Week 09 Lecture

Assignment 2

Assignment 2

2/55

Aim: implement multi-attribute linear hashed files (MALH files)

- placement of tuples in buckets determined by MA hash
- file expansion organised via linear hashing

Each "MALH file" represents one table ...

```
create table R (a0 text, a1 text, ...  an-1 text);
```

Implemented as three physical files ...

- `R.info` ... contains file parameters, e.g. *n, r, b, d, sp, cv*
- `R.data` ... primary data pages, each with *free, ov* and tuples
- `R.ovflow` ... overflow pages, same structure as data pages

... Assignment 2

3/55

Commands:

```
$ ./create  R  3  5  "0,0:0,1:1,0:2,0:1,1:0,2"
... makes new MALH file called R with 3 attrs, 8 data pages, ...

$ ./gendata  1000  3 | ./insert  R
... generates 1000 tuples and inserts them into R files ...

$ ./gendata  500  3  1001  13 | ./insert  R
... generates another 500 tuples and inserts them into R ...

$ ./select  R  "?,eyes,girl"
... finds all tuples with "eyes" as second attribute value ...
...                    and "girl" as third attribute value ...

$ ./select  R  "123,?,?"
... finds all tuples with 123 as first attribute value ...

$ ./stats  R
... display information about the relation/files (debugging) ...
```

... Assignment 2

4/55

Code is structured as a set of modules and ADTs ...

- `Bits` ... functions on 32-bit bit-strings
- `ChVec` ... data structures and operations on choice vectors
- `Page` ... data structures and operations on pages
- `Query` ... data structures and operations for query scans
- `Reln` ... data structures and operations on relations

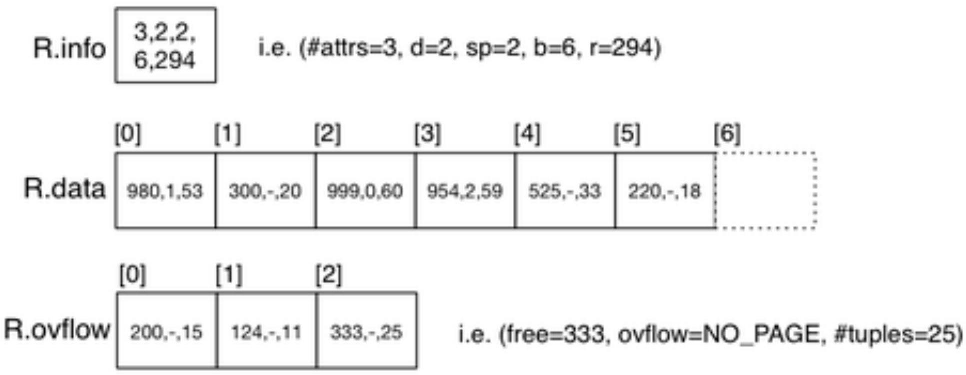
- `Tuple` ... data structures and operations on tuples
- `util` ... miscellaneous helper functions
- `hash` ... hash function (from PostgreSQL)

plus main programs (e.g. `create.c`, `select.c`) for commands

... Assignment 2

5/55

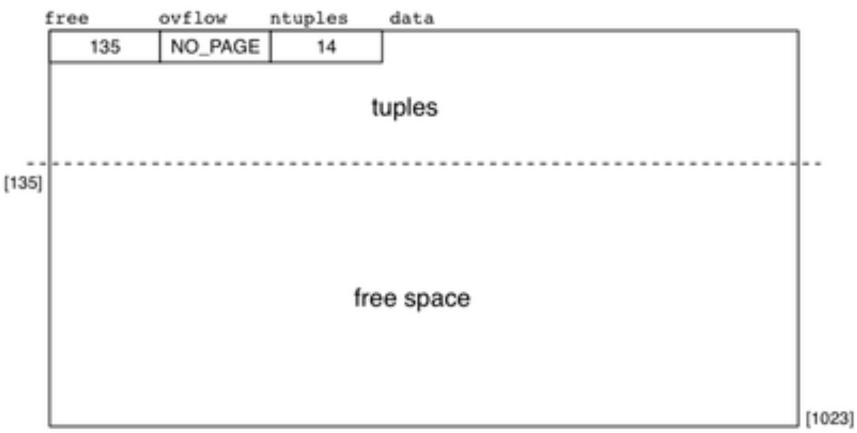
File structure:



... Assignment 2

6/55

Page structure:



... Assignment 2

7/55

Task 1: Multi-attribute hashing

- current tuple hash function uses only first attribute
- modify `tupleHash()` to use CV to build proper MA hash

Task 2: Selection (Querying)

- functions in `query.c` are incomplete
- implement query scan data structure and operations on it

Task 3: Linear Hashing

- current files don't grow primary data file ... just overflow
- implement linear hashing ... split page *sp* after every *c* inserts
- where $c = B/R$ and $B \cong 1024$ and $R = 10n$

... Assignment 2

Notes:

- worth: 14%,   due before: 3pm on Monday 23 May
- work in same pairs as for Assignment 1
- you can change any of the ADTs, except ...
  - do not change `Reln` or `Page` structures
- you are not allowed to change any of the commands
- no need to add any new ADTs
  - but update the `Makefile` appropriately if you do
- submit `Makefile` and code for all ADTs
- MA-hashing, scanning, linear hashing are all discussed in notes

---

Exercise 1: Queries with MA.Hashing

Consider a multi-attributed hashed file with tuples like *(a,b,c)*

where *sp=0*, *d=6*, CV = *<(0,0),(0,1),(1,0),(2,0),(1,1),(0,2), ...>*, and

- *hash* (a) = `...00101101001101`
- *hash* (b) = `...00101101001101`
- *hash* (c) = `...00101101001101`

What are the query hashes for each of the following queries:

- `(a,b,c), (a,?,c), (?,b,c), (a,?,?), (?,?,?)`

Which buckets will be accessed in answering each query?

Tree Indexes for N-d Selection

Multi-dimensional Tree Indexes

Over the last 20 years, from a range of problem areas

- different multi-d tree index schemes have been proposed
- varying primarily in how they partition tuple-space

Consider three popular schemes: kd-trees, Quad-trees, R-trees.

Example data for multi-d trees is based on the following relation:

```
create table Rel (
    X char(1) check (X between 'a' and 'z'),
    Y integer check (Y between 0 and 9)
);
```

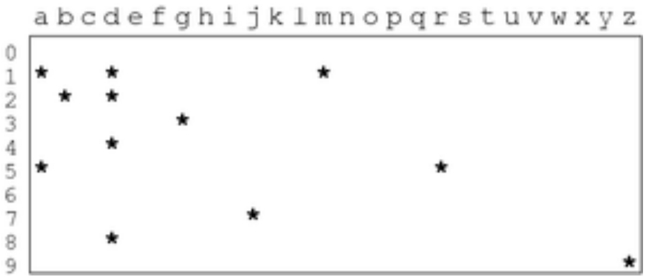
... Multi-dimensional Tree Indexes

12/55

Example tuples:

```
Rel('a',1)  Rel('a',5)  Rel('b',2)  Rel('d',1)
Rel('d',2)  Rel('d',4)  Rel('d',8)  Rel('g',3)
Rel('j',7)  Rel('m',1)  Rel('r',5)  Rel('z',9)
```

The tuple-space for the above tuples:

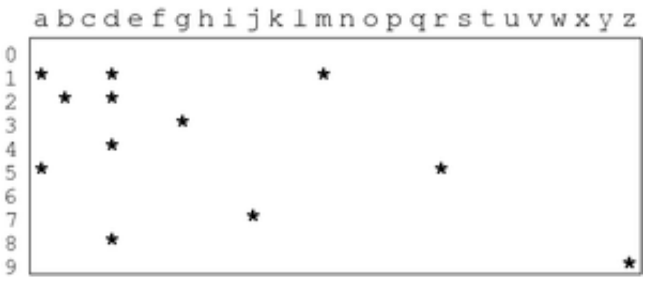


Exercise 2: Query Types and Tuple Space

13/55

Which part of the tuple-space does each query represent?

```
Q1: select * from Rel where X = 'd' and Y = 4
Q2: select * from Rel where 'j' < X ≤ 'r'
Q3: select * from Rel where X > 'm' and Y > 4
Q4: select * from Rel where 'k' ≤ X ≤ 'p' and 3 ≤ Y ≤ 6
```



