

course 3 Scan, Sort, Project

course's aims:

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

tips: what the difference between indexed and hashed in DBMS?



<https://www.geeksforgeeks.org/difference-between-indexing-and-hashing-in-dbms/>

Difference between Indexing and Hashing in DBMS

1. indexing: Indexing, as name suggests, is a technique or mechanism generally used to speed up access of data. Index is basically a type of data structure that is used to locate and access data in database table quickly. Indexes can easily be developed or created using one or more columns of database table.

2. Hashing: Hashing, as name suggests, is a technique or mechanism that uses hash functions with search keys as parameters to generate address of data record. It calculates direct location of data record on disk without using index structure. A good hash functions only uses one-way hashing algorithm and hash cannot be converted back into original key. In simple words, it is a process of converting given key into another value known as hash value or simply hash.

Difference between Indexing and Hashing in DBMS :

 Indexing	 Hashing
It is a technique that allows to quickly retrieve records from database file.	It is a technique that allows to search location of desired data on disk without using index structure.

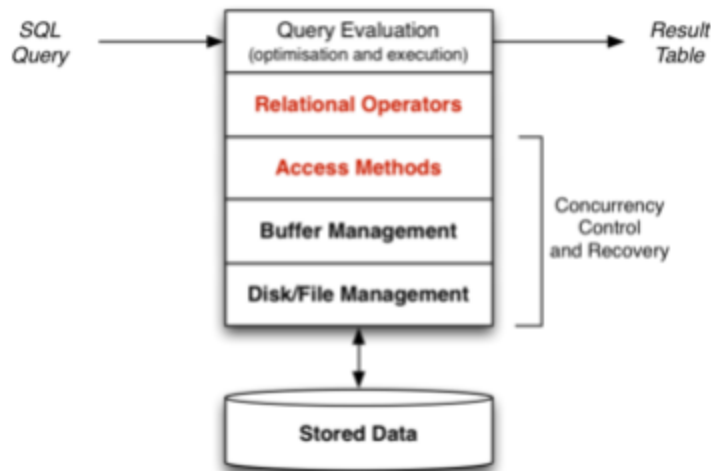
Aa Indexing	≡ Hashing
<u>It is generally used to optimize or increase performance of database simply by minimizing number of disk accesses that are required when a query is processed.</u>	It is generally used to index and retrieve items in database as it is faster to search that specific item using shorter hashed key rather than using its original value.
<u>It offers faster search and retrieval of data to users, helps to reduce table space, makes it possible to quickly retrieve or fetch data, can be used for sorting, etc.</u>	It is faster than searching arrays and lists, provides more flexible and reliable method of data retrieval rather than any other data structure, can be used for comparing two files for quality, etc.
<u>Its main purpose is to provide basis for both rapid random lookups and efficient access of ordered records.</u>	Its main purpose is to use math problem to organize data into easily searchable buckets.
<u>It is not considered best for large databases and its good for small databases.</u>	It is considered best for large databases.
<u>Types of indexing includes ordered indexing, primary indexing, secondary indexing, clustered indexing.</u>	Types of hashing includes static and dynamic hashing.
<u>It uses data reference to hold address of disk block.</u>	It uses mathematical functions known as hash function to calculate direct location of records on disk.
<u>It is important because it protects file and documents of large size business organizations, and optimize performance of database.</u>	It is important because it ensures data integrity of files and messages, takes variable length string or messages and compresses and converts it into fixed length value.

