Scan, Sort, Project

Implementing Relational Operations

Relational Operations

DBMS core = relational engine, with implementations of

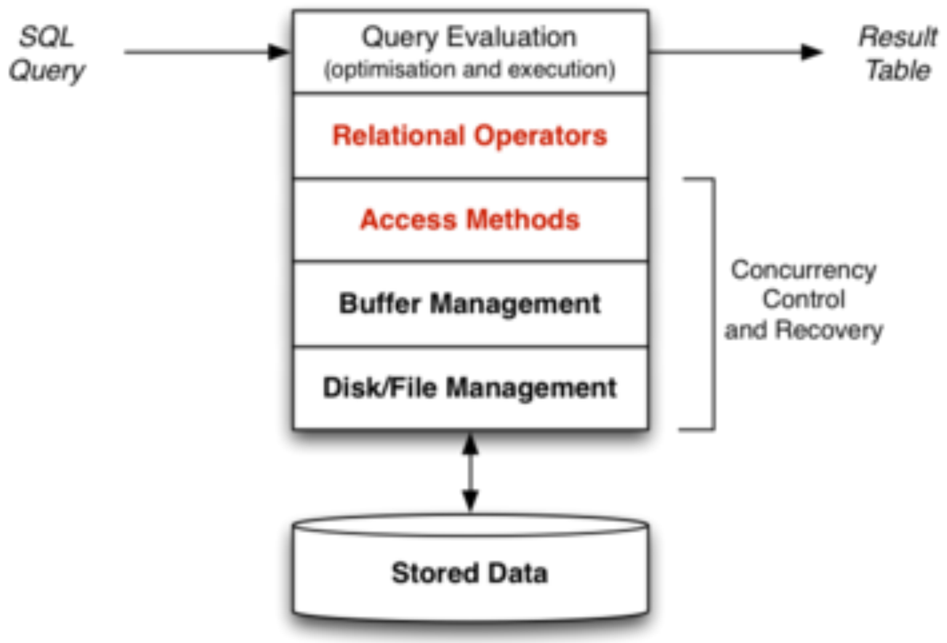
- selection, projection, join, set operations
- scanning, sorting, grouping, aggregation, ...

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

... Relational Operations

Implementation of relational operations in DBMS:



... Relational Operations

All relational operations return a set of tuples.

Can represent a typical operation programmatically as:

```
ResultSet = {}  // initially an empty set
while (t = nextRelevantTuple()) {
   // format tuple according to projection
   t' = formatResultTuple(t,Projection)
   // add next relevant tuple to result set
   ResultSet = ResultSet ∪ t'
}
return ResultSet
```

All of the hard work is in the `nextRelevantTuple()` function.

... Relational Operations

`nextRelevantTuple()` for *selection* operator:

- find next possible result tuple in table
- check whether it satisfies selection condition

`nextRelevantTuple()` for *join* operator:

- find next possible pair of tuples from tables
- check whether pair satisfies join condition

Two ways to handle the `ResultSet`

- build the complete `ResultSet` and then return it
- return each tuple as produced (tuple-by-tuple interface)

... Relational Operations

There are three "dimensions of variation" in this system:

- relational operators (e.g. Sel, Proj, Join, Sort, ...)
- file structures (e.g. heap, indexed, hashed, ...)
- query processing methods (e.g. merge-sort, hash-join, ...)

We consider combinations of these, e.g.

- selection with 0/1 matching tuples on hashed/indexed file
- sort-merge join on ordered heap files
- 2-dimensional range query on an R-tree-indexed file

Also consider updates (insert/delete) on file structures.

Query Types

Queries fall into a number of classes:

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*

Different query classes exhibit different query processing behaviours.

... Query Types

Type	SQL	RelAlg	a.k.a.
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*	-
	`select * from R`	*Sel[a=j ∧ b=k]R*	*pmr*

Sel_{pmr}	`where a=`*j* `and b=`*k*		
$Range_{1d}$	`select * from R` `where a>`*j* `and a<`*k*	*Sel[a>j ∧ a<k]R*	*rng*
$Range_{nd}$	`select * from R` `where a>`*j* `and a<`*k* `and b>`*m* `and b<`*n*	*Sel[...]R*	*space*

... Query Types

Type	**SQL**	**RelAlg**	**a.k.a.**

$Join_1$	`select * from R,S`
	`where R.id = S.r`
		R Join[id=r] S	-

EquiJoin	`select * from R,S` `where R.`*v*`=S.`*w* `and R.`*x*`=S.`*y*	*R Join[v=w ∧ x=y] S*	-
ThetaJoin	`select * from R,S` `where R.`*x op* `S.`*y*	*R Join[...] S*	-
Similar	`select * from R` `where R.* ≅` *Object*	*R ≅ Obj*	sim

Cost Models

Cost Models

An important aspect of this course is

- analysis of cost of various query methods

Won't be using asymptotic complexity (*O(n)*) for this

Rather, we attempt to develop cost models

- for a each query method, over a range of query types
- using a (simplified) model of the behaviour of the DBMS

Cost is measured in terms of number of page reads/writes.

... Cost Models

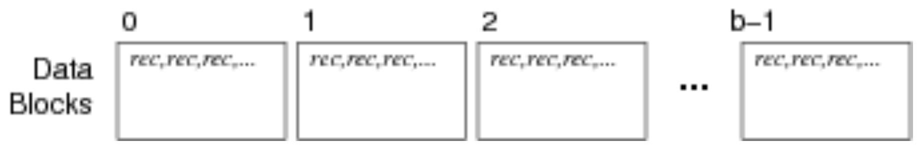
Assumptions in our cost models:

- memory (RAM) is "small", fast, byte-at-a-time
    - e.g. 1GB size, 10^{-7} secs to compare tuples
    - all computation is performed on data loaded into memory
- disk storage is very large, slow, page-at-a-time
    - e.g. 1TB size, 10^{-2} secs to read/write a 4KB page
    - cost of processing a page is 10^{-3} cost of reading a page
- every request to read/write a page results in a read/write
    - no effective buffer-pooling ... 1 memory buffer per relation
    - however, we sometimes consider multiple buffers explicitly

... 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 highest cost in system



... Cost Models

Typical values for measures used in cost models:

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
process page in memory	-	$\cong 0$
# pages containing answers for query q	b_q	≥ 0

... Cost Models

With buffer pool, `request_page()` does not necessarily involve reading

Instead, we assume no buffer pool (worst-case cost analysis)

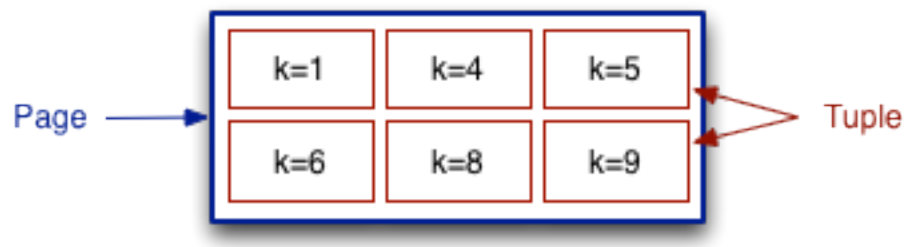
Use either `readPage()` or `get_page()` to get data

```
// Assume data types for Relation, Page

get_page(Relation r, int pid, Page buf)
{
    buf = readPage(r.file, pid);
}

Page readPage(File f, int pid)
{
        Page buf = newPageBuffer();
        lseek(f, pid*PAGE_SIZE, SEEK_SET);
        read(f, buf, PAGE_SIZE);
        return buf;
}
```