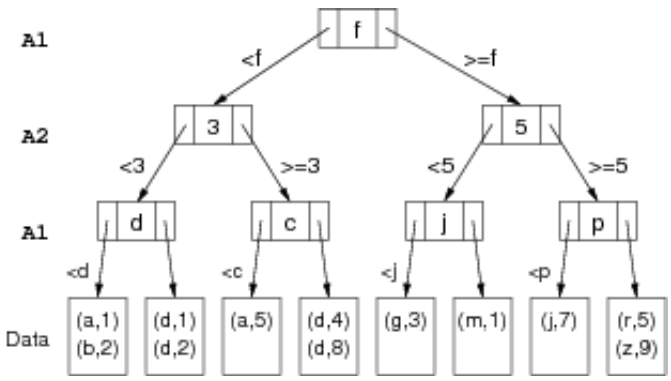
---

kd-Trees

14/55

kd-trees are multi-way search trees where

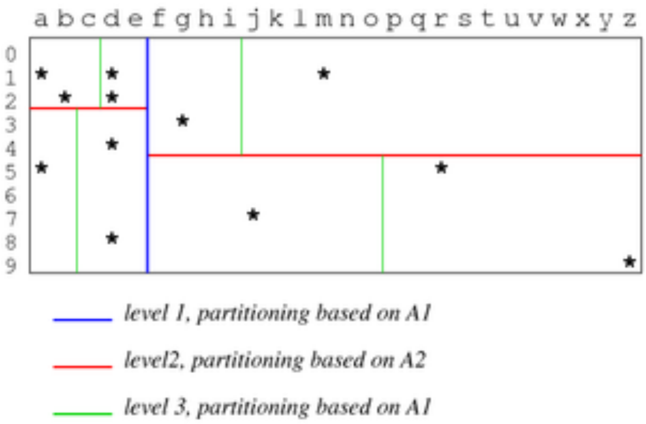
- each level of the tree partitions on a different attribute
- each node contains *n-1* key values, pointers to *n* subtrees



... kd-Trees

15/55

How this tree partitions the tuple space:



Searching in kd-Trees

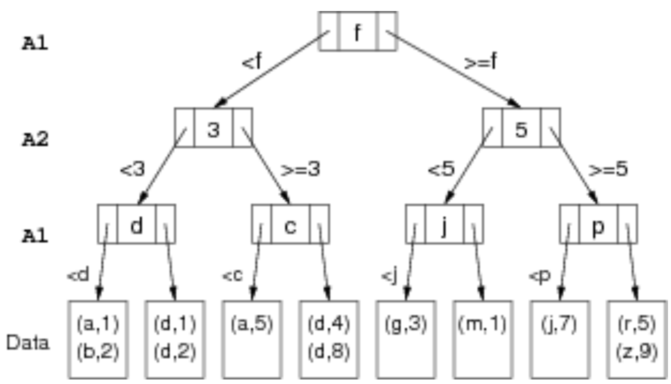
16/55

```
// Started by Search(Q, R, 0, kdTreeRoot)
Search(Query Q, Relation R, Level L, Node N)
{
   if (isDataPage(N)) {
      Buf = getPage(fileOf(R),idOf(N))
      check Buf for matching tuples
   } else {
      a = attrLev[L]
      if (!hasValue(Q,a))
         nextNodes = all children of N
      else {
         val = getAttr(Q,a)
         nextNodes = find(N,Q,a,val)
      }
      for each C in nextNodes
         Search(Q, R, L+1, C)
}  }
```

Exercise 3: Searching in kd-Trees

17/55

Using the following kd-tree index



Answer the queries `(m,1), (a,?), (?,1), (?,?)`

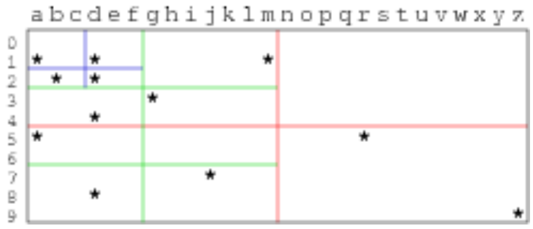
Quad Trees

18/55

Quad trees use regular, disjoint partitioning of tuple space.

