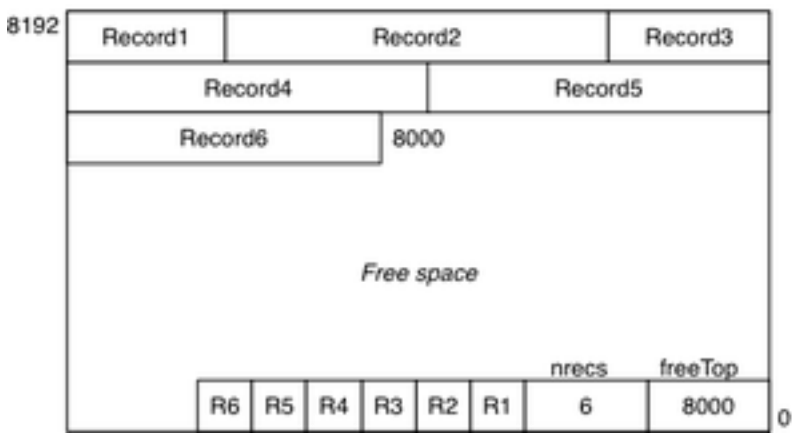
Week 04 Lectures

Tuples

Tuples 2/84

Each *page* contains a collection of *tuples*



What do tuples contain? How are they structured internally?

Records vs Tuples 3/84

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 number(4) references Dept(id)
);
```

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

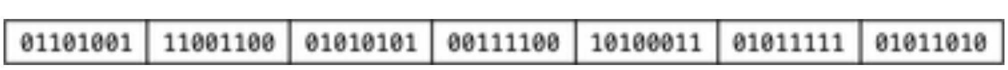
Schema information (meta-data) is stored in the DB catalog

... Records vs Tuples 4/84

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

