SIMC Indexing

❖ Signature-based indexing

Reminder: file organisation for signature indexing (two files)

One signature slot per tuple slot; unused signature slots are zeroed.

❖ Superimposed Codewords (SIMC)

In a superimposed codewords (simc) indexing scheme

- a tuple descriptor is formed by overlaying attribute codewords
- each codeword is *m* bits long and has *k* bits set to 1

A tuple descriptor *desc(t)* is

- a bit-string, *m* bits long, where $j \leq nk$ bits are set to 1
- $desc(t) = cw(A_1)$ **OR** $cw(A_2)$ **OR** ... **OR** $cw(A_n)$

Method (assuming all *n* attributes are used in descriptor):

```
bits desc = 0
for (i = 1; i <= n; i++) {
   bits cw = codeword(A[i],m,k)
   desc = desc | cw
}
```

❖ SIMC Example

Consider the following tuple (from bank deposit database)

Branch	AcctNo	Name	Amount
Perryridge	102	Hayes	400

It has the following codewords/descriptor (for $m = 12, \quad k = 2$)

A_i	$cw(A_i)$
Perryridge	**010000000001**
102	**000000000011**
Hayes	**000001000100**
400	**000010000100**
desc(t)	**010011000111**

❖ SIMC Queries

To answer query *q* in SIMC

- first generate *desc(q)* by OR-ing codewords for known attributes
- then attempt to match *desc(q)* against all signatures in sig file

E.g. consider the query **`(Perryridge, ?, ?, ?)`**.

A_i	$cw(A_i)$
Perryridge	`010000000001`
`?`	`000000000000`
`?`	`000000000000`
`?`	`000000000000`
desc(q)	`010000000001`

❖ SIMC Queries (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}
// scan r signatures
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
// then scan b_sq = b_q + δ pages to check for matches
```

Matching can be implemented efficiently ...

```
#define matches(sig,qdesc) ((sig & qdesc) == qdesc)
```

❖ Example SIMC Query

Consider the query and the example database:

Signature	Deposit Record
010000000001	(Perryridge,?,?,?)
100101001001	(Brighton,217,Green,750)
010011000111	(Perryridge,102,Hayes,400)
101001001001	(Downtown,101,Johnshon,512)
101100000011	(Mianus,215,Smith,700)
010101010101	(Clearview,117,Throggs,295)
100101010011	(Redwood,222,Lindsay,695)

Gives two matches: one true match, one false match.

❖ SIMC Parameters

False match probablity p_F = likelihood of a false match

How to reduce likelihood of false matches?

- use different hash function for each attribute (h_i for A_i)
- increase descriptor size (m)
- choose k so that ≅ half of bits are set

Larger m means larger signature file ⇒ read more signature data.

Having k too high ⇒ increased overlapping.
Having k too low ⇒ increased hash collisions.

❖ SIMC Parameters (cont)

How to determine "optimal" *m* and *k*?

1. start by choosing acceptable p_F

   (e.g. $p_F \leq 10^{-4}$ i.e. one false match in 10,000)

2. then choose *m* and *k* to achieve no more than this p_F.

Formulae to derive *m* and *k* given p_F and *n*:

$$k = 1/\log_e 2 \; . \; \log_e (1/p_F)$$

$$m = (1/\log_e 2)^2 \; . \; n \; . \; \log_e (1/p_F)$$

Formula from Bloom (1970), "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM, 13 (7): 422–426

❖ Query Cost for SIMC

Cost to answer *pmr* query: $Cost_{pmr} = b_D + b_{sq}$

- read r descriptors on b_D descriptor pages
- then read b_{sq} data pages and check for matches

$b_D = ceil(\ r/c_D\ )$ and $c_D = floor(B/ceil(m/8))$

E.g. $m=64$, $B=8192$, $r=10^4$ $\Rightarrow$ $c_D = 1024$, $b_D=10$

b_{sq} includes pages with r_q matching tuples and r_F false matches

Expected false matches = $r_F = (r - r_q).p_F \cong r.p_F$ if r_q *is way smaller than r*

E.g. Worst $b_{sq} = r_q+r_F$, Best $b_{sq} = 1$

❖ Page-level SIMC

SIMC has one descriptor per tuple ... potentially inefficient.

Alternative approach: one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

Size of page descriptor (PD) (clearly larger than tuple descriptor):

- use above formulae but with $c.n$ "attributes"

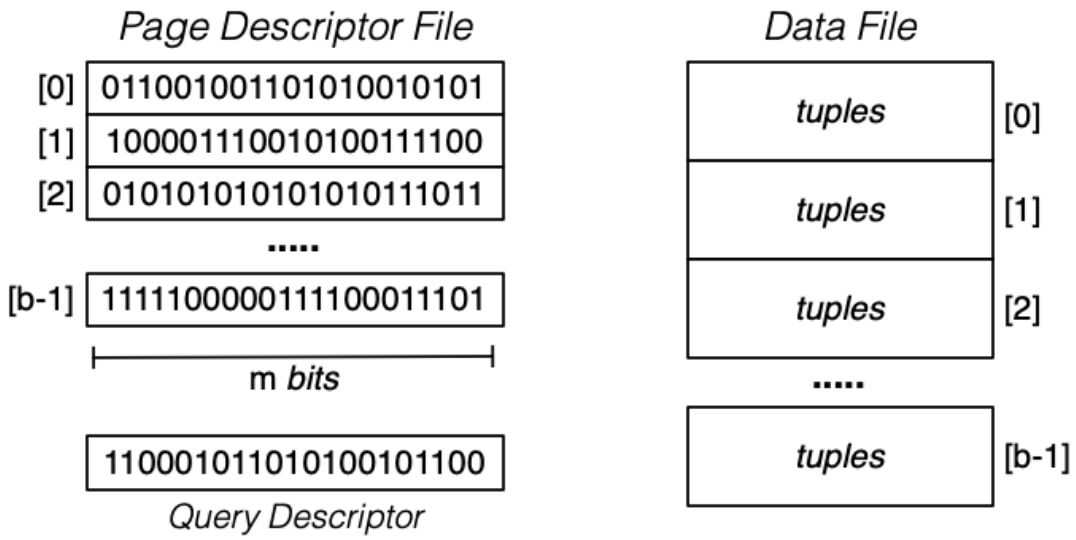
E.g. $n = 4, c = 64, p_F = 10^{-3} \Rightarrow m_p \cong 3680bits \cong 460bytes$

Typically, pages are 1..8KB $\Rightarrow$ 8..64 PD/page (c_{PD}).

E.g. $m_p \cong 460, \quad B = 8192, \quad c_{PD} \cong 17$

❖ Page-level SIMC (cont)

File organisation for page-level superimposed codeword index



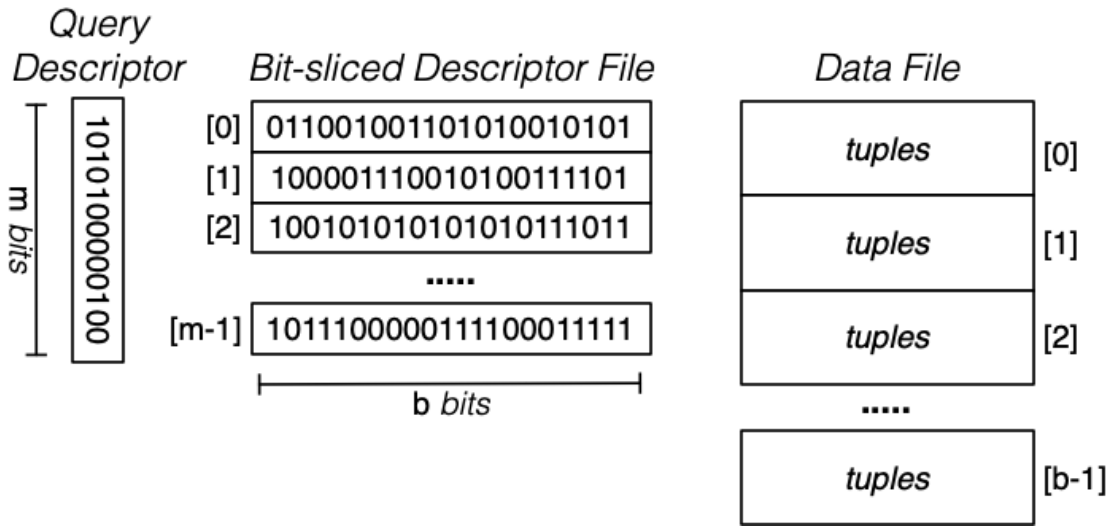
❖ Page-level SIMC (cont)

Algorithm for evaluating *pmr* query using page descriptors

