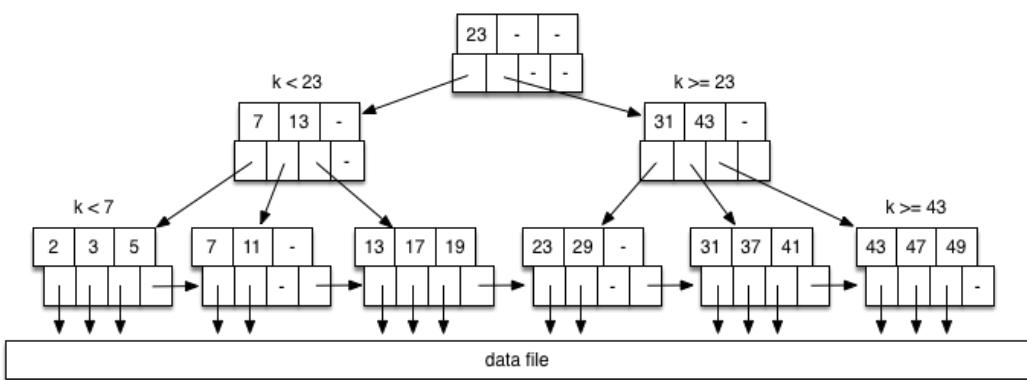
```
Node find(k,tree) {  
    return search(k, root_of(tree))  
}  
Node search(k, node) {  
    // get the page of the node  
    if (is_leaf(node)) return node  
    keys = array of nk key values in node  
    pages = array of nk+1 ptrs to child nodes  
    if (k <= keys[0])  
        return search(k, pages[0])  
    else if (keys[i] < k <= keys[i+1])  
        return search(k, pages[i+1])  
    else if (k > keys[nk-1])  
        return search(k, pages[nk])  
}
```

❖ Selection with B-Trees (cont)

Simplified description of search ...

```
N = B-tree root node
while (N is not a leaf node) N = scanToFindChild(N,K)
tid = scanToFindEntry(N,K)
access tuple T using tid
```

$$\text{Cost}_{\text{one}} = (\textcolor{blue}{D} + \textcolor{red}{1})_r$$

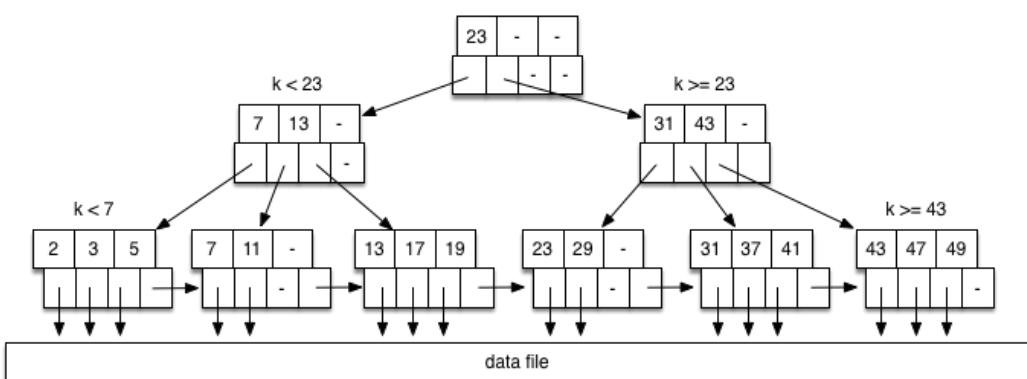


❖ Selection with B-Trees (cont)

For *range* queries (assume sorted on index attribute):

search index to find leaf node for Lo
 for each leaf node entry until Hi found
 add pageOf(tid) to Pages to be scanned
 scan Pages looking for matching tuples

$$\text{Cost}_{\text{range}} = (D + b_i + b_q)r$$



❖ Insertion into B-Trees

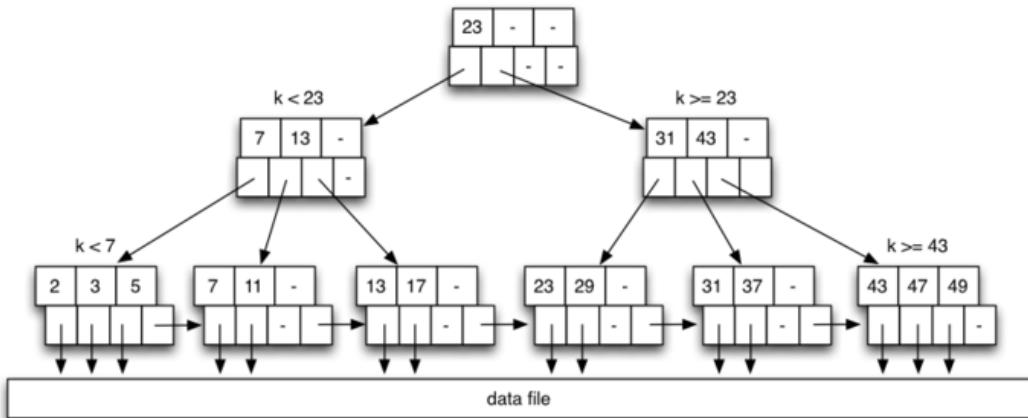
Overview of the method:

1. find leaf node and position in node where new key belongs
2. if node is not full, insert entry into appropriate position
3. if node is full ...
 - promote middle element to parent
 - split node into two half-full nodes (< middle, > middle)
 - insert new key into appropriate half-full node
4. if parent full, split and promote upwards
5. if reach root, and root is full, make new root upwards

Note: if duplicates not allowed and key exists, may stop after step 1.

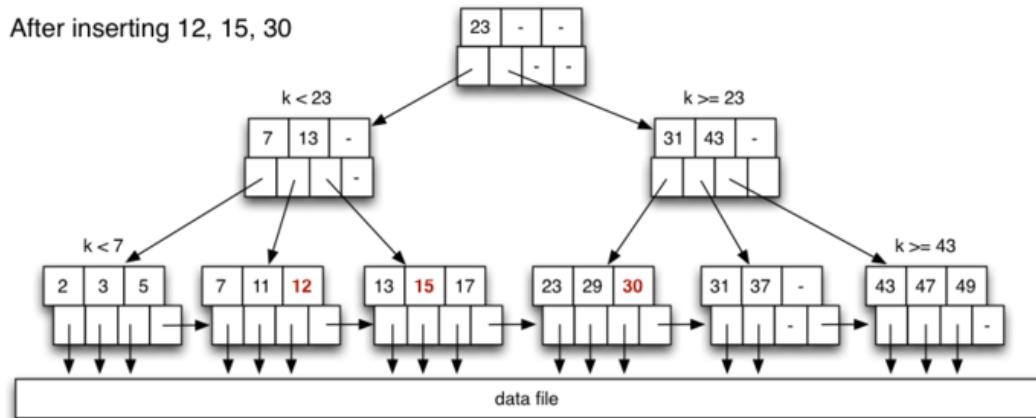
❖ Example: B-tree Insertion

Starting from this tree:

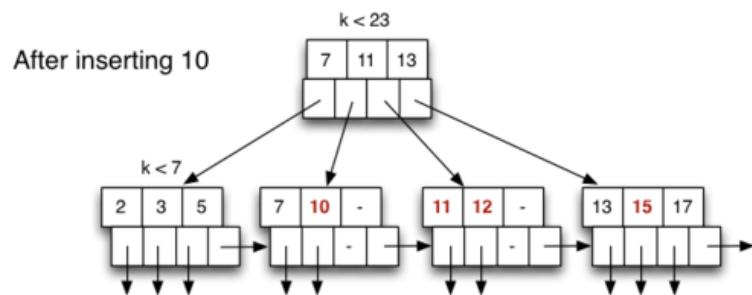
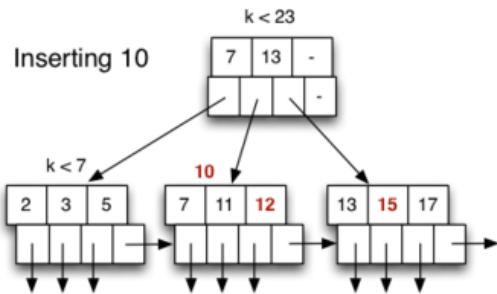


insert the following keys in the given order **12 15 30 10**

❖ Example: B-tree Insertion (cont)



❖ Example: B-tree Insertion (cont)



❖ B-Tree Insertion Cost

Insertion cost = $Cost_{treeSearch}$ + $Cost_{treeInsert}$ + $Cost_{dataInsert}$

Best case: write one page (most of time)

- traverse from root to leaf
- read/write data page, write updated leaf

$$Cost_{insert} = D_r + 1_w + 1_r + 1_w$$

Common case: 3 node writes (rearrange 2 leaves + parent)

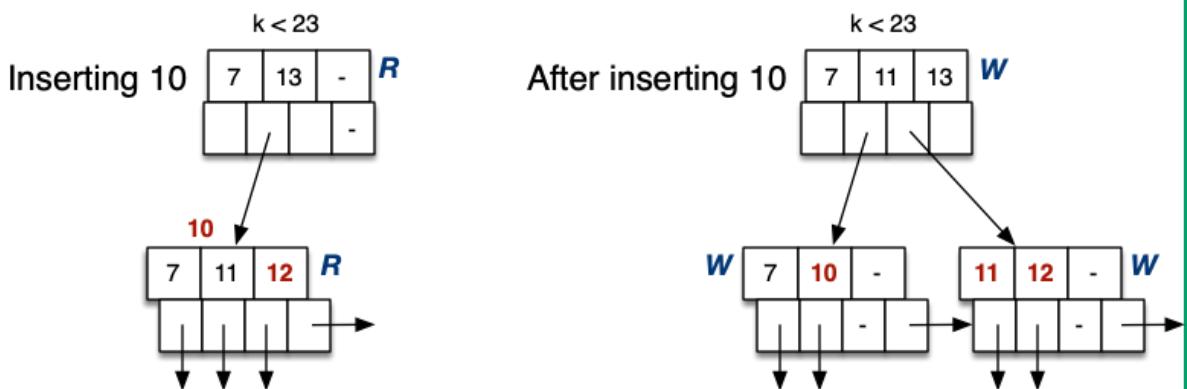
- traverse from root to leaf, holding nodes in buffer
- read/write data page
- update/write leaf, parent and sibling

$$Cost_{insert} = D_r + 3_w + 1_r + 1_w$$

❖ B-Tree Insertion Cost (cont)

Worst case: propagate to root $\text{Cost}_{\text{insert}} = D_r + D \cdot 3_w + 1_r + 1_w$

- traverse from root to leaf
- read/write data page
- update/write leaf, parent and sibling
- repeat previous step $D-1$ times



❖ B-trees in PostgreSQL

PostgreSQL implements \cong Lehman/Yao-style B-trees

- variant that works effectively in high-concurrency environments.

B-tree implementation: **backend/access/nbtree**

- **README** ... comprehensive description of methods
- **nbtree.c** ... interface functions (for iterators)
- **nbtsearch.c** ... traverse index to find key value
- **nbtinsert.c** ... add new entry to B-tree index

Notes:

- stores all instances of equal keys (dense index)
- avoids splitting by scanning right if $\text{key} = \text{max(key)}$ in page
- common insert case: new key is max(key) overall; handled efficiently

❖ B-trees in PostgreSQL (cont)

Changes for PostgreSQL v12

- indexes smaller
 - for composite keys, only store first attribute
 - index entries are smaller, so c_i larger, so tree shallower
- include TID in index key
 - duplicate index entries are stored in "table order"
 - makes scanning table files to collect results more efficient

To explore indexes in more detail:

- `\di+ IndexName`
- `select * from bt_page_items(IndexName,BlockNo)`

❖ B-trees in PostgreSQL (cont)

Interface functions for B-trees

```
// build Btree index on relation
Datum btbuild(rel,index,...)
// insert index entry into Btree
Datum btinsert(rel,key,tupleid,index,...)
// start scan on Btree index
Datum btbeginscan(rel,key,scandesc,...)
// get next tuple in a scan
Datum btgettuple(scandesc,scandir,...)
// close down a scan
Datum btendscan(scandesc)
```

Multi-dimensional Search Trees

- [Multi-dimensional Tree Indexes](#)
- [kd-Trees](#)
- [Searching in kd-Trees](#)
- [Quad Trees](#)
- [Searching in Quad-trees](#)
- [R-Trees](#)
- [Insertion into R-tree](#)
- [Query with R-trees](#)
- [Costs of Search in Multi-d Trees](#)
- [Multi-d Trees in PostgreSQL](#)

❖ Multi-dimensional Tree Indexes

Over the last 20 years, from a range of problem areas

- different multi-d tree index schemes have been proposed
- varying primarily in how they partition tuple-space

Consider three popular schemes: kd-trees, Quad-trees, R-trees.

Example data for multi-d trees is based on the following relation:

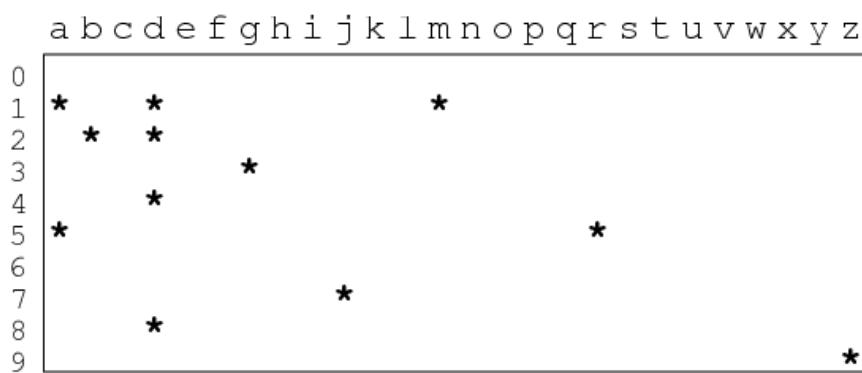
```
create table Rel (
    X char(1) check (X between 'a' and 'z'),
    Y integer check (Y between 0 and 9)
);
```

❖ Multi-dimensional Tree Indexes (cont)

Example tuples:

R('a',1) R('a',5) R('b',2) R('d',1)
R('d',2) R('d',4) R('d',8) R('g',3)
R('j',7) R('m',1) R('r',5) R('z',9)

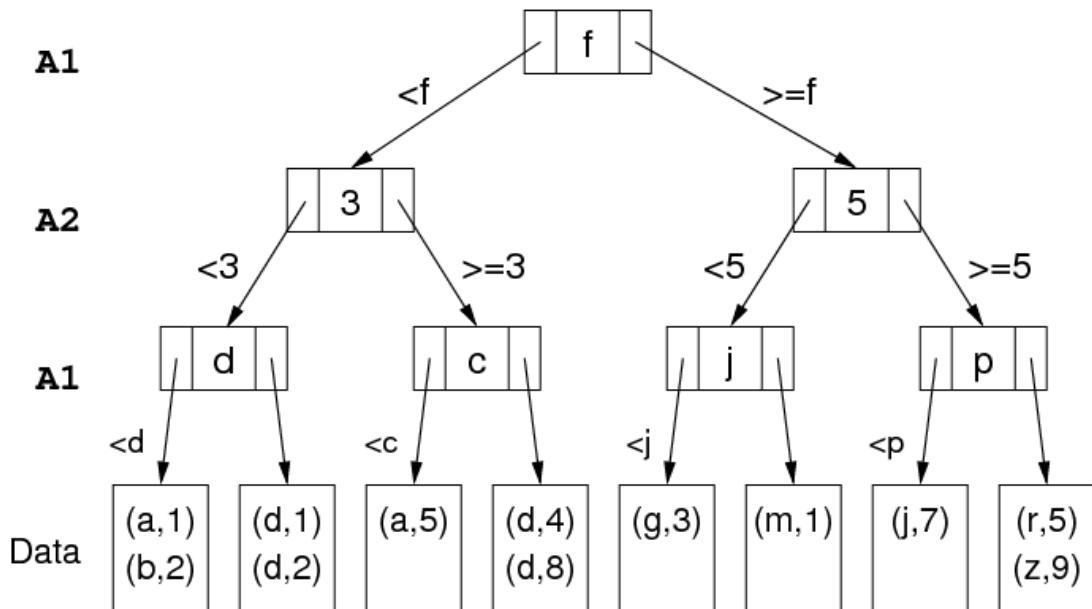
The tuple-space for the above tuples:



❖ kd-Trees

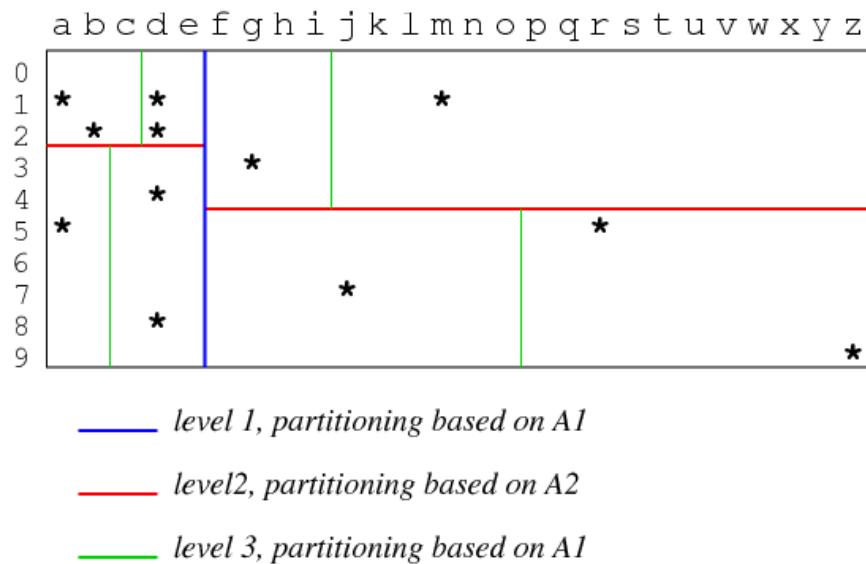
kd-trees are multi-way search trees where

- each level of the tree partitions on a different attribute
- each node contains $n-1$ key values, pointers to n subtrees



❖ kd-Trees (cont)

How this tree partitions the tuple space:



❖ Searching in kd-Trees

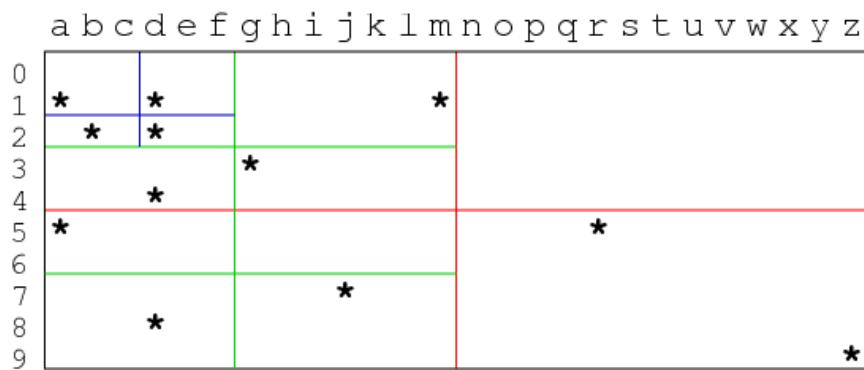
```
// Started by Search(Q, R, 0, kdTreeRoot)
Search(Query Q, Relation R, Level L, Node N)
{
    if (isDataPage(N)) {
        Buf = getPage(fileOf(R), idOf(N))
        check Buf for matching tuples
    } else {
        a = attrLev[L]
        if (!hasValue(Q,a))
            nextNodes = all children of N
        else {
            val = getAttr(Q,a)
            nextNodes = find(N,Q,a,val)
        }
        for each C in nextNodes
            Search(Q, R, L+1, C)
    }
}
```

❖ Quad Trees

Quad trees use regular, disjoint partitioning of tuple space.

- for 2d, partition space into quadrants (NW, NE, SW, SE)
- each quadrant can be further subdivided into four, etc.