Example file structures

When describing file structures

- use a large box to represent a *page*
- sometimes use a small box to represent a *tuple*
- sometimes refer to tuples as rec_i
- sometimes ref to tuples via their *key*
    - mostly, *key* corresponds to the notion of "primary key"
    - sometimes, *key* means "search key" in selection condition

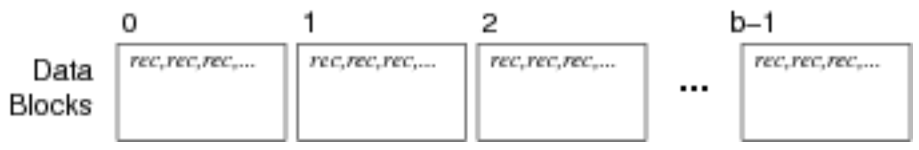


... Example file structures

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

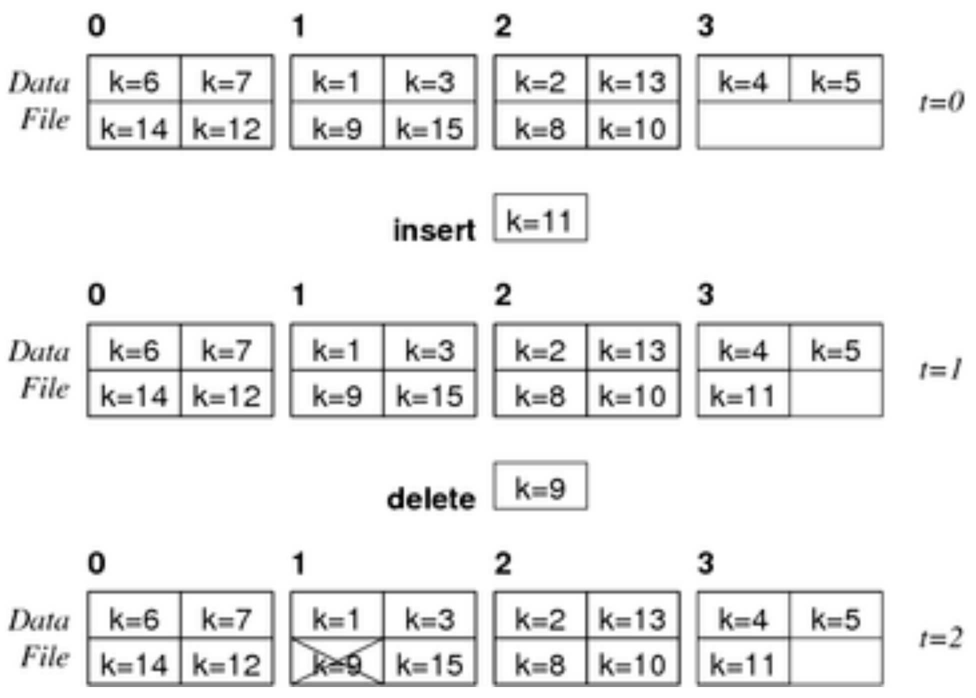
All files are composed of *b* primary blocks/pages



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

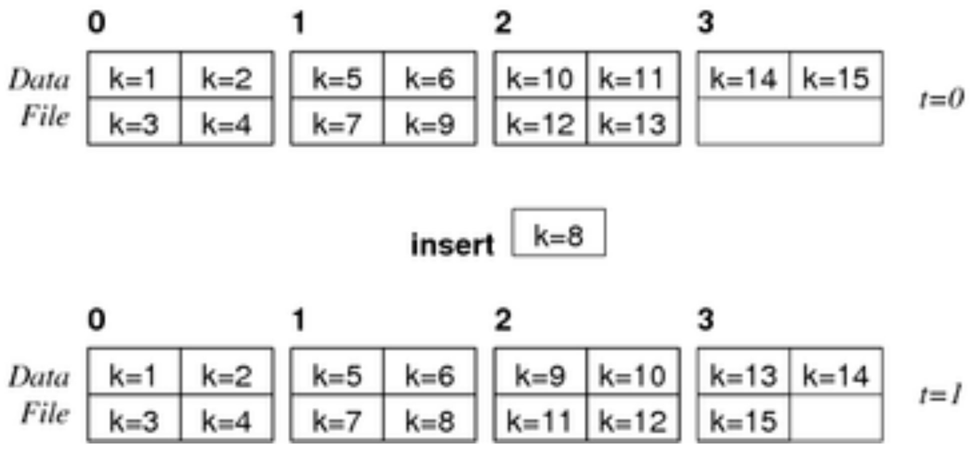
... Example file structures

Heap file with *b = 4, c = 4*:



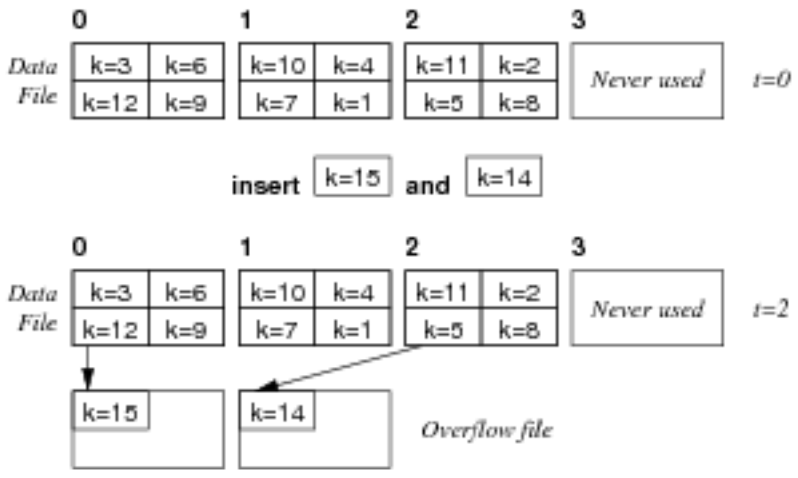
... Example file structures

Sorted file with *b = 4, c = 4*:



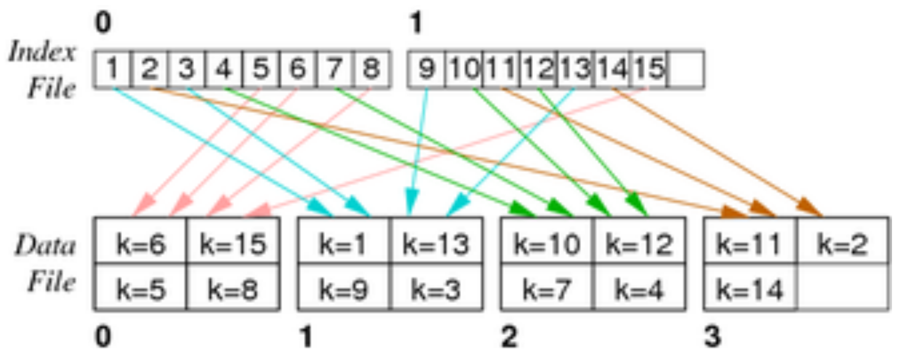
... Example file structures

Hashed file with *b = 3, c = 4, h(k) = k%3*



... Example file structures

Indexed file with $b = 4, c = 4, b_i = 2, c_i = 8$:



Scanning

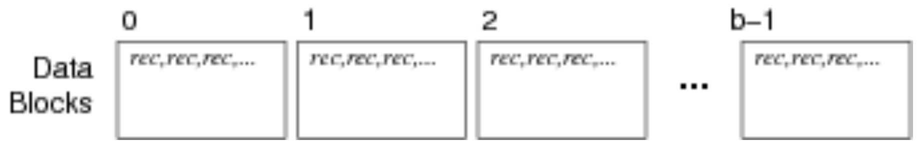
Scanning

Consider the query:

```
select * from T;
```

Conceptually:

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



... Scanning

Implemented via iteration over file containing T:

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

Cost: read every data page once

Cost = b.T_r

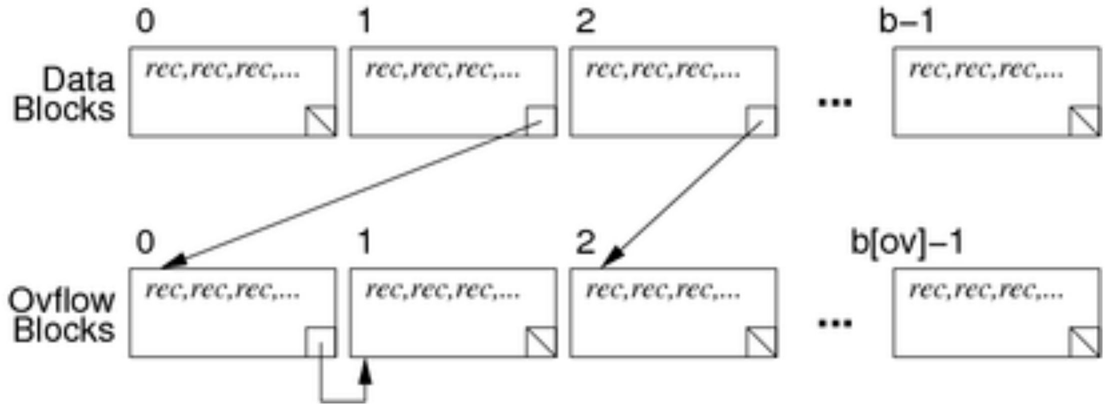
... Scanning

In terms of file operations:

```
// implementation of "select * from T"

File inf;   // data file handle
int p;      // input file page number
Buffer buf; // input file buffer
int i;      // current record in input buf
Tuple t;    // data for current record

inf = openFile(fileName("T"), READ)
for (p = 0; p < nPages(inf); p++) {
    buf = readPage(inf,p);
    for (i = 0; i < nTuples(buf); i++) {
        t = getTuple(buf,i);
        add t to result set
}   }
```

... Scanning

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



... Scanning

In this case, the implementation changes to:

```
for each page P in file of relation T {
    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 and overflow page once

Cost = $(b + b_{Ov}).T_r$

where b_{Ov} = total number of overflow pages

... Scanning

In terms of file operations:

```
// implementation of "select * from T"

File inf;   // data file handle
File ovf;   // overflow file handle
int p;      // input file page number
int ovp;    // overflow file page number
Buffer buf; // input file buffer
int i;      // current record in input buf
Tuple t;    // data for current record

inf = openFile(fileName("T"), READ)
ovf = openFile(ovFileName("T"), READ)
for (p = 0; p < nPages(inf); p++) {
    buf = readPage(inf,p);
    for (i = 0; i < nTuples(buf); i++) {
        t = getTuple(buf,i);
        add t to result set
    }
    ovp = ovflow(buf);
    while (ovp != NO_PAGE) {
        buf = readPage(ovf,ovp);
        for (i = 0; i < nTuples(buf); i++) {
            t = getTuple(buf,i);
            add t to result set
        }
        ovp = ovflow(buf);
    }
}
```

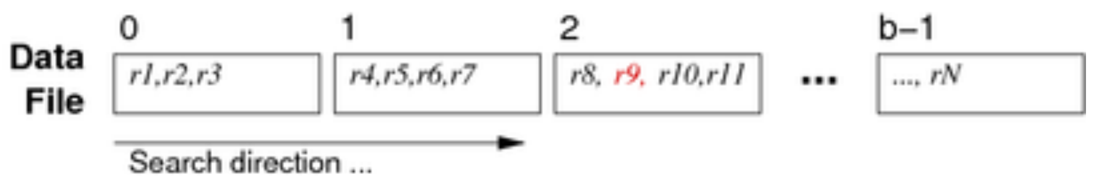
Cost: read data+ovflow page   *Cost* = $(b+b_{ov}).T_r$

Selection via Scanning

Consider a *one* query like:

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

In an unordered file, search for matching record requires:



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

... Selection via Scanning

In terms of file operations (assuming var delcarations as before):

```
inf = openFile(fileName("Employee"), READ);
for (p = 0; p < nPages(inf); p++)
    buf = readPage(inf,p);
    for (i = 0; i < nTuples(buf); i++) {
        t = getTuple(buf,i);
        if (getField(t,"id") == 762288)
           return t;
}   }
```

For different selection condition, simply replace `(getField(t,"id")==762288)`

... Selection via Scanning

Cost analysis for *one* searching in unordered file

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

Assumptions:

- negligible cost for scanning tuples in page
- negligible cost for checking condition on each record

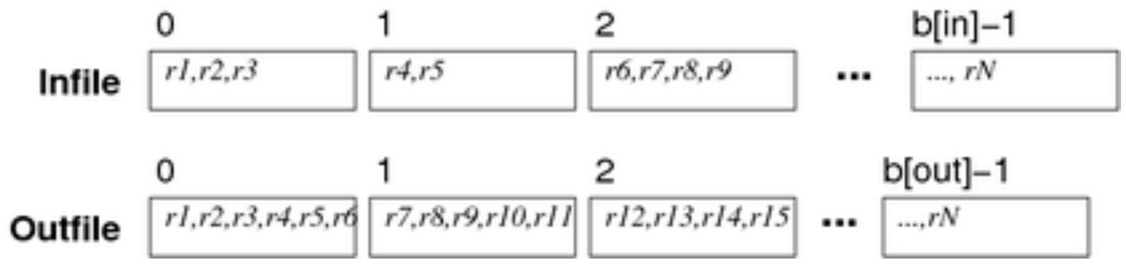
$Cost_{avg} = T_r b/2 \quad Cost_{min} = T_r \quad Cost_{max} = T_r b$

File Copying

Consider an SQL statement like:

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

Effectively, copies data from one file to another.



Conceptually:

```
make empty relation T
for each tuple t in relation S {
    append tuple t to relation T
}
```

... File Copying

In terms of previously defined relation/page/tuple operations:

```
Relation in;        // relation handle (incl. files)
Relation out;       // relation handle (incl. files)
int ipid,opid;      // input/output page indexes
int tid;            // record/tuple index on current page
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++) {
    get_page(in, ipid, ibuf);
    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 1: 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

Iterators

Higher-levels of DBMS are given a view of scanning as:

```
cursor = initScan(relName,condition);
while (tup = getNextTuple(cursor)) {
        process tup
}
endScan(cursor);
```

Also known as *iterator*.

