Implementing Join

Join

1/87

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

Filtering is achieved via selection and projection.

The *join* operation ($\bowtie$) is the primary means of *combining* information.

Because *join* is

- such an important operation in database applications/systems
- potentially very expensive to execute

many methods have been developed for its implementation.

(We use a running example to compare costs of the various join processing methods)

... Join

2/87

Types of join:

- simple equijoin (single-equality condition)

  ```
  select * from R,S where R.i = S.j
  ```

- partial-match join (conjunction of equality conditions)

  ```
  select * from R,S where R.a = S.b and R.c = S.d ...
  ```

- theta join (arbitrary expression as condition)

  ```
  select * from R,S where R.a < S.b and R.c <> S.d ...
  ```

Focus on simple equijoin, since common in practice (`R.pk=S.fk`)

Join Example

3/87

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,  ...
);
```

And the following request on this database:

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

... Join Example

4/87

The result of this request would look like:

```
Subj      Name
--------  -----------------
COMP1011  Chen Hwee Ling
```

```
COMP1011  John Smith
COMP1011  Ravi Shastri
...
COMP1021  David Jones
COMP1021  Stephen Mao
...
COMP3311  Dean Jones
COMP3311  Mark Taylor
COMP3311  Sashin Tendulkar
```

... Join Example

5/87

An 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*)))

The core of the query is the join *Join*[id=stude](*Student*,*Enrolled*)

To simplify writing of formulae, *S* = *Student*, *E* = *Enrolled*.

... Join Example

6/87

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

7/87

`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`

Join via Cross-product

8/87

Join can be defined as a cross-product followed by selection:

Join[Cond](R,S) = *Select[Cond](R × S)*

For the example query, could implement

Join$_{[id=stude]}$*(Student,Enrolled)*

as

Select[id=stude](Student × Enrolled)

Cross-product contains *20,000 × 80,000 = 1,600,000,000* tuples.

... Join via Cross-product

9/87

For `Temp` = *(Student × Enrolled)*

I/O costs:

- size of `Temp` relation $r = 16 \times 10^8$ records
- assuming $C_{Temp}=16$, then $b_{Temp} = 10^8$
- `Temp` is written once, then scanned once
- total I/O = $10^8.(T_w+T_r)$

Assuming $T_w=T_r=0.01s$, this will take around 500 hours!

... Join via Cross-product

10/87

Because

- cross-products are infrequent in practice (except to describe join)
- cross-products are large (typically **much** larger than the final join result)

DBMSs do **not** implement join via cross-product.

DBMSs implement only join and provide cross-product as:

R × S = *Join[true](R,S)*

or, in SQL

```
select * from R,S
```

Nested-Loop Join

Nested Loop Join

12/87

The simplest join algorithm:

- iteratively generates the cross-product
- checks join condition on each tuple

Algorithm to compute *Join[Cond](R,S)*:

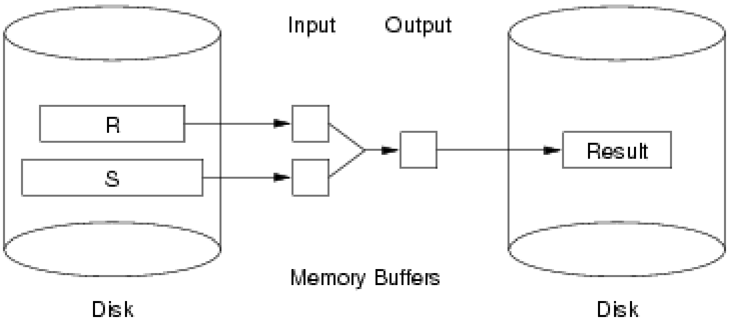
```
for each tuple r in R {
    for each tuple s in S {
        if ((r,s) satisfies join condition) {
            add (r,s) to result
}   }   }
```

R is the *outer* relation; *S* is the *inner* relation.

... Nested Loop Join

13/87

Requires (at least) three memory buffers (2 input, 1 output).



... Nested Loop Join

14/87

Abstract algorithm for *Join[Cond](R,S)* (with 3 memory buffers):

```
for each page of relation R {
    read into buffer rBuf
    for each page of relation S {
        read into buffer sBuf
        for each record r in rBuf {
            for each record s in sBuf {
                if ((r,s) satisfies Cond) {
                    add combined(r,s) to OutBuf
                    write Outbuf when full
}   }   }   }   }
```

... Nested Loop Join

15/87

Detailed algorithm for *Join[Cond](R,S)* (with 3 memory buffers):