```
(33357462, 'Neil Young', 'Musician', 0277)
```

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



Bytes need to be interpreted relative to schema to get tuple

Converting Records to Tuples 5/84

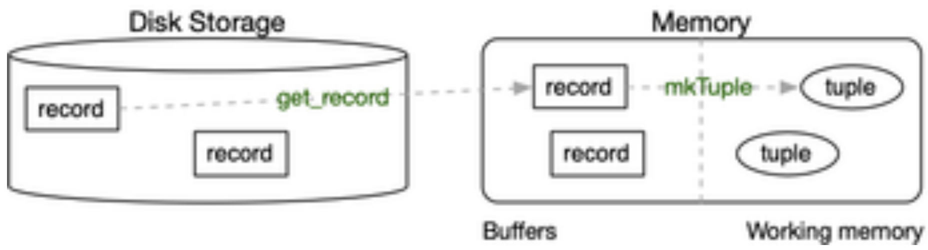
A `Record` is an array of bytes (`byte[]`)

- 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

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 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)`

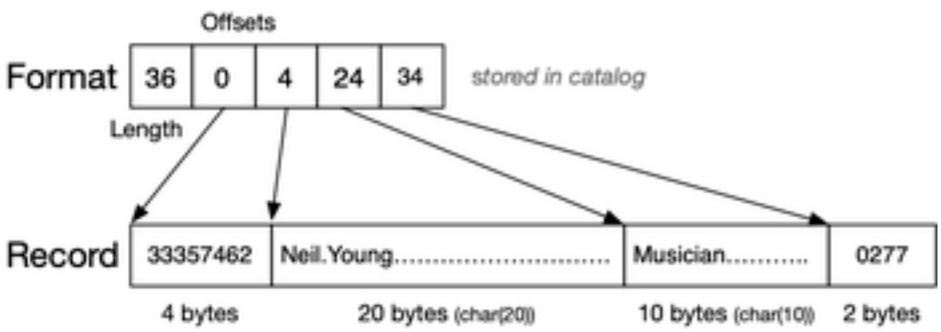
- extract the `i`'th field from a `Tuple` as a value of type *Typ*

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.

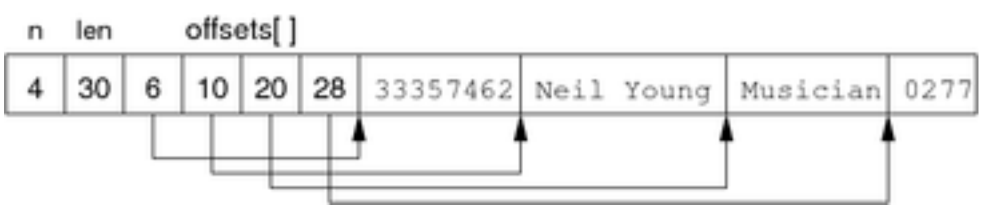
Variable-length Records

Possible encoding schemes for variable-length records:

- Prefix each field by length

- Terminate fields by delimiter

- Array of offsets



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(`*n*`)`	`int[]` (?)
`VARCHAR(`*n*`)`	`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

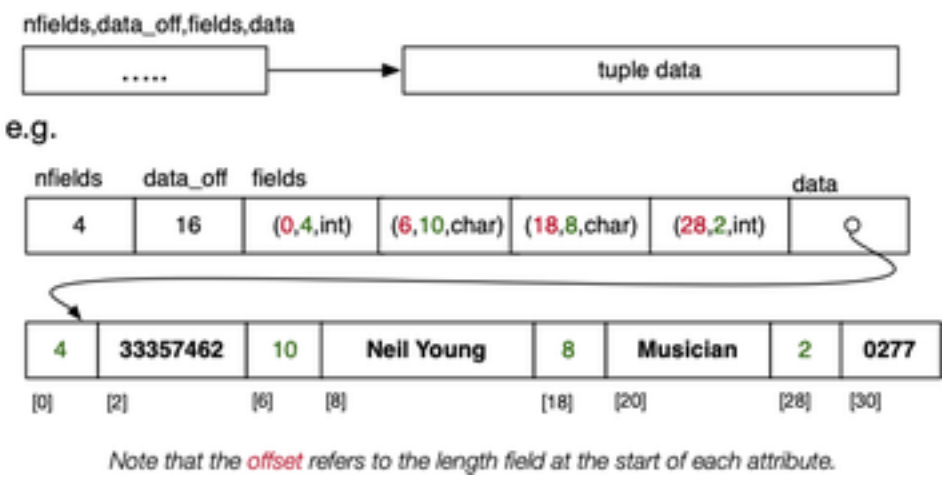
```