... Iterators

Implementation of simple scan iterator (via file operations):

```
typedef struct {
    File   inf;    // data file handle
    Buffer buf;    // input buffer
    int    curp;   // current page number
    int    curi;   // current record number
    Expr   cond;   // representation of condition
} Cursor;
```

... Iterators

Implementation of simple scan iterator (continued):

```
Cursor *initScan(char *rel, char *cond)
{
    Cursor *c;
    c = malloc(sizeof(Cursor));
    c->inf = openFile(fileName(rel),READ);
    c->buf = readPage(c->inf,0);
    c->curp = 0;
    c->curi = 0;
    c->cond = makeTestableCondition(cond);
    return c;
}
void endScan(Course *c)
{
    closeFile(c->inf);
    freeExpr(c->cond);
    free(c);
}
```

... Iterators

Implementation of simple scan iterator (continued):

```
Tuple getNextTuple(Cursor *c)
{
getNextTuple:
    if (c->curi < nTuples(c->buf))
        return getTuple(c->buf, c->curi++);
    else {
        // no more tuples in this page; get next page
        c->curp++;
        if (c->curp == nPages(c->inf))
            return NULL;  // no more pages
        else {
            c->buf = readPage(c->inf,c->curp);
            c->curi = 0;
            goto getNextTuple;
        }
    }
}
```

... Iterators

Implementation of full iterator interface via file operations:

```
typedef struct {
    File   inf;   // data file handle
    File   ovf;   // overflow file handle
    Buffer buf;   // input buffer
    int    curp;  // current page number
    int    curop; // current ovflow page number
    int    curi;  // current record number
    Expr   cond;  // representation of condition
} Cursor;
```

... Iterators

Implementation of full iterator interface (continued):

```
Cursor *initScan(char *rel, char *cond)
{
    Cursor *c;

    c = malloc(sizeof(Cursor));
    c->inf = openFile(fileName(rel),READ);
    c->ovf = openFile(ovFileName(rel),READ);
    c->buf = readPage(c->inf,0);
    c->curp = 0;
    c->curop = NO_PAGE;
    c->curi = 0;
    c->cond = makeTestableCondition(cond)
    return c;
}
void endScan(Course *c)
{
    closeFile(c->inf);
    if (c->ovf) closeFile(c->ovf);
    freeExpr(c->cond);
    free(c);
}
```

... Iterators

Implementation of scanning interface (continued):

```
Tuple getNextTuple(Cursor *c)
{
getNextTuple:
    if (c->curi < nTuples(c->buf))
        return getTuple(c->buf, c->curi++);
    else {
        // no more tuples in this page; get next page
        if (c->curop == NO_PAGE) {
            c->curop = ovflow(c->buf);
            if (c->curop != NO_PAGE) {
                // start ovflow chain scan
getNextOvPage:
                c->buf = readPage(c->ovf,c->curop);
                c->curi = 0;
                goto getNextTuple;
            }
            else {
getNextDataPage:
                c->curp++;
                if (c->curp == nPages(c->inf))
                    return NULL;  // no more pages
                else {
                    c->buf = readPage(c->inf,c->curp);
                    c->curi = 0;
                    goto getNextTuple;
                }
            }
        }
        else {
            // continue ovflow chain scan
            c->curop = ovflow(c->buf);
            if (c->curop == NO_PAGE)
                goto getNextDataPage;
            else
                goto getNextOvPage;
        }
    }
}
```

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)`**

  ... uses **`initscan()`** to do half the work (shared with rescan)
- **`tup = heap_getnext(scan, direction)`**

  ... uses **`heapgettup()`** to do most of the work
- **`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 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

Above examples are for *heap* files

- simple, unordered, no index, no hashing

Other access file structures in PostgreSQL:

- **`btree`**, **`hash`**, **`gist`**, **`gin`**
- each implements:
  - startscan, getnext, endscan
  - insert, delete
  - other file-specific operators

Sorting

The Sort Operation

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

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

More important, sorting is used internally in other operations:

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

External Sorting

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

For data on disks, need *external sorting* techniques.

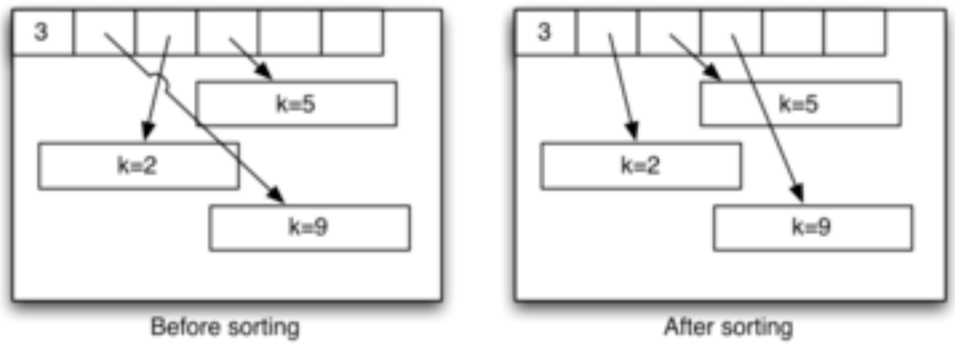
The standard external sorting method (*merge sort*) works by

- reading pages of data into memory buffers
- use in-memory sort to order items within buffers
- merging sorted buffers to produce output
- possibly requiring multiple passes over the data

... External Sorting

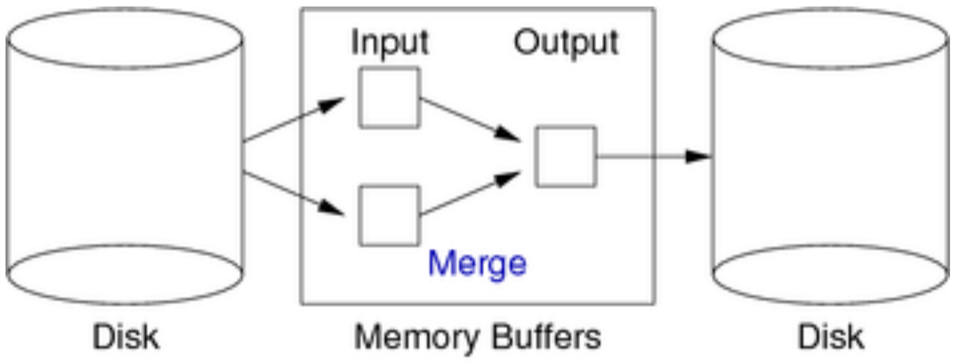
Sorting tuples within pages

- need to extract sort key from each tuple
- no need to physically move tuples
- simply swap entries in page directory



Two-way Merge Sort

Requires three in-memory buffers:



Assumption: cost of merge on two buffers ≃ *0*.