Example:



❖ Quad Trees (cont)

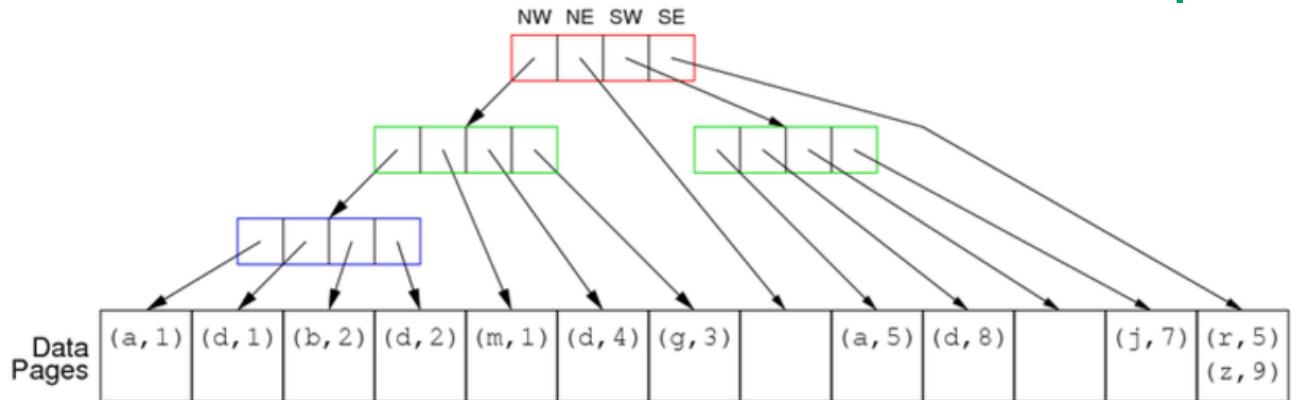
Basis for the partitioning:

- a quadrant that has no sub-partitions is a **leaf quadrant**
- each leaf quadrant maps to a single data page
- subdivide until points in each quadrant fit into one data page
- ideal: same number of points in each leaf quadrant (balanced)
- point density varies over space
⇒ different regions require different levels of partitioning
- this means that the tree is not necessarily balanced

Note: effective for $d \leq 5$, ok for $6 \leq d \leq 10$, ineffective for $d > 10$

❖ Quad Trees (cont)

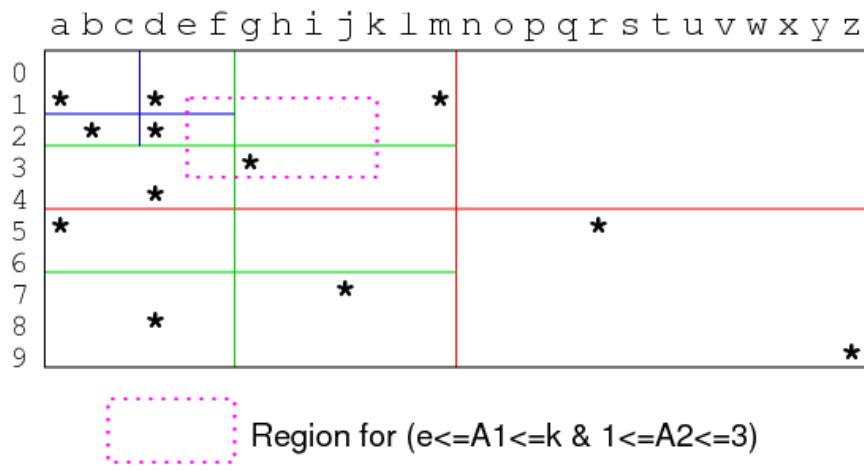
The previous partitioning gives this tree structure, e.g.



In this and following examples, we give coords of top-left, bottom-right of a region

❖ Searching in Quad-trees

Space query example:



Need to traverse: red(NW), green(NW,NE,SW,SE),
blue(NE,SE).

❖ Searching in Quad-trees (cont)

Method for searching in Quad-tree:

- find all regions in current node that query overlaps with
- for each such region, check its node
 - if node is a leaf, check corresponding page for matches
 - else recursively repeat search from current node

Note that query region may be a single point.

❖ R-Trees

R-trees use a flexible, overlapping partitioning of tuple space.

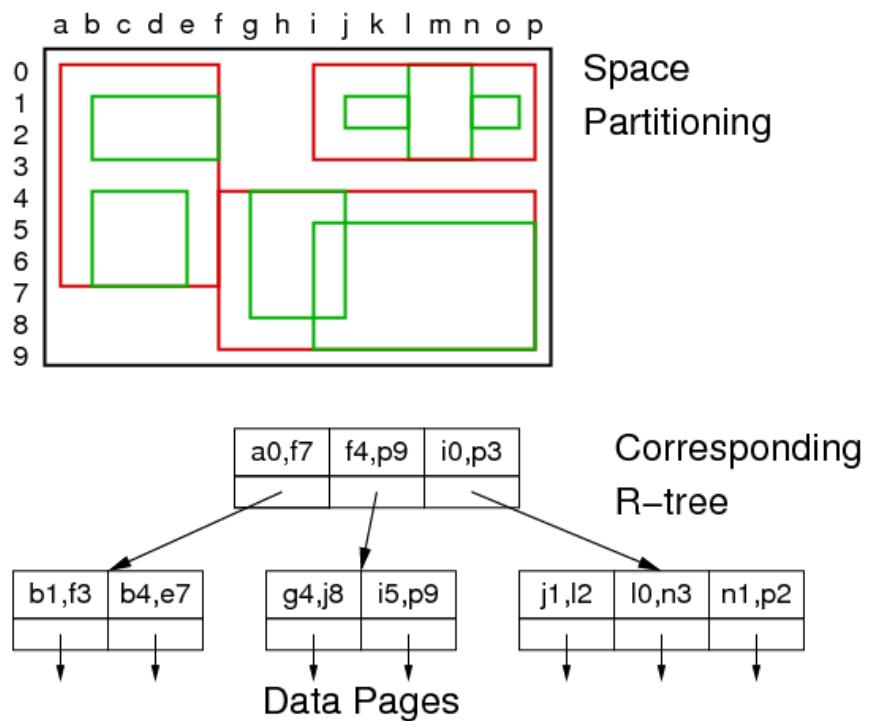
- each node in the tree represents a kd hypercube
- its children represent (possibly overlapping) subregions
- the child regions do not need to cover the entire parent region

Overlap and partial cover means:

- can optimize space partitioning wrt data distribution
- so that there are similar numbers of points in each region

Aim: height-balanced, partly-full index pages (cf. B-tree)

❖ R-Trees (cont)



❖ Insertion into R-tree

Insertion of an object R occurs as follows:

- start at root, look for children that completely contain R
- if no child completely contains R , **choose one** of the children and expand its boundaries so that it does contain R
- if several children contain R , **choose one** and proceed to child
- repeat above containment search in children of current node
- once we reach data page, insert R if there is room
- if no room in data page, replace by two data pages
- **partition** existing objects between two data pages
- update node pointing to data pages
(may cause B-tree-like propagation of node changes up into tree)

Note that R may be a point or a polygon.

❖ Query with R-trees

Designed to handle *space* queries and "where-am-I" queries.

"Where-am-I" query: find all regions containing a given point P :

- start at root, select all children whose subregions contain P
- if there are zero such regions, search finishes with P not found
- otherwise, recursively search within node for each subregion
- once we reach a leaf, we know that region contains P

Space (region) queries are handled in a similar way

- we traverse down any path that intersects the query region

❖ Costs of Search in Multi-d Trees

Cost depends on type of query and tree structure

Best case: *pmr* query where all attributes have known values

- in kd-trees and quad-trees, follow single tree path
- cost is equal to depth D of tree
- in R-trees, may follow several paths (overlapping partitions)

Typical case: some attributes are unknown or defined by range

- need to visit multiple sub-trees
- how many depends on: range, choice-points in tree nodes

❖ Multi-d Trees in PostgreSQL

Up to version 8.2, PostgreSQL had R-tree implementation

Superseded by **GiST** = Generalized Search Trees

GiST indexes parameterise: data type, searching, splitting

- via seven user-defined functions (e.g. **picksplit()**)

GiST trees have the following structural constraints:

- every node is at least fraction f full (e.g. 0.5)
- the root node has at least two children (unless also a leaf)
- all leaves appear at the same level

Details: Chapter 68 in PG Docs or [src/backend/access/gist](#)

Multi-dimensional Hashing

- [Hashing and pmr](#)
- [MA.Hashing Example](#)
- [MA.Hashing Hash Functions](#)
- [Queries with MA.Hashing](#)
- [MA.Hashing Query Algorithm](#)
- [Query Cost for MA.Hashing](#)
- [Optimising MA.Hashing Cost](#)

❖ Hashing and *pmr*

For a *pmr* query like

```
select * from R where  $a_1 = C_1$  and ... and  $a_n = C_n$ 
```

- if one a_i is the hash key, query is very efficient
- if no a_i is the hash key, need to use linear scan

Can be alleviated using *multi-attribute hashing (mah)*

- form a composite hash value involving all attributes
- at query time, some components of composite hash are known
(allows us to limit the number of data pages which need to be checked)

MA.hashing works in conjunction with any dynamic hashing scheme.

❖ Hashing and *pmr* (cont)

Multi-attribute hashing parameters:

- file size = $b = 2^d$ pages \Rightarrow use d -bit hash values
- relation has n attributes: a_1, a_2, \dots, a_n
- attribute a_i has hash function h_i
- attribute a_i contributes d_i bits (to the combined hash value)
- total bits $d = \sum_{i=1..n} d_i$
- a **choice vector** (*cv*) specifies for all $k \in 0..d-1$
bit j from $h_i(a_i)$ contributes bit k in combined hash
value

7	6	5	4	3	2	1	0
$b_{2,2}$	$b_{1,2}$	$b_{3,1}$	$b_{2,1}$	$b_{1,1}$	$b_{3,0}$	$b_{2,0}$	$b_{1,0}$

❖ MA.Hashing Example

Consider relation

Deposit(branch,acctNo,name,amount)

Assume a small data file with 8 main data pages (plus overflows).

Hash parameters: $d=3$ $d_1=1$ $d_2=1$ $d_3=1$ $d_4=0$

Note that we ignore the **amount** attribute ($d_4=0$)

Assumes that nobody will want to ask queries like

```
select * from Deposit where amount=533
```

Choice vector is designed taking expected queries into account.

❖ MA.Hashing Example (cont)

Choice vector:

	7	6	5	4	3	2	1	0
	$b_{2,2}$	$b_{1,2}$	$b_{3,1}$	$b_{2,1}$	$b_{1,1}$	$b_{3,0}$	$b_{2,0}$	$b_{1,0}$

This choice vector tells us:

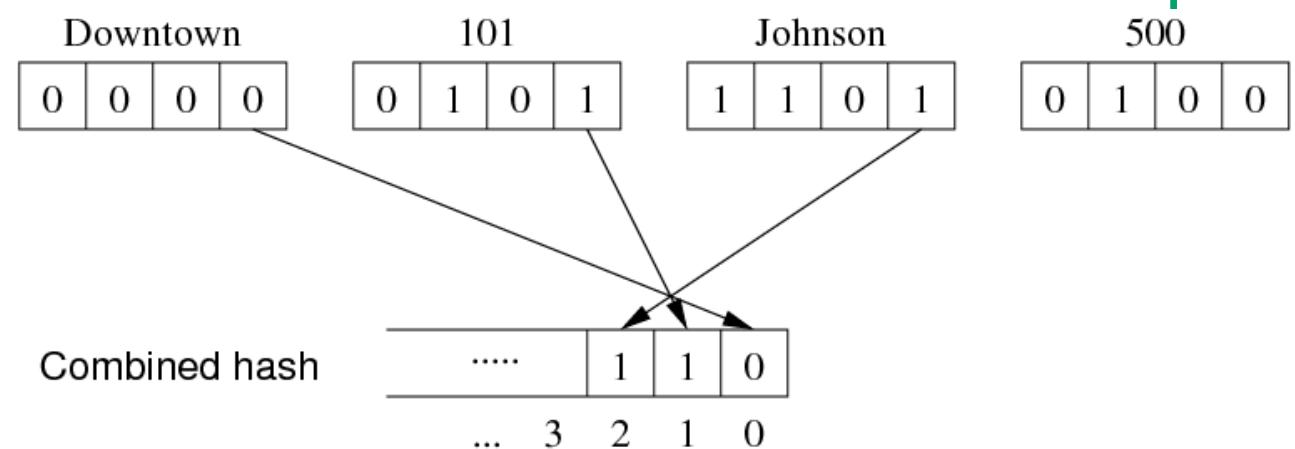
- bit 0 in hash comes from bit 0 of $\text{hash}_1(a_1)$ ($b_{1,0}$)
- bit 1 in hash comes from bit 0 of $\text{hash}_2(a_2)$ ($b_{2,0}$)
- bit 2 in hash comes from bit 0 of $\text{hash}_3(a_3)$ ($b_{3,0}$)
- bit 3 in hash comes from bit 1 of $\text{hash}_1(a_1)$ ($b_{1,1}$)
- etc. etc. etc. (up to as many bits of hashing as required, e.g. 32)

❖ MA.Hashing Example (cont)

Consider the tuple:

branch	acctNo	name	amount
Downtown	101	Johnston	512

Hash value (page address) is computed by:



❖ MA.Hashing Hash Functions

Auxiliary definitions:

```
#define MaxHashSize 32
typedef unsigned int HashVal;

// extracts i'th bit from hash value
#define bit(i,h) (((h) & (1 << (i))) >> (i))

// choice vector elems
typedef struct { int attr, int bit } CVelem;
typedef CVelem ChoiceVec[MaxHashSize];

// hash function for individual attributes
HashVal hash_any(char *val) { ... }
```

❖ MA.Hashing Hash Functions (cont)

Produce combined d -bit hash value for tuple t :

```
HashVal hash(Tuple t, ChoiceVec cv, int d)
{
    HashVal h[nAttr(t)+1]; // hash for each attr
    HashVal res = 0, oneBit;
    int i, a, b;
    for (i = 1; i <= nAttr(t); i++)
        h[i] = hash_any(attrVal(t,i));
    for (i = 0; i < d; i++) {
        a = cv[i].attr;
        b = cv[i].bit;
        oneBit = bit(b, h[a]);
        res = res | (oneBit << i);
    }
    return res;
}
```

❖ Queries with MA.Hashing

In a partial match query:

- values of some attributes are known
- values of other attributes are unknown

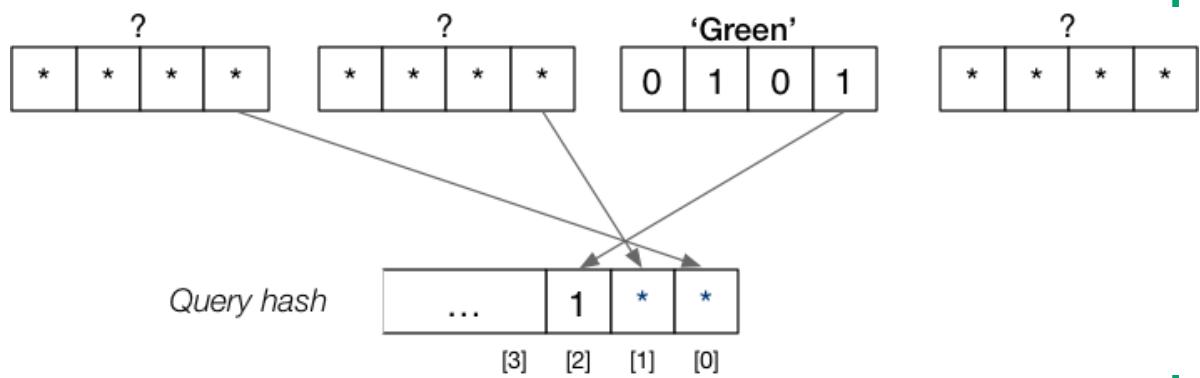
E.g.

```
select amount
from Deposit
where branch = 'Brighton' and name = 'Green'
```

for which we use the shorthand (**Brighton, ?, Green, ?**)

❖ Queries with MA.Hashing (cont)

Consider query: select amount from Deposit where name='Green'



Matching tuples must be in pages: **100**, **101**, **110**, **111**.

❖ MA.Hashing Query Algorithm

```
// Builds the partial hash value (e.g. 10*0*1)
// Treats query like tuple with some attr values missing

ns = 0; // # unknown bits = # stars
for each attribute i{
    if (hasValue(Q,i)) {
        set d[i] bits in composite hash
            using choice vector and hash(Q,i)
    } else {
        set d[i] *'s in composite hash
            using choice vector
        ns += d[i]
    }
}
...
...
```

❖ MA.Hashing Query Algorithm (cont)

```
...
// Use the partial hash to find candidate pages

r = openRelation("R",READ);
for (i = 0; i < 2ns; i++) {
    P = composite hash
    replace '*'s in P
        using ns bits in i
    Buf = getPage(fileOf(r), P);
    for each tuple T in Buf {
        if (T satisfies pmr query)
            add T to results
    }
}
```

❖ Query Cost for MA.Hashing

Multi-attribute hashing handles a range of query types,
e.g.

```
select * from R where a=1
select * from R where d=2
select * from R where b=3 and c=4
select * from R where a=5 and b=6 and c=7
```

A relation with n attributes has 2^n different query types.

Different query types have different costs (different no. of '*'s)

$$\text{Cost}(Q) = 2^s \text{ where } s = \sum_{i \notin Q} d_i \quad (\text{alternatively } \text{Cost}(Q) = \prod_{i \notin Q} 2^{d_i})$$

Query distribution gives probability p_Q of asking each query type Q .

❖ Query Cost for MA.Hashing (cont)

Min query cost occurs when all attributes are used in query

$$\text{Min Cost}_{pmr} = 1$$

Max query cost occurs when no attributes are specified

$$\text{Max Cost}_{pmr} = 2^d = b$$

Average cost is given by weighted sum over all query types:

$$\text{Avg Cost}_{pmr} = \sum_Q p_Q \prod_{i \notin Q} 2^{d_i}$$

Aim to minimise the weighted average query cost over possible query types

❖ Optimising MA.Hashing Cost

For a given application, useful to minimise $Cost_{pmr}$.

Can be achieved by choosing appropriate values for d_i (cv)

Heuristics:

- distribution of query types (more bits to frequently used attributes)
- size of attribute domain (\leq #bits to represent all values in domain)
- discriminatory power (more bits to highly discriminating attributes)

Trade-off: making Q_j more efficient makes Q_k less efficient.

This is a combinatorial optimisation problem
(solve via standard optimisation techniques e.g. simulated annealing)

Signature-based Indexing

- [Indexing with Signatures](#)
- [Signatures](#)
- [Generating Codewords](#)
- [Superimposed Codewords \(SIMC\)](#)
- [Concatenated Codewords \(CATC\)](#)
- [Queries using Signatures](#)
- [False Matches](#)
- [SIMC vs CATC](#)

❖ Indexing with Signatures

Signature-based indexing:

- designed for *pmr* queries (conjunction of equalities)
- does not try to achieve better than $O(n)$ performance
- attempts to provide an "efficient" linear scan

Each tuple is associated with a **signature**

- a compact (lossy) descriptor for the tuple
- formed by combining information from multiple attributes
- stored in a signature file, parallel to data file

Instead of scanning/testing tuples, do pre-filtering via signatures.

❖ Indexing with Signatures (cont)

File organisation for signature indexing (two files)

Signature File

[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	
[7]	
[8]	
...	

Data File

[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	
[7]	
[8]	
...	

One signature slot per tuple slot; unused signature slots are zeroed.

Signatures do not determine record placement ⇒ can use with other indexing.