typedef struct {
  ushort    nfields;   // number of fields/attrs
  ushort    data_off;  // offset in struct for data
  FieldDesc fields[];  // field descriptions
  Record    data;      // pointer to record in buffer
} Tuple;
```

Fields are derived from relation descriptor + record instance data.

... Field Descriptors

Tuple `data` could be

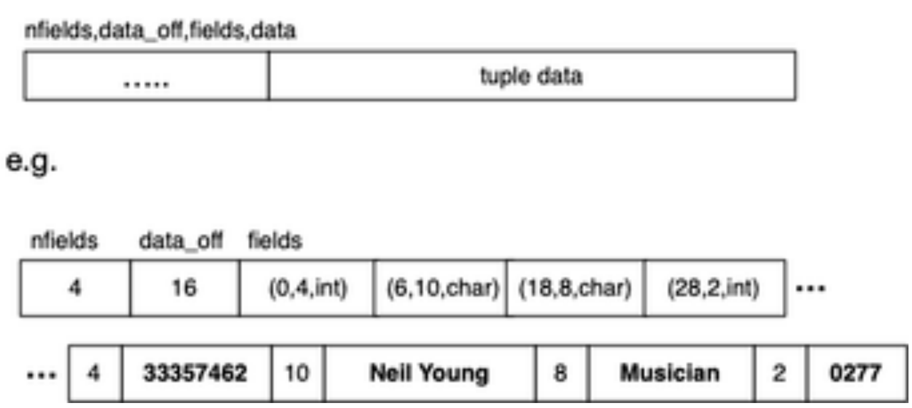
- a pointer to bytes stored elsewhere in memory



... Field Descriptors

Or, tuple `data` could be ...

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



Exercise 1: 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)

PostgreSQL Tuples

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

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

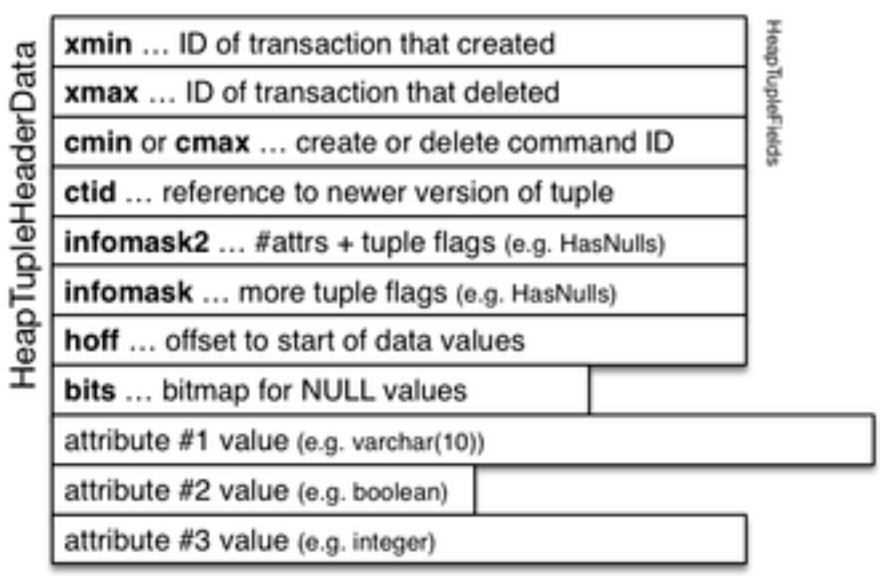
- `HeapTuple heap_form_tuple(desc,values[],isnull[])`
- `heap_deform_tuple(tuple,desc,values[],isnull[])`

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

Tuple structure:



... PostgreSQL Tuples

Tuple-related data types: (cont)