- for *2d*, partition space into quadrants (NW, NE, SW, SE)
- each quadrant can be further subdivided into four, etc.

Example:



... Quad Trees

19/55

Basis for the partitioning:

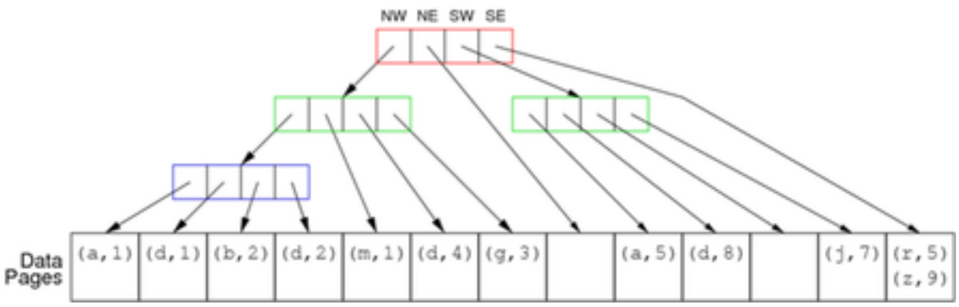
- a quadrant that has no sub-partitions is a *leaf quadrant*
- each leaf quadrant maps to a single data page
- subdivide until points in each quadrant fit into one data page
- ideal: same number of points in each leaf quadrant (balanced)
- point density varies over space
  ⇒ different regions require different levels of partitioning
- this means that the tree is not necessarily balanced

Note: effective for *d≤5*, ok for *6≤d≤10*, ineffective for *d>10*

... Quad Trees

20/55

The previous partitioning gives this tree structure, e.g.



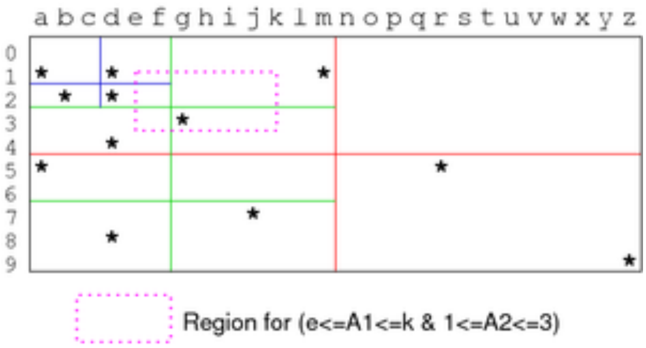
In this and following examples, we give coords of top-left,bottom-right of a region

---

Searching in Quad-tree

21/55

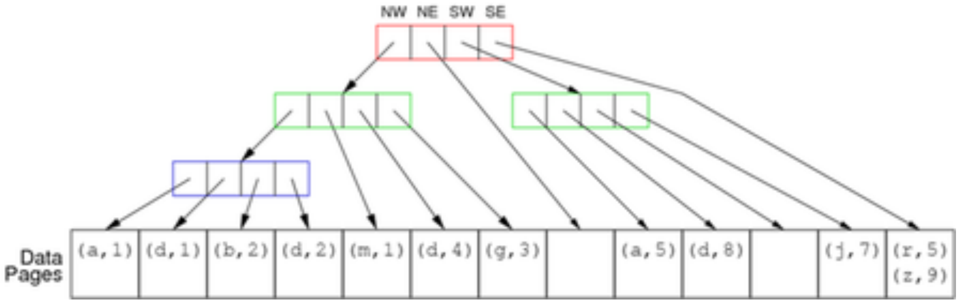
Space query example:



Need to traverse: red(NW), green(NW,NE,SW,SE), blue(NE,SE).