❖ Signatures

A **signature** "summarises" the data from one tuple

A tuple consists of n attribute values $A_1 \dots A_n$

A **codeword** $cw(A_i)$ is

- a bit-string, m bits long, where k bits are set to 1 ($k \ll m$)
- derived from the value of a single attribute A_i

A **tuple descriptor** (signature) is built

- by combining $cw(A_i)$, $i=1..n$
- aim to have roughly half of the bits set to 1

Two strategies for building signatures: overlay, concatenate

❖ Generating Codewords

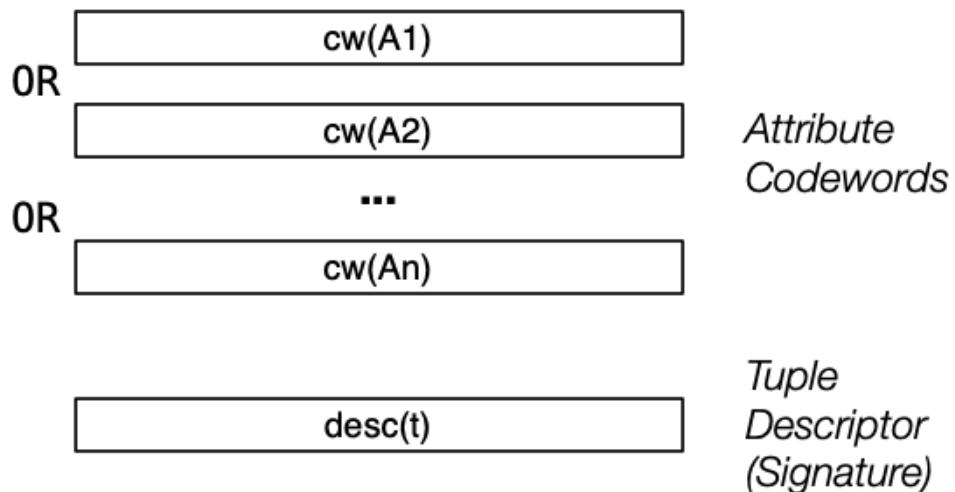
Generating a k -in- m codeword for attribute A_i

```
bits codeword(char *attr_value, int m, int k)
{
    int nbits = 0;      // count of set bits
    bits cword = 0;     // assuming m <= 32 bits
    srand(hash(attr_value));
    while (nbits < k) {
        int i = random() % m;
        if (((1 << i) & cword) == 0) {
            cword |= (1 << i);
            nbits++;
        }
    }
    return cword; // m-bits with k 1-bits and m-k 0-bits
}
```

❖ Superimposed Codewords (SIMC)

In a superimposed codewords (simc) indexing scheme

- tuple descriptor formed by *overlaying* attribute codewords (bitwise-OR)



❖ Superimposed Codewords (SIMC) (cont)

A SIMC tuple descriptor $desc(t)$ is

- a bit-string, m bits long, where $j \leq nk$ bits are set to 1
- $desc(t) = cw(A_1) \text{ OR } cw(A_2) \text{ OR } \dots \text{ OR } cw(A_n)$

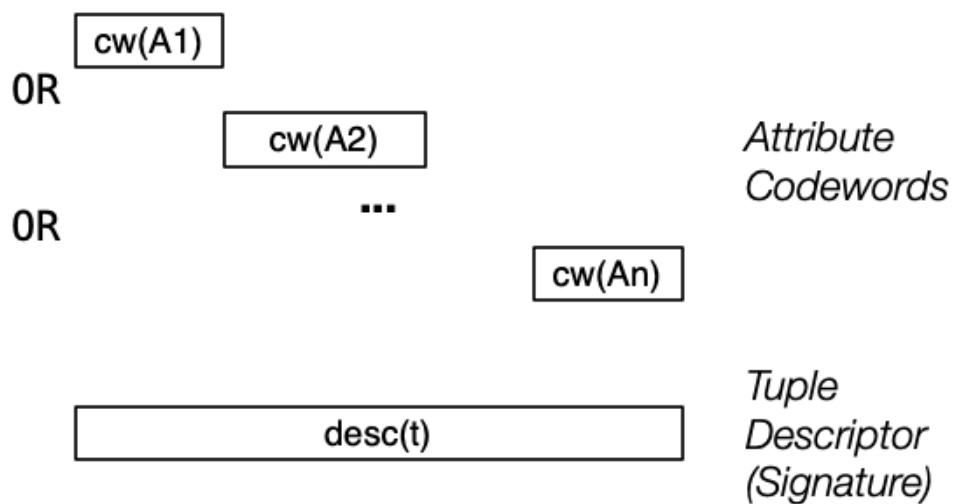
Method (assuming all n attributes are used in descriptor):

```
Bits desc = 0
for (i = 1; i <= n; i++) {
    bits cw = codeword(A[i], m, k)
    desc = desc | cw
}
```

❖ Concatenated Codewords (CATC)

In a concatenated codewords (catc) indexing schema

- tuple descriptor formed by *concatenating* attribute codewords



❖ Concatenated Codewords (CATC) (cont)

A CATC tuple descriptor $desc(t)$ is

- a bit-string, m bits long, where $j = nk$ bits are set to 1
- $desc(t) = cw(A_1) + cw(A_2) + \dots + cw(A_n)$ (+ is concatenation)

Each codeword is $p = m/n$ bits long, with $k = p/2$ bits set to 1

Method (assuming all n attributes are used in descriptor):

```
Bits desc = 0 ;  int p = m/n
for (i = 1; i <= n; i++) {
    bits cw = codeword(A[i],p,k)
    desc = desc | (cw << p*(n-i))
}
```

❖ Queries using Signatures

To answer query q with a signature-based index

- first generate a **query descriptor** $desc(q)$
- then scan the signature file using the query descriptor
- if sig_i matches $desc(q)$, then tuple i may be a match

$desc(q)$ is formed from codewords of known attributes.

Effectively, any unknown attribute A_i has $cw(A_i) = 0$

E.g. for SIMC $(a, ?, c, ?, e) = cw(a) \text{ OR } 0 \text{ OR } cw(c) \text{ OR } 0 \text{ OR } cw(e)$

❖ Queries using Signatures (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}
// scan r descriptors
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
// scan bq + δ data pages
for each pid in pagesToCheck {
    Buf = getPage(dataFile,pid)
    check tuples in Buf for answers
}
```

❖ False Matches

Both SIMC and CATC can produce **false matches**

- **matches($D[i]$, $desc(q)$)** is true, but **$Tup[i]$** is not a solution for q

Why does this happen?

- signatures are based on hashing, and it is possible that

$\text{hash(key}_1\text{)} == \text{hash(key}_2\text{)}$ even though $\text{key}_1 \neq \text{key}_2$

- for SIMC, overlaying could also produce "unfortunate" bit-combinations

To mitigate this, need to choose "good" m and k

❖ SIMC vs CATC

Both build m -bit wide signatures, with $\sim 1/2$ bits set to 1

Both have codewords with $\sim m/2n$ bits set to 1

CATC: codewords are $m/n = p$ -bits wide

- shorter codewords \rightarrow more hash collisions

SIMC: codewords are also m -bits wide

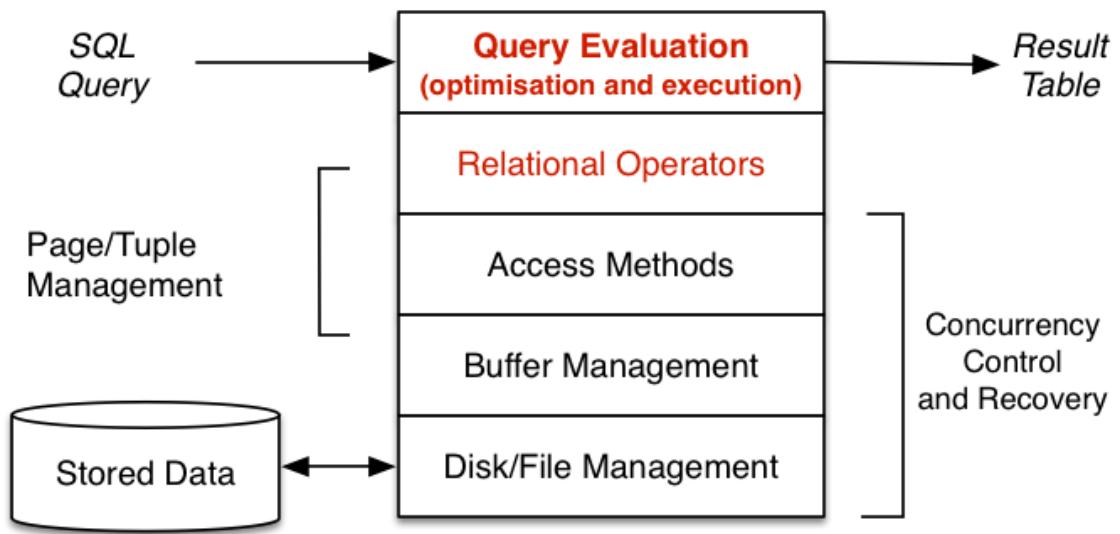
- longer codewords \Rightarrow less hash collisions, but also has overlay collisions

CATC has option of different length codeword p_i for each A_i ($\sum p_i = m$)

Query Processing

- [Query Processing](#)
- [Terminology Variations](#)

❖ Query Processing



❖ Query Processing (cont)

A **query** in SQL:

- states *what* kind of answers are required (declarative)
- does not say *how* they should be computed (procedural)

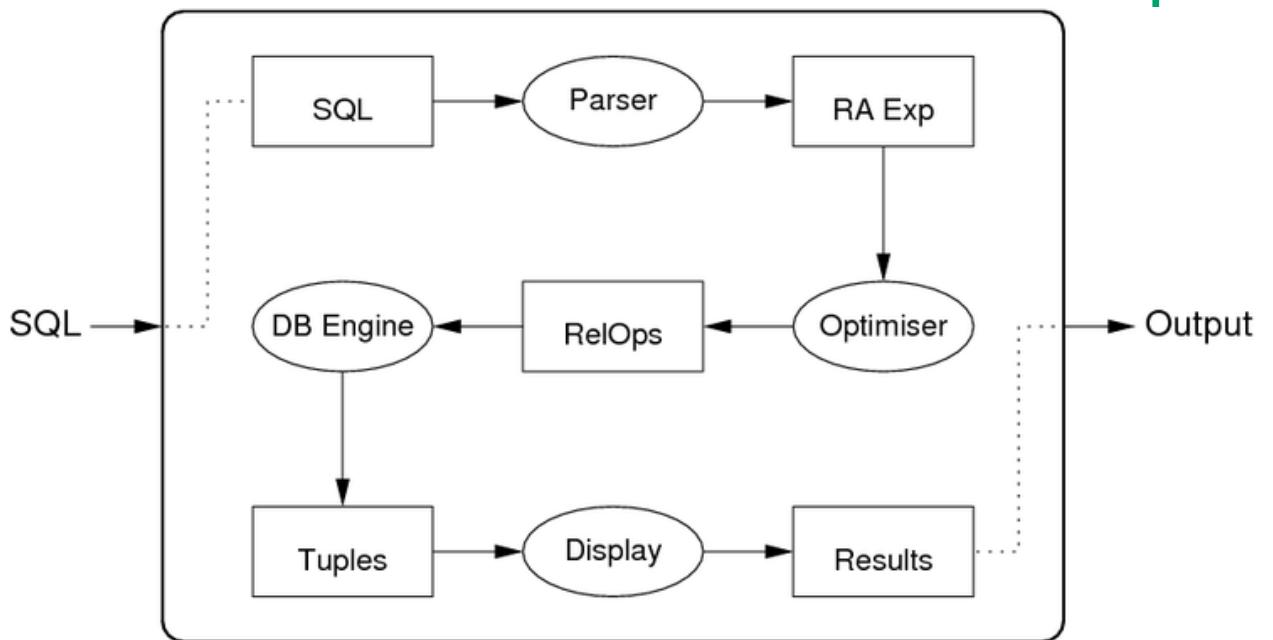
A **query evaluator/processor** :

- takes declarative description of query (in SQL)
- parses query to internal representation (relational algebra)
- determines plan for answering query (expressed as DBMS ops)
- executes method via DBMS engine (to produce result tuples)

Some DBMSs can save query plans for later re-use.

❖ Query Processing (cont)

Internals of the query evaluation "black-box":



❖ Query Processing (cont)

DBMSs provide several "flavours" of each RA operation.

For example:

- several "versions" of selection (σ) are available
- each version is effective for a particular kind of selection, e.g

```
select * from R where id = 100    -- hashing
select * from S                      -- Btree index
where age > 18 and age < 35
select * from T                      -- MATH file
where a = 1 and b = 'a' and c = 1.4
```

Similarly, π and \bowtie have versions to match specific query types.

❖ Query Processing (cont)

We call these specialised version of RA operations **RelOps**.

One major task of the query processor:

- given a RA expression to be evaluated
- find a combination of RelOps to do this efficiently

Requires the query translator/optimiser to consider

- information about relations (e.g. sizes, primary keys, ...)
- information about operations (e.g. selection reduces size)

RelOps are realised at execution time

- as a collection of inter-communicating **nodes**
- communicating either via pipelines or temporary relations

❖ Terminology Variations

Relational algebra expression of SQL query

- intermediate query representation
- logical query plan

Execution plan as collection of RelOps

- query evaluation plan
- query execution plan
- physical query plan

Representation of RA operators and expressions

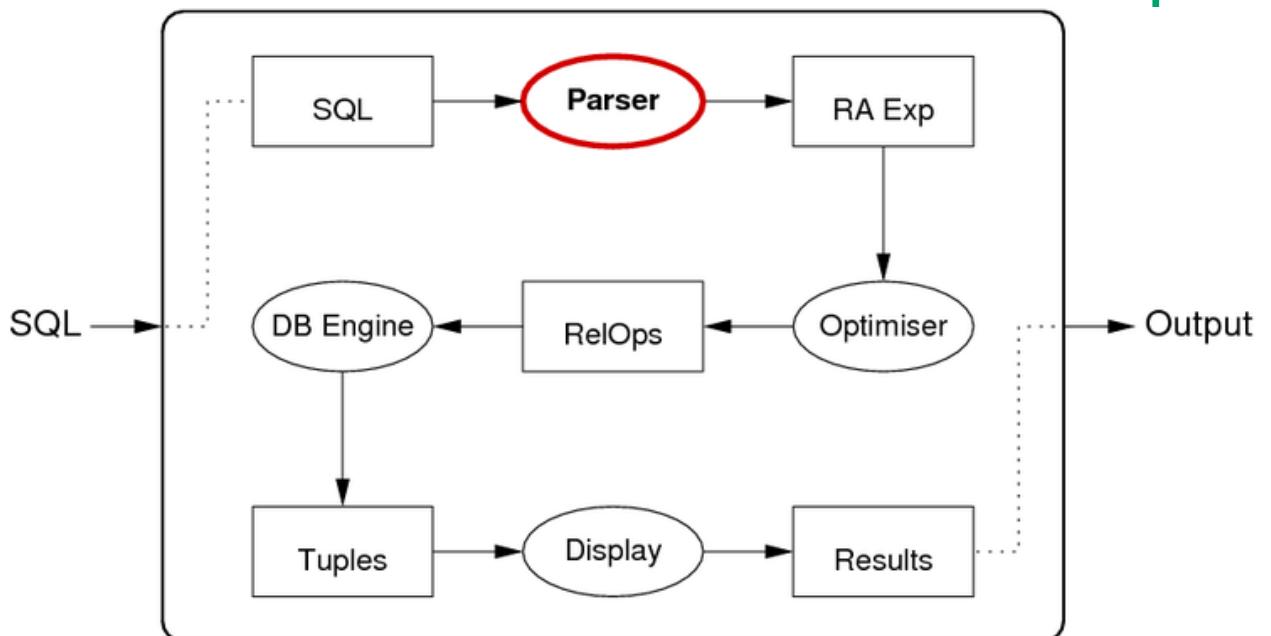
- $\sigma = \text{Select} = \text{Sel}$, $\pi = \text{Project} = \text{Proj}$
- $R \bowtie S = R \text{ Join } S = \text{Join}(R, S)$, $\wedge = \&$, $\vee = |$

Query Translation

- [Query Translation](#)
- [Parsing SQL](#)
- [Expression Rewriting Rules](#)
- [Relational Algebra Laws](#)
- [Query Rewriting](#)

❖ Query Translation

Query translation: SQL statement text → RA expression



❖ Query Translation (cont)

Translation step: SQL text → RA expression

Example:

SQL: select name from Students where id=7654321;
-- is translated to
RA: Proj[name](Sel[id=7654321]Students)

Processes: lexer/parser, mapping rules, rewriting rules.

Mapping from SQL to RA may include some optimisations, e.g.

select * from Students where id = 54321 and age > 50;
-- is translated to
Sel[age>50](Sel[id=54321]Students)
-- rather than ... because of index on id
Sel[id=54321&age>50](Students)

❖ Parsing SQL

Parsing task is similar to that for programming languages.

Language elements:

- keywords: `create`, `select`, `from`, `where`, ...
- identifiers: `Students`, `name`, `id`, `CourseCode`, ...
- operators: `+`, `-`, `=`, `<`, `>`, `AND`, `OR`, `NOT`, `IN`, ...
- constants: `'abc'`, `123`, `3.1`, `'01-jan-1970'`, ...

PostgreSQL parser ...

- implemented via lex/yacc ([src/backend/parser](#))
- maps all identifiers to lower-case (A-Z → a-z)
- needs to handle user-extendable operator set
- makes extensive use of catalog ([src/backend/catalog](#))

❖ Expression Rewriting Rules

Since RA is a well-defined formal system

- there exist many algebraic laws on RA expressions
- which can be used as a basis for expression rewriting
- in order to produce **equivalent** (more-efficient?) expressions

Expression transformation based on such rules can be used

- to simplify/improve SQL→RA mapping results
- to generate new plan variations to check in query optimisation

❖ Relational Algebra Laws

Commutative and Associative Laws:

- $R \bowtie S \leftrightarrow S \bowtie R, \quad (R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$
(natural join)
- $R \cup S \leftrightarrow S \cup R, \quad (R \cup S) \cup T \leftrightarrow R \cup (S \cup T)$
- $R \bowtie_{Cond} S \leftrightarrow S \bowtie_{Cond} R \quad (\text{theta join})$
- $\sigma_c(\sigma_d(R)) \leftrightarrow \sigma_d(\sigma_c(R))$

Selection splitting (where c and d are conditions):

- $\sigma_{c \wedge d}(R) \leftrightarrow \sigma_c(\sigma_d(R))$
- $\sigma_{c \vee d}(R) \leftrightarrow \sigma_c(R) \cup \sigma_d(R)$

❖ Relational Algebra Laws (cont)

Selection pushing ($\sigma_c(R \cup S)$ and $\sigma_c(R \cap S)$):

- $\sigma_c(R \cup S) \leftrightarrow \sigma_c R \cup \sigma_c S, \quad \sigma_c(R \cap S) \leftrightarrow \sigma_c R \cap \sigma_c S$

Selection pushing with join ...

- $\sigma_c(R \bowtie S) \leftrightarrow \sigma_c(R) \bowtie S$ (if c refers only to attributes from R)
- $\sigma_c(R \bowtie S) \leftrightarrow R \bowtie \sigma_c(S)$ (if c refers only to attributes from S)

If *condition* contains attributes from both R and S :

- $\sigma_{c' \wedge c''}(R \bowtie S) \leftrightarrow \sigma_{c'}(R) \bowtie \sigma_{c''}(S)$
- c' contains only R attributes, c'' contains only S attributes

❖ Relational Algebra Laws (cont)

Rewrite rules for projection ...

All but last projection can be ignored:

- $\Pi_{L_1} (\Pi_{L_2} (\dots \Pi_{L_n} (R))) \rightarrow \Pi_{L_1} (R)$

Projections can be pushed into joins:

- $\Pi_L (R \bowtie_c S) \leftrightarrow \Pi_L (\Pi_M(R) \bowtie_c \Pi_N(S))$

where

- M and N must contain all attributes needed for c
- M and N must contain all attributes used in L ($L \subset M \cup N$)

❖ Query Rewriting

Subqueries \Rightarrow convert to a join

Example: (on schema Courses(id,code,...),
Enrolments(cid,sid,...), Students(id,name,...))

```
select c.code, count(*)
from Courses c
where c.id in (select cid from Enrolments)
group by c.code
```

becomes

```
select c.code, count(*)
from Courses c join Enrolments e on c.id = e.cid
group by c.code
```

❖ Query Rewriting (cont)

But not all subqueries can be converted to join, e.g.