```
// 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        tdhasoid;    // 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

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 Tuples

`HeapTupleData` contains information about a stored tuple

```
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

PostgreSQL stores a single block of data for tuple

- containing a tuple header, followed by data `byte[]`

```
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

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 `NULL`s is stored in t_bits[] array

... PostgreSQL Tuples

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 Attribute Values

Values of attributes in PostgreSQL tuples are packaged as `Datum`s

```
// 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

Attribute values can be extracted as `Datum` from `HeapTuple`s

```
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

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))
```

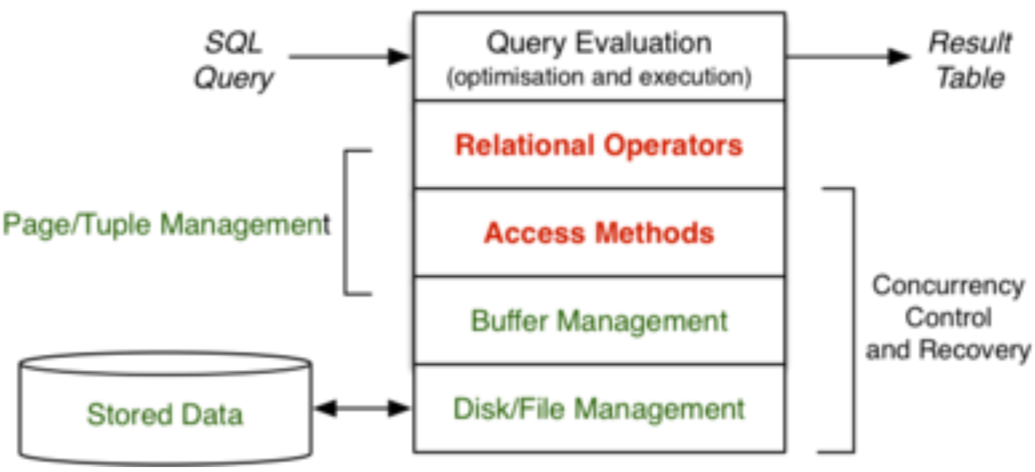
For details, see `include/postgres.h`

Implementing Relational Operations

DBMS Architecture (revisited)

28/84

Implementation of relational operations in DBMS:



Relational Operations

29/84

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

30/84

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)
- e.g. two-dimensional range query on R-tree indexed file

As well as query costs, consider update costs (insert/delete).

... Relational Operations

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

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

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!!

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

... Cost Models

In developing cost models, we also assume:

- 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
- the tuples which answer query q are contained in b_q pages
- data is transferred disk↔memory in whole pages
- cost of disk↔memory transfer $T_{r/w}$ is very high

... Cost Models

36/84

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 ≅ 4 000 000 people
- Average household size ≅ 3 ∴ 1 300 000 households
- Let's say that 1 in 10 households owns a piano
- Therefore there are ≅ 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 ≅ 130000/2/500 = 130 tuners

Actual number of tuners in Yellow Pages = 120

Example borrowed from Alan Fekete at Sydney University.

Query Types

37/84

Type	SQL	RelAlg	a.k.a.
Scan	`select * from R`	*R*	-
Proj	`select` *x,y* `from R`	*Proj[x,y]R*	-
Sort	`select * from R order by` *x*	*Sort[x]R*	*ord*
Sel_1	`select * from R where id =` *k*	*Sel[id=k]R*	*one*
Sel_n	`select * from R where a =` *k*	*Sel[a=k]R*	-
$Join_1$	`select * from R,S where R.id = S.r`	*R Join[id=r] S*	-

Different query classes exhibit different query processing behaviours.

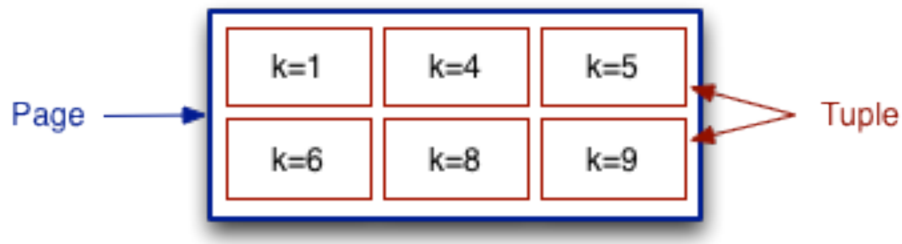
Example File Structures

38/84

When describing file structures

- use a large box to represent a *page*
- use either a small box or tup_i (or rec_i) to represent a *tuple*

- sometimes refer to tuples via their *key*
    - mostly, *key* corresponds to the notion of "primary key"
    - sometimes, *key* means "search key" in selection condition



... Example File Structures

Consider three simple file structures:

- *heap file* ... tuples added to any page which has space
- *sorted file* ... tuples arranged in file in key order
- *hash file* ... tuples placed in pages using hash function

All files are composed of *b* primary blocks/pages

Some records in each page may be marked as "deleted".

Exercise 2: 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)

Scanning

Scanning

Consider the query:

```
select * from Rel;
```

Operational view:

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