```
// rf: file for R, sf: file for S, of: output file
outp = 0; clearBuf(oBuf);
for (rp = 0; rp < nPages(rf); rp++) {
   readPage(rf, rp, rBuf);
   for (sp = 0; sp < nPages(sf); sp++) {
      readPage(sf, sp, sBuf);
      for (i = 0; i < nTuples(rBuf); i++) {
         rTup = getTuple(rBuf, i);
         for (j = 0; j < nTuples(sBuf); j++) {
            sTup = getTuple(sBuf, j);
            if (satisfies(rTup,sTup,Cond)) {
            rsTup = combine(rTup,sTup);
            addTuple(oBuf, rsTup);
            if (isFull(oBuf)) {
               writePage(of, outp++, oBuf);
               clearBuf(oBuf);
}   }   }   }   }   }
```

... Nested Loop Join

16/87

The three-memory-buffer nested loop join requires:

- read all b_R pages of R once
- for each of page of R, read b_S pages of S

Cost = $b_R + b_R\, b_S$

If we use S as the outer relation in the join

Cost = $b_S + b_S\, b_R$

It is (slightly) better to use smaller relation as outer relation.

Nested Loop Join on Example

17/87

If `Student` is outer relation and `Enrolled` is inner:

$$\begin{aligned} \text{Cost} &= b_S + b_S\, b_E \\ &= 1{,}000 + 1{,}000 \times 2{,}000 = 2{,}001{,}000 \end{aligned}$$

If `Enrolled` is outer relation and `Student` is inner:

$$\begin{aligned} \text{Cost} &= b_E + b_E\, b_S \\ &= 2{,}000 + 2{,}000 \times 1{,}000 = 2{,}002{,}000 \end{aligned}$$

Cost of nested-loop join is too high (5 hours, if T_r=*0.01 sec*)

Implementing Join Better

18/87

Aims of effective join computation:

- generate only relevant tuples from the cross-product
- generate these tuples with minimal disk I/O

Range of costs for *Join(R,S)*

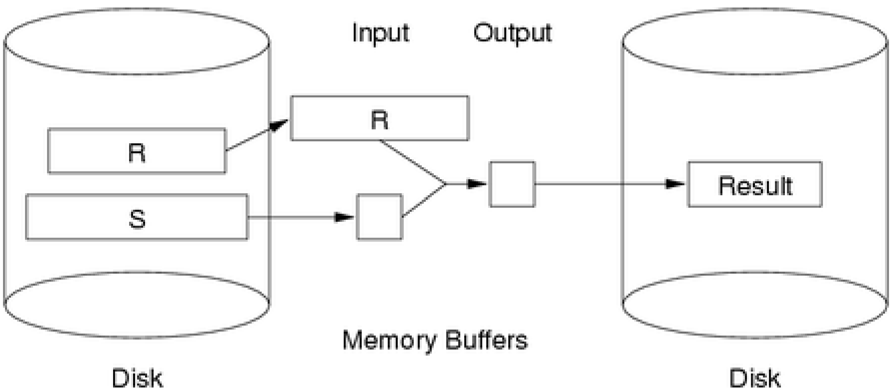
- worst case cost = $b_R + b_R b_S$ (nested loop join)
- best case cost = $b_R + b_S$ (read each page once)

Block Nested Loop Join

19/87

If at least b_R+2 memory buffers available:

- read the entire *R* relation into memory
- for each *S* page, check join condition on all `(r,s)` pairs



... Block Nested Loop Join

20/87

Algorithm for nested loop join with b_R+2 memory buffers:

```
read all of R's pages into memory buffers
for each page of relation S {
    read page into S's input buffer
    for each tuple s in S's buffer {
        for each tuple r in R's memory buffers {
            if ((r,s) satisfies JoinCond)) {
                add (r,s) to output buffer
                write output buffer when full
}   }   }   }
```

Note that *R* effectively becomes the inner relation in this scheme.

... Block Nested Loop Join

21/87

This method requires:

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

Cost = $b_R + b_S$

Notes:

- minimal I/O cost, but considers all *(r,s)* pairs
- thus, requires $r_R.r_S$ checks of the join condition

... Block Nested Loop Join

22/87

Further performance improvements:

- must reduce number of R tuples matched against each S tuple
- use access method to find small set of R tuples matching S tuple

Example:

- each S joins with $k \ll r_R$ tuples of R
- R tuples are stored in sorted array of memory buffers
- for each S tuple, use binary search to find matching buffer
- scan around that buffer to find all matching *(R,S)* pairs
- requires approx $C_R.r_S$ checks of join condition

Block Nested Loop Join on Example

23/87

If ≥ *1002* memory buffers are available:

- read `Student` relation into memory
- scan `Enrolled` relation, computing join

$$\begin{aligned} \text{Cost} &= b_S + b_E \\ &= 1{,}000 + 2{,}000 = 3{,}000 \end{aligned}$$

This is considerably better than 10^6 (30 secs vs 5 hours).

But what if we have only N memory buffers, where $N < b_R$, $N < b_S$?

... Block Nested Loop Join on Example

24/87

In general case, read *outer* relation in runs of *N-2* pages