```
select e.sid as student_id, e.cid as course_id  
from Enrolments e  
where e.sid = (select max(id) from Students)
```

has to be evaluated as

$Val = \max[id]Students$

$Res = \Pi_{(sid,cid)}(\sigma_{sid=Val}Enrolments)$

❖ Query Rewriting (cont)

In PostgreSQL, views are implemented via rewrite rules

- a reference to view in SQL expands to its definition in RA

Example:

```
create view COMP9315studes as
select stu,mark from Enrolments where course='COMP9315';
-- students who passed
select stu from COMP9315studes where mark >= 50;
```

is represented in RA by

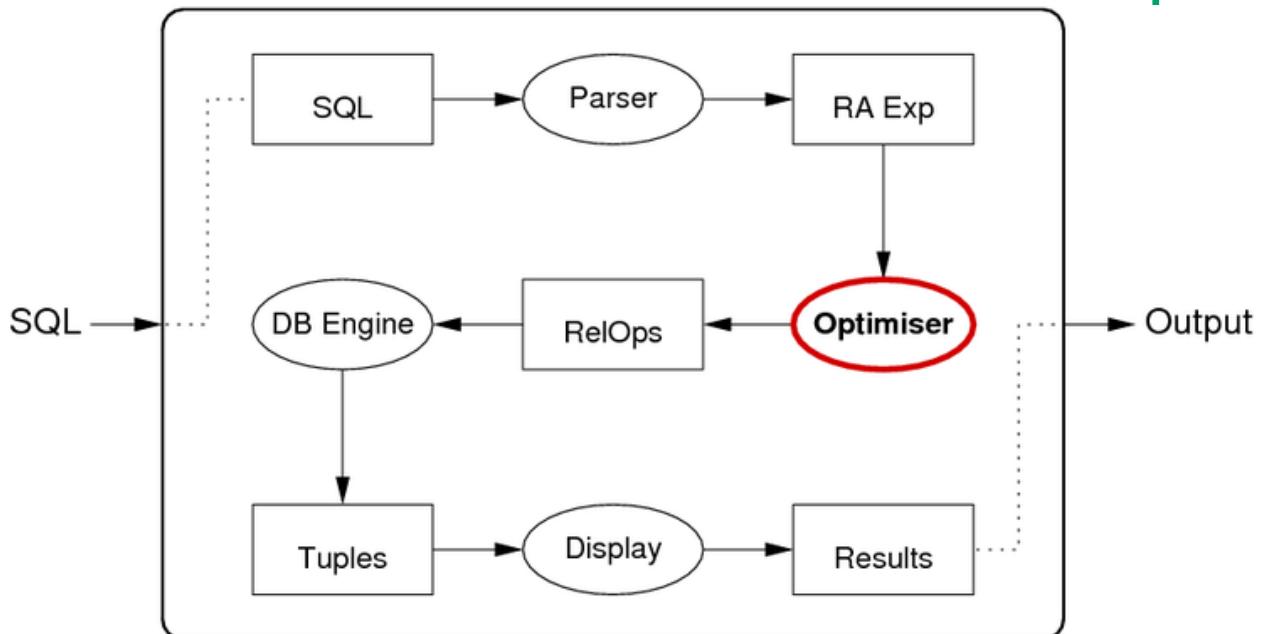
```
COMP9315studes
  = Proj[stu,mark](Sel[course=COMP9315](Enrolments))
  -- with query ...
Proj[stu](Sel[mark>=50](COMP9315studes))
  -- becomes ...
Proj[stu](Sel[mark>=50](
  Proj[stu,mark](Sel[course=COMP9315](Enrolments))))
  -- which could be rewritten as ...
Proj[stu](Sel[mark>=50 & course=COMP9315]Enrolments)
```

Query Optimisation

- [Query Optimisation](#)
- [Approaches to Optimisation](#)
- [Cost-based Query Optimiser](#)
- [Cost Models and Analysis](#)
- [Choosing Access Methods \(RelOps\)](#)
- [PostgreSQL Query Optimization](#)

❖ Query Optimisation

Query optimiser: RA expression → efficient evaluation plan



❖ Query Optimisation (cont)

Query optimisation is a critical step in query evaluation.

The query optimiser

- takes relational algebra expression from SQL compiler
- produces sequence of RelOps to evaluate the expression
- **query execution plan** should provide efficient evaluation

"Optimisation" is a misnomer since query optimisers

- aim to find a good plan ... but maybe not optimal

Observed Query Time = Planning time + Evaluation time

❖ Query Optimisation (cont)

Why do we not generate optimal query execution plans?

Finding an optimal query plan ...

- requires exhaustive search of a **space of possible plans**
- for each possible plan, need to estimate cost (not cheap)

Even for relatively small query, search space is *very large*.

Compromise:

- do limited search of query plan space (guided by heuristics)
- *quickly choose a reasonably efficient execution plan*

❖ Approaches to Optimisation

Three main classes of techniques developed:

- algebraic (equivalences, rewriting, heuristics)
- physical (execution costs, search-based)
- semantic (application properties, heuristics)

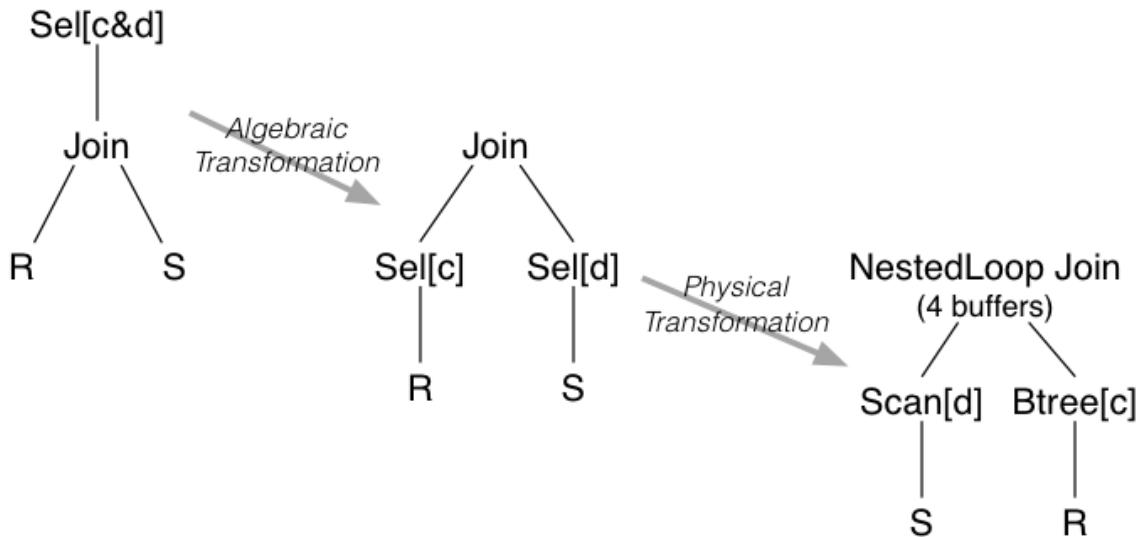
All driven by aim of minimising (or at least reducing) "cost".

Real query optimisers use a combination of algebraic+physical.

Semantic QO is good idea, but expensive/difficult to implement.

❖ Approaches to Optimisation (cont)

Example of optimisation transformations:



For join, may also consider sort/merge join and hash join.

❖ Cost-based Query Optimiser

Approximate algorithm for cost-based optimisation:

```
translate SQL query to RAexp
for enough transformations RA' of RAexp {
    while (more choices for RelOps) {
        Plan = {}; i = 0; cost = 0
        for each node e of RA' (recursively) {
            ROp = select RelOp method for e
            Plan = Plan ∪ ROp
            cost += Cost(ROp) // using child info
        }
        if (cost < MinCost)
            { MinCost = cost; BestPlan = Plan }
    }
}
```

Heuristics: push selections down, consider only left-deep join trees.

❖ Cost Models and Analysis

The cost of evaluating a query is determined by:

- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves *estimating*:

- size of intermediate results
- number of disk reads/writes

❖ Choosing Access Methods (RelOps)

Performed for each node in RA expression tree ...

Inputs:

- a single RA operation (σ , π , \bowtie)
- information about file organisation, data distribution,
...
- list of operations available in the database engine

Output:

- specific DBMS operation to implement this RA operation

❖ Choosing Access Methods (RelOps) (cont)

Example:

- RA operation: $\text{Sel}_{[\text{name}='John' \wedge \text{age}>21]}(\text{Student})$
 - **Student** relation has B-tree index on **name**
 - database engine (obviously) has B-tree search method
- giving

```
tmp[i] := BtreeSearch[name='John'](Student)
tmp[i+1] := LinearSearch[age>21](tmp[i])
```

Where possible, use pipelining to avoid storing **tmp[i]** on disk.

❖ Choosing Access Methods (RelOps) (cont)

Rules for choosing σ access methods:

- $\sigma_{A=c}(R)$ and R has index on A \Rightarrow
indexSearch[A=c](R)
- $\sigma_{A=c}(R)$ and R is hashed on A \Rightarrow
hashSearch[A=c](R)
- $\sigma_{A=c}(R)$ and R is sorted on A \Rightarrow
binarySearch[A=c](R)
- $\sigma_{A \geq c}(R)$ and R has clustered index on A
 \Rightarrow **indexSearch[A=c](R)** then
scan
- $\sigma_{A \geq c}(R)$ and R is hashed on A
 \Rightarrow **linearSearch[A>=c](R)**

❖ Choosing Access Methods (RelOps) (cont)

Rules for choosing \bowtie access methods:

- $R \bowtie S$ and R fits in memory buffers \Rightarrow
 $\text{bnlJoin}(R, S)$
- $R \bowtie S$ and S fits in memory buffers \Rightarrow
 $\text{bnlJoin}(S, R)$
- $R \bowtie S$ and R, S sorted on join attr \Rightarrow **$\text{smJoin}(R, S)$**
- $R \bowtie S$ and R has index on join attr \Rightarrow
 $\text{inlJoin}(S, R)$
- $R \bowtie S$ and no indexes, no sorting \Rightarrow
 $\text{hashJoin}(R, S)$

(**bnl** = block nested loop; **inl** = index nested loop; **sm** = sort merge)

❖ PostgreSQL Query Optimization

Input: tree of **Query** nodes returned by parser

Output: tree of **Plan** nodes used by query **executor**

- wrapped in a **PlannedStmt** node containing state info

Intermediate data structures are trees of **Path** nodes

- a path tree represents one evaluation order for a query

All **Node** types are defined in **include/nodes/*.h**

❖ PostgreSQL Query Optimization (cont)

Query optimisation proceeds in two stages (after parsing)...

Rewriting:

- uses PostgreSQL's **rule** system
- query tree is expanded to include e.g. view definitions

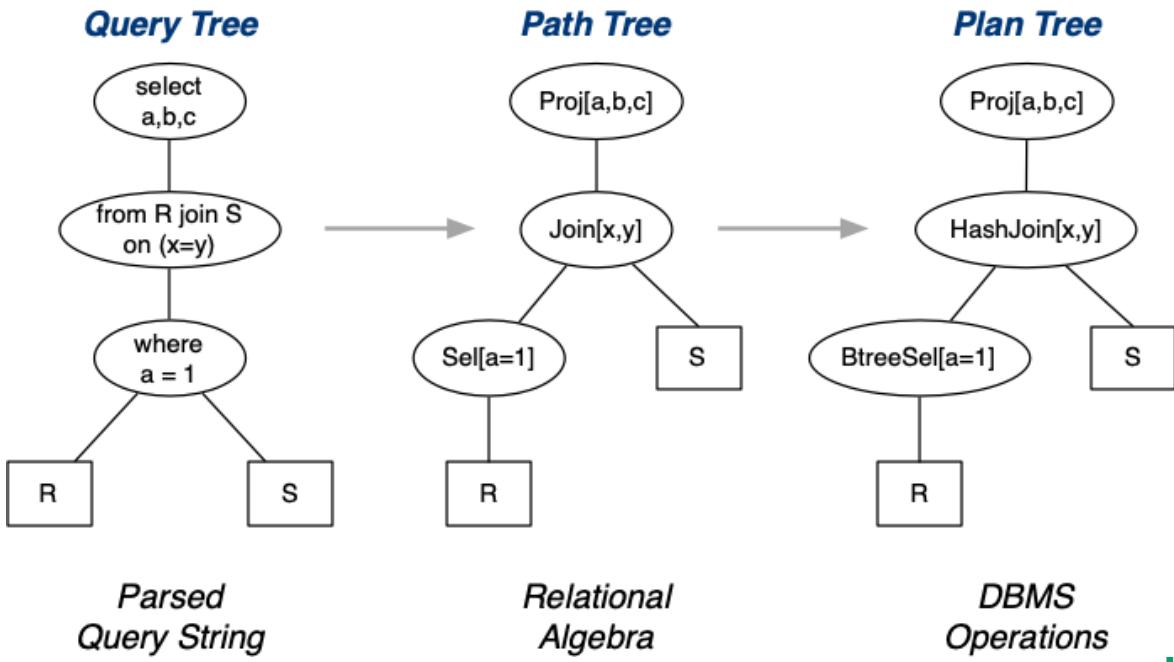
Planning and optimisation:

- using cost-based analysis of generated paths
- via one of **two** different path generators
- chooses least-cost path from all those considered

Then produces a **Plan** tree from the selected path.

❖ PostgreSQL Query Optimization (cont)

```
select a,b,c from R join S on (x=y) where a = 1
```



Query Cost Estimation

- [Query Cost Estimation](#)
- [Estimating Projection Result Size](#)
- [Estimating Selection Result Size](#)
- [Estimating Join Result Size](#)
- [Cost Estimation: Postscript](#)

❖ Query Cost Estimation

Without executing a plan, cannot always know its precise cost.

Thus, query optimisers **estimate** costs via:

- cost of performing operation (dealt with in earlier lectures)
- size of result (which affects cost of performing next operation)

Result size estimated by statistical measures on relations, e.g.

r_S cardinality of relation S

R_S avg size of tuple in relation S

$V(A, S)$ # distinct values of attribute A

$\min(A, S)$ min value of attribute A

$\max(A, S)$ max value of attribute A

❖ Estimating Projection Result Size

Straightforward, since we know:

- number of tuples in output

$$r_{out} = | \Pi_{a,b,\dots}(T) | = | T | = r_T \quad (\text{in SQL, because of bag semantics})$$

- size of tuples in output

$$R_{out} = \text{sizeof}(a) + \text{sizeof}(b) + \dots + \text{tuple-overhead}$$

Assume page size B , $b_{out} = \lceil r_T / c_{out} \rceil$, where $c_{out} = \lfloor B/R_{out} \rfloor$

If using **select distinct** ...

- $|\Pi_{a,b,\dots}(T)|$ depends on proportion of duplicates produced

❖ Estimating Selection Result Size

Selectivity = fraction of tuples expected to satisfy a condition.

Common assumption: attribute values uniformly distributed.

Example: Consider the query

```
select * from Parts where colour='Red'
```

If $V(\text{colour}, \text{Parts})=4$, $r=1000 \Rightarrow |$

$$|\sigma_{\text{colour}=\text{red}}(\text{Parts})| = 250$$

In general, $|\sigma_{A=c}(R)| \approx r_R / V(A, R)$

Heuristic used by PostgreSQL: $|\sigma_{A=c}(R)| \approx r/10$

❖ Estimating Selection Result Size (cont)

Estimating size of result for e.g.

```
select * from Enrolment where year > 2015;
```

Could estimate by using:

- uniform distribution assumption, r , min/max years

Assume: $\min(\text{year})=2010$, $\max(\text{year})=2019$,
 $|Enrolment|=10^5$

- 10^5 from 2010-2019 means approx 10000 enrolments/year
- this suggests 40000 enrolments since 2016

Heuristic used by some systems: $|\sigma_{A>c}(R)| \cong r/3$

❖ Estimating Selection Result Size (cont)

Estimating size of result for e.g.

```
select * from Enrolment where course <> 'COMP9315';
```

Could estimate by using:

- uniform distribution assumption, r , domain size

e.g. $|V(\text{course}, \text{Enrolment})| = 2000$, $|\sigma_{A <> c}(E)| = r * 1999/2000$

Heuristic used by some systems: $|\sigma_{A <> c}(R)| \cong r$

❖ Estimating Selection Result Size (cont)

How to handle non-uniform attribute value distributions?

- collect statistics about the values stored in the attribute/relation
- store these as e.g. a histogram in the meta-data for the relation

So, for part colour example, might have distribution like:

White: 35% **Red**: 30% **Blue**: 25% **Silver**: 10%

Use histogram as basis for determining # selected tuples.

Disadvantage: cost of storing/maintaining histograms.

❖ Estimating Selection Result Size (cont)

Summary: analysis relies on operation and data distribution:

E.g. `select * from R where a = k;`

Case 1: $\text{uniq}(R.a) \Rightarrow$ 0 or 1 result

Case 2: r_R tuples $\&\& \text{size}(\text{dom}(R.a)) = n \Rightarrow r_R / n$ results

E.g. `select * from R where a < k;`

Case 1: $k \leq \min(R.a) \Rightarrow$ 0 results

Case 2: $k > \max(R.a) \Rightarrow \approx r_R$ results

Case 3: $\text{size}(\text{dom}(R.a)) = n \Rightarrow ? \min(R.a) \dots k \dots \max(R.a) ?$

❖ Estimating Join Result Size

Analysis relies on semantic knowledge about data/relations.

Consider equijoin on common attr: $R \bowtie_a S$

Case 1: $\text{values}(R.a) \cap \text{values}(S.a) = \emptyset \Rightarrow \text{size}(R \bowtie_a S) = 0$

Case 2: $\text{uniq}(R.a)$ and $\text{uniq}(S.a) \Rightarrow \text{size}(R \bowtie_a S) \leq \min(|R|, |S|)$

Case 3: $\text{pkey}(R.a)$ and $\text{fkey}(S.a) \Rightarrow \text{size}(R \bowtie_a S) \leq |S|$

❖ Cost Estimation: Postscript

Inaccurate cost estimation can lead to poor evaluation plans.

Above methods can (sometimes) give inaccurate estimates.

To get more accurate cost estimates:

- more time ... complex computation of selectivity
- more space ... storage for histograms of data values

Either way, optimisation process costs more (more than query?)

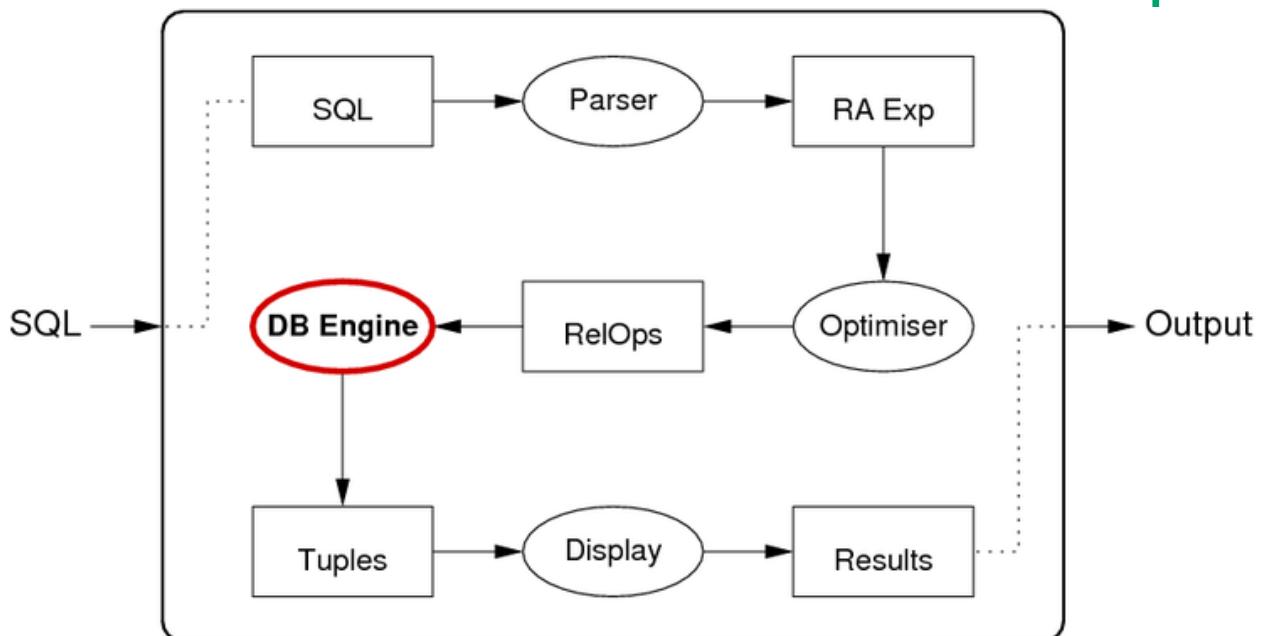
Trade-off between optimiser performance and query performance.

Query Execution

- [Query Execution](#)
- [Materialization](#)
- [Pipelining](#)
- [Iterators \(reminder\)](#)
- [Pipelining Example](#)
- [Disk Accesses](#)
- [PostgreSQL Query Execution](#)
- [PostgreSQL Executor](#)
- [Example PostgreSQL Execution](#)

❖ Query Execution

Query execution: applies evaluation plan → result tuples



❖ Query Execution (cont)

Example of query translation:

```
select s.name, s.id, e.course, e.mark  
from Student s, Enrolment e  
where e.student = s.id and e.semester = '05s2';
```

maps to

$$\begin{aligned} &\Pi_{name,id,course,mark}(Stu \bowtie_{e.student=s.id} \\ &(\sigma_{semester=05s2}Enr)) \end{aligned}$$

maps to