```
pagesToCheck = {}
// scan b mp-bit page descriptors
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = i
        pagesToCheck = pagesToCheck ∪ pid
    }
}
// read and scan bsq data pages
for each pid in pagesToCheck {
    Buf = getPage(dataFile,pid)
    check tuples in Buf for answers
}
```

❖ Bit-sliced SIMC

Improvement: store *b* *m*-bit page descriptors as *m* *b*-bit "bit-slices"



❖ Bit-sliced SIMC (cont)

Algorithm for evaluating *pmr* query using bit-sliced descriptors

```
matches = ~0  //all ones
// scan m r-bit slices
for each bit i set to 1 in desc(q) {
   slice = fetch bit-slice i
   matches = matches & slice
}
for each bit i set to 1 in matches {
   fetch page i
   scan page for matching records
}
```

Effective because *desc(q)* typically has less than half bits set to 1

❖ Comparison of Approaches

Tuple-based

- *r* signatures, *m*-bit signatures, *k* bits/attribute
- read all pages of signature file in filtering for a query

Page-based

- *b* signatures, m_p-bit signatures, *k* bits/attribute
- read all pages of signature file in filtering for a query

Bit-sliced

- *m* signatures, *b*-bit slices, *k* bits/attribute
- read less than half of the signature file in filtering for a query

All signature files are roughly the same size, for a given p_F

❖ Signature-based Indexing in PostgreSQL

PostgreSQL supports signature based indexing via the **`bloom`** module

(Signature-based indexes like this are often called Bloom filters)

Creating a Bloom index

```
create index Idx on R using bloom (a1,a2,a3)
with   (length=64, col1=3, col2=4, col3=2);
```

Example: 10000 tuples, query = select * from R where a1=55 and a2=42, no matching tuples, random numeric values for attributes