```
    }
}
```

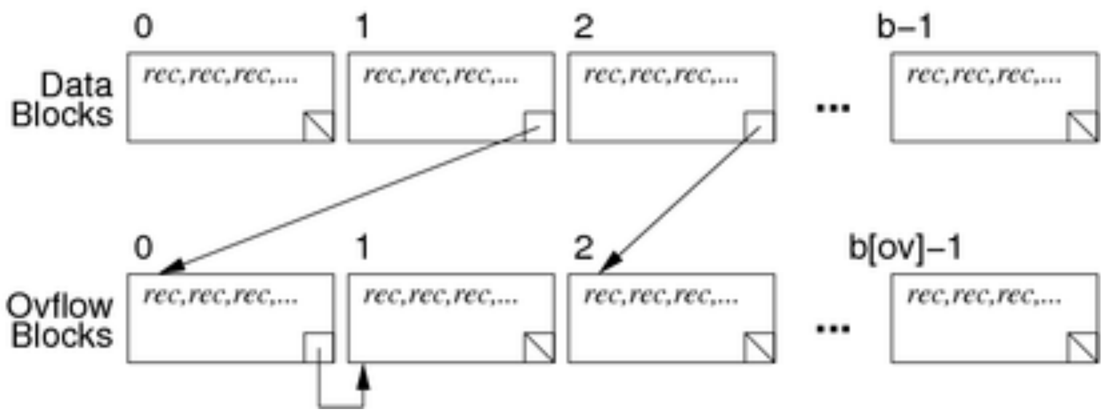
Cost: read every data page once

Time Cost = b.T_r*,*   *Page Cost = b*

... Scanning

43/84

Scan implementation when file has overflow pages, e.g.



... Scanning

44/84

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

$Cost = b + b_{Ov}$

where b_{Ov} = total number of overflow pages

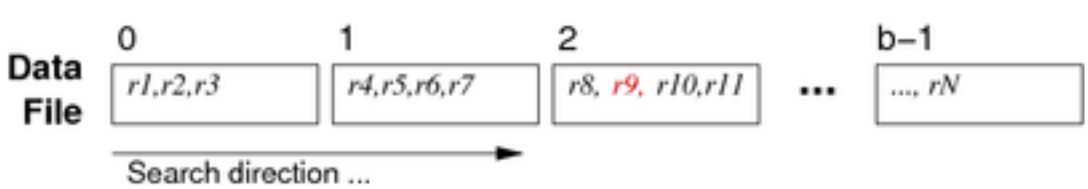
Selection via Scanning

45/84

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

46/84

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

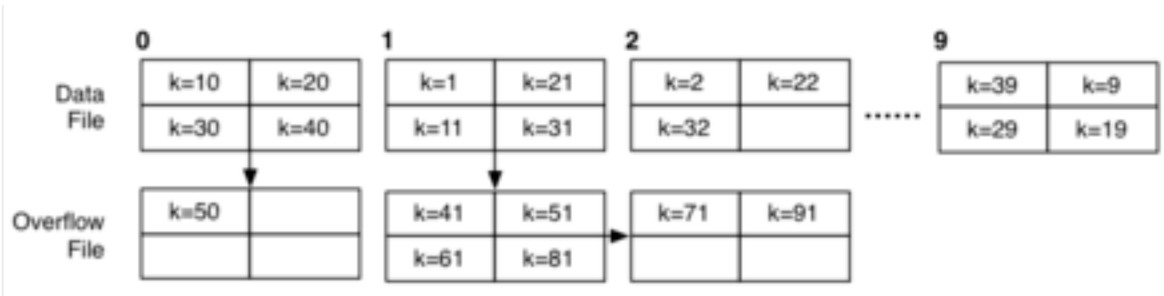
- best case: read one page, find tuple
- worst case: read all *b* pages, find in last (or don't find)
- average case: read half of the pages (*b/2*)

Page Costs: $Cost_{avg} = b/2$ $Cost_{min} = 1$ $Cost_{max} = b$

Exercise 3: Cost of Search in Hashed File

47/84

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)

Iterators

48/84

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

49/84

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);
}
```

Exercise 4: Implement next_tuple()

Consider the following possible **`Scan`** data structure

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

Assume tuples are indexed 0..`nTuples(p)`

Assume pages are indexed 0..`nPages(rel)`

Implement the **`Tuple next_tuple(Scan)`** function

P.S. What's in a `Relation` object?

Relation Copying

Consider an SQL statement like:

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

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

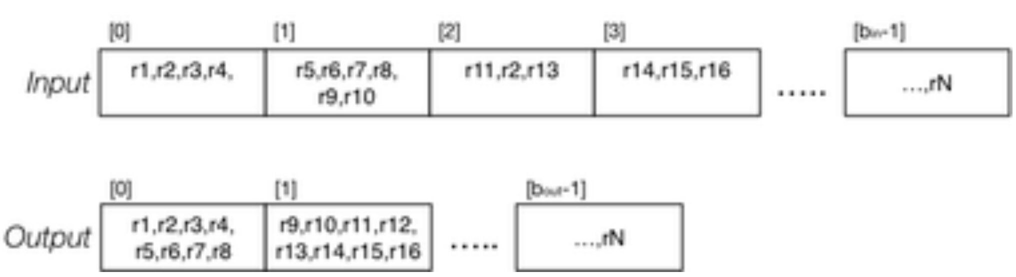
Process:

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

... Relation Copying

Possible that `T` is smaller than `S`

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



... Relation Copying

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);
```


Exercise 5: 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

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

```
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_npages;    // number of pages to scan
  PageNumber    rs_startpage; // page # to start at
  ...
```

```
  // scan current state, initally 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