1. Implementing Relational Operations

1.1 Relational operations

DBMS core = relational engine, with implementation of:

- selection, projection, join, set operation
- scanning, sorting, grouping, aggregation



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`, and for different operator, this function means different operations

- for selection operator
 - find next tuple in the table and check whether it satisfied selection condition or not
- for join operator
 - find next pair of tuples in the table and check whether it satisfied join condition or not

there are three “dimension of variation” in this system:

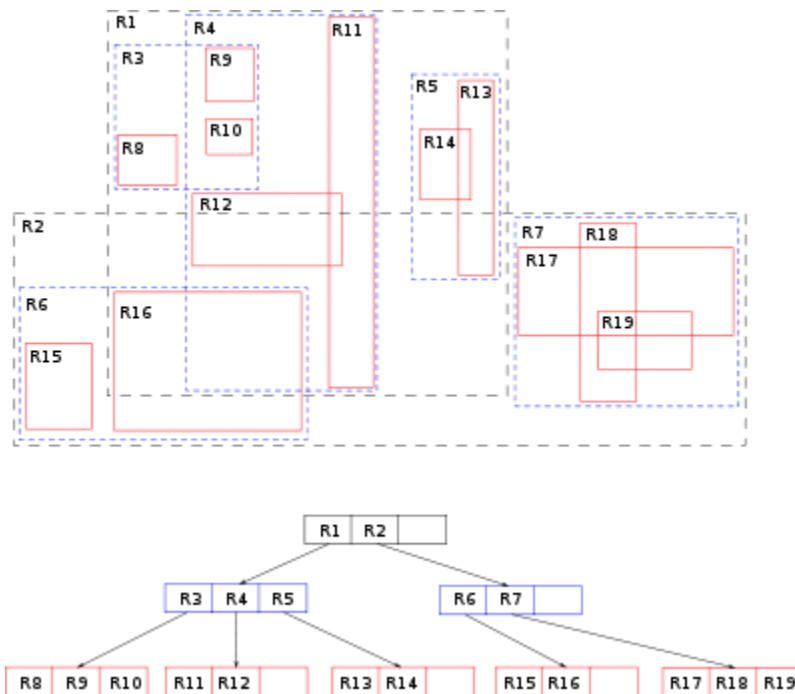
- relational operators: selection, projection, join, sort
- file structure: heap, indexed, hashed
- query processing methods: merge-sort, hash-join

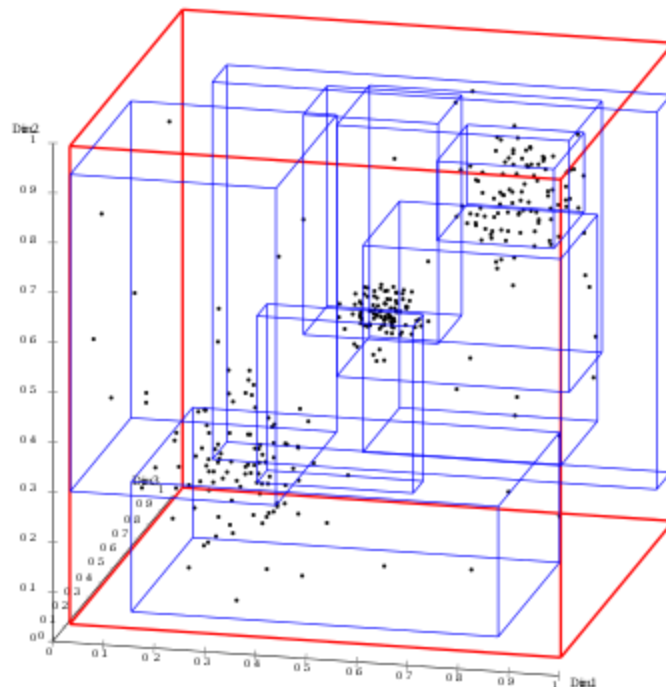
we consider combination of these, e.g.:

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

Notes: R-tree is a special datastructure used for spacial data access methods

<https://en.wikipedia.org/wiki/R-tree>





1.2 Query Types

queries fall into a number of classes:

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</code> <code>order by x</code>	$Sort[x]R$	<i>ord</i>

1.3 Cost Models

in DBMS query evaluation, we don't use asymptotic complexity, instead, we measure it in terms of number of page reads / writes

we have to define some symbols for convenience

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	–	≈ 0
# pages containing answers for query q	b_q	≥ 0