```
for each run of N-2 pages from R {
    read N-2 of R's pages into memory buffers
    for each page of relation S {
        read page into S's input buffer
        for each tuple s in S's buffer do
            for each tuple r in R's memory buffers {
                if ((r,s) satisfies JoinCond)) {
                    add (r,s) to output buffer
                    write output buffer when full
}   }   }   }   }
```

... Block Nested Loop Join on Example

25/87

Block nested loop join requires

- read $\lceil b_R/N\text{-}2 \rceil$ runs from R
- for each run, scan b_S pages of S

Cost = $b_R + b_S \cdot \lceil b_R/N\text{-}2 \rceil$

Notes:

- the final run will typically be "short" (i.e. < *N-2* pages)
- unless index/hash is used, we still do $r_R.r_S$ tuple comparisons

---

... Block Nested Loop Join on Example

26/87

Costs for various buffer pool sizes:

N	Inner	Outer	#runs	Cost
22	`Student`	`Enrolled`	50	101,000
52	`Student`	`Enrolled`	20	41,000
102	`Student`	`Enrolled`	10	21,000
1002	`Student`	`Enrolled`	1	3,000
22	`Enrolled`	`Student`	100	102,000
52	`Enrolled`	`Student`	40	42,000
102	`Enrolled`	`Student`	20	22,000
1002	`Enrolled`	`Student`	2	4,000

Block Nested Loop Join in Practice

27/87

Why block nested loop join is very 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)

Join Conditions and Methods

28/87

Nested loop join makes no assumptions about join conditions.

```
for each pair of tuples (r,s) {
    check join condition on (r,s)
    if satisfied, add to results
}
```

To improve join:

- reduce the number of tuple pairs considered
- but not easy to do for arbitrary join condition

As noted above, simple equijoin is a common join condition.

Thus, a range of other join algorithms has been developed specifically for equality join conditions.

Index Nested Loop Join

29/87

Most joins considered so far have a common problem:

- repeated scans of *entire* inner relation *S* are required

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
}   }
```

(For ordered indexes (e.g. Btree), this also assists join conditions like $R.i<S.j$)

... Index Nested Loop Join

30/87

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*).

... Index Nested Loop Join

31/87

For index lookup:

- cost of locating first matching tuple
  - for B+ trees, typically 2-4 page reads
  - for hashing, typically 1-2 page reads
- cost of finding other matching tuples
  - if clustered, typically 1-2 page reads
  - if unclustered, up to b_q page reads

Note: building an index "on the fly" to perform a join can be very cost-effective.

Index Nested Loop Join on Example

32/87

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

- `Student` is outer and `Enrolled` is inner
- `Enrolled` has a clustered B+ tree index on `stude` field
- B+ tree has depth 3 (root + internal + leaf)
- most of the time, the four matching records are in a single page

$$\begin{aligned} Cost &= b_S + r_S\ btree_E \\ &= 1{,}000 + 20{,}000 \times (3+1.01) = 80{,}000 \end{aligned}$$

... Index Nested Loop Join on Example

33/87

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

- `Student` is outer and `Enrolled` is inner
- `Enrolled` has an unclustered B+ tree index on `stude` field
- B+ tree has depth 3 (root + internal + leaf)
- assume worst case; matching records are all on different pages

$$\begin{aligned} Cost &= b_S + r_S\ btree_E \\ &= 1{,}000 + 20{,}000 \times (3+4) = 150{,}000 \end{aligned}$$

... Index Nested Loop Join on Example

34/87

Case 3: *Join[id=stude](Student,Enrolled)*

- `Enrolled` is outer and `Student` is inner
- `Student` is hashed on `id` field (e.g. linear hashing)
- there may be (short) overflow chains (e.g. 1.1 page reads/bucket)