```
No indexes ... execution time 15ms
B-tree index on all attributes ... execution time 12ms
Bloom index ... execution time 0.4ms
```

For more details, see PostgreSQL doc, Appendix F.7

CATC Indexing

❖ Signature-based indexing

Reminder: file organisation for signature indexing (two files)

One signature slot per tuple slot; unused signature slots are zeroed.

We use the terms "signature" and "descriptor" interchangeably

❖ Concatenated Codewords (CATC)

In a concatenated codewords (catc) indexing scheme

- a tuple signature is formed by concatenating attribute codewords
- the signature is m bits long, with $\cong m/2$ bits set to 1
- codeword for $attr_i$ is u_i bits long and has $\cong u_i/2$ bits set to 1
- each codeword could be different length, but always $\sum_{1..n} u_i = m$

A tuple descriptor (signature) *desc(t)* is

- $desc(t) = cw(A_n) + cw(A_{n-1}) \ldots + cw(A_+) + cw(A_1)$
- where "+" represents bit-string concatentation

The order that the concenated codewords appears doesn't matter, as long as it's done consistently

❖ CATC Example

Consider the following tuple (from bank deposit database)

Branch	AcctNo	Name	Amount
Perryridge	102	Hayes	400

It has the following codewords/descriptor (for $m = 16$, $u_i = 4$)

A_i	$cw(A_i)$
Perryridge	0101
102	1001
Hayes	1010
400	1100
desc(t)	1100101010010101

❖ CATC Queries

To answer query *q* in CATC

- first generate *desc(q)* by combining codewords for all attributes
- for known A_i use $cw(A_i)$; for "unknown" A_i, use $cw(A_i) = 0$

E.g. consider the query **(Perryridge, ?, Hayes, ?)**.

A_i	$cw(A_i)$
Perryridge	**0101**
?	**0000**
Hayes	**1010**
?	**0000**
desc(q)	**0000101000000101**

❖ CATC Queries (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}
// scan r signatures
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
// then scan b_sq = b_q + δ pages to check for matches
```

Matching can be implemented efficiently ...

```
#define matches(sig,qdesc) ((sig & qdesc) == qdesc)
```

❖ Example CATC Query

Consider the query and the example database:

Signature	Deposit Record
0000101000000101	(Perryridge,?,Hayes,?)
1010100101101001	(Brighton,217,Green,750)
1100101010010101	(Perryridge,102,Hayes,400)
1010011010010110	(Downtown,101,Johnshon,512)
0110101001010011	(Mianus,215,Smith,700)
1010101011000101	(Clearview,117,Throggs,295)
1001010100111001	(Redwood,222,Lindsay,695)

Gives two matches: one true match, one false match.

❖ CATC Parameters

False match probablity p_F = likelihood of a false match

How to reduce likelihood of false matches?

- increase descriptor size (m)

Larger m means larger signature file ⇒ read more signature data.

Since u_i 's are relatively small, hash collisions may be a serious issue

But making u_i's means larger signatures ⇒ optimisation problem

❖ CATC Parameters (cont)

How to determine "optimal" m and u ?

1. start by choosing acceptable p_F
   (e.g. $p_F \leq 10^{-4}$ i.e. one false match in 10,000)
2. then choose m to achieve no more than this p_F.

Formulae to derive "good" m: $m = (1/\log_e 2)^2 . n . \log_e (1/p_F)$

Choice of u_i values

- each A_i has same u_i, or
- allocate u_i based on size of attribute domains

❖ Query Cost for CATC

Cost to answer *pmr* query: $Cost_{pmr} = b_D + b_{sq}$

- read r descriptors on b_D descriptor pages
- then read b_{sq} data pages and check for matches

$b_D = ceil(\ r/c_D\ )$ and $c_D = floor(B/ceil(m/8))$

E.g. $m=64$, $B=8192$, $r=10^4$ $\Rightarrow$ $c_D = 1024$, $b_D=10$

b_{sq} includes pages with r_q matching tuples and r_F false matches

Expected false matches = $r_F = (r - r_q).p_F \cong r.p_F$ if r_q *is way smaller than r*

E.g. Worst $b_{sq} = r_q+r_F$, Best $b_{sq} = 1$

❖ Variations on CATC

CATC has one descriptor per tuple ... potentially inefficient.

Alternative approach: one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

Size of page descriptor $m_p = (1/\log_e 2)^2 . c.n . \log_e (1/p_F)$

Size of codewords is proportionally larger (unless attribute domain small)

E.g. $n = 4, c = 64, p_F = 10^{-3} \Rightarrow m_p \cong 3680bits \cong 460bytes$

Typically, pages are 1..8KB ⇒ 8..64 PD/page (c_{PD}).

E.g. $m_p \cong 460, \quad B = 8192, \quad c_{PD} \cong 17$

❖ Variations on CATC (cont)

Improvement: store $r \times m$ -bit descriptors as $m \times r$-bit "bit-slices"

If $r = 2^x$ then uses same storage as tuple descriptors

Query cost: scan $u_i / 2$ bit-slices for each known attribute

If k is set of known attribute values, #slices = $\sum_{i \in k} u_i / 2$

E.g. $r = 128$, $m = 64$, $n = 4$, $u_i = 16$

`(a,?,c,?)` requires scan of 2×8 128-bit (16-byte) slices

compared to scan of 128 tuple descriptors, where each descriptor is 64-bits (8-bytes)

❖ Comparison with SIMC

Assume same m , p_F , n for each method ...

CATC has u_i-bit codewords, each has $\cong u_i / 2$ bits set to 1

SIMC has m-bit codewords, each has k bits set to 1

Signatures for both have m bits, with $\cong m / 2$ bits set to 1

CATC has flexibility in u_i, but small(er) codewords so more hash collisions

SIMC has less hash collisions, but has errors from "unfortunate" overlays

Implementing Join

❖ Join

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 sorted on join attributes
- hash-based ... requires good hash function and sufficient buffering

❖ Join Example

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,  ...
);
```

We use this example for each join implementation, by way of comparison

❖ Join Example (cont)

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

SQL query to provide this information:

```
select E.subj, S.name
from   Student S
       join Enrolled E on (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 (cont)

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 later, N = number of memory buffers.