```
Temp1 = BtreeSelect[semester=05s2](Enr)  
Temp2 = HashJoin[e.student=s.id](Stu,Temp1)  
Result = Project[name,id,course,mark](Temp2)
```

❖ Query Execution (cont)

A query execution plan:

- consists of a **collection of RelOps**
- executing together to produce a set of result tuples

Results may be passed from one operator to the next:

- **materialization** ... writing results to disk and reading them back
- **pipelining** ... generating and passing via memory buffers

❖ Materialization

Steps in **materialization** between two operators

- first operator reads input(s) and writes results to disk
- next operator treats tuples on disk as its input
- in essence, the **Temp** tables are produced as real tables

Advantage:

- intermediate results can be placed in a file structure (which can be chosen to speed up execution of subsequent operators)

Disadvantage:

- requires disk space/writes for intermediate results
- requires disk access to read intermediate results

❖ Pipelining

How **pipelining** is organised between two operators:

- operators execute "concurrently" as producer/consumer pairs
- structured as interacting iterators (open; while(next); close)

Advantage:

- no requirement for disk access (results passed via memory buffers)

Disadvantage:

- higher-level operators access inputs via linear scan, or
- requires sufficient memory buffers to hold all outputs

❖ Iterators (reminder)

Iterators provide a "stream" of results:

- **iter = startScan(*params*)**
 - set up data structures for iterator (create state, open files, ...)
 - *params* are specific to operator (e.g. reln, condition, #buffers, ...)
- **tuple = nextTuple(iter)**
 - get the next tuple in the iteration; return null if no more
- **endScan(iter)**
 - clean up data structures for iterator

Other possible operations: reset to specific point, restart,

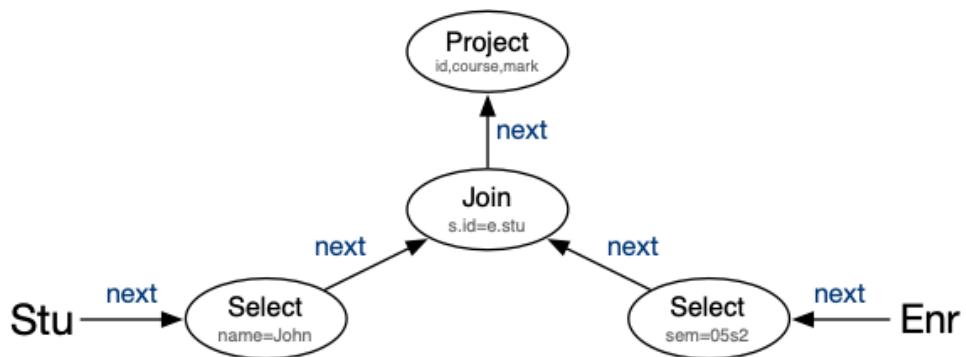
...

❖ Pipelining Example

Consider the query:

```
select s.id, e.course, e.mark
from Student s, Enrolment e
where e.student = s.id and
      e.semester = '05s2' and s.name = 'John';
```

Evaluated via communication between RA tree nodes:



Note: likely that projection is combined with join in PostgreSQL

❖ Disk Accesses

Pipelining cannot avoid all disk accesses.

Some operations use multiple passes (e.g. merge-sort, hash-join).

- data is written by one pass, read by subsequent passes

Thus ...

- **within** an operation, disk reads/writes are possible
- **between** operations, no disk reads/writes are needed

❖ PostgreSQL Query Execution

Defs: `src/include/executor` and
`src/include/nodes`

Code: `src/backend/executor`

PostgreSQL uses pipelining (as much as possible) ...

- query plan is a tree of `Plan` nodes
- each type of node implements one kind of RA operation
(node implements specific access method via iterator interface)
- node types e.g. `Scan`, `Group`, `Indexscan`, `Sort`, `HashJoin`
- execution is managed via a tree of `PlanState` nodes
(mirrors the structure of the tree of Plan nodes; holds execution state)

❖ PostgreSQL Executor

Modules in `src/backend/executor` fall into two groups:

execXXX (e.g. `execMain`, `execProcnode`, `execScan`)

- implement generic control of plan evaluation (execution)
- provide overall plan execution and dispatch to node iterators

nodeXXX (e.g. `nodeSeqscan`, `nodeNestloop`, `nodeGroup`)

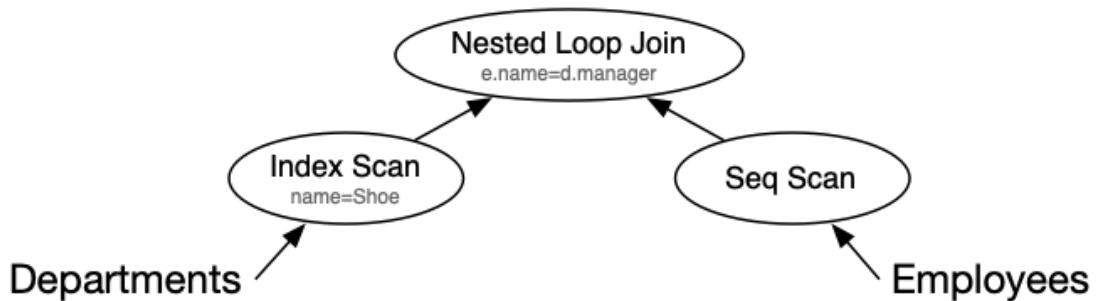
- implement iterators for specific types of RA operators
- typically contains **ExecInitXXX**, **ExecXXX**, **ExecEndXXX**

❖ Example PostgreSQL Execution

Consider the query:

```
-- get manager's age and # employees in Shoe department
select e.age, d.nemps
from Departments d, Employees e
where e.name = d.manager and d.name = 'Shoe'
```

and its execution plan tree



❖ Example PostgreSQL Execution (cont)

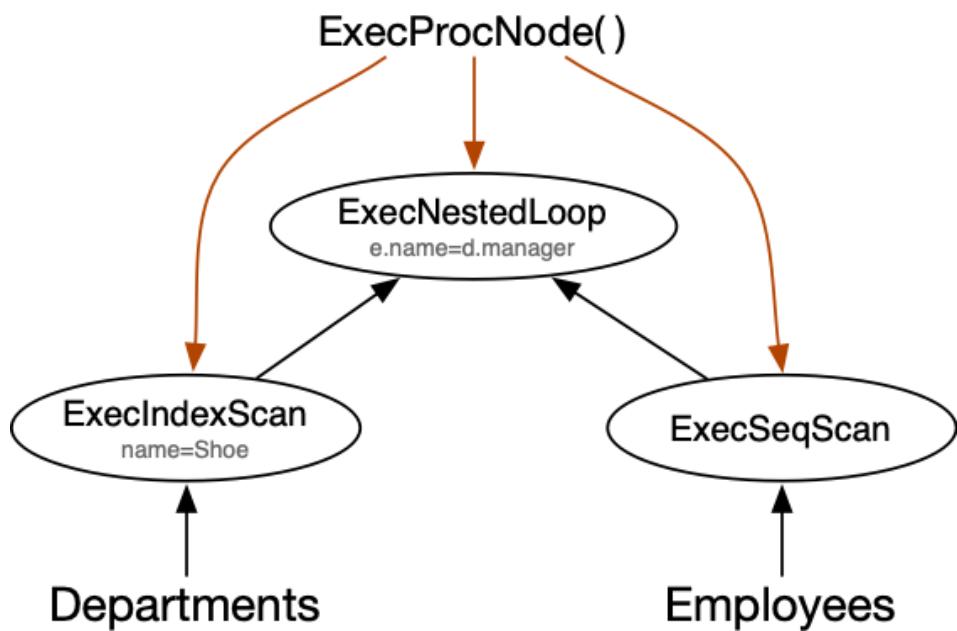
Initially **InitPlan()** invokes **ExecInitNode()** on plan tree root.

ExecInitNode() sees a **NestedLoop** node ...
so dispatches to **ExecInitNestLoop()** to set up iterator
then invokes **ExecInitNode()** on left and right sub-plans
in left subPlan, **ExecInitNode()** sees an **IndexScan** node
so dispatches to **ExecInitIndexScan()** to set up iterator
in right sub-plan, **ExecInitNode()** sees a **SeqScan** node
so dispatches to **ExecInitSeqScan()** to set up iterator

Result: a plan *state* tree with same structure as plan tree.

❖ Example PostgreSQL Execution (cont)

Plan state tree (collection of iterators):



❖ Example PostgreSQL Execution (cont)

Then **ExecutePlan()** repeatedly invokes **ExecProcNode()**.

ExecProcNode() sees a **NestedLoop** node ...
so dispatches to **ExecNestedLoop()** to get next tuple
which invokes **ExecProcNode()** on its sub-plans
in left sub-plan, **ExecProcNode()** sees an
IndexScan node
so dispatches to **ExecIndexScan()** to get next
tuple
if no more tuples, return END
for this tuple, invoke **ExecProcNode()** on right
sub-plan
ExecProcNode() sees a **SeqScan** node
so dispatches to **ExecSeqScan()** to get
next tuple
check for match and return joined tuples if
found
continue scan until end
reset right sub-plan iterator

Result: stream of result tuples returned via
ExecutePlan()

Query Performance Tuning

- [Query Performance Tuning](#)
- [PostgreSQL Query Tuning](#)
- [EXPLAIN examples](#)

❖ Query Performance Tuning

What to do if the DBMS takes "too long" to answer some queries?

Improving performance may involve any/all of:

- making applications using the DB run faster
- lowering response time of queries/transactions
- improving overall transaction throughput

Remembering that, to some extent ...

- the query optimiser removes choices from DB developers
- by making its own decision on the optimal execution plan

❖ Query Performance Tuning (cont)

Tuning requires us to consider the following:

- which queries and transactions will be used?
(e.g. check balance for payment, display recent transaction history)
- how frequently does each query/transaction occur?
(e.g. 80% withdrawals; 1% deposits; 19% balance check)
- are there time constraints on queries/transactions?
(e.g. EFTPOS payments must be approved within 7 seconds)
- are there uniqueness constraints on any attributes?
(define indexes on attributes to speed up insertion uniqueness check)
- how frequently do updates occur?
(indexes slow down updates, because must update table *and* index)

❖ Query Performance Tuning (cont)

Performance can be considered at two times:

- *during* schema design
 - typically towards the end of schema design process
 - requires schema transformations such as **denormalisation**
- *outside* schema design
 - typically after application has been deployed/used
 - requires adding/modifying data structures such as **indexes**

Difficult to predict what query optimiser will do, so ...

- implement queries using methods which *should* be efficient
- observe execution behaviour and modify query accordingly

❖ PostgreSQL Query Tuning

PostgreSQL provides the **explain** statement to

- give a representation of the query execution plan
- with information that may help to tune query performance

Usage:

```
EXPLAIN [ANALYZE] Query
```

Without **ANALYZE**, **EXPLAIN** shows plan with estimated costs.

With **ANALYZE**, **EXPLAIN** executes query and prints real costs.

Note that runtimes may show considerable variation due to buffering.

❖ EXPLAIN examples

Using the following database ...

```
CourseEnrolments(student, course, mark, grade, ...)
Courses(id, subject, semester, homepage)
People(id, family, given, title, name, ..., birthday)
ProgramEnrolments(id, student, semester, program, wam, ...)
Students(id, stype)
Subjects(id, code, name, longname, uoc, offeredby, ...)
```

with a view defined as

```
create view EnrolmentCounts as
select s.code, c.semester, count(e.student) as nstudes
  from Courses c join Subjects s on c.subject=s.id
    join Course_enrolments e on e.course = c.id
 group by s.code, c.semester;
```

❖ EXPLAIN examples (cont)

Some database statistics:

tab_name	n_records
courseenrolments	503120
courses	71288
people	36497
programenrolments	161110
students	31048
subjects	18799

❖ EXPLAIN examples (cont)

Example: Select on non-indexed attribute

```
uni=# explain
uni=# select * from Students where stype='local';
          QUERY PLAN
-----
Seq Scan on students
(cost=0.00..562.01 rows=23544 width=9)
  Filter: ((stype)::text = 'local'::text)
```

where

- **Seq Scan** = operation (plan node)
- **cost**=*StartUpCost..TotalCost*
- **rows**=*NumberOfResultTuples*
- **width**=*SizeOfTuple* (# bytes)

❖ EXPLAIN examples (cont)

More notes on **explain** output:

- each major entry corresponds to a plan node
 - e.g. **Seq Scan**, **Index Scan**, **Hash Join**, **Merge Join**, ...
- some nodes include additional qualifying information
 - e.g. **Filter**, **Index Cond**, **Hash Cond**, **Buckets**, ...
- **cost** values in **explain** are estimates (notional units)
- **explain analyze** also includes actual time costs (ms)
- costs of parent nodes include costs of all children
- estimates of #results based on sample of data

❖ EXPLAIN examples (cont)

Example: Select on non-indexed attribute with actual costs

```
uni=# explain analyze
uni=# select * from Students where stype='local';
          QUERY PLAN
-----
Seq Scan on students
  (cost=0.00..562.01 rows=23544 width=9)
    (actual time=0.052..5.792 rows=23551 loops=1)
  Filter: ((stype)::text = 'local'::text)
  Rows Removed by Filter: 7810
Planning time: 0.075 ms
Execution time: 6.978 ms
```

❖ EXPLAIN examples (cont)

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=100250;
                                QUERY PLAN
-----
Index Scan using student_pkey on student
  (cost=0.00..8.27 rows=1 width=9)
    (actual time=0.049..0.049 rows=0 loops=1)
  Index Cond: (id = 100250)
Planning Time: 0.274 ms
Execution Time: 0.109 ms
```

❖ EXPLAIN examples (cont)

Example: Select on indexed, unique attribute

```
uni=# explain analyze
uni-# select * from Students where id=1216988;
                                QUERY PLAN
-----
Index Scan using students_pkey on students
  (cost=0.29..8.30 rows=1 width=9)
    (actual time=0.011..0.012 rows=1 loops=1)
  Index Cond: (id = 1216988)
Planning time: 0.273 ms
Execution time: 0.115 ms
```

❖ EXPLAIN examples (cont)

Example: Join on a primary key (indexed) attribute (2016)

```
uni=# explain analyze
uni=# select s.id,p.name
uni=# from Students s, People p where s.id=p.id;
                                QUERY PLAN
-----
Hash Join (cost=988.58..3112.76 rows=31048 width=19)
          (actual time=11.504..39.478 rows=31048 loops=1)
  Hash Cond: (p.id = s.id)
    -> Seq Scan on people p
        (cost=0.00..989.97 rows=36497 width=19)
        (actual time=0.016..8.312 rows=36497 loops=1)
    -> Hash (cost=478.48..478.48 rows=31048 width=4)
        (actual time=10.532..10.532 rows=31048 loops=1)
        Buckets: 4096  Batches: 2  Memory Usage: 548kB
    -> Seq Scan on students s
        (cost=0.00..478.48 rows=31048 width=4)
        (actual time=0.005..4.630 rows=31048 loops=1)
Planning Time: 0.691 ms
Execution Time: 44.842 ms
```

❖ EXPLAIN examples (cont)

Example: Join on a primary key (indexed) attribute (2018)

```
uni=# explain analyze
uni=# select s.id,p.name
uni=# from Students s, People p where s.id=p.id;
                                QUERY PLAN
-----
Merge Join  (cost=0.58..2829.25 rows=31361 width=18)
            (actual time=0.044..25.883 rows=31361 loops=1)
  Merge Cond: (s.id = p.id)
    -> Index Only Scan using students_pkey on students s
        (cost=0.29..995.70 rows=31361 width=4)
        (actual time=0.033..6.195 rows=31361 loops=1)
          Heap Fetches: 31361
    -> Index Scan using people_pkey on people p
        (cost=0.29..2434.49 rows=55767 width=18)
        (actual time=0.006..6.662 rows=31361 loops=1)
Planning time: 0.259 ms
Execution time: 27.327 ms
```

Transaction Processing

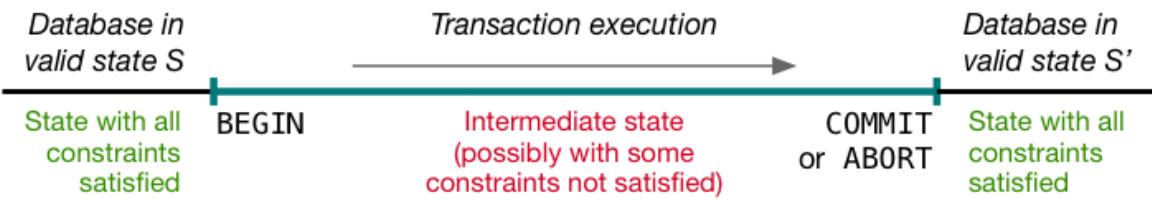
- [Transaction Processing](#)
- [Transaction Terminology](#)
- [Schedules](#)
- [Transaction Anomalies](#)

❖ Transaction Processing

A **transaction** (tx) is ...

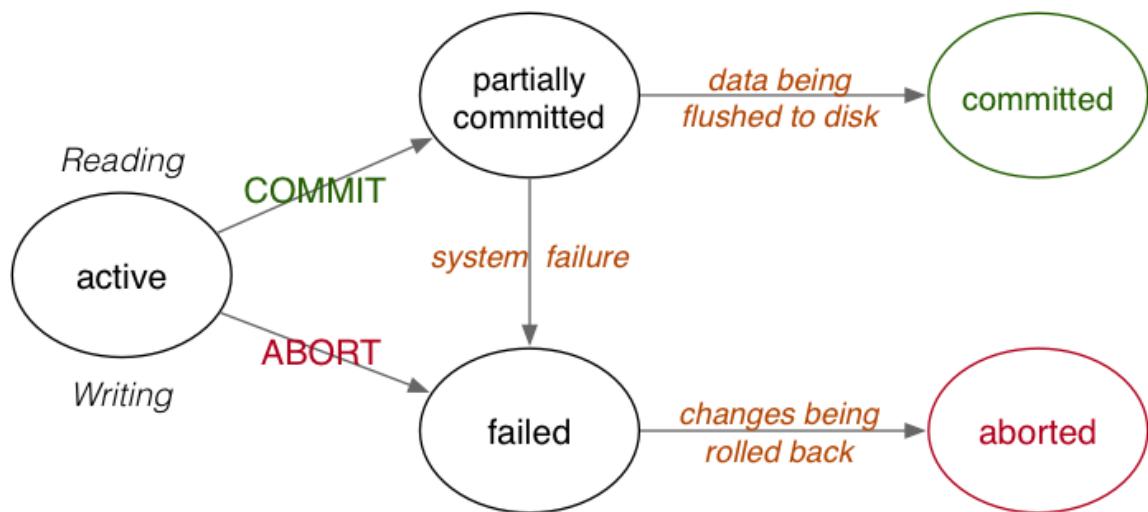
- a single application-level operation
- performed by a sequence of database operations

A transaction effects a state change on the DB



❖ Transaction Processing (cont)

Transaction states:



COMMIT ⇒ all changes preserved, **ABORT** ⇒ database unchanged

❖ Transaction Processing (cont)

Concurrent transactions are

- desirable, for improved performance (throughput)
- problematic, because of potential unwanted interactions

To ensure problem-free concurrent transactions:

- **A**tomic ... whole effect of tx, or nothing
- **C**onsistent ... individual tx's are "correct" (wrt application)
- **I**solated ... each tx behaves as if no concurrency
- **D**urable ... effects of committed tx's persist

❖ Transaction Processing (cont)

Transaction processing:

- the study of techniques for realising ACID properties

Consistency is the property:

- a tx is correct with respect to its own specification
- a tx performs a mapping that maintains all DB constraints

Ensuring this must be left to application programmers.