Cost = $b_E + r_E\ hash_S$

= $2{,}000 + 80{,}000 \times 1.1 = 90{,}000$

Optimised Index Nested Loop Join

35/87

Consider the following scenario for *Join[R.i=S.j](R,S)*:

- `R.i` is not a primary key (so many tuples have same `R.i` value)
- R is sorted on `R.i` (or could be efficiently sorted on `R.i`)
- each `R.i` value does not match very many tuples

Could save repeated index scans with the same `R.i` value

- cache results of index scan for `R.i`=*k* in buffer
- if next R tuple also has `R.i`=*k*, re-use scan results

... Optimised Index Nested Loop Join

36/87

Abstract algorithm for optimised index nested loop join:

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

Cost savings depend on repetition factor, #buffers, size of index scans

Sort-Merge Join

Sort-Merge Join

38/87

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 (relations may be sorted on join key?)

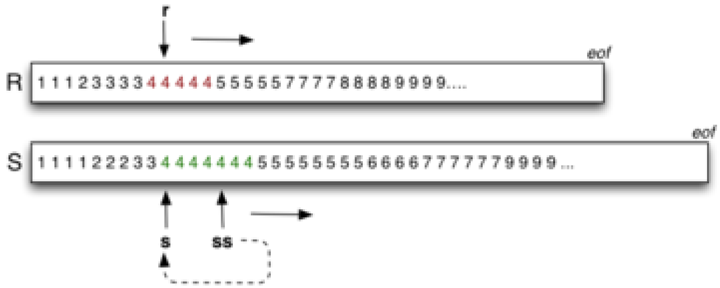
- some rescanning required when long runs of *S* tuples

... Sort-Merge Join

39/87

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

40/87

Abstract algorithm for merge phase of *Join*[*R.i*=*S.j*](*R,S*):

```
r = first tuple in R
s = first tuple in S
while (r != eof and s != eof) {
    // align cursors to start of next common run
    while (r != eof and r.i < s.j) { r = next tuple in R }
    while (s != eof and r.i > s.j) { s = next tuple in S }
    // scan common run, generating result tuples
    while (r != eof and r.i == s.j) {
        ss = s   // set to start of run
        while (ss != eof and ss.j == r.i) {
            add (r,s) to result
            ss = next tuple in S
        }
        r = next tuple in R
    }
    s = ss   // start search for next run
}
```

Sidetrack: Iterators

41/87

Sort-merge join implementation is simplified by use of iterators.

- iterators give the appearance of tuple-at-a-time
- even when the underlying data is page-by-page
- and even in the pesence of auxiliary index structures

Typical usage of iterator:

```
Iterator iter; Tuple tup;
iter = startScan("Rel","i=5");
while ((tup = nextTuple(iter)) != NULL) {
    process(tuple);
}
endScan(iter);
```

... Sidetrack: Iterators

42/87

```
typedef struct {
    File   inf;  // input file
    Buffer buf;  // buffer holding current page
    int    curp; // current page during scan
```

```
    int    curr; // index of current record in page
} Iterator;

// simple linear scan; no condition
Iterator *startScan(char *relName) {
    Iterator *iter = malloc(sizeof(Iterator));
    iter->inf  = openFile(fileName(relName),READ);
    iter->curp = 0;
    iter->curr = -1;
    readPage(iter->inf, iter->curp, iter->buf);
}
```

... Sidetrack: Iterators


```
Tuple nextTuple(Iterator *iter) {
    // check if reached end of current page
    if (iter->curr == nTuples(iter->buf)-1) {
        // check if reached end of data file
        if (iter->curp == nPages(iter->inf)-1)
            return NULL;
        iter->curp++;
        iter->buf = readPage(iter->inf, iter->curp);
        iter->curr = -1;
    }
    iter->curr++;
    return getTuple(iter->buf, iter->curr);
}
// curp and curr hold indexes of most recently read page/record
```

... Sidetrack: Iterators


```
TupleID scanCurrent(Iterator *iter) {
    // form TupleID for current record
    return iter->curp + iter->curr;
}

void setScan(Iterator *iter, int page, int rec) {
    assert(page >= 0 && page < nPages(iter->inf));
    if (iter->curp != page) {
        iter->curp = page;
        readPage(iter->inf, iter->curp, iter->buf);
    }
    assert(rec >= 0 && rec < nTuples(iter->buf));
    iter->curr = rec;
}

void endScan(Iterator *iter) {
    closeFile(iter->buf);
    free(iter);
}
```