❖ Join Example (cont)

Relation statistics for **`Out`** = *Student* ⋈ *Enrolled*

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

❖ Implementing Join

A naive join implementation strategy

```
for each tuple T_S in Students {
   for each tuple T_E in Enrolled {
      if (testJoinCondition(C,T_S,T_E)) {
         T1 = concat(T_S,T_E)
         T2 = project([subj,name],T1)
         ResultSet = ResultSet ∪ {T2}
} } }
```

Problems:

- join condition is tested $r_E.r_S = 16{\times}10^8$ times
- tuples scanned = $r_S + r_S.r_E$ = 20000 + 20000.80000 = 1600020000

❖ Implementing Join (cont)

An alternative naive join implementation strategy

```
for each tuple T_E in Enrolled {
   for each tuple T_S in Students {
      if (testJoinCondition(C,T_S,T_E)) {
         T1 = concat(T_S,T_E)
         T2 = project([subj,name],T1)
         ResultSet = ResultSet ∪ {T2}
} } }
```

Relatively minor performance difference ...

- tuples scanned = $r_E + r_E.r_S$ = 80000 + 80000.20000 = 1600080000

Terminology: relation in outer loop is outer; other relation is inner

❖ Join Summary

None of nested-loop/sort-merge/hash join is superior in some overall sense.

Which strategy is best for a given query depends on:

- sizes of relations being joined, size of buffer pool
- any indexing on relations, whether relations are sorted
- which attributes and operations are used in the query
- number of tuples in *S* matching each tuple in *R*
- distribution of data values (uniform, skew, ...)

Given query *Q*, choosing the "best" join strategy is critical;
the cost difference between best and worst case can be very large.

E.g. *Join[id=stude](Student,Enrolled)*: 3,000 ... 2,000,000 page reads

❖ Join in PostgreSQL

Join implementations are under:
`src/backend/executor`

PostgreSQL suports the three join methods that we discuss:

- nested loop join (**`nodeNestloop.c`**)
- sort-merge join (**`nodeMergejoin.c`**)
- hash join (**`nodeHashjoin.c`**) (hybrid hash join)

The query optimiser chooses appropriate join, by considering

- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)

Nested-loop Join

❖ Join Example

SQL query on student/enrolment database:

```
select E.subj, S.name
from   Student S join Enrolled E on (S.id = E.stude)
order  by E.subj
```

And its relational algebra equivalent:

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

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$, $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with N buffers

❖ Nested Loop Join

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

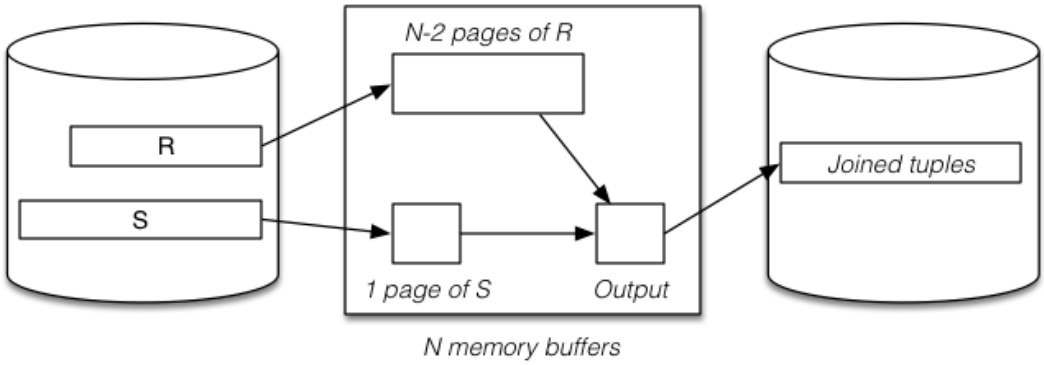
Terminology: R is outer relation, S is inner relation

Cost = b_R . b_S ... ouch!

❖ Block Nested Loop Join

Method (for *N* memory buffers):

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



❖ Block Nested Loop Join (cont)

Best-case scenario: $b_R \leq N-2$

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

Cost = $b_R + b_S$

Typical-case scenario: $b_R > N-2$

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

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

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

❖ Cost on Example Query

With N = 12 buffers, and S as outer and E as inner

- Cost = $b_S + b_E.ceil(b_S/(N-2))$ = $1000 + 2000.ceil(1000/10)$ = 201000

With N = 12 buffers, and E as outer and S as inner

- Cost = $b_E + b_S.ceil(b_E/(N-2))$ = $2000 + 1000.ceil(2000/10)$ = 202000

With N = 102 buffers, and S as outer and E as inner

- Cost = $b_S + b_E.ceil(b_S/(N-2))$ = $1000 + 2000.ceil(1000/100)$ = 21000

With N = 102 buffers, and E as outer and S as inner

- Cost = $b_E + b_S.ceil(b_E/(N-2))$ = $2000 + 1000.ceil(2000/100)$ = 22000

❖ Block Nested Loop Join

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

Many queries have the form

```
select *
from   R join S on (R.i = S.j)
where  R.x = K
```

This would typically be evaluated as

```
Tmp = Sel[x=K](R)
Res = Join[i=j](Tmp, S)
```

If **`Tmp`** is small ⇒ may fit in memory (in small #buffers)

❖ Index Nested Loop Join

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[i=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 (cont)

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 $b_R < r_R$ and $Sel_S < b_S$

Sort-merge Join

- Sort-Merge Join
- Sort-Merge Join on Example

❖ Sort-Merge Join

Basic approach:

- sort both relations on join attribute (reminder: *Join [i=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