57/84

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

59/84

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`

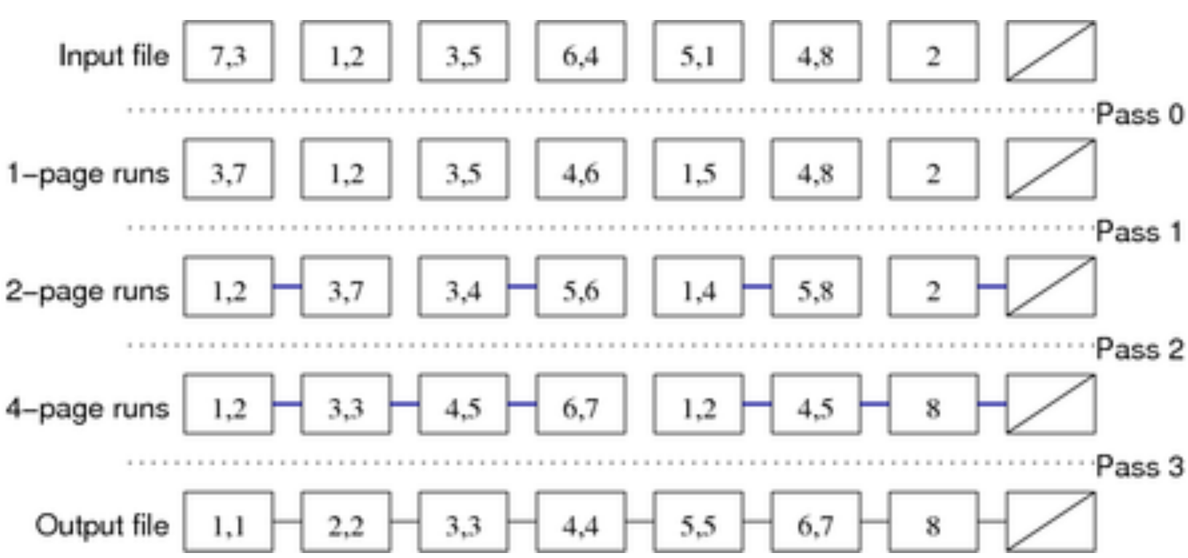
Sort methods such as quicksort are designed for in-memory data.

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

Two-way Merge Sort

60/84

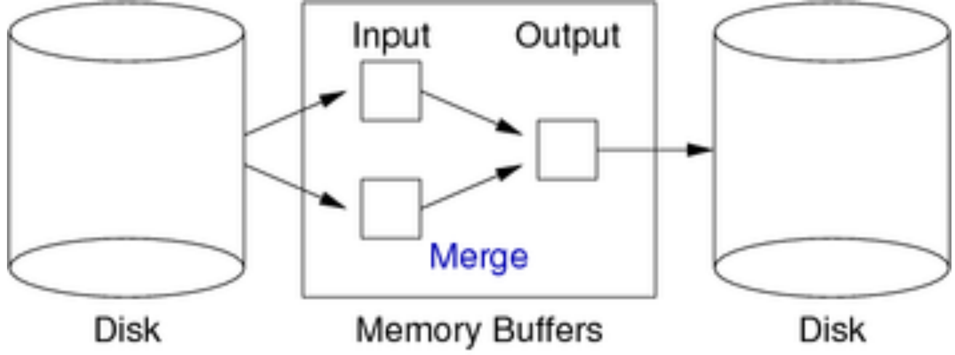
Example:



... Two-way Merge Sort

61/84

Requires three in-memory buffers:



Assumption: cost of Merge operation on two in-memory buffers $\cong 0$.

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

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 $ceil(log_2 b)$ passes to sort,
- each pass requires *b* page reads, *b* page writes

Gives total cost: $2.b.ceil(log_2 b)$

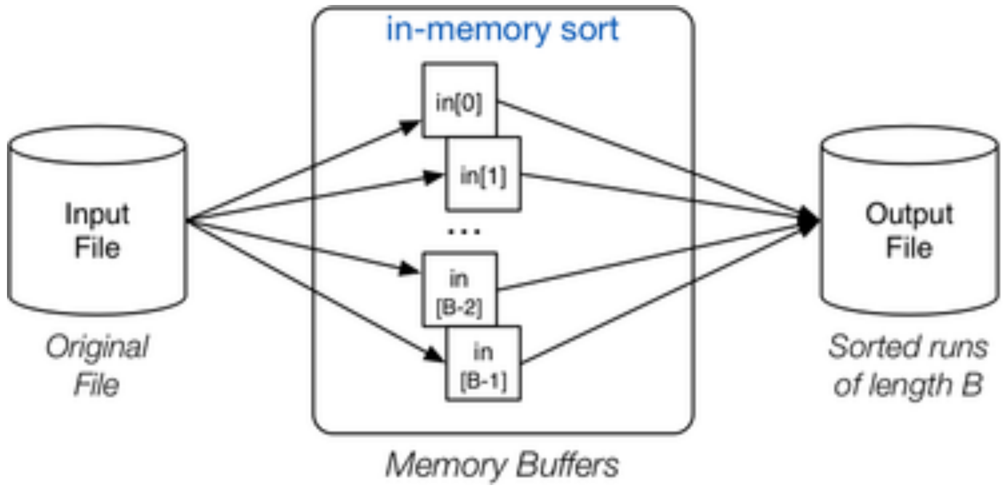
Example: Relation with $r=10^5$ and $c=50$ $\Rightarrow$ $b=2000$ pages.