Sort-Merge Join

Concrete algorithm using iterators:

```
Iterator *ri, *si;  Tuple rup, stup;

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

```
            stup = nextTuple(si);
        if (stup == NULL) break;
            // must have (r.i == s.j) here
...
```

... Sort-Merge Join

46/87

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

... Sort-Merge Join

47/87

Buffer requirements:

- for sort phase:
  - as many as possible (remembering that cost is $O(log_{\#Bufs})$)
  - 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

48/87

Cost of sort-merge join.

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

- Cost = $2.b_R\ (1 + log_{N-1}(b_R\ /N))\ +\ 2.b_S\ (1 + log_{N-1}(b_S\ /N))$
  (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

49/87

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

- *Student* and *Enrolled* already sorted on *id#*
- memory buffers $N=4$; all runs are of length < 2

Cost = $b_S + b_E\ =\ 3{,}000$ (i.e. minimal cost)

... Sort-Merge Join on Example

50/87

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

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

$$\begin{aligned} \text{Cost} &= sort(S) + sort(E) + b_S + b_E \\ &= b_S \lceil \log_{30} b_S \rceil + b_E \lceil \log_{30} b_E \rceil + b_S + b_E \\ &= 1{,}000 \times 3 + 2{,}000 \times 3 + 1{,}000 + 2{,}000 \\ &= 12{,}000 \end{aligned}$$

... Sort-Merge Join on Example

51/87

Case 3: *Join[id=stude](Student,Enrolled)*

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

Cost depends on which relation is outer and which is inner.

... Sort-Merge Join on Example

52/87

Case 3 (continued) ...

If E is outer relation:

- Cost $= b_E + b_S = 3{,}000$

If S is outer relation:

- one-quarter of E runs require two page reads
- each E run is processed once for matching *S.id* value
- Cost $= b_S + b_E + r_S/4 = 8{,}000$

Sidetrack 2: More on Iterators

53/87

Above description of iterators:

- involved simple scan of a single table
- with no condition to select tuples

In the general case, an iterator involves:

- one (selection) or two (join) tables
- with a condition to determine relevant tuples

A typical SQL query involves many iterators

- one for each relational operator in query plan
- connected in a demand-driven network of query nodes

... Sidetrack 2: More on Iterators

54/87

Requires a more general definition of execution state:

```
typedef struct {
    Oper   op;     // operation (sel,sort,join,...)
    Reln   r1;     // first relation
    Reln   r2;     // second relation (if any)
    Buffer *bufs;  // buffers used by operation
```

```
    int     curp1; // index of current page for r1
    int     curr1; // index of current record in page
    int     curp2; // index of current page for r2
    int     curr2; // index of current record in page
    Cond    cond;  // condition for choosing tuple(s)
} Iterator;
```

For PostgreSQL details, see **`include/nodes/execnodes.h`**

Hash Join

Hash Join

56/87

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

57/87

Basic approach:

- hash the inner relation *R* into memory buffers (build)
- scan the outer 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

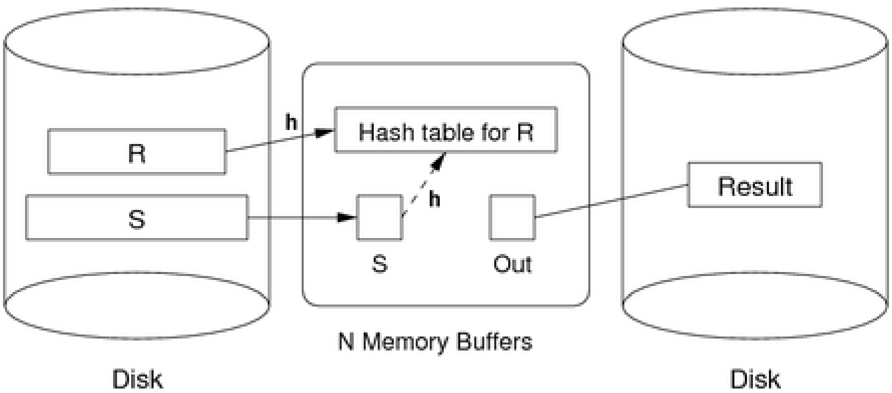
Makes the assumption: whole of *S* hashes into memory

- requires *R* to be smaller than memory buffers
- requires a uniform hash function (no overflows)

... Simple Hash Join

58/87

Data flow:



... Simple Hash Join

59/87

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