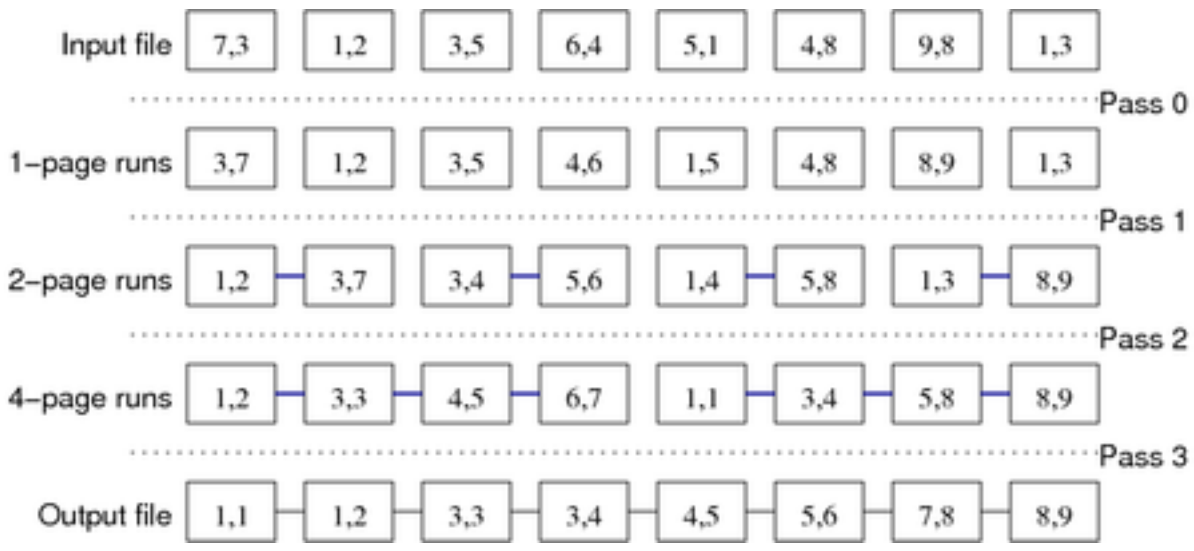
... Two-way Merge Sort

Two-way merge-sort method:

```
read each page into buffer, sort it, write it
numberOfRuns = b; runLength = 1;
while (numberOfRuns > 1) {
    for each pair of adjacent runs {
        merge the pair of runs to output, by
          - read pages from runs into input
              buffers, one page at a time
          - apply merge algorithm to transfer
              tuples to output buffer
          - flush output buffer when full and
              when merge finished
   }
   numberOfRuns = numberOfRuns / 2
   runLength = runLength * 2
}
```

... Two-way Merge Sort

Example:



... Two-way Merge Sort

Two-way merge-sort method (improved):

```
numberOfRuns = b; runLength = 1;
while (numberOfRuns > 1) {
    for each pair of adjacent runs {
        merge the pair of runs to output, by
          - read pages from runs into input
              buffers, one page at a time
          - if (runLength == 1)
              sort contents of each input buffer
          - apply merge algorithm to transfer
              tuples to output buffer
          - flush output buffer when full and
              when merge finished
   }
   numberOfRuns = numberOfRuns / 2
   runLength = runLength * 2
}
```

Avoids first pass to sort contents of individual pages.

... Two-way Merge Sort

Consider file where $b = 2^k$:

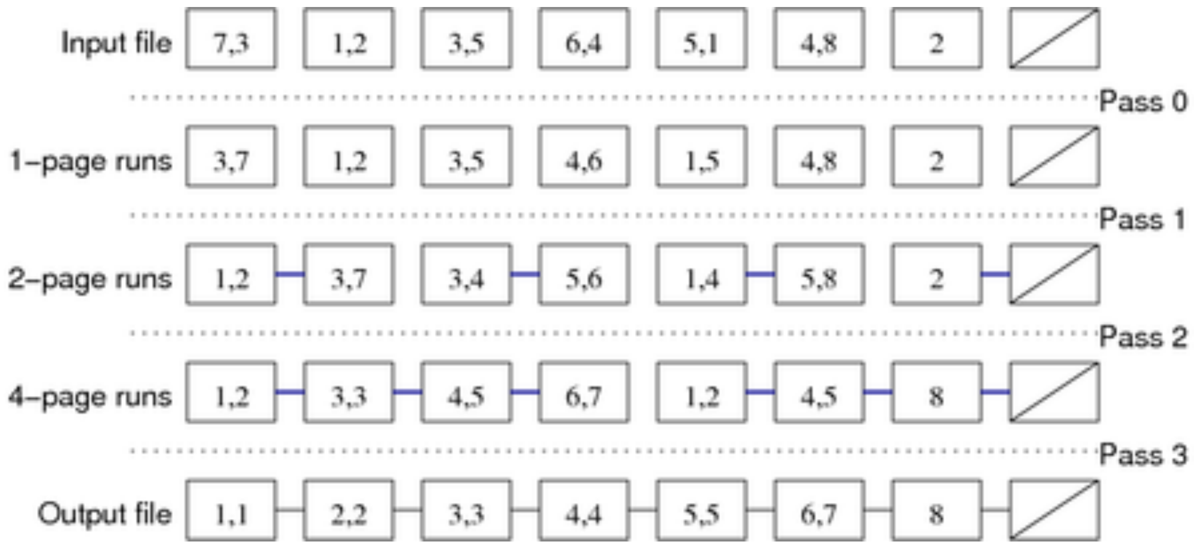
- pass 0 produces 2^k sorted runs of 1 page
- pass 1 produces 2^{k-1} sorted runs of 2 pages
- pass 2 produces 2^{k-2} sorted runs of 4 page
- and so on, until
- pass k produces *1* sorted run of 2^k pages

Method also works ok when

- $b \neq 2^k$ ... last run simply has less pages than others
- pages are not completely full (nextTuple() function)

... Two-way Merge Sort

Example:



Merging Two Sorted Pages

Method using operations on files and buffers:

```
// Pre:  buffers B1,B2; outfile position op
// Post: tuples from B1,B2 output in order
i1 = i2 = 0; clear(Out);
R1 = getTuple(B1,i1); R2 = getTuple(B2,i2);
while (i1 < nTuples(B1) && i2 < nTuples(B2)) {
    if (lessThan(R1,R2))
        { addTuple(R1,Out); i1++; R1 = getTuple(B1,i1); }
    else
        { addTuple(R2,Out); i2++; R2 = getTuple(B2,i2); }
    if (isFull(Out))
        { writePage(outf,op++,Out); clear(Out); }
}
for (i1=i1; i1 < nTuples(B1); i1++) {
    addTuple(getTuple(B1,i1), Out);
    if (isFull(Out))
        { writePage(outf,op++,Out); clear(Out); }
}
for (i2=i2; i2 < nTuples(B2); i2++) {
    addTuple(getTuple(B2,i2), Out);
    if (isFull(Out))
        { writePage(outf,op++,Out); clear(Out); }
}
if (nTuples(Out) > 0) writePage(outf,op,Out);
```

Merging Runs vs Merging Pages

In the above, we merged two input buffers.