1.4 Example file structure

when describing file structure:

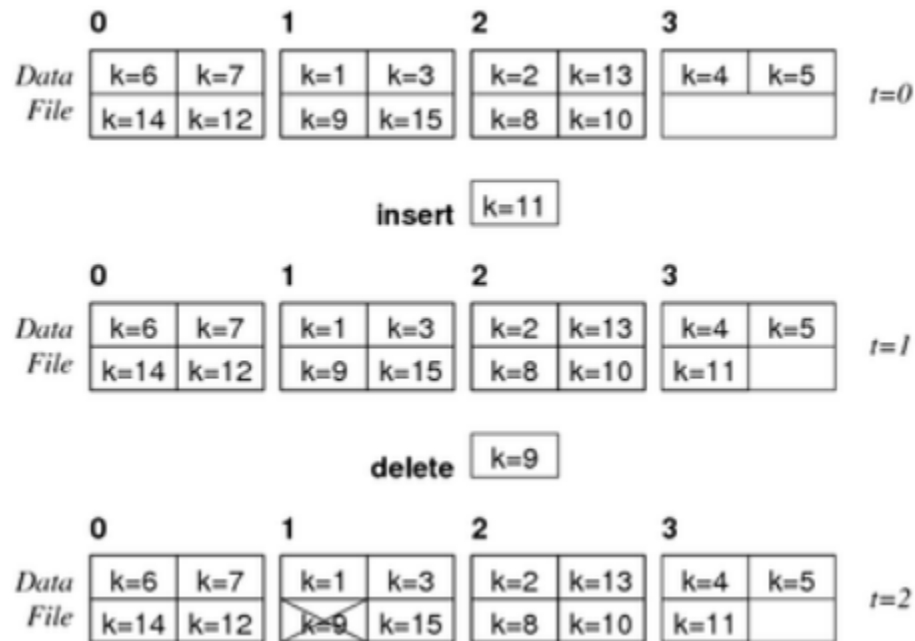
- use a large box as Page
- use small box to represent tuple
- we sometimes refer to tuples as `rec_i`
- we sometimes refer to tuples via their key
 - mostly, key corresponds to the notion of “primary key”
 - sometimes, key means “search key” in selection condition



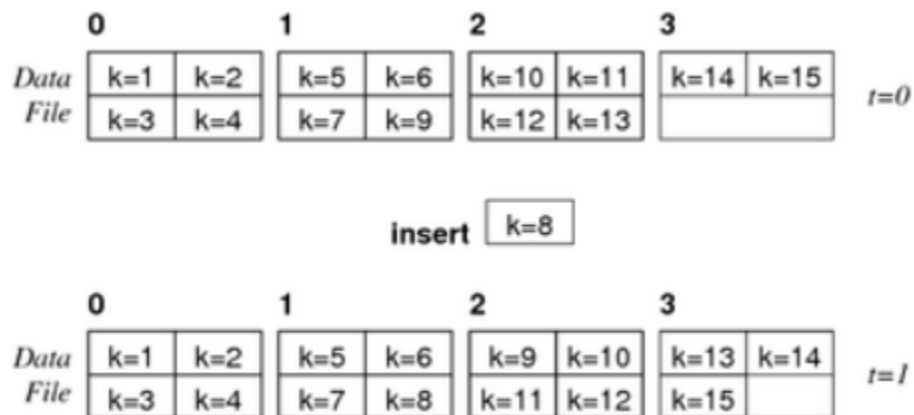
consider several kinds of file structure:

- heap file: tuples will be added in any page which has space

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

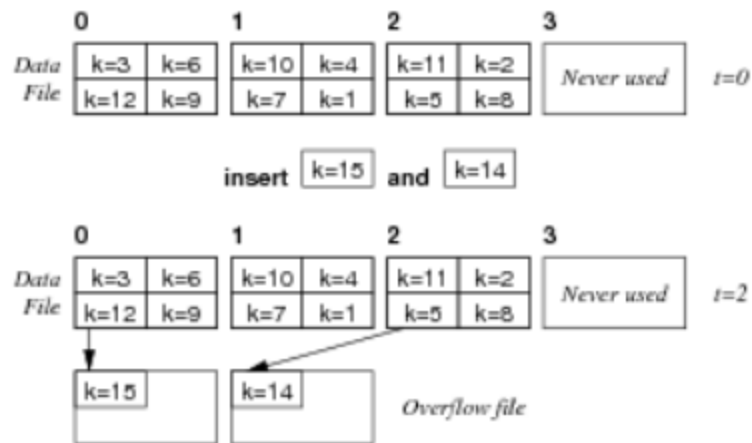


- sorted file: tuples arranged in file in key order



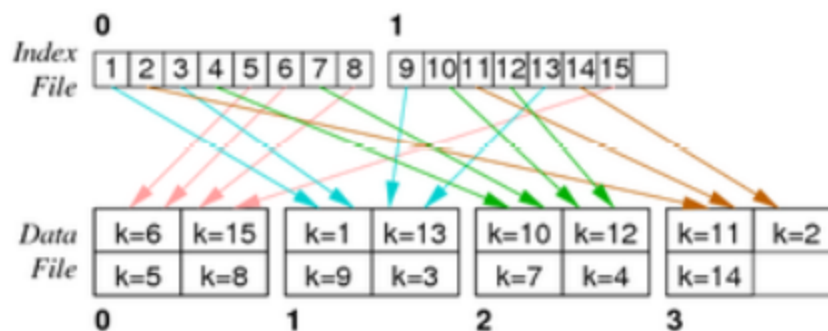
- hash file: tuples placed in pages using hash function

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



- index file: indexed file's tuples will spread to several pages, linked by pointer

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



2. Scanning

let's talk about scanning with an clear examples

```
select * from T;
```

this sql will be implemented via iteration over file containing T

```
for each tuple in Page
    for each Page in file
```



```
for each file in relation T
```

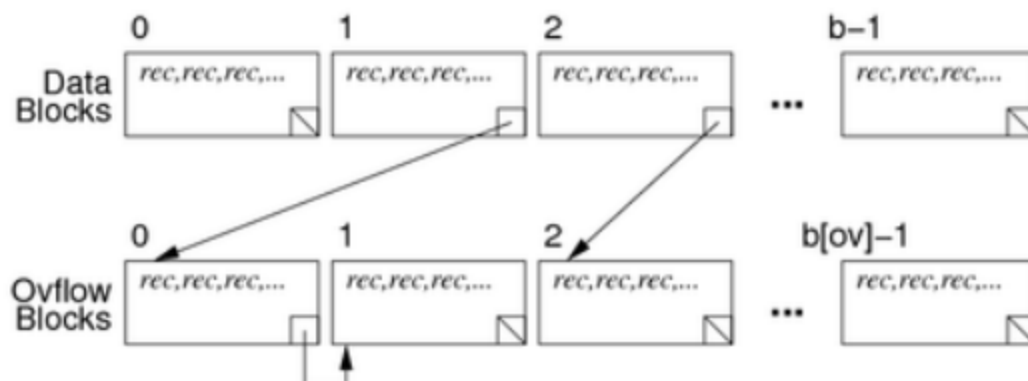
and in terms of file operations:

```
// 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
    }
}
```

and if file has overflow page, the overflow part will be insert in a Overflow blocks:



and it will increase the cost of scanning

and for some operations like selection, in an unordered file, it still needs to scan all blocks to find the correct one

2.1 File Copying

consider a SQL statement like that:

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

and in file operation:

```
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);
```

2.2 Iterators

in high-level of DBMS, it define a Iterators to scan a relation:

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

3. Sorting

3.1 the sort operation

sorting is used essentially in many operations:

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

3.2 external sorting

some algorithm like quick-sorting are designed for in-mem data

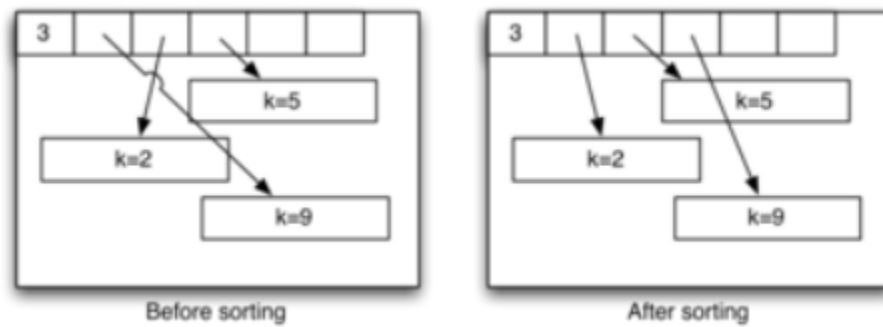
for data in disk, we need external sorting techniques

the standard external sorting method is merge sort, it works by:

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

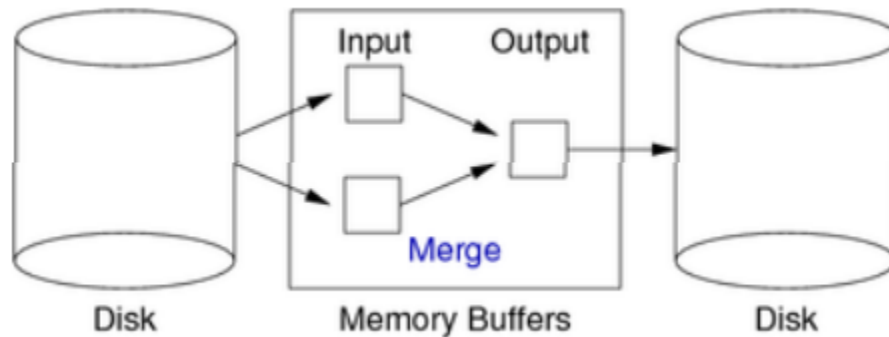
so what about sorting tuples within page?

- first, we need to extract sort-key from each tuple
- tips: no need to physically move tuples
- and then simply swap entries in page dir

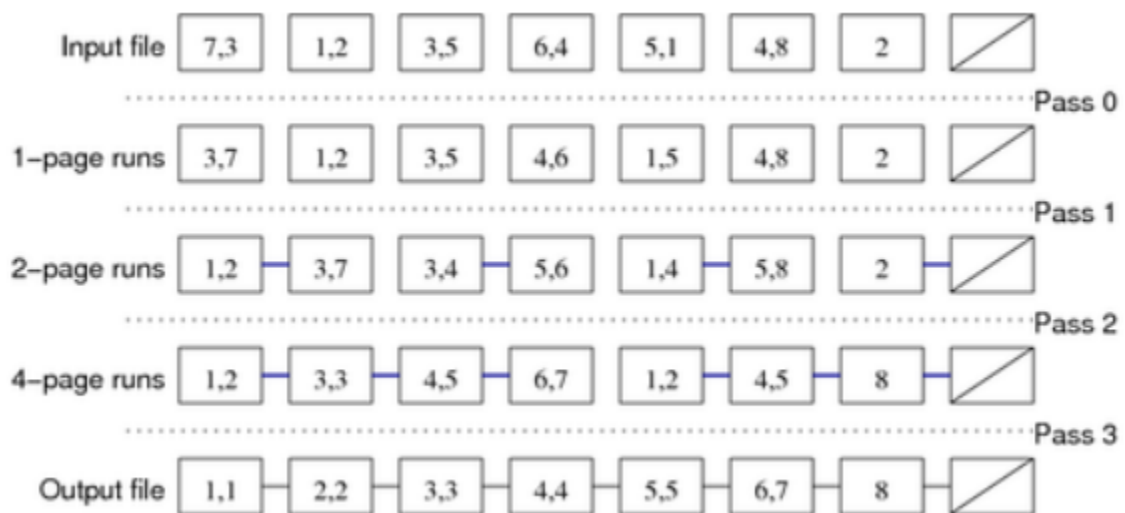


3.3 Two-way merge sort

this algorithm requires three in-mem buffers



let's assume that there are cost for merging two buffer, the algorithm's illustration is shown below:



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);
```

for a file containing b data pages:

- require $\log_2 b$ passes to sort
- each pass requires b page reads, b page writes

so the entire cost is $2 \cdot b \cdot \log_2 b$

so, can we do it faster?

3.4 n-way merge sort

merge passes use: $n(B)$ input buffers, 1 out buffer

```

// 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
}

```

and the cost of n-way merge sort is :

- first pass: read/write b pages, gives $b_0 = b/B$ runs
- then need $\log_n b_0$ passes until sorted
- each pass reads and writes b pages

so the entire cost = $2 \cdot b \cdot (1 + \log_n(b_0))$, where $b_0 = b/B$

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 (just like the cache mechanism in I/O in OS course)

3.5 sorting in PostgreSQL

in postgresQL, 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 data-size fits mem, then call qsort(), else use disk-based sort

disk-based sort has several phases:

- divide input into several runs using HeapSort
- merge use N buffers, one for output
- $N =$ as many buffers as workMem allows

4. implementing Projection

4.1 the projection operation

consider the query like:

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

the projection operation needs to:

- scan the entire relation as input:
 - straightforward, whichever file organisation is used
- remove the unwanted attributes in output
 - straightforward, manipulating internal record structure

- eliminate any duplicates produced
 - **this task is not simple, and there are two approaches : sorting or hashing**

4.2 sort-based projection

this algorithm is simple:

- scan input relation Rel and produce a file of tuples containing only the projected attributes
- sort this file of tuples using the combination of all attributes as the sort key
- scan the sorted result, comparing adjacent tuples, and discard duplicates

and the cost of this algorithm is:

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

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

4.3 improving sort-base projection

there are some approaches for improving algorithm:

- do projection during first phase of sort
- reduce sorting costs:
 - using more buffer of mem
 - eliminating duplicates during the merge phase

4.4 Hash-based projection

overview of the method:

- scan input relation Rel1 and produce a set of hash partitions based on the projected attributes

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.

- scan each hash partition looking for duplicates

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
}
```

- once each partition is duplicate-free, write out the remaining tuples

this method requires:

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

the hash function do a job:

- maps attribute values → page address

there are several issues that we will face:

- the range of values is typically large than range of page address
 - this problem we can finish by using **mod** operators
- try to spread address calculated by hash_function uniformly
- and make address computation more cheap

and the cost of the algorithm is:

The total cost is the sum of the following:

- scanning original relation **Rel**: b_R
- writing partitions: $b_P \geq b_R$, but likely $b_P = b_R$
- 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

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)

4.5 Index-only Projection ?

we use this algorithm under conditions:

- relation is indexed on (A1, A2, A3, ..., An)
- projected attributes are a prefix of (A1, A2, A3, ..., An)

in this condition, we can use index-only projection

basic idea:

- attribute values for (A1, A2, ..., An) are stored in the index
- scan through index file (which is already sorted on the attributes)
- duplicates are already adjacent in index, so easy to skip

```
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: $b_I + b_{Out} \ll b_R + b_{Out}$

4.5 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 $B+1$ in-memory buffers):

- index-only: $b_I + b_{Out} \ll b_R + b_{Out}$

-
- hash-based: $b_R + 2.b_P + b_{Out} \approx 3.b_R + b_{Out}$
 - sort-based: $b_R + 2.b_T(2 + \log_B b_0) + b_{Out}$