**Lecture 4**

**Scanning in PostgreSQL**

A screen shot of a social media post

Description automatically generated

**Reminder on Cost Analyses**

* When showing the cost of operations, don’t include Tr and Tw
* In counting reads and writes, assume minimal buffering
  + Each **request\_page**()causes a read
  + Each **release\_page**() causes a write (if page is dirty)

**The Sort Operation**

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

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

**Two-way Merge Sort**

A picture containing object

Description automatically generated

For a file containing *b* data pages:

* Require *ceil(log2b)* passes to sort,
* Each pass requires *b* page reads, *b* page writes

Total cost: *2\*b\* ceil(log2b)*

**n-Way Merge Sort**

Merge passes use: *B* memory buffers, *n* input buffers, *B-n* output buffers

A close up of a glass

Description automatically generated

Typically, consider only one output buffer, i.e. *B = n + 1*

**Cost of n-Way Merge Sort**

For *b* data pages and *n = B – 1* input buffers

* First pass: read/writes *b* pages, gives *b0 = ceil(b/B)* runs
* Then need *ceil(lognb0)* passes until sorted
* Each pass reads and writes *b* pages (*2\*b*)

*Cost = 2\*b\*(1 + ceil(lognb0))*, where *b0 = ceil(b/B)*

**Sorting in PostgreSQL**

Sort uses a polyphase merge-sort

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

Disk-based sort has phases:

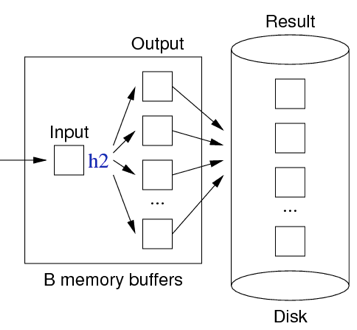
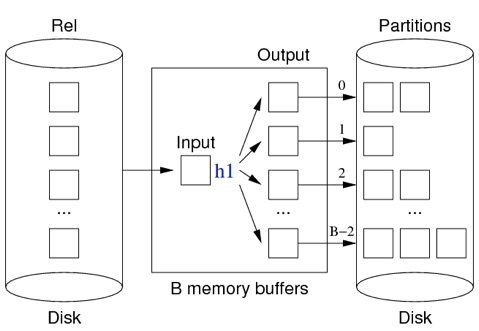
* Divide input into sorted runs using HeapSort
* Merge using *N* buffers, one output buffer
* *N* = as many buffers as **workMem** allows

**Sort-based Projection**

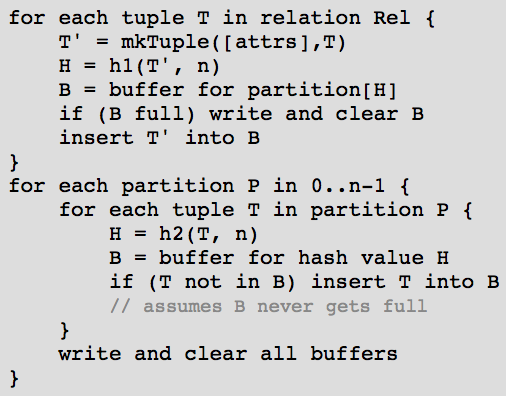
Requires a temporary file/relation (Temp)

A screenshot of a cell phone

Description automatically generated

**Hash-based Projection**



Algorithm for both phases:



**Cost of Hash-based Projection**

Cost = *bR + 2bP + bOut*

Allocate at least *sqrt(bR)*+1 buffer

**Index-only Projection**

Can do projection without accessing data file iff …

* Relation is indexed on *(A1, A2, …, An)*
* Projected attributes are a prefix of *(A1, A2, …, An)*

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 *bi* pages (where *bi* *<<* *bR*)
* Cost = *bi* reads + *bOut* writes

**Comparison of Projection Methods**

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

A screenshot of a cell phone

Description automatically generated

**Projection in PostgreSQL**

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

**Implementing Select Efficiently**

Two basic approaches:

Physical arrangement of tuples

* Sorting (ssearch strategy)
* Hashing (static, dynamic, n-dimensional)

Additional indexing information

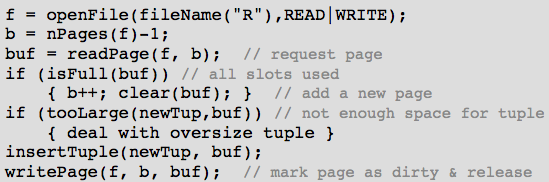
* Index files (primary, secondary, trees)
* Signatures (superimposed, disjoint)