In general, we need to merge sorted "runs" of pages.

The only difference that this makes to the above method:

```
R1 = getTuple(B1,i1);
```

becomes

```
if (i1 == nTuples(B1)) {
    B1 = readPage(inf,ip++); i1 = 0;
}
R1 = getTuple(B1,i1);
```

Comparison for Sorting

Above assumes that we have a function to compare tuples.

Mechanism needs to be generic, to handle all of:

```
select * from Employee order by eid;
select * from Employee order by name;
select * from Employee order by age;
```

Envisage a function `tupCompare(r1,r2,f)` (cf. C's `strcmp`)

- takes two tuples `r1`, `r2` and a field name `f`
- returns negative value if `r1.f < r2.f`
- returns positive value if `r1.f > r2.f`
- returns zero value if `r1.f == r2.f`

... 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 $\lceil log_2 b \rceil$ passes to sort,
- each pass requires *b* page reads, *b* page writes

Gives total cost: $2.b.\lceil log_2 b \rceil$

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

Number of passes for sort: $\lceil log_2 2000 \rceil = 11$

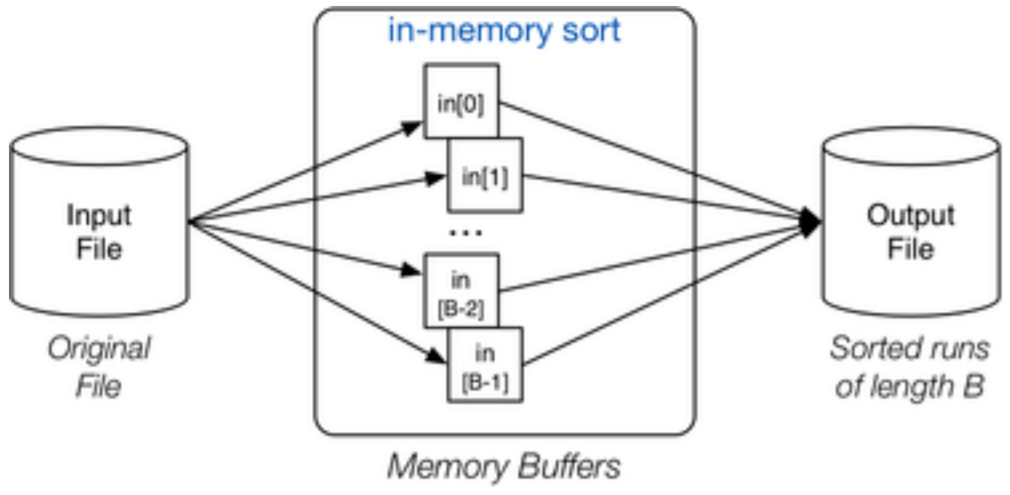
Reads/writes entire file 11 times! Can we do better?

---

n-Way Merge Sort

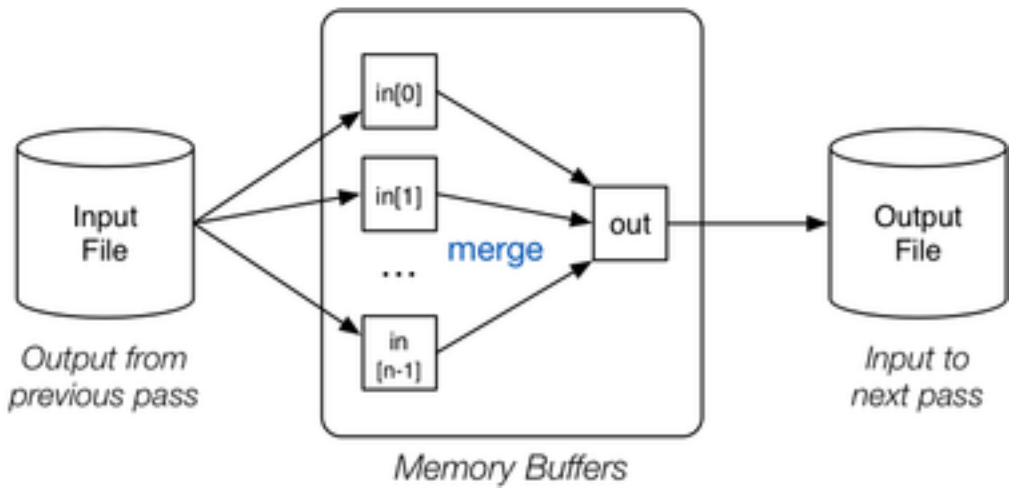
Initial pass uses: *B* total buffers

- read B pages into memory buffers
- sort tuples across all B pages in memory
- write out B-page-long run of sorted tuples



... 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 pages into memory buffers
    sort group in memory
    write pages out to Temp
}
// Merge runs until everything sorted
// n-way merge, where n=B-1
numberOfRuns = ⌈b/B⌉
while (numberOfRuns > 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
}
```

... n-Way Merge Sort

Method for merging n runs (n input buffers, 1 output buffer):