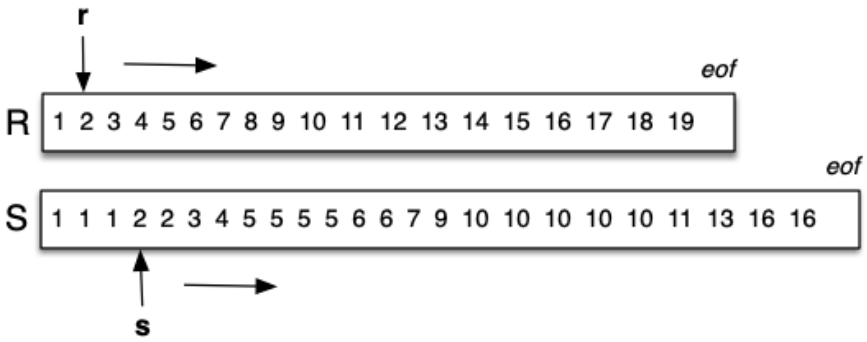
Disadvantages:

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

❖ Sort-Merge Join (cont)

Standard merging requires two cursors:

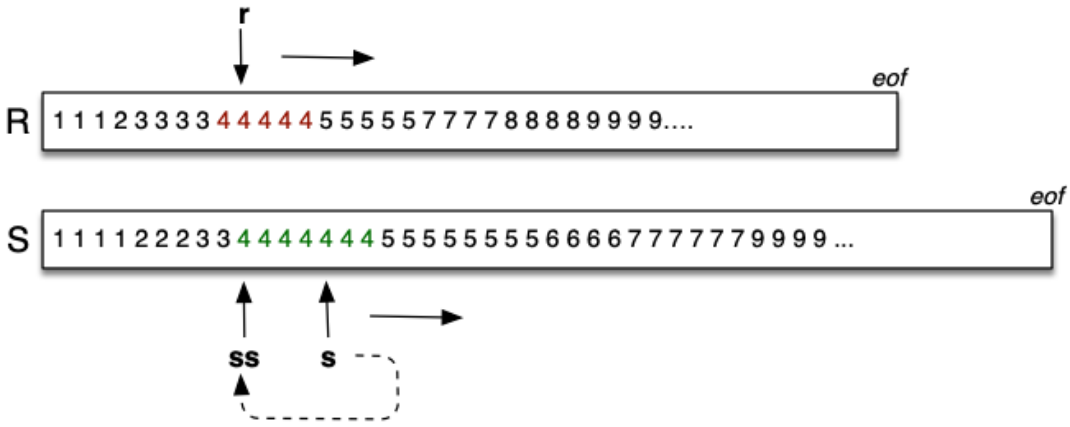
```
while (r != eof && s != eof) {
    if (r.val ≤ s.val) { output(r.val); next(r); }
    else { output(s.val); next(s); }
}
while (r != eof) { output(r.val); next(r); }
while (s != eof) { output(s.val); next(s); }
```



❖ Sort-Merge Join (cont)

Merging for join requires 3 cursors to scan sorted relations:

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



❖ Sort-Merge Join (cont)

Algorithm using query iterators/scanners:

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

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

❖ Sort-Merge Join (cont)

```
...
    // 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 (cont)

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 (cont)

Cost of sort-merge join.

Step 1: sort each relation (if not already sorted):

- Cost = $2.b_R\ (1 + ceil(log_{N-1}(b_R/N)))\ +\ 2.b_S\ (1 + ceil(log_{N-1}(b_S/N)))$
  (where N = number of memory 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

SQL query on student/enrolment database:

```
select E.subj, S.name
from   Student S join Enrolled E on (S.id = E.stude)
order  by E.subj
```

And its relational algebra equivalent:

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

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$, $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with N buffers

❖ Sort-Merge Join on Example (cont)

Case 1: *Join[id=stude](Student,Enrolled)*

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

Cost = $sort(S) + sort(E) + b_S + b_E$

= $2b_S(1+\log_{31}(b_S/32)) + 2b_E(1+\log_{31}(b_E/32)) + b_S + b_E$

= $2\times1000\times(1+2) + 2\times2000\times(1+2) + 1000 + 2000$

= $6000 + 12000 + 1000 + 2000$

= $21{,}000$

❖ Sort-Merge Join on Example (cont)

Case 2: *Join[id=stude](Student,Enrolled)*

- *Student* and *Enrolled* already sorted on *id#*
- memory buffers *N=4* (*S* input, 2 × *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)

Hash Join

❖ Hash Join

Basic idea:

- use hashing as a technique to partition relations
- to avoid having to consider all pairs of tuples

Requires sufficent memory buffers

- to hold substantial portions of partitions
- (preferably) to hold largest partition of outer relation

Other issues:

- works only for equijoin `R.i=S.j` (but this is a common case)
- susceptible to data skew (or poor hash function)

Variations: *simple*, *grace*, *hybrid*.

❖ Simple Hash Join

Basic approach:

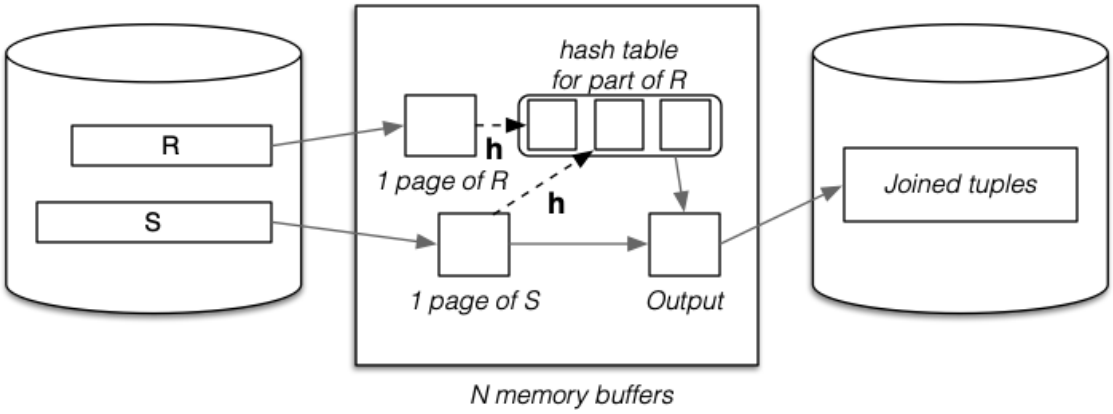
- hash part of outer relation *R* into memory buffers (build)
- scan inner relation *S*, using hash to search (probe)
  - if R.i=S.j, then h(R.i)=h(S.j) (hash to same buffer)
  - only need to check one memory buffer for each *S* tuple
- repeat until whole of *R* has been processed

No overflows allowed in in-memory hash table

- works best with uniform hash function
- can be adversely affected by data/hash skew

❖ Simple Hash Join (cont)

Data flow in hash join:



❖ Simple Hash Join (cont)

Algorithm for simple hash join *Join[R.i=S.j](R,S)*:

```
for each tuple r in relation R {
   insert r into buffer[h(R.i)]
   if (buffer[h(R.i)] is full or
                r is the last tuple in R) {
      for each tuple s in relation S {
         for each tuple rr in buffer[h(S.j)] {
            if ((rr,s) satisfies join condition) {
               add (rr,s) to result
      }  }  }
      clear all hash table buffers
   }
}
```

Best case: # join tests ≤ $r_S.c_R$ (cf. nested-loop $r_S.r_R$)

❖ Simple Hash Join (cont)

Cost for simple hash join ...

Best case: all tuples of R fit in the hash table

- Cost = $b_R + b_S$
- Same page reads as block nested loop, but less join tests

Good case: refill hash table m times (where $m \geq ceil(b_R / (N-3))$)

- Cost = $b_R + m.b_S$
- More page reads than block nested loop, but less join tests

Worst case: everything hashes to same page

- Cost = $b_R + b_R.b_S$

❖ Grace Hash Join

Basic approach (for $R \bowtie S$):

- partition both relations on join attribute using hashing (*h1*)
- load each partition of *R* into N-3*buffer hash table (*h2*)
- scan through corresponding partition of *S* to form results
- repeat until all partitions exhausted

For best-case cost ($O(b_R + b_S)$):

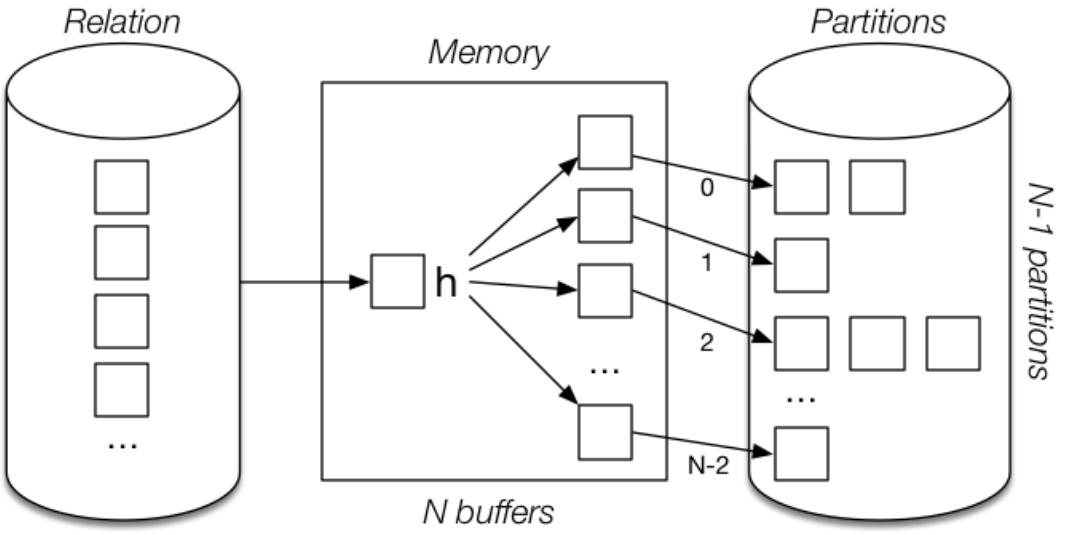
- need ≥ $\sqrt{b_R}$ buffers to hold largest partition of outer relation

If < $\sqrt{b_R}$ buffers or poor hash distribution

- need to scan some partitions of *S* multiple times

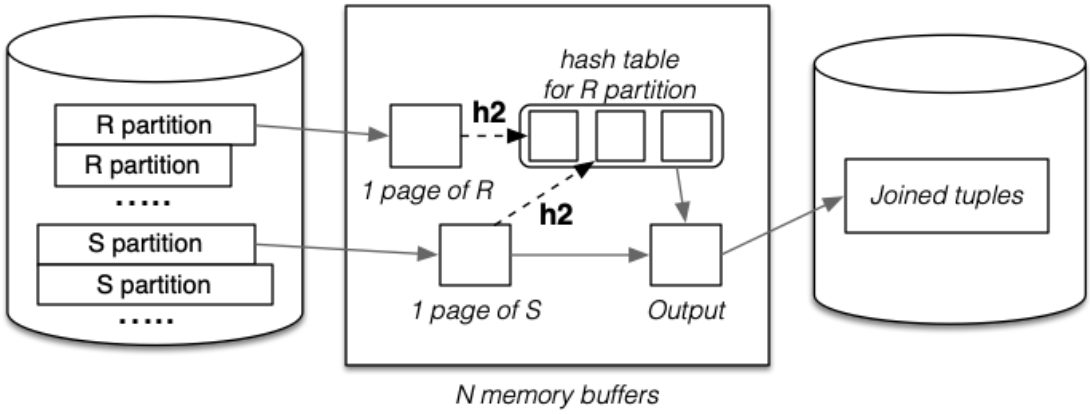
❖ Grace Hash Join (cont)

Partition phase (applied to both *R* and *S*):



❖ Grace Hash Join (cont)

Probe/join phase:



The second hash function (**h2**) simply speeds up the matching process. Without it, would need to scan entire *R* partition for each record in *S* partition.

❖ Grace Hash Join (cont)

Cost of grace hash join:

- #pages in all partition files of *Rel* ≅ b_{Rel} (maybe slightly more)