Our analyses assume: 1 input buffer available for each relation

If more buffers are available, more methods benefit

**Heap Files**

**Selection in Heaps** **Insertion in Heaps**

A picture containing person

Description automatically generated**

*Costrange = Costpmr = b Costinsert = 1r + 1w*

*Costone: Best = 1* Plus possible extra write for oversize tuples,

*Average = b/2* e.g. PostgreSQL’s TOAST files

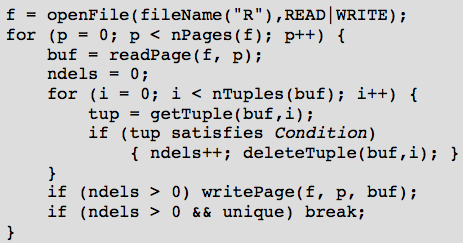
*Worst = b*

PostgreSQL’s tuple insertion:

A close up of a blue wall

Description automatically generated

* Finds page which has enough free space for **newtup**
* 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**



If buffers, **read = request, write = mark-as-dirty**

PostgreSQL tuple deletion:

A close up of a blue wall

Description automatically generated

* Gets page containing tuple into buffer pool and locks it
* Sets flags, commandID and **xmax** in tuple; dirties buffer
* Writes indication of deletion to transaction log

**Updates in Heaps**

Analysis for updates is similar to that for deletion

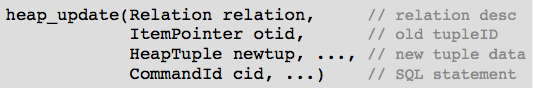
* Scan all pages
* Replace any updated tuples (within each page)
* Write affected pages to disk

*Costupdate = br + bqw*

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

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

PostgreSQL tuple update:



* Essentially does **delete(otid)**, then **insert(newtup)**
* Also, sets old tuple’s **ctid** field to reference new tuple
* Can also update-in-place if no referencing transactions

**Heaps in PostgreSQL**

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 varlen attributes)

**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

A screenshot of a cell phone

Description automatically generatedA picture containing object

Description automatically generated

In order to mitigate insertion costs, use overflow blocks

A picture containing screenshot

Description automatically generated

Total number of overflow blocks = bov

Average overflow chain length = OV = bov/b

Bucket = data page + its overflow page(s)

**Selection in Sorted Files**

For *one* query on sort key, use binary search

A screenshot of a cell phone

Description automatically generated

Where **f** is file for relation; **mid, lo, hi** are page indexes;

**k** is a field/attr; **val, loVal, hiVal** are values for **k**

Search a page and its overflow chain for a key value

A picture containing indoor

Description automatically generated

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

A screenshot of a cell phone

Description automatically generated

**Selection in Sorted Files Cost**

Costone: Best = *1* Worst = *log2b + bov*

*Costpmr = Costone + (bq-1) \* (1+OV)*

*Costrange = Costone + (bq-1) \* (1+OV)*

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

If condition contains attribute j, not the sort key

Costone, Costrange, Costpmr as for heap files

**Updates to Sorted Files**

**Insertion** approach:

* Find appropriate page for tuple (via binary search)
* If page not full, insert into page
* Otherwise, insert into next overflow block with space

Thus, Costinsert = Costone + δW (where δW = 1 or 2)

**Deletion** strategy:

* Find matching tuple(s)
* Mark them as deleted

Thus, Costdelete = Costselect + bqw

**Hashed Files**

**Hashing**

PostgreSQL hash function (simplified):

A screenshot of a cell phone on a table

Description automatically generated

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

If *b = 2k*, bitwise AND with *k* low-order bits set to one

A black sign with white text

Description automatically generated

Otherwise, use mod to produce value in range *0..b-1*

A close up of a logo

Description automatically generated

**Hashing Performance**

Aims:

* Distribute tuples evenly amongst buckets
* Have most buckets nearly full (attempt to minimise wasted space)

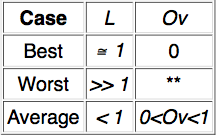
Note: if data distribution not uniform, address distribution can’t be uniform

Use overflow pages to handle “overfull” buckets

Two important measures for hash files:

* Load factor: *L = r/bc*
* Average overflow chain length: *OV = bOV/b*

Three cases for distribution of tuples in a hashed file:

 (\*\* performance is same as Heap File)

To achieve average case, aim for *0.75 ≤ L ≤ 0.9*

**Selection with Hashing**

Best *Costone = 1* (find in data page)

Average *Costone = 1 + OV/2* (scan half of overflow chain)

Worst *Costone = 1 + max(OvLen)* (find in last page of overflow chain)

*Costpmr = 1 + OV*

Costrange = b + bOV // Hashing does not help with range queries

*Costone, Costrange, Costpmr = b + bOV* //Selection on attribute *j* which is not hash key

**Insertion with Hashing**

Costinsert: Best: *1r + 1w* Worst: *1 + max(OvLen)r +2w*

**Deletion with Hashing (**Similar to selection)