Our discussion focusses on: Atomicity, Durability, Isolation

❖ Transaction Processing (cont)

Atomicity is handled by the **commit** and **abort** mechanisms

- **commit** ends tx and ensures all changes are saved
- **abort** ends tx and *undoes* changes "already made"

Durability is handled by implementing **stable storage**, via

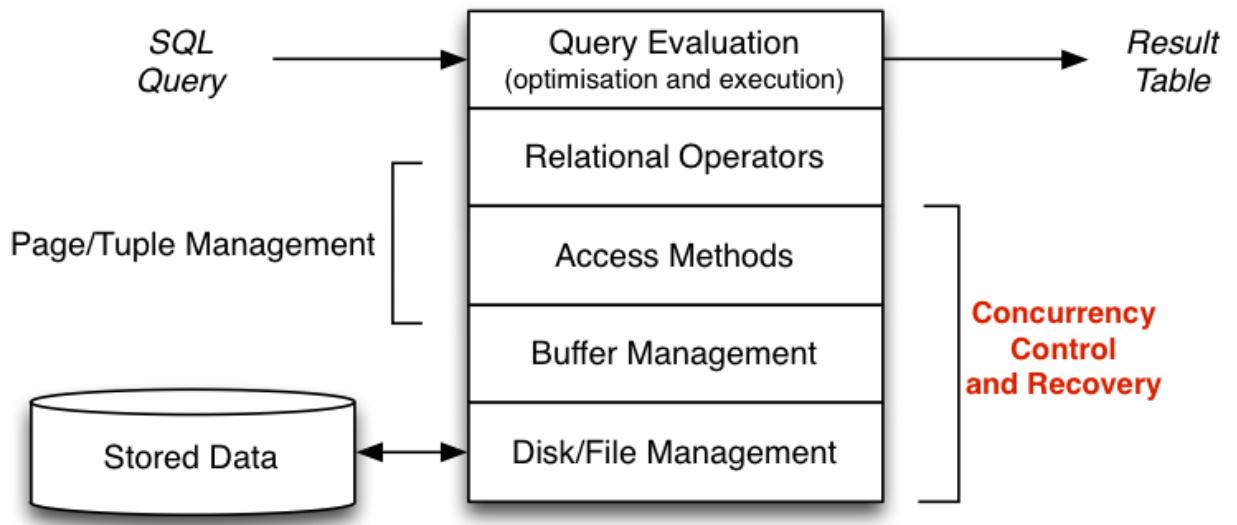
- redundancy, to deal with hardware failures
- logging/checkpoint mechanisms, to recover state

Isolation is handled by **concurrency control** mechanisms

- possibilities: lock-based, timestamp-based, check-based
- various levels of isolation are possible (e.g. serializable)

❖ Transaction Processing (cont)

Where transaction processing fits in the DBMS:



❖ Transaction Terminology

To describe transaction effects, we consider:

- **READ** - transfer data from "disk" to memory
- **WRITE** - transfer data from memory to "disk"
- **ABORT** - terminate transaction, unsuccessfully
- **COMMIT** - terminate transaction, successfully

Relationship between the above operations and SQL:

- **SELECT** produces **READ** operations on the database
- **UPDATE** and **DELETE** produce **READ** then **WRITE** operations
- **INSERT** produces **WRITE** operations

❖ Transaction Terminology (cont)

More on transactions and SQL

- **BEGIN** starts a transaction
 - the **begin** keyword in PLpgSQL is not the same thing
- **COMMIT** commits and ends the current transaction
 - some DBMSs e.g. PostgreSQL also provide **END** as a synonym
 - the **end** keyword in PLpgSQL is not the same thing
- **ROLLBACK** aborts the current transaction, undoing any changes
 - some DBMSs e.g. PostgreSQL also provide **ABORT** as a synonym

In PostgreSQL, tx's cannot be defined inside functions (e.g. PLpgSQL)

❖ Transaction Terminology (cont)

The **READ**, **WRITE**, **ABORT**, **COMMIT** operations:

- occur in the context of some transaction T
- involve manipulation of data items X, Y, \dots (READ and WRITE)

The operations are typically denoted as:

$R_T(X)$ read item X in transaction T

$W_T(X)$ write item X in transaction T

A_T abort transaction T

C_T commit transaction T

❖ Schedules

A **schedule** gives the sequence of operations from ≥ 1 tx

Serial schedule for a set of tx's $T_1 \dots T_n$

- all operations of T_i complete before T_{i+1} begins

E.g. $R_{T_1}(A) \ W_{T_1}(A) \ R_{T_2}(B) \ R_{T_2}(A) \ W_{T_3}(C) \ W_{T_3}(B)$

Concurrent schedule for a set of tx's $T_1 \dots T_n$

- operations from individual T_i 's are interleaved

E.g. $R_{T_1}(A) \ R_{T_2}(B) \ W_{T_1}(A) \ W_{T_3}(C) \ W_{T_3}(B) \ R_{T_2}(A)$

❖ Schedules (cont)

Serial schedules guarantee database consistency

- each T_i commits before T_{i+1} starts
- prior to T_i database is consistent
- after T_i database is consistent (assuming T_i is correct)
- before T_{i+1} database is consistent ...

Concurrent schedules interleave tx operations arbitrarily

- and may produce a database that is not consistent
- after all of the transactions have committed successfully

❖ Transaction Anomalies

What problems can occur with (uncontrolled) concurrent tx's?

The set of phenomena can be characterised broadly under:

- **dirty read:**
reading data item written by a concurrent uncommitted tx
- **nonrepeatable read:**
re-reading data item, since changed by another concurrent tx
- **phantom read:**
re-scanning result set, finding it changed by another tx

Properties of Schedules

- [Schedule Properties](#)
- [Serializable Schedules](#)
- [Transaction Failure](#)
- [Recoverability](#)
- [Cascading Aborts](#)
- [Strictness](#)
- [Classes of Schedules](#)

❖ Schedule Properties

If a concurrent schedule on a set of tx's TT ...

- produces the same effect as a serial schedule on TT
- then we say that the schedule is **serializable**

A goal of isolation mechanisms (see later) is

- arrange execution of individual operations in tx's in TT
- to ensure that a serializable schedule is produced

Serializability is one property of a schedule, focusing on isolation

Other properties of schedules focus on recovering from failures

❖ Serializable Schedules

Producing a serializable schedule

- eliminates all update anomalies
- may reduce opportunity for concurrency
- may reduce overall throughput of system

If DB programmers know update anomalies are unlikely/tolerable

- serializable schedules may not be necessary
- some DBMSs offer less strict isolation levels (e.g. repeatable read)
- allowing more opportunity for concurrency

❖ Transaction Failure

So far, have implicitly assumed that all transactions commit.

Additional problems can arise when transactions abort.

Consider the following schedule where transaction T1 fails:

T1: R(X) W(X) A

T2: R(X) W(X) C

Abort *will* rollback the changes to X, but ...

Consider three places where the rollback might occur:

T1: R(X) W(X) A [1] [2] [3]

T2: R(X) W(X) C

❖ Transaction Failure (cont)

Abort / rollback scenarios:

T1: R(X) W(X) A [1] [2] [3]
T2: R(X) W(X) C

Case [1] is ok

- all effects of T1 vanish; final effect is simply from T2

Case [2] is problematic

- some of T1's effects persist, even though T1 aborted

Case [3] is also problematic

- T2's effects are lost, even though T2 committed

❖ Recoverability

Consider the serializable schedule:

T1:	R(X)	W(Y)	C
T2:	W(X)		A

(where the final value of Y is dependent on the X value)

Notes:

- the final value of X is valid (change from T_2 rolled back)
- T_1 reads/uses an X value that is eventually rolled-back
- even though T_2 is correctly aborted, it has produced an effect

Produces an invalid database state, even though serializable.

❖ Recoverability (cont)

Recoverable schedules avoid these kinds of problems.

For a schedule to be recoverable, we require additional constraints

- all tx's T_i that write values used by T_j must commit before T_j commits

and this property must hold for all transactions T_j

Note that recoverability does not prevent "dirty reads".

In order to make schedules recoverable in the presence of dirty reads and aborts, may need to abort multiple transactions.

❖ Cascading Aborts

Recall the earlier non-recoverable schedule:

T1:	R(X)	W(Y)	C
T2:	W(X)		A

To make it recoverable requires:

- delaying T_1 's commit until T_2 commits
- if T_2 aborts, cannot allow T_1 to commit

T1:	R(X)	W(Y)	...	C?	A!
T2:	W(X)			A	

Known as **cascading aborts** (or **cascading rollback**).

❖ Cascading Aborts (cont)

Example: T_3 aborts, causing T_2 to abort, causing T_1 to abort

T1:	R(Y)	W(Z)	A
T2:	R(X)	W(Y)	A
T3:	W(X)		A

Even though T_1 has no direct connection with T_3 (i.e. no shared data).

This kind of problem ...

- can potentially affect very many concurrent transactions
- could have a significant impact on system throughput

❖ Cascading Aborts (cont)

Cascading aborts can be avoided if

- transactions can only read values written by committed transactions

(alternative formulation: no tx can read data items written by an uncommitted tx)

Effectively: eliminate the possibility of reading dirty data

Downside: reduces opportunity for concurrency

GUW call these **ACR** (avoid cascading rollback) schedules.

All ACR schedules are also recoverable.

❖ Strictness

Strict schedules also eliminate the chance of *writing* dirty data.

A schedule is **strict** if

- no tx can read values written by another uncommitted tx (ACR)
- no tx can write a data item written by another uncommitted tx

Strict schedules simplify the task of rolling back after aborts.

❖ Strictness (cont)

Example: non-strict schedule

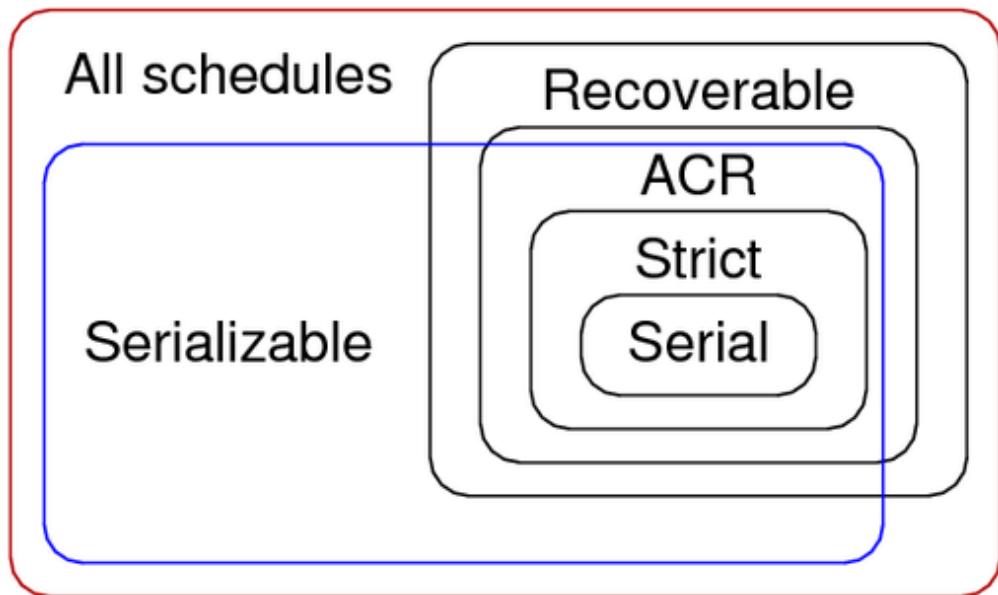
T1:	W(X)	A
T2:		W(X) A

Problems with handling rollback after aborts:

- when T_1 aborts, don't rollback (need to retain value written by T_2)
- when T_2 aborts, need to rollback to pre- T_1 (not just pre- T_2)

❖ Classes of Schedules

Relationship between various classes of schedules:



Schedules ought to be serializable and strict.

But more serializable/strict \Rightarrow less concurrency.

DBMSs allow users to trade off "safety" against performance.

Transaction Isolation

- [Transaction Isolation](#)
- [Serializability](#)
- [Checking Serializability](#)
- [Transaction Isolation Levels](#)
- [Concurrency Control](#)

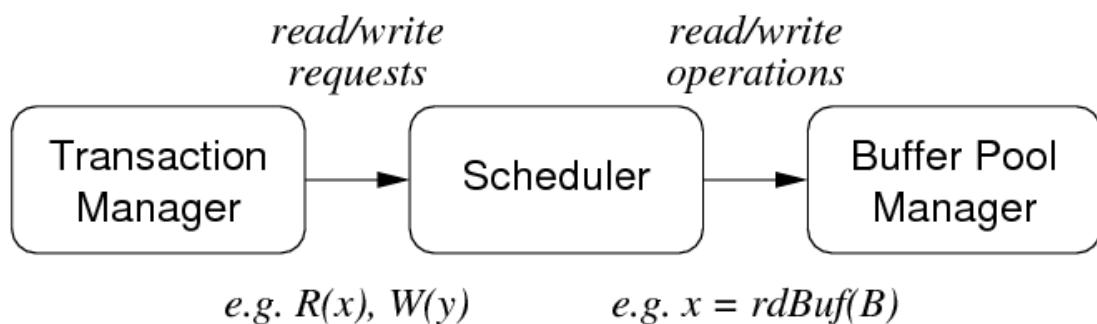
❖ Transaction Isolation

Simplest form of isolation: **serial** execution ($T_1 ; T_2 ; T_3 ; \dots$)

Problem: serial execution yields poor throughput.

Concurrency control schemes (CCSs) aim for "safe" concurrency

Abstract view of DBMS concurrency mechanisms:



❖ Serializability

Consider two schedules S_1 and S_2 produced by

- executing the same set of transactions $T_1..T_n$ concurrently
- but with a non-serial interleaving of R/W operations

S_1 and S_2 are **equivalent** if $\text{StateAfter}(S_1) = \text{StateAfter}(S_2)$

- i.e. final state yielded by S_1 is same as final state yielded by S_2

S is a **Serializable schedule** (for a set of concurrent tx's $T_1..T_n$) if

- S is equivalent to some serial schedule S_S of $T_1..T_n$

Under these circumstances, consistency is guaranteed (assuming no aborted transactions and no system failures)

❖ Serializability (cont)

Two formulations of serializability:

- **conflict serializability**
 - i.e. conflicting R/W operations occur in the "right order"
 - check via precedence graph; look for absence of cycles
- **view serializability**
 - i.e. read operations see the correct version of data
 - checked via VS conditions on likely equivalent schedules

View serializability is strictly weaker than conflict serializability.

❖ Checking Serializability

Conflict serializability checking:

```
make a graph with just nodes, one for each Ti
for each pair of operations across transactions {
    if (Ti and Tj have conflicting ops on variable X) {
        put a directed edge between Ti and Tj
        where the direction goes from
        first tx to access X to second tx to access X
        if this new edge forms a cycle in the graph
            return "Not conflict serializable"
    }
}
return "Conflict serializable"
```

❖ Checking Serializability (cont)

View serializability checking:

```
// TC,i denotes transaction i in concurrent schedule  
for each serial schedule S {  
    // TS,i denotes transaction i in serial schedule  
    for each shared variable X {  
        if TC,i reads same version of X as TS,i  
            (either initial value or value written by Tj)  
            continue  
        else  
            give up on this serial schedule  
        if TC,i and TS,i write the final version of X  
            continue  
        else  
            give up on this serial schedule  
    }  
    return "View serializable"  
}  
return "Not view serializable"
```

❖ Transaction Isolation Levels

SQL programmers' concurrency control mechanism ...

```
set transaction
    read only -- so weaker isolation may be ok
    read write -- suggests stronger isolation needed
isolation level
    -- weakest isolation, maximum concurrency
read uncommitted
read committed
repeatable read
serializable
    -- strongest isolation, minimum concurrency
```

Applies to current tx only; affects how scheduler treats this tx.

❖ Transaction Isolation Levels (cont)

Implication of transaction isolation levels:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read Uncommitted	Possible	Possible	Possible
Read Committed	Not Possible	Possible	Possible
Repeatable Read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

❖ Transaction Isolation Levels (cont)

For transaction isolation, PostgreSQL

- provides syntax for all four levels
- treats **read uncommitted** as **read committed**
- **repeatable read** behaves *like* **serializable**
- default level is **read committed**

Note: cannot implement **read uncommitted** because of MVCC

For more details, see [PostgreSQL Documentation section 13.2](#)

- extensive discussion of semantics of **UPDATE**, **INSERT**, **DELETE**

❖ Transaction Isolation Levels (cont)

A PostgreSQL tx consists of a sequence of SQL statements:

```
BEGIN  $S_1$ ;  $S_2$ ; ...  $S_n$ ; COMMIT;
```

Isolation levels affect view of DB provided to each S_i :

- in *read committed* ...
 - each S_i sees snapshot of DB at start of S_i
- in *repeatable read* and *serializable* ...
 - each S_i sees snapshot of DB at start of tx
 - serializable checks for extra conditions

Transactions fail if the system detects violation of isolation level.

❖ Transaction Isolation Levels (cont)

Example of **repeatable read** vs **serializable**

- table R(class,value) containing (1,10) (1,20) (2,100) (2,200)
- T1: $X = \text{sum}(\text{value}) \text{ where class}=1$; insert R(2,X); commit
- T2: $X = \text{sum}(\text{value}) \text{ where class}=2$; insert R(1,X); commit
- with **repeatable read**, both transactions commit, giving
 - updated table: (1,10) (1,20) (2,100) (2,200) (1,300) (2,30)
- with **serial** transactions, only one transaction commits
 - T1;T2 gives (1,10) (1,20) (2,100) (2,200) (2,30) (1,330)
 - T2;T1 gives (1,10) (1,20) (2,100) (2,200) (1,300) (2,330)
- PG recognises that committing both gives serialization violation

❖ Concurrency Control

Isolation requires some method to control concurrency

Possible approaches to implementing concurrency control:

- **Lock-based**
 - Synchronise tx execution via locks on relevant part of DB.
- **Version-based** (multi-version concurrency control)
 - Allow multiple consistent versions of the data to exist.
Each tx has access only to version existing at start of tx.
- **Validation-based** (optimistic concurrency control)
 - Execute all tx's; check for validity problems on commit.
- **Timestamp-based**
 - Organise tx execution via timestamps assigned to actions.

Lock-based Concurrency Control

- [Lock-based Concurrency Control](#)
- [Two-Phase Locking](#)
- [Problems with Locking](#)
- [Deadlock](#)

❖ Lock-based Concurrency Control

Requires read/write **lock** operations which act on database objects.

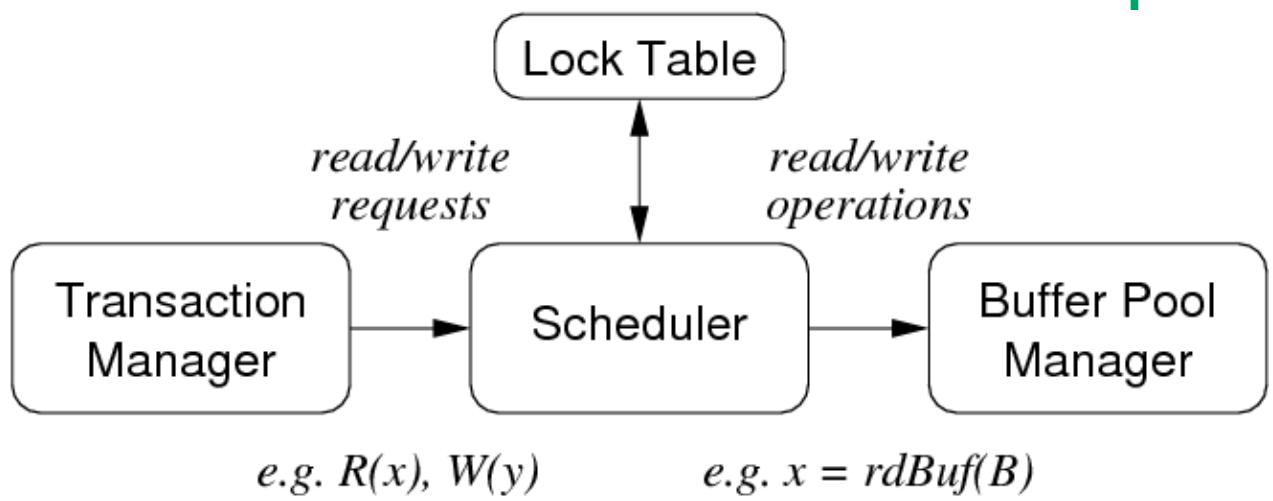
Synchronise access to shared DB objects via these rules:

- before reading X , get read (shared) lock on X
- before writing X , get write (exclusive) lock on X
- a tx attempting to get a read lock on X is blocked if another tx already has write lock on X
- a tx attempting to get a write lock on X is blocked if another tx has any kind of lock on X

These rules alone do not guarantee serializability.

❖ Lock-based Concurrency Control (cont)

Locks introduce additional mechanisms in DBMS:



The Lock Manager manages the locks requested by the scheduler

❖ Lock-based Concurrency Control (cont)

Lock table entries contain:

- object being locked (DB, table, tuple, field)
- type of lock: read/shared, write/exclusive
- FIFO queue of tx's requesting this lock
- count of tx's currently holding lock (max 1 for write locks)

Lock and unlock operations *must* be atomic.

Lock **upgrade**:

- if a tx holds a read lock, and it is the only tx holding that lock
- then the lock can be converted into a write lock

❖ Lock-based Concurrency Control (cont)

Consider the following schedule, using locks:

T1(a): $L_r(Y)$ R(Y) continued

T2(a): $L_r(X)$ R(X) U(X) continued

T1(b): U(Y) $L_w(X)$ W(X) U(X)

T2(b): $L_w(Y)$. . . W(Y) U(Y)

(where L_r = read-lock, L_w = write-lock, U = unlock)

Locks correctly ensure controlled access to X and Y.

Despite this, the schedule is not serializable.

(Ex: prove this)

❖ Two-Phase Locking

To guarantee serializability, we require an additional constraint:

- in every tx, all *lock* requests precede all *unlock* requests

Each transaction is then structured as:

- **growing** phase where locks are acquired
- **action** phase where "real work" is done
- **shrinking** phase where locks are released

Clearly, this reduces potential concurrency ...

◆ Problems with Locking

Appropriate locking can guarantee serializability..

However, it also introduces potential undesirable effects:

- **Deadlock**
 - No transactions can proceed; each waiting on lock held by another.
- **Starvation**
 - One transaction is permanently "frozen out" of access to data.
- **Reduced performance**
 - Locking introduces delays while waiting for locks to be released.

❖ Deadlock

Deadlock occurs when two tx's wait for a lock on an item held by the other.

Example:

T1: $L_w(A)$	$R(A)$	$L_w(B)$	
T2:	$L_w(B)$	$R(B)$	$L_w(A)$

How to deal with deadlock?