- partition relation *R* ... Cost = *read*(b_R) + *write*(≅b_R) = $2b_R$
- partition relation *S* ... Cost = *read*(b_S) + *write*(≅b_S) = $2b_S$
- probe/join requires one scan of each (partitioned) relation Cost = $b_R + b_S$
- all hashing and comparison occurs in memory ⇒ tiny cost

Total Cost = $2b_R + 2b_S + b_R + b_S$ = $3\,(b_R + b_S)$

❖ Hybrid Hash Join

A variant of grace hash join if we have $\sqrt{b_R} < N < b_R+2$

- create $k<N$ partitions, *1* in memory, *k-1* on disk
- buffers: 1 input, *k-1* output, $p = N-k-2$ for in-memory partition

When we come to scan and partition *S* relation

- any tuple with hash *0* can be resolved (using in-memory partition)
- other tuples are written to one of *k* partition files for *S*

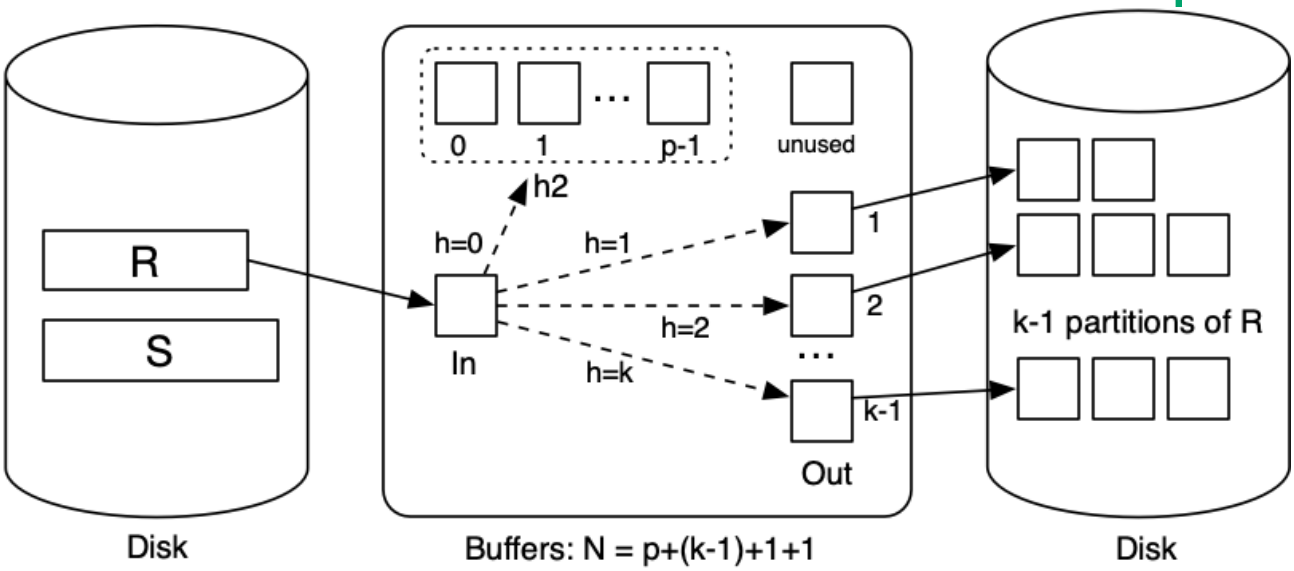
Final phase is same as grace join, but with only *k-1* partitions.

Comparison:

- grace hash join creates *N-1* partitions on disk
- hybrid hash join creates *1* (memory) + *k-1* (disk) partitions

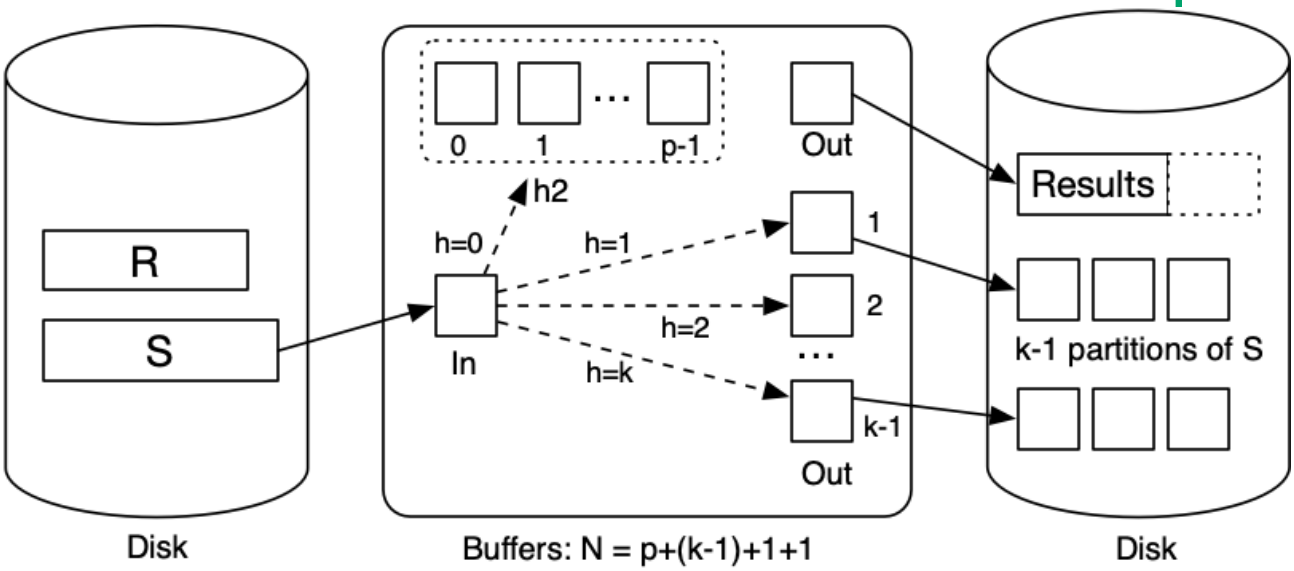
❖ Hybrid Hash Join (cont)

First phase of hybrid hash join (partitioning *R*):



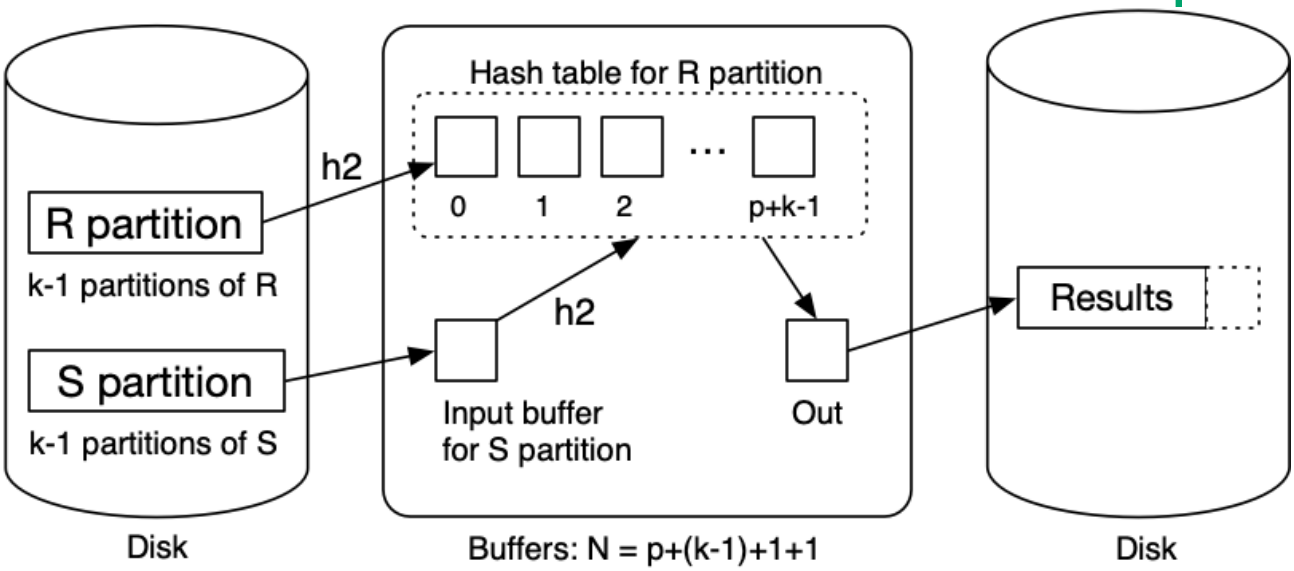
❖ Hybrid Hash Join (cont)

Next phase of hybrid hash join (partitioning *S*):



❖ Hybrid Hash Join (cont)

Final phase of hybrid hash join (finishing join):



❖ Hybrid Hash Join (cont)

Some observations:

- with *k* partitions, each partition has expected size *ceil*(b_R/*k*)
- holding 1 partition in memory needs *ceil*(b_R/*k*) buffers
- trade-off between in-memory partition space and #partitions

Other notes:

- if $N = b_R+2$, using block nested loop join is simpler
- cost depends on *N* (but less than grace hash join)

For *k* partitions, Cost = $(3-2/k).(b_R+b_S)$

❖ Costs for Join Example

SQL query on student/enrolment database:

```
select E.subj, S.name
from   Student S join Enrolled E on (S.id = E.stude)
order  by E.subj
```

And its relational algebra equivalent:

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

Database: $r_S = 20000$, $c_S = 20$, $b_S = 1000$, $r_E = 80000$, $c_E = 40$, $b_E = 2000$

We are interested only in the cost of *Join*, with N buffers

❖ Costs for Join Example (cont)

Costs for hash join variants on example (N=103):

Hash Join Method	Cost Analysis	Cost
Hybrid Hash Join	$(3-2/k).(b_S+b_E)$ = 2.8((1000+2000) assuming k = 10 ... and one partition fits in 91 pages	8700
Grace Hash Join	$3(b_S+b_E)$ = 3(1000+2000)	9000
Simple Hash Join	$b_S + b_E.ceil(b_R/(N-3))$ = 1000 + ceil(1000/100).2000 = 1000 + 10.2000	21000
Sort-merge Join	$sort(S) + sort(E) + b_S + b_E$ = 2.1000.2 + 2.2000.2 + 1000 + 2000	11000
Nested-loop Join	$b_S + b_E.ceil(b_S/(N-2))$ = 1000 + 2000.ceil(1000/101) = 1000 + 10.2000	21000

❖ Join in PostgreSQL

Join implementations are under: **`src/backend/executor`**

PostgreSQL suports three kinds of join:

- nested loop join (**`nodeNestloop.c`**)
- sort-merge join (**`nodeMergejoin.c`**)
- hash join (**`nodeHashjoin.c`**) (hybrid hash join)

Query optimiser chooses appropriate join, by considering

- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)