```
for i = 1..n {
   read first page of run[i] into a buffer[i]
   set current tuple cur[i] to first tuple in buffer[i]
}
while (more than 1 run still has tuples) {
   s = find buffer with smallest tuple as cur[i]
   copy tuple cur[i] to output buffer
   if (output buffer full) { write it and clear it}
   advance cur[i] to next tuple
   if (no more tuples in buffer[i]) {
      if (no more pages in run[i])
         mark run[i] as complete
      else {
         read next page of run[i] into buffer[i]
         set cur[i] to first tuple in buffer[i]
}  }  }
copy tuples in non-empty buffer to output
```

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 1
    - 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
- each pass reads and writes *b* pages (i.e. *2.b* page accesses)

$Cost = 2.b.(1 + \lceil \log_n b_0 \rceil)$, where $b_0 = \lceil b/B \rceil$

... Cost of n-Way Merge Sort

Costs (number of passes) for varying *b* and *B* (*n=B-1*):

b	*B=3*	*B=16*	*B=128*
100	7	2	1
1000	10	3	2
10,00	13	4	2
100,000	17	5	3
1,000,000	20	5	3

In the above, we assume that

- the first pass uses all *B* buffers as inputs
- subsequent merging passes use *n=B-1* input buffers, and one output buffer

Elapsed time could be reduced by double-buffering

- fill one output buffer while the other is being flushed to disk
- but this needs two output buffers => n-1-way merging, so maybe more merge passes

---

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 seven *N* buffers, one output buffer
- *N* = as many buffers as `workMem` allows

Many references to "tapes" since Knuth's original algorithm was described in terms of merging data from magnetic tapes.

Effectively, a "tape" is a 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 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

   (straightforward, whichever file organisation is used)

2. remove unwanted attributes in output

   (straightforward, manipulating internal record structure)

3. eliminate any duplicates produced

   (not as simple as other operations ...)

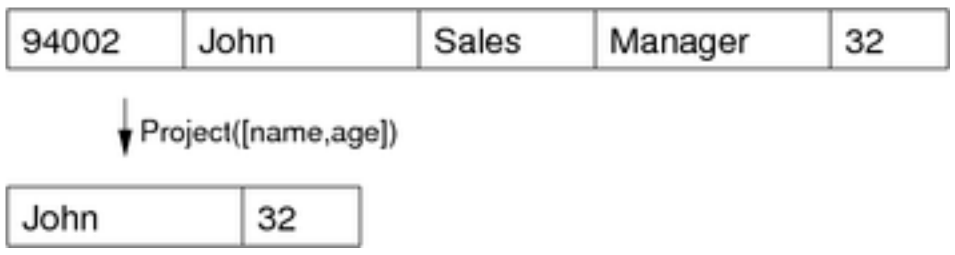
There are two approaches for task 3: sorting or hashing.

Removing Attributes

Projecting attributes involves creating a new tuple, using only some values from the original tuple.

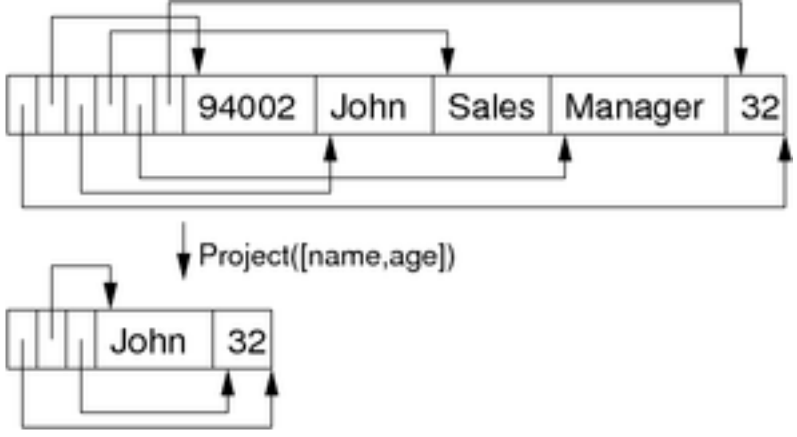
Precisely how to achieve this depends on tuple internals.

Removing attributes from fixed-length tuples:



... Removing Attributes

Removing attributes from variable-length tuples:



Sort-based Projection

Overview of the method:

1. Scan input relation `Rel` and produce a file of tuples containing only the projected attributes
2. Sort this file of tuples using the combination of all attributes as the sort key
3. Scan the sorted result, comparing adjacent tuples, and discard duplicates

Requires a temporary file/relation (`Temp`)

... Sort-based Projection

The method, in detail:

```
// Inputs: relName, attrList
inf = openFile(fileName(relName),READ);
tempf = openFile(tmpName,CREATE);
clear(outbuf); j = 0;
for (p = 0; p < nPages(inf); p++) {
    buf = readPage(inf,p);
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        newtup = project(tup,attrList);
        addTuple(newtup,outbuf);
        if (isFull(outbuf)) {
            writePage(tempf,j++,outbuf);
            clear(outbuf);
        }
    }
}
mergeSort(tempf);
```

(continued ...)

... Sort-based Projection

(... continued)

```
tempf = openFile(tmpName,READ);
outf = openFile(result,CREATE);
clear(outbuf); prev = EMPTY; j = 0;
for (p = 0; p < nPages(tempf); p++) {
    buf = readPage(tempf,p);
    for (i = 0; i < nTuples(buf); i++) {
        tup = getTuple(buf,i);
        if (tupCompare(tup,prev) != 0) {
            addTuple(tup,outbuf);
            if (isFull(outbuf)) {
                writePage(outf,j++,outbuf);
                clear(outbuf);
            }
            prev = tup;
        }
    }
}
```

Cost of Sort-based Projection

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

- scanning original relation `Rel`: b_R
- writing `Temp` relation: b_T
- sorting `Temp` relation: $2.b_T(1 + \lceil log_B b_0 \rceil)$ where $b_0 = \lceil b_T/B \rceil$
- removing duplicates from `Temp`: b_T
- writing the result relation: b_{Out}

Total cost = sum of above = $b_R + 2.b_T + 2.b_T(1 + \lceil log_B b_0 \rceil) + b_{Out}$

Note that we often ignore cost of writing the result; especially when comparing different algorithms for the same relational operation.

Improving Sort-based Projection

Some approaches for improving the cost:

- remove first stage; do projection during first phase of sort
- reduce sorting costs by:
  - using more memory buffers (but there is a limit)
  - eliminating duplicates during the merge phase
- minimise scanning cost by laying pages out on disk appropriately
  (generally, we don't have this luxury since the O/S handles it for us)

Hash-based Projection

Overview of the method:

1. Scan input relation `Rel` and produce a set of hash partitions based on the projected attributes
2. Scan each hash partition looking for duplicates
3. Once each partition is duplicate-free, write out the remaining tuples

The method requires:

- two different hash functions using all projected fields
- "sufficient" main memory buffers and good hash functions

Hash Functions

Hash function `h(tuple,range)`:

- maps attribute values → page address

Implementation issues for hash functions:

- range of values is typically larger than range of page addresses
- use `mod` function to "fit" hash value into address range
- expect many tuples to hash to one page (but not too many)
- try to spread addresses *uniformly* (impossible if data distrib is skew)
- make address computation cheap

... Hash Functions

Usual approach in hash function:

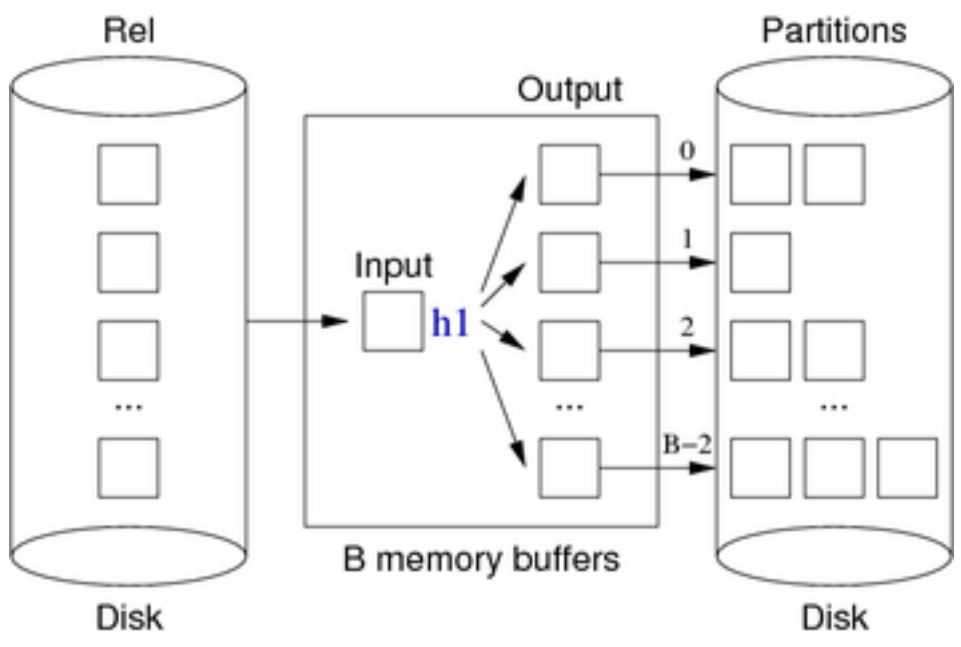
- convert key into numeric value (method depends on key type)
- fit into page address space

Example hash function for character strings:

```
unsigned int hash(char *val, int b)
{
    char *cp;
    unsigned int v, sum = 0;
    for (c = val; *c != '\0'; c++) {
        v = *c + (*(c+1) << 8);
        sum += (sum + 2153*v) % 19937;
    }
    return(sum % b);
}
```

Hash-based Projection

Partitioning phase:



... Hash-based Projection

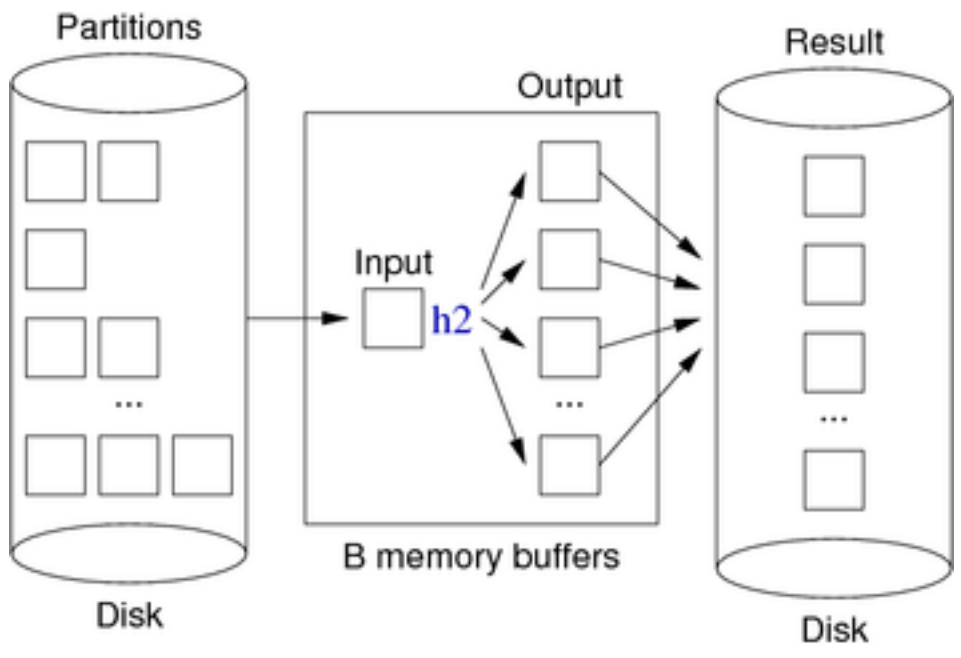
Algorithm for partitioning phase:

```
for each page P in relation Rel {
    for each tuple t in page P {
        t' = project(t, attrList)
        H = h1(t', B-1)
        write t' to partition[H]
}   }
```

Each partition could be implemented as a simple data file.

... Hash-based Projection

Duplicate elimination phase:



... Hash-based Projection

Algorithm for duplicate elimination phase:

```
for each partition P in 0..B-2 {
    for each tuple t in partition P {
        H = h2(t, B-1)
        if (!(t occurs in buffer[H]))
            append t to buffer H
    }
    output contents of all buffers
    clear all buffers
}
```

Cost of Hash-based Projection

The total cost is the sum of the following:

- scanning original relation `Rel`: b_R
- writing partitions: $b_P \geq b_R$, but likely $b_P \cong b_R$
- re-reading partitions: b_P
- writing the result relation: b_{Out}

To ensure that *B* is larger than the largest partition ...

- use hash functions (h1,h2) with uniform spread
- allocate at least *sqrt(b_R)* buffers

... Cost of Hash-based Projection

If the largest partition had more than *B-1* pages

- some in-memory hash buckets would fill up
- overflow would then need to be dumped to disk
- for each subsequent record hashing to that bucket
    - look for duplicates in contents of in-memory hash bucket
    - and *read* dumped bucket contents and look for duplicates

This would potentially increase the cost by a large amount
(worst case is one additional page read for every record after hash bucket fills)

Index-only Projection

Under the conditions:

- relation is indexed on $(A_1,A_2,...A_n)$
- projected attributes are a prefix of $(A_1,A_2,...A_n)$

can do projection without accessing data file.

Basic idea:

- attribute values for $(A_1,A_2,...A_n)$ are stored in the index
- scan through index file (which is already sorted on attributes)
- duplicates are already adjacent in index, so easy to skip

... Index-only Projection

Method:

```
for each entry I in index file {
    tup = project(I.key, attrList)
    if (tupCompare(tup,prev) != 0) {
        addTuple(outbuf,tup)
        if (isFull(outbuf)) {
            writePage(outf,op++,outbuf);
            clear(outbuf);
        }
        prev = tup;
}   }
```

"for each index entry": loop over index pages and loop over entries in each page

Cost of Index-only Projection

Assume that the index (see details later):

- is a file containing values of indexing keys
- consisting of b_i pages (where $b_i \ll b_R$)

Costs involved in index-only projection:

- scanning whole index file `Index`: b_i
- writing tuples to Result: b_{Out}

Total cost: $b_i + b_{Out} \ll b_R + b_{Out}$

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 ⇒ use as default

Best case scenario for each (assuming *B+1* in-memory buffers):

- index-only: $b_i + b_{Out} \ll b_R + b_{Out}$
- hash-based: $b_R + 2.b_P + b_{Out} \cong 3.b_R + b_{Out}$
- sort-based: $b_R + 2.b_T(2+\log_B b_0) + b_{Out}$

Projection in PostgreSQL

Code for projection forms part of execution iterators:

- backend/executor/execQual.c

Functions involved with projection:

- **`ExecProject(projInfo,...)`** ... extracts/stores projected data
- **`ExecTargetList(...)`** ... makes new tuple from old tuple + projection info
- **`ExecStoreTuple(newTuple,...)`** ... save tuple in output slot