```
for each tuple r in relation R
   { insert r into buffer[h(R.i)] }
for each tuple s in relation S {
   for each tuple r in buffer[h(S.j)] {
      if ((r,s) satisfies join condition) {
         add (r,s) to result
      }
   }
}
```

Cost = $b_R + b_S$ (minimum possible cost)

... Simple Hash Join

60/87

Consider that we have *N* buffers available.

If $b_R \le N\text{-}2$ buffers, no need to hash   (use nested loop).

In practice, size of hash table $b_{hR} > b_R$ (e.g. data skew)
⇒ hash table for *R* is even less likely to fit in memory

Can be handled by a variation on above algorithm:

- scan *R*, making hash table with *N-2* buffers
- once hash table built, scan *S*   (standard probe phase)
- if more *R* tuples, build new table and repeat

... Simple Hash Join

61/87

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

```
for each tuple r in relation R {
   if (buffer[h(R.i)] is full) {
      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
   }
   insert r into buffer[h(R.i)]
}
```

Note: requires multiple passes over the *S* relation.

... Simple Hash Join

62/87

Cost depends on *N* and on properties of data/hash.

Worst case:

- $h(i)=k$ so read only C_R tuples before hash table "full"
- each hash table for *R* occupies one buffer with C_R tuples
- degenerates to nested-loop-with-3-buffers case ⇒ $b_R + b_R b_S$

Best case:

- perfect uniform distribution of hash values
- each hash table of *R* holds $(N\text{-}2)C_R$ tuples from *N-2* pages
- number of hash tables built = $n_{hR} = \lceil b_R / (N\text{-}2) \rceil$
- read all of *S* for each hash table ⇒ $b_R + n_{hR}.b_S$

Grace Hash Join

63/87

Basic approach:

- partition both relations on join attribute using hashing
- scan through corresponding pairs of partitions to form results

Similar approach to sort-merge join, except:

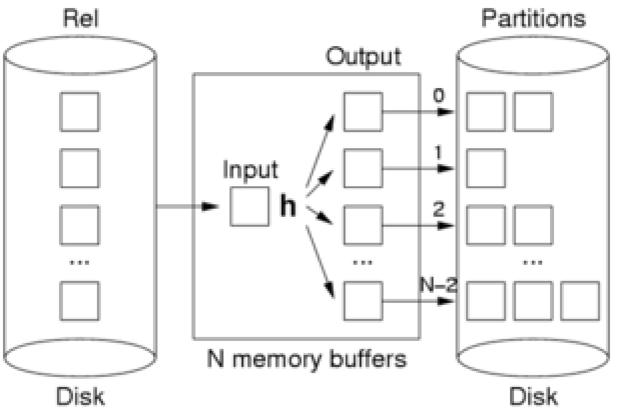
- sort-merge: partitioning achieved by sorting (runs)
- hash: partitioning achieved by hashing

Requires enough buffer space to hold largest partition of inner relation.

... Grace Hash Join

64/87

Partition phase:

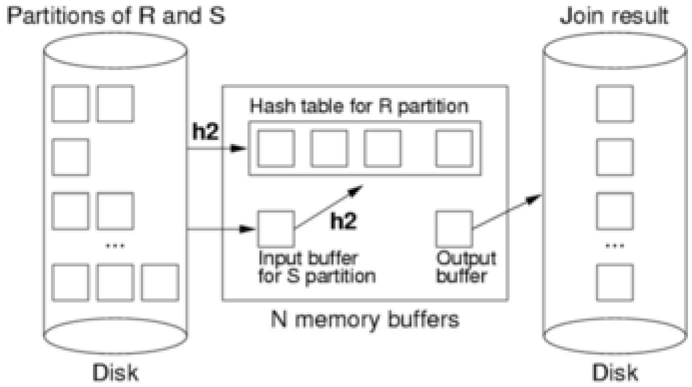


This is applied to each relation *R* and *S*.

... Grace Hash Join

65/87

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

66/87

Abstract algorithm for *Join[R.i=S.j](R,S)*:

```
// assume h(val) generates [0..N-2]
// assume h2(val) generates [0..N-3]

// Partition phase (each relation -> N-1 partitions)
// 1 input buffer, N-1 output buffers

for each tuple r in relation R
    add r to partition h(r.i) in output file R'
```

```
for each tuple s in relation S
    add s to partition h(s.j) in output file S'
...
```

... Grace Hash Join

67/87

Abstract algorithm for *Join[R.i=S.j](R,S)* (cont.)