Exercise 4: Searching in Quad-trees

Using the following quad-tree index



Answer the queries `(m,1), (a,?), (?,1), (?,?)`

R-Trees

R-trees use a flexible, overlapping partitioning of tuple space.

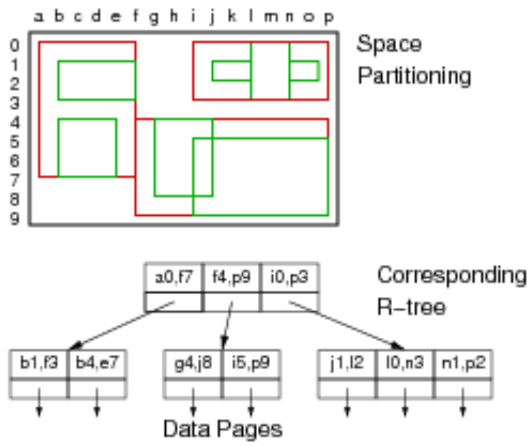
- each node in the tree represents a *k*d hypercube
- its children represent (possibly overlapping) subregions
- the child regions do not need to cover the entire parent region

Overlap and partial cover means:

- can optimize space partitioning wrt data distribution
- so that there are similar numbers of points in each region

Aim: height-balanced, partly-full index pages (cf. B-tree)

... R-Trees



Insertion into R-tree

25/55

Insertion of an object *R* occurs as follows:

- start at root, look for children that completely contain *R*
- if no child completely contains *R*, *choose one* of the children and expand its boundaries so that it does contain *R*
- if several children contain *R*, *choose one* and proceed to child
- repeat above containment search in children of current node
- once we reach data page, insert *R* if there is room
- if no room in data page, replace by two data pages
- *partition* existing objects between two data pages
- update node pointing to data pages
  (may cause B-tree-like propagation of node changes up into tree)

Note that *R* may be a point or a polygon.

Query with R-trees

26/55

Designed to handle *space* queries and "where-am-I" queries.

"Where-am-I" query: find all regions containing a given point *P*:

- start at root, select all children whose subregions contain *P*
- if there are zero such regions, search finishes with *P* not found
- otherwise, recursively search within node for each subregion
- once we reach a leaf, we know that region contains *P*

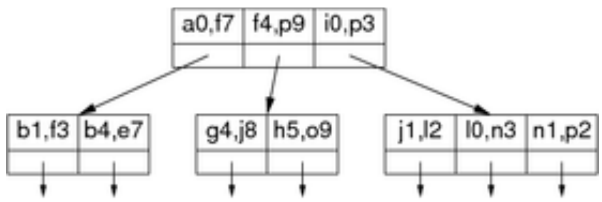
Space (region) queries are handled in a similar way

- we traverse down any path that intersects the query region

Exercise 5: Query with R-trees

27/55

Using the following R-tree:



Show how the following queries would be answered:

```
Q1: select * from Rel where X='a' and Y=4
Q2: select * from Rel where X='i' and Y=6
Q3: select * from Rel where 'c'≤X≤'j' and Y=5
Q4: select * from Rel where X='c'
```

Multi-d Trees in PostgreSQL

28/55

Up to version 8.2, PostgreSQL had R-tree implementation

Superseded by *GiST* = Generalized Search Trees

GiST indexes parameterise: data type, searching, splitting

- via seven user-defined functions (e.g. `picksplit()`)

GiST trees have the following structural constraints:

- every node is at least fraction *f* full (e.g. 0.5)
- the root node has at least two children (unless also a leaf)
- all leaves appear at the same level

Details: src/backend/access/gist

Costs of Search in Multi-d Trees

29/55

Difficult to determine cost precisely.

Best case: *pmr* query where all attributes have known values

- in kd-trees and quad-trees, follow single tree path
- cost is equal to depth *D* of tree
- in R-trees, may follow several paths (overlapping partitions)

Typical case: some attributes are unknown or defined by range

- need to visit multiple sub-trees
- how many depends on: range, choice-points in tree nodes

Note: can view unknown value `X=?` as range $min(X) \le X \le max(X)$

Implementing Join

Join

31/55

DBMSs are engines to *store*, *combine* and *filter* information.