Number of passes for sort: $ceil(log_2 2000) = 11$

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

n-Way Merge Sort

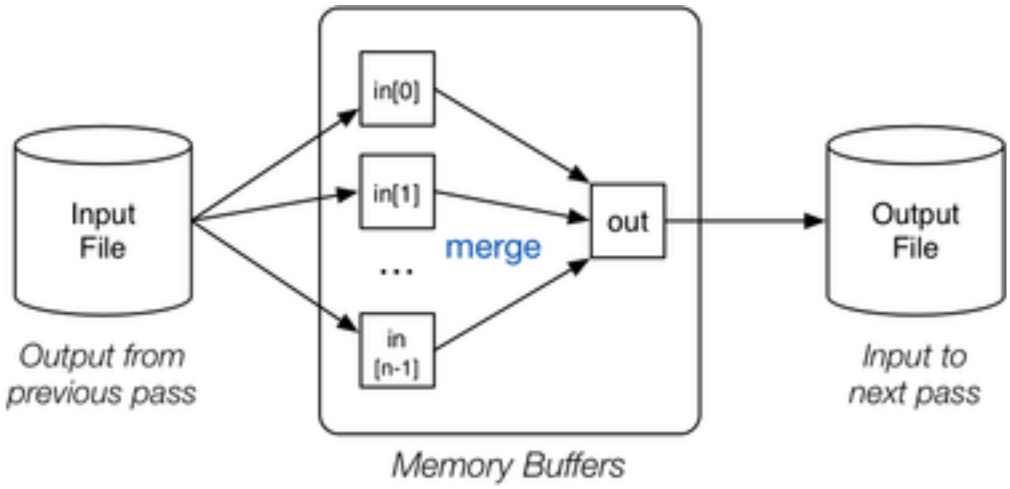
Initial pass uses: *B* total buffers



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

... n-Way Merge Sort

Merge passes use: *n* input buffers,   *1* output buffer



... n-Way Merge Sort

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 = ⌈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 = ⌈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

Generalising from previous example ...

For b data pages and B buffers

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

$Cost = 2.b.(1 + \lceil \log_n b_0 \rceil)$, where b_0 and n are defined above

---

Exercise 6: 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
- D = 1 presence bit per tuple in page directory
- 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

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

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" ≅ sorted run)

Implementation of "tapes": backend/utils/sort/logtape.c

... Sorting in PostgreSQL

Sorting comparison operators are obtained via catalog (in *`Type`*`.o`):

```
// gets pointer to function via pg_operator
struct Tuplesortstate { ... SortTupleComparator ... };

// 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()`

Implementing Projection

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

The projection operation needs to:

1. scan the entire relation as input
   - already seen how to do scanning
2. remove unwanted attributes in output tuples
   - 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   (if `distinct`)
   - two approaches: sorting or hashing

Sort-based Projection

Requires a temporary file/relation (`Temp`)

```
for each tuple T in Rel {
    T' = mkTuple([attrs],T)
    write T' to Temp
}

sort Temp on [attrs]