```
// Probe/join phase
// 1 input buffer for S, 1 output buffer
// N-2 buffers to build hash table for R partition

for each partition p = 0 .. N-2 {
    // Build in-memory hash table for partition p of R'
    for each tuple r in partition p of R'
        insert r into buffer h2(r.i)

    // Scan partition p of S', probing for matching tuples
    for each tuple s in partition p of S' {
        b = h2(s.j)
        for all matching tuples r in buffer b
            add (r,s) to result
}   }
```

... Grace Hash Join

68/87

Concrete algorithm for partitioning:

```
Buffer iBuf, oBuf[N-1];
File inf, outf[N-1]; char rel[100];
int i, r, h, ip, op[N-1]; Tuple tup;
for (i = 0; i < N-1; i++) {
    clearBuf(oBuf[i]);   op[i] = 0;
    rel = sprintf("%s%d","Rel",i);
    outf[i] = openFile(fileName(rel),WRITE));
}
inf = openFile(fileName("Rel"),READ);
for (ip = 0; ip < nPages(inf); ip++) {
    iBuf = readPage(inf, ip);
    for (r = 0; r < nTuples(iBuf); r++) {
        tup = getTuple(iBuf, r);
        h = hash(tup.i, N-1);
        addTuple(oBuf[h], tup);
        if (isFull(oBuf[h])) {
            writePage(outf[h], op[h]++, oBuf[h]);
            clearBuf(oBuf[h]);
}   }   }
```

... Grace Hash Join

69/87

Cost of grace hash join:

- #pages in all partition files of *Rel* $\cong b_{Rel}$ (maybe slightly more)
- partition relation *R* ... Cost = $b_R.T_r + b_R.T_w = 2b_R$
- partition relation *S* ... Cost = $b_S.T_r + b_S.T_w = 2b_S$
- probe/join requires one scan of each (partitioned) relation
  Cost = $b_R + b_S$
- all hashing and comparison occurs in memory ⇒ $\cong 0$ cost

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

... Grace Hash Join

70/87

The above cost analysis assumes:

- every partition of R fits in memory buffers at once

We achieve this situation if:

- data has uniform distribution
- hash function gives uniform distribution
  (all partitions are similar size)
- we have $N\text{-}1 \geq \lceil \sqrt{b} \rceil$ memory buffers
  (giving $N\text{-}1$ partitions, each with $\cong b_R/(N\text{-}1)$ pages)

... Grace Hash Join

71/87

Possibilities for dealing with "over-long" partitions of R

- handle each over-long partition via scanning
  - requires over-long partitions to be scanned multiple times
  - essentially, such partitions are treated via nested loop join
- apply hash join recursively to over-long partitions
  - increases i/o by needing to partition parts of file multiple times
- use a different hash function with better distribution properties
  - but difficult to find such hash functions "on the fly"
- use the relation with the best partitioning as the "outer" relation

---

Grace Hash Join on Example

72/87

For the example $Join_{[id=stude]}(Student,Enrolled)$:

- assume that we have a good hash function and $N = \sqrt{1000} = 32$

$$\begin{aligned} \text{Cost} &= 3\,(b_S + b_E) \\ &= 3\,(1{,}000 + 2{,}000) = 9{,}000 \end{aligned}$$

Hybrid Hash Join

73/87

An optimisation if we have $\sqrt{b_R} < N < b_R+2$

- create k partitions using N buffers where $k \ll N$
- with grace join, would use k output buffers (one per partition)