Join (⋈) is the primary means of *combining* information.

Join is important and potentially expensive

Most common join condition: equijoin, e.g. `(R.pk = S.fk)`

Join varieties (natural, inner, outer, semi, anti) all behave similarly.

We consider three strategies for implementing join

- *nested loop* ... simple, widely applicable, inefficient without buffering
- *sort-merge* ... works best if tables are soted on join attributes
- *hash-based* ... requires good hash function and sufficient buffering

---

Join Example

32/55

Consider a university database with the schema:

```
create table Student(
   id     integer primary key,
   name   text,  ...
);
create table Enrolled(
   stude  integer references Student(id),
   subj   text references Subject(code),  ...
);
create table Subject(
   code   text primary key,
   title  text,  ...
);
```

... Join Example

33/55

List names of students in all subjects, arranged by subject.

SQL query to provide this information:

```
select E.subj, S.name
from   Student S, Enrolled E
where  S.id = E.stude
order  by E.subj, S.name;
```

And its relational algebra equivalent:

Sort[subj] (Project[subj,name] (Join[id=stude](Student,Enrolled)))

To simplify formulae, we denote `Student` by *S* and `Enrolled` by *E*

... Join Example

34/55

Some database statistics:

Sym	Meaning	Value
r_S	# student records	20,000
r_E	# enrollment records	80,000
c_S	`Student` records/page	20
c_E	`Enrolled` records/page	40
b_S	# data pages in `Student`	1,000
b_E	# data pages in `Enrolled`	2,000

Also, in cost analyses below, *N* = number of memory buffers.

... Join Example

35/55

`Out` = *Student* ⋈ *Enrolled* relation statistics:

Sym	Meaning	Value
r_{Out}	# tuples in result	80,000
C_{Out}	result records/page	80
b_{Out}	# data pages in result	1,000

Notes:

- r_{Out} ... one result tuple for each `Enrolled` tuple
- C_{Out} ... result tuples have only `subj` and `name`
- in analyses, ignore cost of writing result ... same in all methods

Nested Loop Join

Nested Loop Join

37/55

Basic strategy (R.a ⋈ S.b):

```
Result = {}
for each page i in R {
   pageR = getPage(R,i)
   for each page j in S {
      pageS = getPage(S,j)
      for each pair of tuples tR,tS
                        from pageR,pageS {
         if (tR.a == tS.b)
            Result = Result ∪ (tR:tS)
} } }
```

Needs input buffers for R and S, output buffer for "joined" tuples

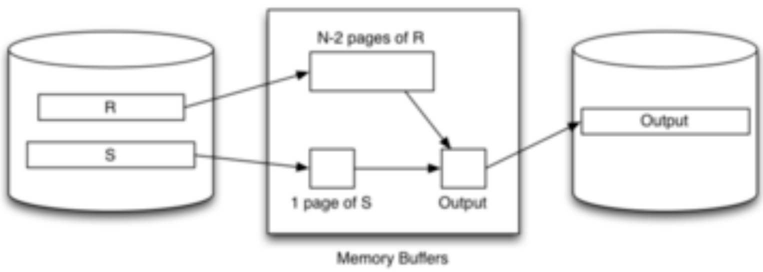
Terminology: R is outer relation, S is inner relation

Block Nested Loop Join

38/55

Method (for *N* memory buffers):

- read *N-2* page chunks of *R* relation into memory
- for each *S* page, check join condition on all (t_R, t_S) pairs



... Block Nested Loop Join

39/55

Best-case scenario: $b_R \le N\text{-}2$

- read b_R pages of relation R into buffers
- while R is buffered, read b_S pages of S

Cost = $b_R + b_S$

Typical-case scenario: $b_R > N\text{-}2$

- read $ceil(b_R/N\text{-}2)$ chunks of pages from R
- for each chunk, read b_S pages of S