for each tuple T in Temp {
    if (T == Prev) continue
    write T to Result
    Prev = T
}
```


Exercise 7: Cost of 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 `R` tuples (i.e. c_R=3), 6 `T` tuples (i.e. c_T=6)

Show how sort-based projection would execute this statement.

Cost of Sort-based Projection

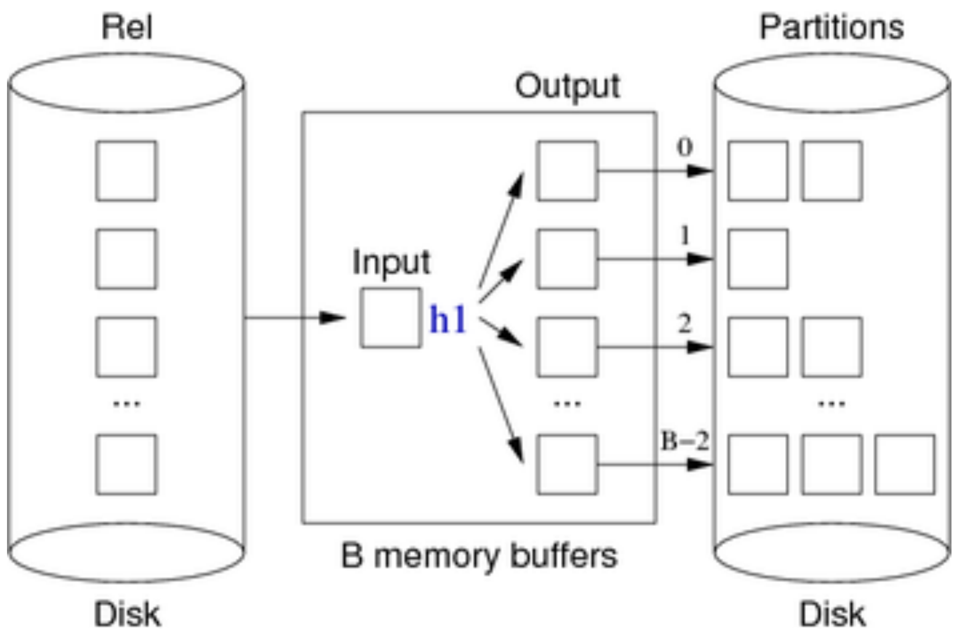
The costs involved are (assuming $B=n+1$ buffers for sort):

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

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

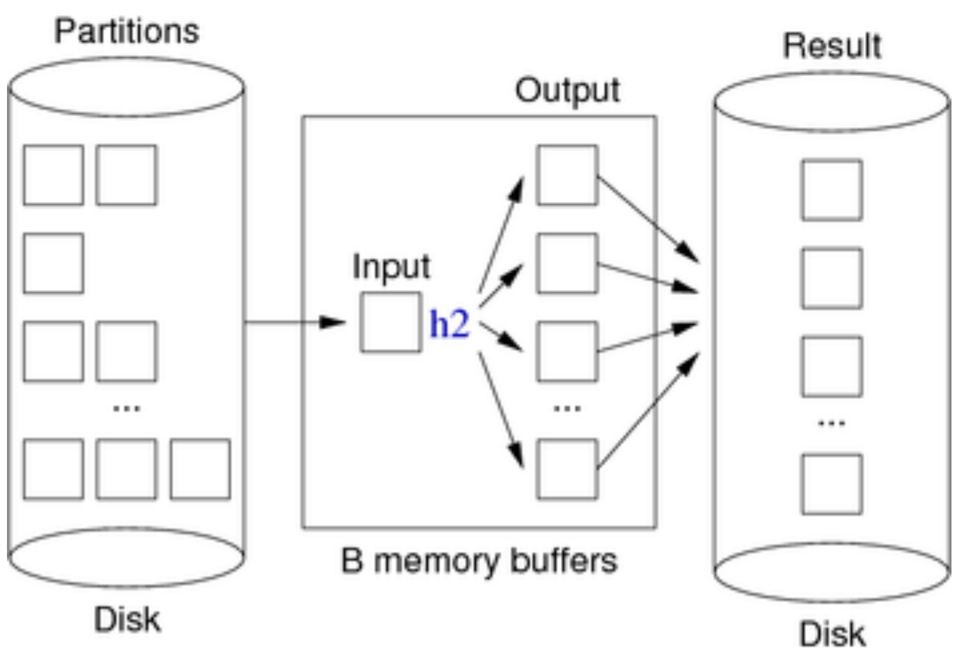
Hash-based Projection

Partitioning phase:



... Hash-based Projection

Duplicate elimination phase:



... Hash-based Projection

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
}
```

Exercise 8: Cost of 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 `R` tuples (i.e. $c_R=3$), 4 `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.

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}

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