- prevent it happening in the first place
- let it happen, detect it, recover from it

❖ Deadlock (cont)

Handling deadlock involves forcing a transaction to "back off"

- select process to roll back
 - choose on basis of how far tx has progressed, # locks held, ...
- roll back the selected process
 - how far does this it need to be rolled back?
 - worst-case scenario: abort one transaction, then retry
- prevent starvation
 - need methods to ensure that same tx isn't always chosen

❖ Deadlock (cont)

Methods for managing deadlock

- **timeout** : set max time limit for each tx
- **waits-for graph** : records T_j waiting on lock held by T_k
 - *prevent* deadlock by checking for new cycle \Rightarrow abort T_i
 - *detect* deadlock by periodic check for cycles \Rightarrow abort T_i
- **timestamps** : use tx start times as basis for priority
 - scenario: T_j tries to get lock held by T_k
 - ...
 - **wait-die**: if $T_j < T_k$, then T_j waits, else T_j rolls back
 - **wound-wait**: if $T_j < T_k$, then T_k rolls back, else T_j waits

❖ Deadlock (cont)

Properties of deadlock handling methods:

- both wait-die and wound-wait are fair
- wait-die tends to
 - roll back tx's that have done little work
 - but rolls back tx's more often
- wound-wait tends to
 - roll back tx's that may have done significant work
 - but rolls back tx's less often
- timestamps easier to implement than waits-for graph
- waits-for minimises roll backs because of deadlock

Optimistic Concurrency Control

- [Optimistic Concurrency Control](#)
- [Validation](#)
- [Validation Check](#)

❖ Optimistic Concurrency Control

Locking is a pessimistic approach to concurrency control:

- limit concurrency to ensure that conflicts don't occur

Costs: lock management, deadlock handling, contention.

In scenarios where there are far more reads than writes

...

- don't lock (allow arbitrary interleaving of operations)
- check just before commit that no conflicts occurred
- if problems, roll back conflicting transactions

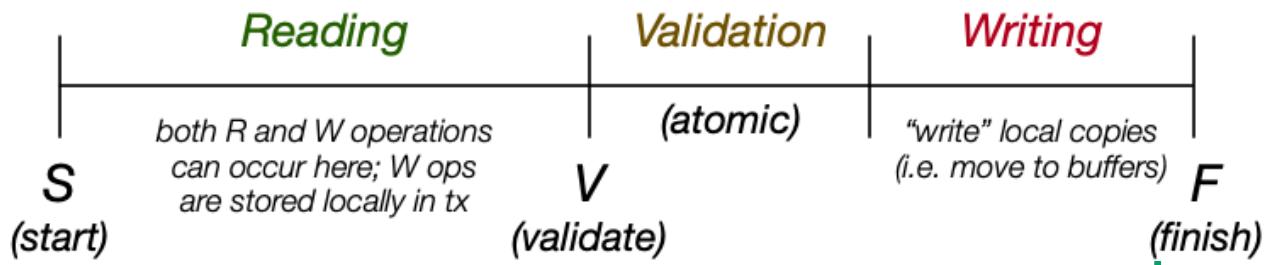
Optimistic concurrency control (OCC) is a strategy to realise this.

❖ Optimistic Concurrency Control (cont)

Under OCC, transactions have three distinct phases:

- **Reading**: read from database, modify local copies of data
- **Validation**: check for conflicts in updates
- **Writing**: commit local copies of data to database

Timestamps are recorded at points S , V , F :



❖ Validation

Data structures needed for validation:

- S ... set of txs that are reading data and computing results
- V ... set of txs that have reached validation (not yet committed)
- F ... set of txs that have finished (committed data to storage)
- for each T_i , timestamps for when it reached S, V, F
- $RS(T_i)$ set of all data items read by T_i
- $WS(T_i)$ set of all data items to be written by T_i

Use the V timestamps as ordering for transactions

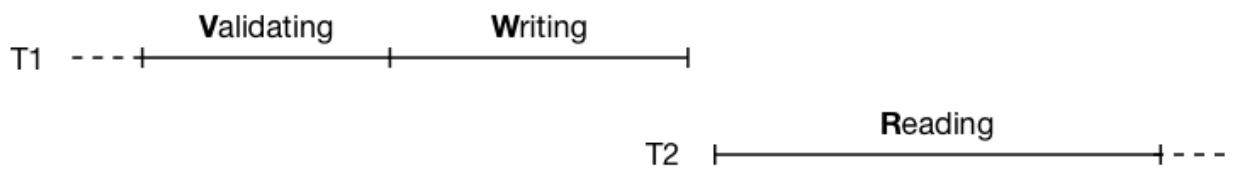
- assume serial tx order based on ordering of $V(T_i)$'s

❖ Validation (cont)

Two-transaction example:

- allow transactions T_1 and T_2 to run without any locking
- check that objects used by T_2 are not being changed by T_1
- if they are, we need to roll back T_2 and retry

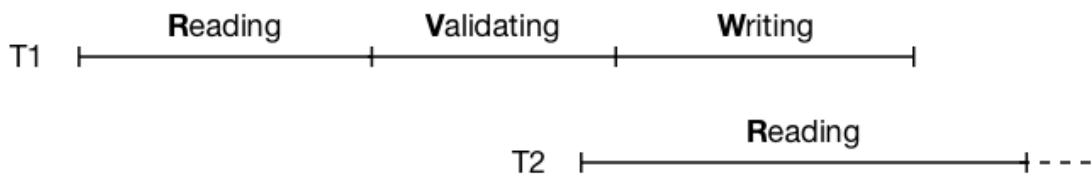
Case 0: serial execution ... no problem



❖ Validation (cont)

Case 1: reading overlaps validation/writing

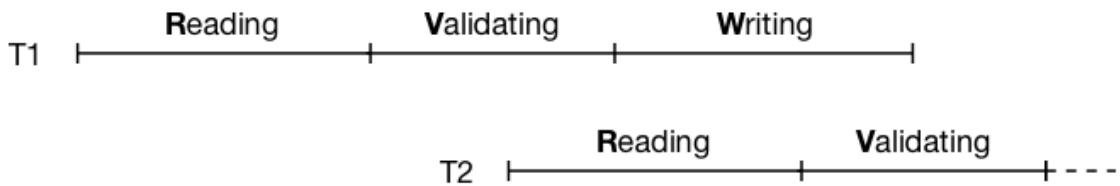
- T_2 starts while T_1 is validating/writing
- if some X being read by T_2 is in $WS(T_1)$
- then T_2 may not have read the updated version of X
- so, T_2 must start again



❖ Validation (cont)

Case 2: reading/validation overlaps validation/writing

- T_2 starts validating while T_1 is validating/writing
- if some X being written by T_2 is in $WS(T_1)$
- then T_2 may end up overwriting T_1 's update
- so, T_2 must start again



❖ Validation Check

Validation check for transaction T

- for all transactions $T_i \neq T$
 - if $T \in S \text{ & } T_i \in F$, then ok
 - if $T \notin V \text{ & } V(T_i) < S(T) < F(T_i)$,
then check $WS(T_i) \cap RS(T)$ is empty
 - if $T \in V \text{ & } V(T_i) < V(T) < F(T_i)$,
then check $WS(T_i) \cap WS(T)$ is empty

If this check fails for any T_i , then T is rolled back.

❖ Validation Check (cont)

OCC prevents: T reading dirty data, T overwriting T_i 's changes

Problems with OCC:

- increased roll backs**
- tendency to roll back "complete" tx's
- cost to maintain S, V, F sets

** "Roll back" is relatively cheap

- changes to data are purely local before Writing phase
- no requirement for logging info or undo/redo (see later)

Multi-version Concurrency Control

- [Multi-version Concurrency Control](#)
- [Concurrency Control in PostgreSQL](#)

❖ Multi-version Concurrency Control

Multi-version concurrency control (MVCC) aims to

- retain benefits of locking, while getting more concurrency
- by providing multiple (consistent) versions of data items

Achieves this by

- readers access an "appropriate" version of each data item
- writers make new versions of the data items they modify

Main difference between MVCC and standard locking:

- read locks do not conflict with write locks
⇒
- reading never blocks writing, writing never blocks reading

❖ Multi-version Concurrency Control (cont)

WTS = timestamp of tx that wrote this data item

Chained tuple versions: $tup_{oldest} \rightarrow tup_{older} \rightarrow tup_{newest}$

When a reader T_i is accessing the database

- ignore any data item D created after T_i started
 - checked by: $\text{WTS}(D) > \text{TS}(T_i)$
- use only newest version V accessible to T_i
 - determined by: $\max(\text{WTS}(V)) < \text{TS}(T_i)$

❖ Multi-version Concurrency Control (cont)

When a writer T_i attempts to change a data item

- find newest version V satisfying $\text{WTS}(V) < \text{TS}(T_i)$
- if no later versions exist, create new version of data item
- if there are later versions, then abort T_i

Some MVCC versions also maintain RTS (TS of last reader)

- don't allow T_i to write D if $\text{RTS}(D) > \text{TS}(T_i)$

❖ Multi-version Concurrency Control (cont)

Advantage of MVCC

- locking needed for serializability considerably reduced

Disadvantages of MVCC

- visibility-check overhead (on every tuple read/write)
- reading an item V causes an update of $RTS(V)$
- storage overhead for extra versions of data items
- overhead in removing out-of-date versions of data items

Despite apparent disadvantages, MVCC is very effective.

❖ Multi-version Concurrency Control (cont)

Removing old versions:

- V_j and V_k are versions of same item
- $WTS(V_j)$ and $WTS(V_k)$ precede $TS(T_i)$ for all T_i
- remove version with smaller $WTS(V_x)$ value

When to make this check?

- every time a new version of a data item is added?
- periodically, with fast access to blocks of data

PostgreSQL uses the latter (**vacuum**).

❖ Concurrency Control in PostgreSQL

PostgreSQL uses two styles of concurrency control:

- multi-version concurrency control (MVCC)
(used in implementing SQL DML statements (e.g. `select`))
- two-phase locking (2PL)
(used in implementing SQL DDL statements (e.g. `create table`))

From the SQL (PLpgSQL) level:

- can let the lock/MVCC system handle concurrency
- can handle it explicitly via **LOCK** statements

❖ Concurrency Control in PostgreSQL (cont)

PostgreSQL provides **read committed** and **Serializable** isolation levels.

Using the serializable isolation level, a **select**:

- sees only data committed before the transaction began
- never sees changes made by concurrent transactions

Using the serializable isolation level, an update fails:

- if it tries to modify an "active" data item
(active = affected by some other tx, either committed or uncommitted)

The transaction containing the update must then rollback and re-start.

❖ Concurrency Control in PostgreSQL (cont)

Implementing MVCC in PostgreSQL requires:

- a log file to maintain current status of each T_i
- in every tuple:
 - **xmin** = ID of the tx that created the tuple
 - **xmax** = ID of the tx that replaced/deleted the tuple (if any)
 - **xnew** = link to newer versions of tuple (if any)
- for each transaction T_i :
 - a transaction ID (timestamp)
 - SnapshotData: list of active tx's when T_i started

❖ Concurrency Control in PostgreSQL (cont)

Rules for a tuple to be visible to T_i :

- the **xmin** (creation transaction) value must
 - be committed in the log file
 - have started before T_i 's start time
 - not be active at T_i 's start time
- the **xmax** (delete/replace transaction) value must
 - be blank or refer to an aborted tx, or
 - have started after T_i 's start time, or
 - have been active at SnapshotData time

For details, see:

`backend/access/heap/heapam_visibility.c`

❖ Concurrency Control in PostgreSQL (cont)

Tx's always see a consistent version of the database.

But may not see the "current" version of the database.

E.g. T1 does select, then concurrent T2 deletes some of T1's selected tuples

This is OK unless tx's communicate outside the database system.

E.g. T1 counts tuples; T2 deletes then counts; then counts are compared

Use locks if application needs every tx to see same current version

- **LOCK TABLE** locks an entire table
- **SELECT FOR UPDATE** locks only the selected rows

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

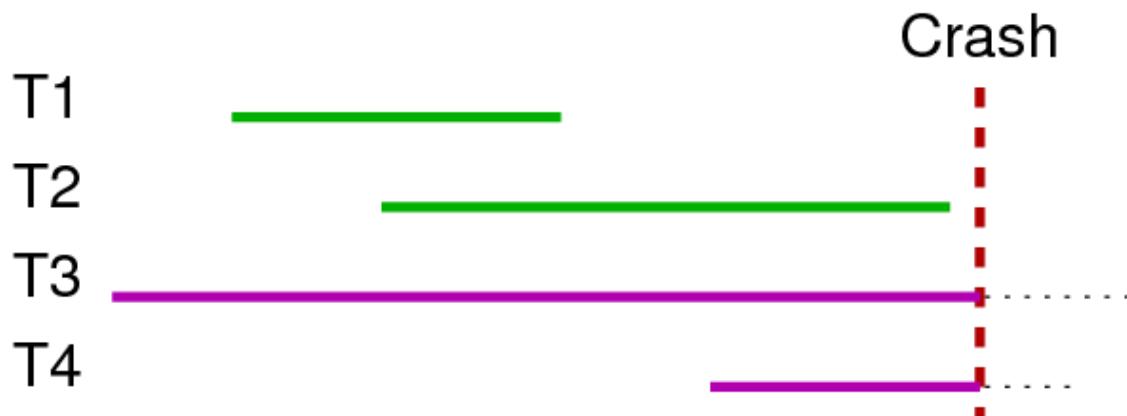
What kinds of "system failures" do we need to deal with?

- 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)

Durability 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:

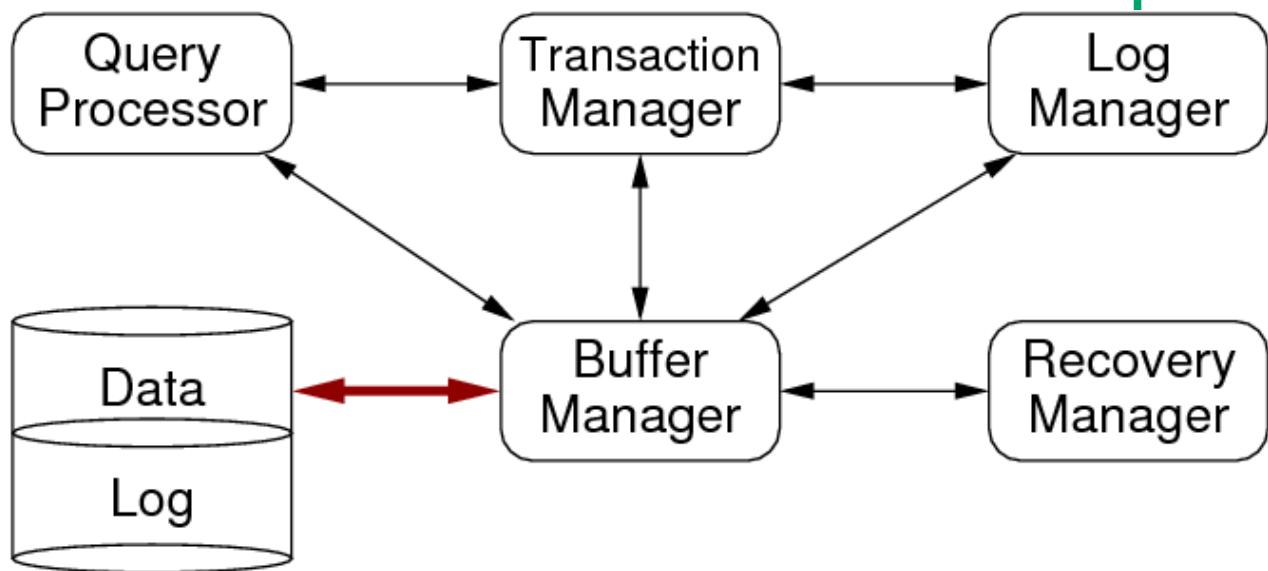
- failure of DBMS processes or operating system
- failure of transactions (**ABORT**)

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, ..., T_k)>**
(contains references to all active transactions \Rightarrow active tx table)
2. continue normal processing (e.g. new tx's can start)
3. when all of **T₁, ..., T_k** 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 $\langle T, X, v' \rangle$, 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 $\langle T, X, v \rangle$ and T is committed, set X to v on disk
 - if $\langle \text{START } T \rangle$ and T not committed, put $\langle \text{ABORT } T \rangle$ 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
 $\langle T, X, v, v' \rangle$ 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 logging 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 $\langle \text{ABORT } T \rangle$ to log
- scan **backwards** through log
 - if $\langle T, X, v, w \rangle$ and T is not committed, set X to v on disk
- scan **forwards** through log
 - if $\langle T, X, v, w \rangle$ 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 number (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 (petabytes, 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

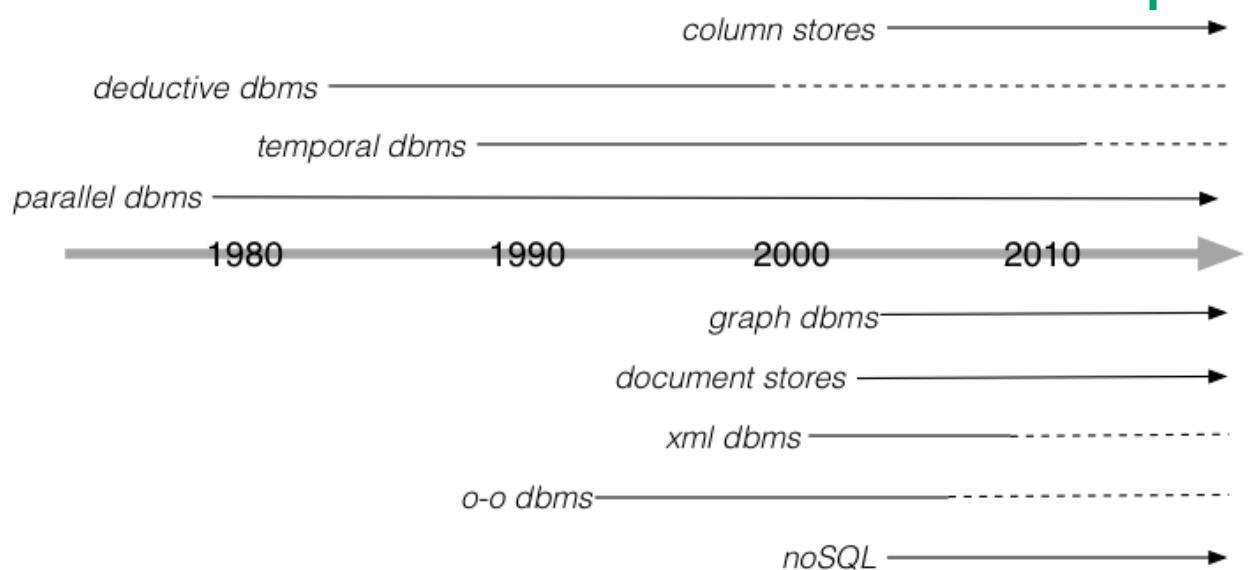
❖ Future of Database (cont)

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 *keys* 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/regexprs)

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? ($image_1 \approx image_2$)
- 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

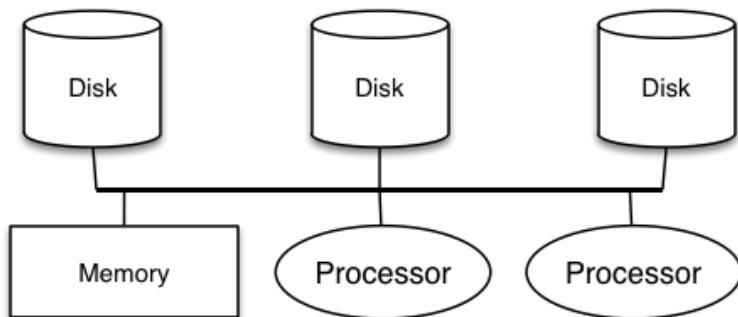
Parallelism in Databases

- [Parallel DBMSs](#)
- [Parallel Architectures](#)
- [Distributed Architectures](#)
- [Parallel Databases \(PDBs\)](#)
- [Data Storage in PDBs](#)
- [Parallelism in DB Operations](#)

❖ Parallel DBMSs

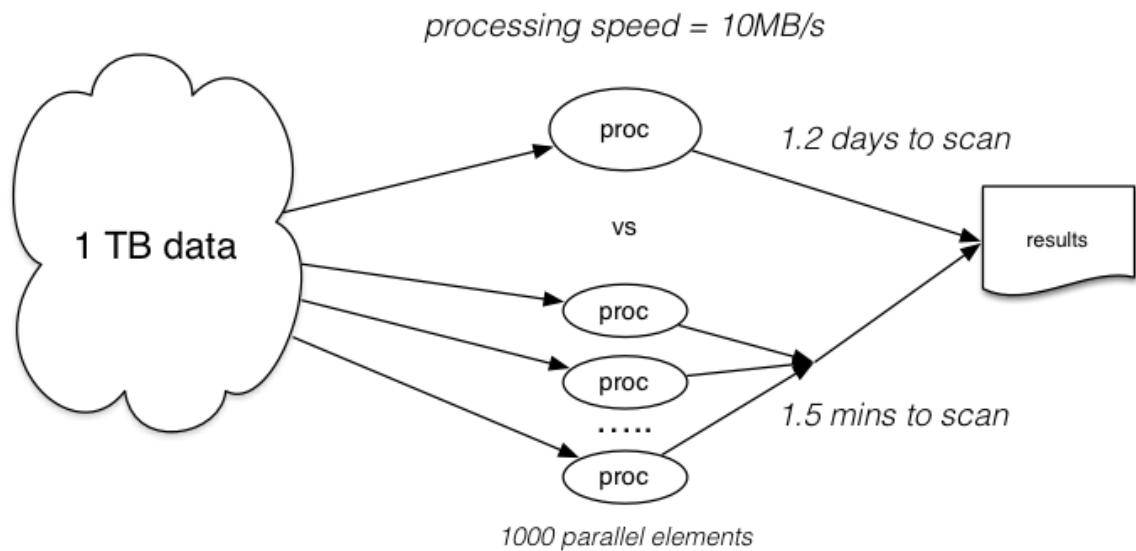
RDBMS discussion so far has revolved around systems

- 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

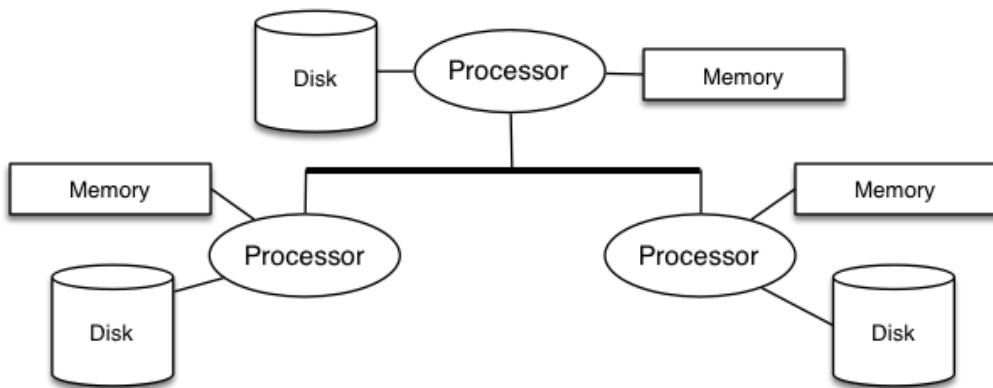
- can process many data elements (tuples) at the same time
- can create pipelines of query evaluation steps
- don't require special hardware
- can hide parallelism within the query evaluator
 - application programmers don't need to change habits

Compare this with effort to do parallel programming.