Cost = $b_R + b_S \, . \, ceil(b_R/N\text{-}2)$

Note: requires $r_R . r_S$ checks of the join condition

Exercise 6: Nested Loop Join Cost

40/55

Consider executing *Join[i=j](S,T)* with the following parameters:

- $r_S = 1000$, $b_S = 50$, $r_T = 3000$, $b_T = 150$
- *S.i* is primary key, and *T* has index on *T.j*
- *T* is sorted on *T.j*, each *S* tuple joins with 2 *T* tuples
- DBMS has $N = 42$ buffers available for the join

Calculate the cost for evaluating the above join

- using block nested loop join
- compute #pages read/written
- compute #join-condition checks performed

Exercise 7: Nested Loop Join Cost (ii)

41/55

Compute the cost (# pages fetched) of $(S \bowtie E)$

Sym	Meaning	Value
r_S	# student records	20,000
r_E	# enrollment records	80,000
c_S	`Student` records/page	20
c_E	`Enrolled` records/page	40
b_S	# data pages in `Student`	1,000
b_E	# data pages in `Enrolled`	2,000

for *N = 22, 202, 2002* and different inner/outer combinations

---

Exercise 8: Nested Loop Join Cost (cont)

42/55

If the query in the above example was:

```
select j.code, j.title, s.name
from   Student s
       join Enrolled e on (s.id=e.student)
       join Subject j on (e.subj=j.code)
```

how would this change the previous analysis?

What join combinations are there?

Assume 2000 subjects, with $c_J = 10$

How large would the intermediate tuples be? What assumptions?

Compute the cost (# pages fetched, # pages written) for $N = 22$

Block Nested Loop Join in Practice

43/55

Why block nested loop join is actually useful in practice ...

Many queries have the form

```
select * from R,S where r.i=s.j and r.x=k
```

This would typically be evaluated as

Join [i=j] ((Sel[r.x=k](R)), S)

If *|Sel[r.x=k](R)|* is small ⇒ may fit in memory (in small #buffers)

Index Nested Loop Join

44/55

A problem with nested-loop join:

- needs repeated scans of *entire* inner relation *S*

If there is an index on *S*, we can avoid such repeated scanning.

Consider *Join[R.i=S.j](R,S)*:

```
for each tuple r in relation R {
    use index to select tuples
        from S where s.j = r.i
    for each selected tuple s from S {
        add (r,s) to result
}   }
```

... Index Nested Loop Join

45/55

This method requires:

- one scan of *R* relation (b_R)
  - only one buffer needed, since we use *R* tuple-at-a-time
- for each *tuple* in *R* (r_R), one index lookup on *S*
  - cost depends on type of index and number of results
  - best case is when each *R.i* matches few *S* tuples

Cost = $b_R + r_R.Sel_S$ (Sel_S is the cost of performing a select on *S*).

Typical Sel_S = 1-2 (hashing) .. b_q (unclustered index)

Trade-off: $r_R.Sel_S$ vs $b_R.b_S$, where $b_R \ll r_R$ and $Sel_S \ll b_S$

Sort-Merge Join

Sort-Merge Join

47/55

Basic approach:

- sort both relations on join attribute (reminder: *Join[R.i=S.j](R,S)*)
- scan together using *merge* to form result `(r,s)` tuples

Advantages:

- no need to deal with "entire" *S* relation for each *r* tuple
- deal with runs of matching *R* and *S* tuples

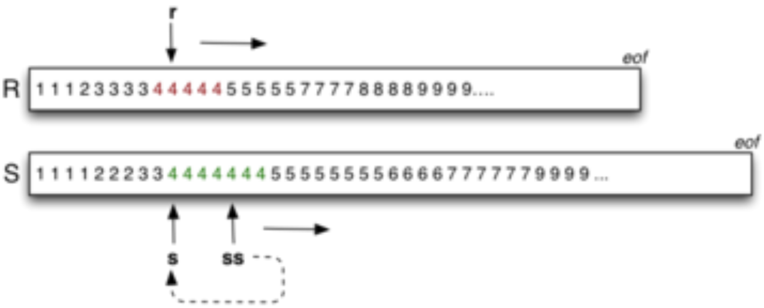
Disadvantages:

- cost of sorting both relations (already sorted on join key?)
- some rescanning required when long runs of *S* tuples

... Sort-Merge Join

Method requires several cursors to scan sorted relations:

- `r` = current record in *R* relation
- `s` = start of current run in *S* relation
- `ss` = current record in current run in *S* relation



... Sort-Merge Join

Algorithm using query iterators/scanners:

```
Query ri, si;  Tuple r,s;

ri = startScan("SortedR");
si = startScan("SortedS");
while ((r = nextTuple(ri)) != NULL
       && (s = nextTuple(si)) != NULL) {
    // align cursors to start of next common run
    while (r != NULL && r.i < s.j)
           r = nextTuple(ri);
    if (r == NULL) break;
    while (s != NULL && r.i > s.j)
           s = nextTuple(si);
    if (s == NULL) break;
    // must have (r.i == s.j) here
...
```

... Sort-Merge Join


```
...
    // remember start of current run in S
    TupleID startRun = scanCurrent(si)
    // scan common run, generating result tuples
    while (r != NULL && r.i == s.j) {
        while (s != NULL and s.j == r.i) {
            addTuple(outbuf, combine(r,s));
            if (isFull(outbuf)) {
                writePage(outf, outp++, outbuf);
                clearBuf(outbuf);
            }
            s = nextTuple(si);
        }
        r = nextTuple(ri);
```

```
        setScan(si, startRun);
    }
}
```

... Sort-Merge Join

51/55

Buffer requirements:

- for sort phase:
  - as many as possible (remembering that cost is $O(\log_N)$)
  - if insufficient buffers, sorting cost can dominate
- for merge phase:
  - one output buffer for result
  - one input buffer for relation *R*
  - (preferably) enough buffers for longest run in *S*

... Sort-Merge Join

52/55

Cost of sort-merge join.

Step 1: sort each relation that is not already sorted:

- Cost = $\sum_i 2.b_i (1 + \log_{N-1}(b_i /N))$ (with *N* buffers)

Step 2: merge sorted relations:

- if every run of values in *S* fits completely in buffers,
  merge requires single scan,   Cost = $b_R + b_S$
- if some runs in of values in *S* are larger than buffers,
  need to re-scan run for each corresponding value from *R*

Sort-Merge Join on Example

53/55

Case 1: $Join_{[id=stude]}(Student,Enrolled)$

- relations are not sorted on *id#*
- memory buffers *N=32*; all runs are of length < *30*

$$
\begin{aligned}
\text{Cost} &= sort(S) + sort(E) + b_S + b_E \\
&= 2b_S(1+\log_{31}(b_S/32)) + 2b_E(1+\log_{31}(b_E/32)) + b_S + b_E \\
&= 2\times1000\times(1+2) + 2\times2000\times(1+2) + 1000 + 2000 \\
&= 6000 + 12000 + 1000 + 2000 \\
&= 21{,}000
\end{aligned}
$$

... Sort-Merge Join on Example

54/55

Case 2: $Join_{[id=stude]}(Student,Enrolled)$

- *Student* and *Enrolled* already sorted on *id#*
- memory buffers *N=3* (*S* input, *E* input, output)
- 5% of the "runs" in *E* span two pages
- there are no "runs" in *S*, since *id#* is a primary key

For the above, no re-scans of *E* runs are ever needed

Cost = 2,000 + 1,000 = 3,000   (regardless of which relation is outer)

Exercise 9: Sort-merge Join Cost

55/55

Consider executing *Join[i=j](S,T)* with the following parameters:

- $r_S = 1000$, $b_S = 50$, $r_T = 3000$, $b_T = 150$
- *S.i* is primary key, and *T* has index on *T.j*
- *T* is sorted on *T.j*, each *S* tuple joins with 2 *T* tuples
- DBMS has $N = 42$ buffers available for the join

Calculate the cost for evaluating the above join

- using sort-merge join
- compute #pages read/written
- compute #join-condition checks performed

Produced: 4 May 2016