❖ Parallel Architectures

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

Example shared-nothing architecture:

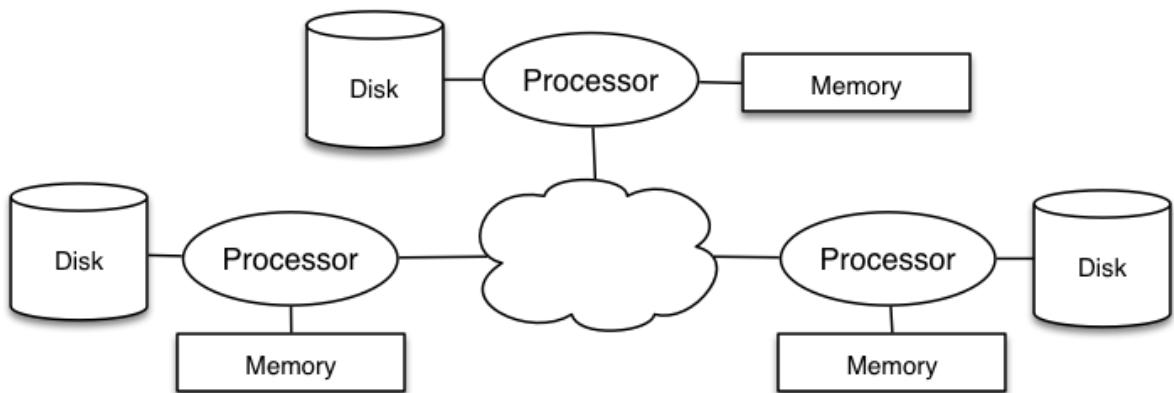


Typically same room/LAN (data transfer cost ~ 100's of μ secs .. msecs)

❖ 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

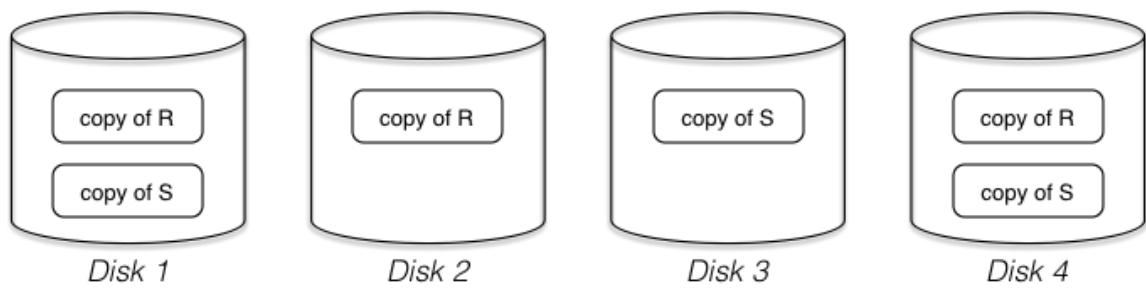
❖ Data Storage in PDBs

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

Can distribute data across multiple storage devices

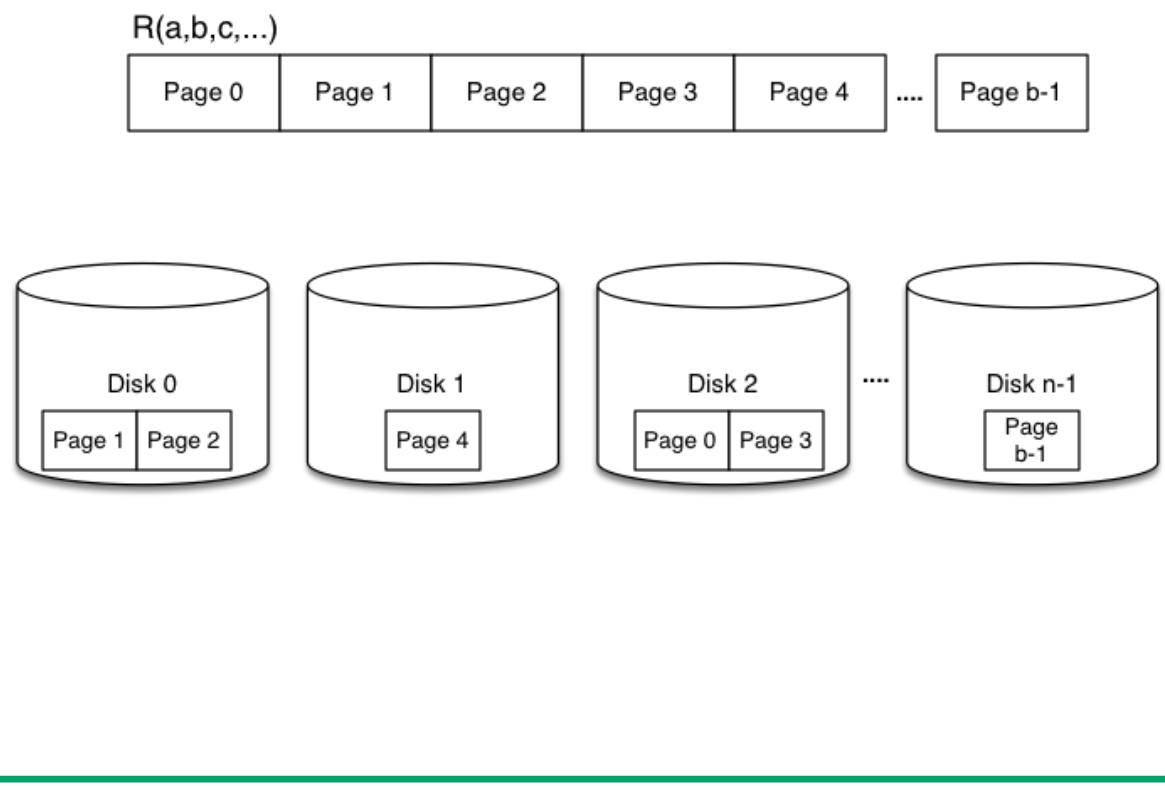
- duplicate all pages from a relation (replication)
- store some pages on one store, some on others (partitioning)

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: ($\{\text{Nodes}\}$, *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 $(i \bmod n)^{\text{th}}$ 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. $i = \text{hash}(\text{tuple})$ 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 node₀, 11-20 on node₁, ..., 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

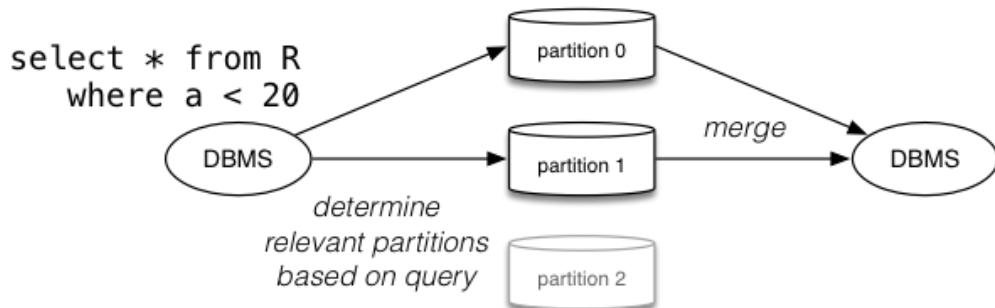
- intra-operator parallelism
 - get all machines working to compute a given operation
(scan, sort, join)
- inter-operator parallelism
 - each operator runs concurrently on a different processor
(exploits pipelining)
- Inter-query parallelism
 - different queries run on different processors

❖ 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

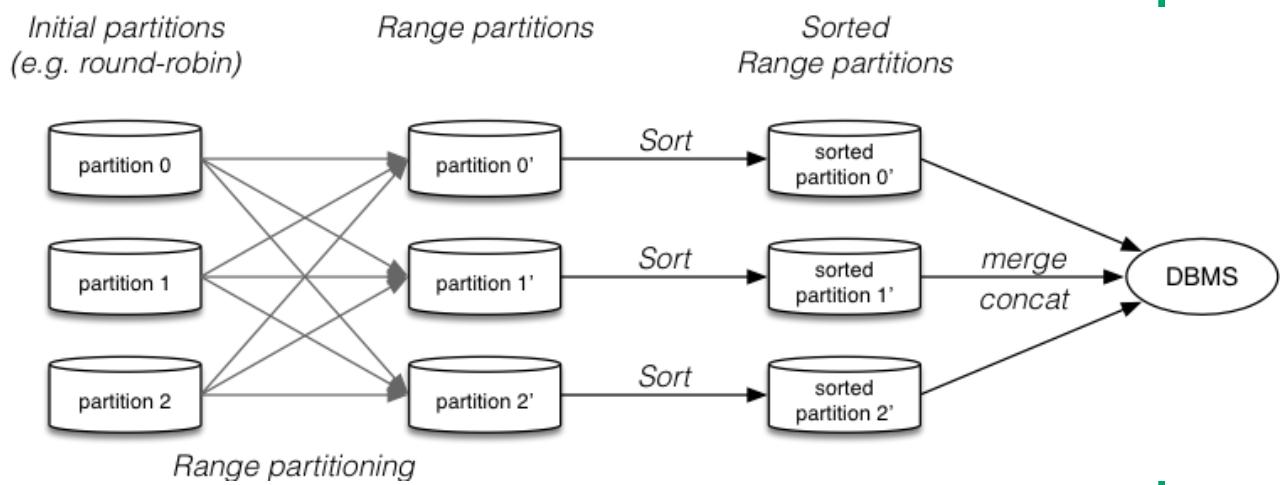
- scan in parallel, range-partition during scan
- pipeline into local sort on each processor
- merge sorted partitions in order

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

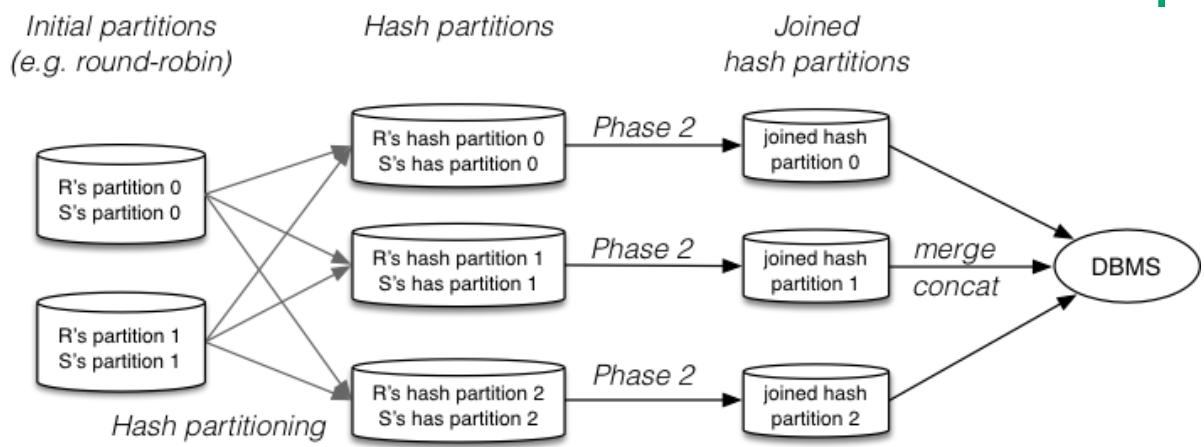
- distribute partitions to different processors
- partition 0 of R goes to same node as partition 0 of S
- join phase can be done in parallel on each processor
- then results need to be merged
- very effective for equijoin

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 Data Models](#)
- [GDb Queries](#)
- [Example Graph Queries](#)

❖ Graph Databases

Graph Databases (GDbs):

- DBMSs that use **graphs** as the data model

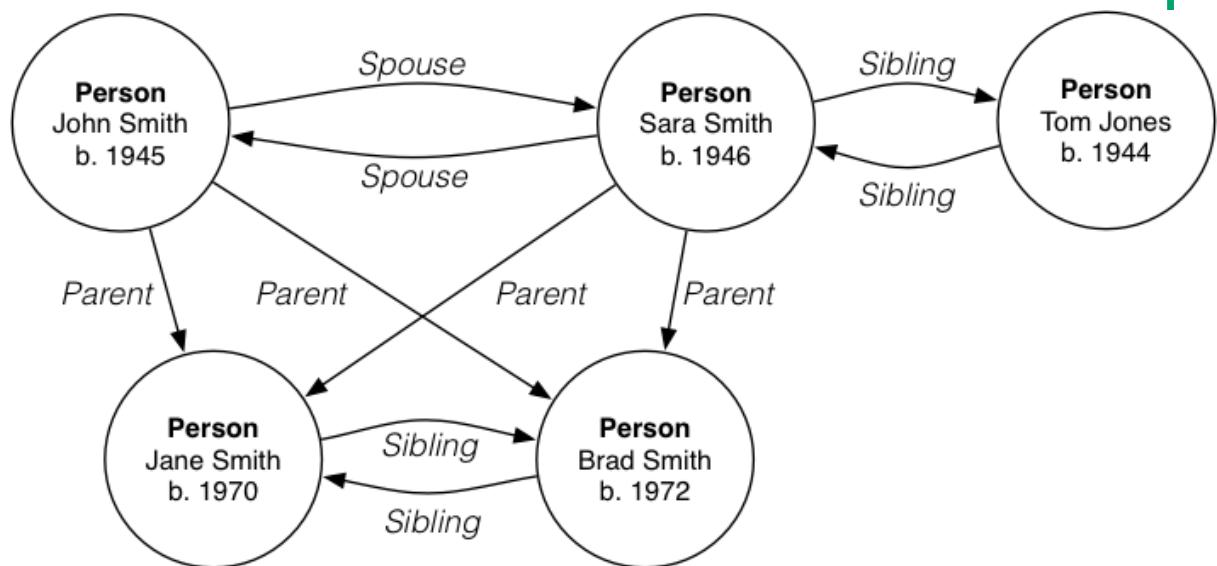
But what kind of "graphs"?

- 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

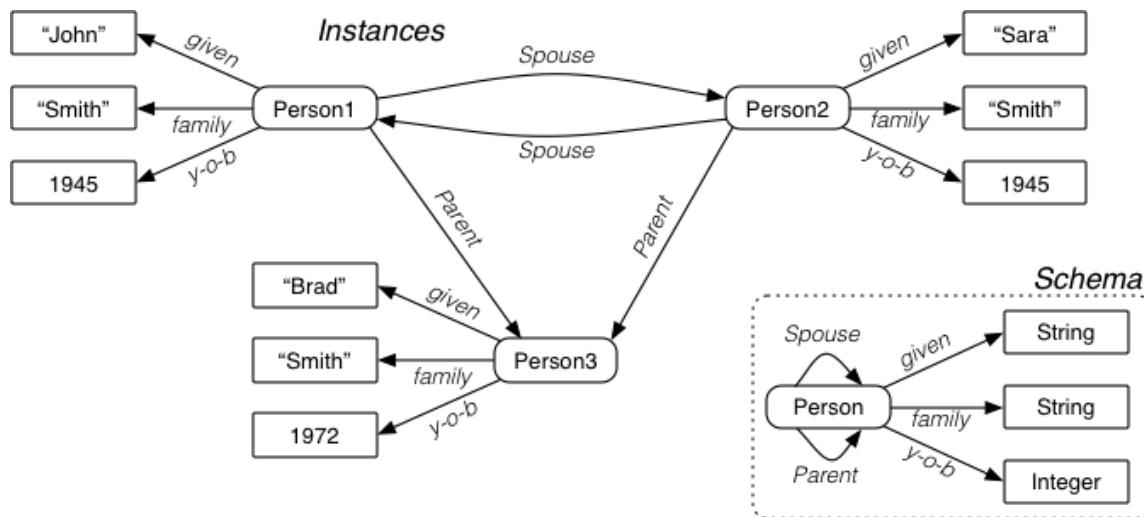
- directed, labelled graphs
- nodes have identifiers (constant values, incl. URIs)
- edges are labelled with the relationship
- can have multiple edges between nodes (diff. labels)
- can store multiple graphs in one database
- datatypes based on W3C XML Schema datatypes

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)

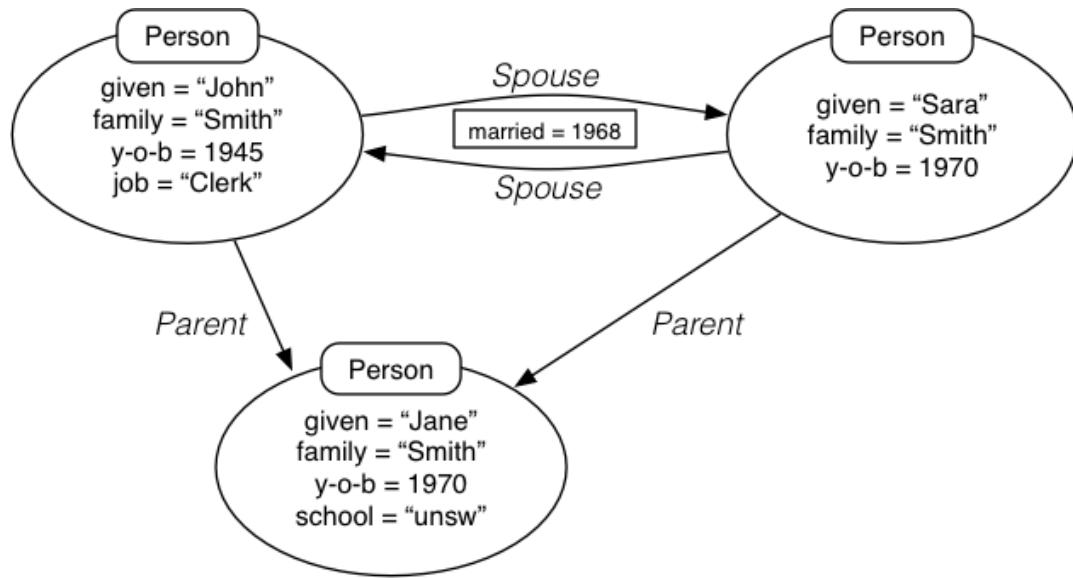
Property Graph

- 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
```