- what to do with $N\text{-}k$ remaining buffers? (ignore input buffer)
- use them to hold m partitions of R in memory (no disk writes)
- other partitions are handled as before (using $k\text{-}m$ output buffers)

When we come to scan and partition S relation

- any tuple with hash in range $0..m\text{-}1$ can be resolved
- other tuples are written to one of $k\text{-}m$ partition files for S

Final phase is same as grace join, but with only $k\text{-}m$ partitions.

... Hybrid Hash Join

74/87

Some observations:

- for k partitions, each partition has expected size $ceil(b_R/k)$
- holding m partitions in memory needs $m \times ceil(b_R/k)$ buffers
- since we have $k\text{-}m$ output buffers, we must have $mb_R/k + (k\text{-}m) \leq N$
- for every partition/block held in memory, we save on disk i/o
- saving is $m/k \times 2(b_R + b_S)$

Other notes:

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

... Hybrid Hash Join

75/87

Need to choose appropriate *m* and *k* to minimise cost

- base cost: $3 \times (b_R + b_S)$ (grace join)
- i/o saving: $m/k \times 2(b_R + b_S)$
- constraint: $mb_R/k + (k-m) \le N$

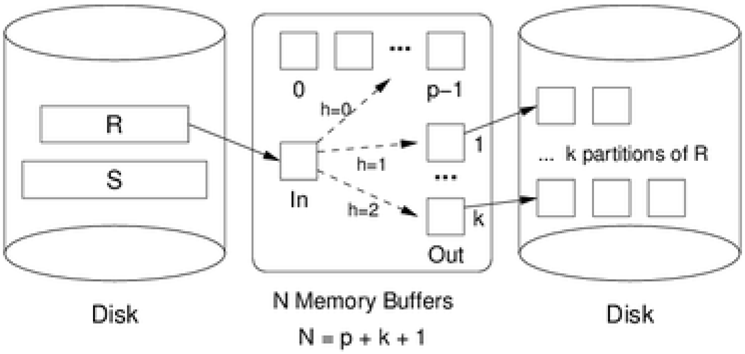
Approach to maximise saving:

- have one large in-memory partition ($m = 1$)
- use as many as possible of *N* buffers for partition
- use as few output buffers as possible (minimise *k*)

... Hybrid Hash Join

76/87

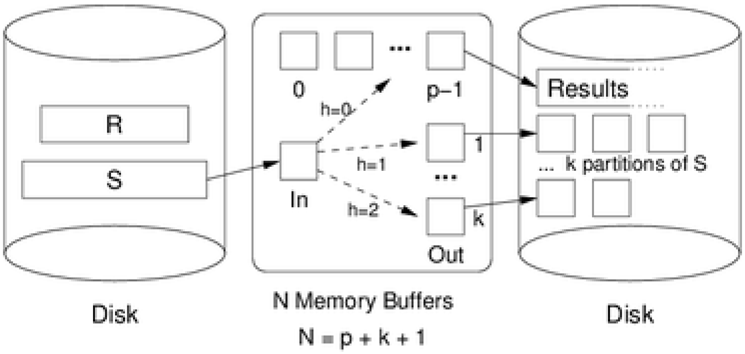
Data flow for hybrid hash join (partitioning *R*):



... Hybrid Hash Join

77/87

Data flow for hybrid hash join (partitioning *S*):



After this, proceed as for grace hash join.

... Hybrid Hash Join

78/87

Cost of hybrid hash join:

- assume: large *N* total buffers, *m* partitions in memory, *k* partitions on disk
- read both tables: $b_R + b_S$
- total partitions for each table: $m+k$
- assuming uniform hashing, #pages in each R partition $P_R = ceil(b_R/(m+k))$
- assuming uniform hashing, #pages in each S partition $P_S = ceil(b_S/(m+k))$
- in Pass 1, $k*P_R + k*P_S$ pages written to disk partitions
- all joining of *m* in-memory partitions is handled in memory
- in Pass2, $k*P_R + k*P_S$ pages read back from disk partitions

$$\begin{aligned} \text{Cost} &= b_R + b_S + k*P_R + k*P_S + k*P_R + k*P_S \\ &= b_R + b_S + 2 * k * (P_R + P_S) \end{aligned}$$

$= b_R + b_S + 2 * k * (ceil(b_R/(m+k)) + ceil(b_S/(m+k)))$

How to determine k:

- *set m=1 and so size of partition ≅ N ⇒ k ≅ b_R/N*
- *need to ensure that $\lceil b_R/k \rceil + k \leq N$ (allowing for input buffer)*
- *choose k close to b_R/N but satisfying constraint*

Hybrid Hash Join on Example

Case 1: N = 100 buffers, b_R = 1000

- *k = 10 ⇒ 1000/10 + 10 = 110 buffers; not less than 100*
- *k = 12 ⇒ 1000/12 + 12 = 96 buffers*
- *Cost = (3-2/12).(1000+2000) = 8500*

Case 2: N = 200 buffers, b_R = 1000

- *k = 5 ⇒ 1000/5 + 5 = 205 buffers; not less than 200*
- *k = 6 ⇒ 1000/6 + 6 = 173 buffers*
- *Cost = (3-2/6).(1000+2000) = 8000*

Case 3: N = 502 buffers, b_R = 1000

- *k = 2 ⇒ 1000/2 + 2 = 502 buffers*
- *Cost = (3-2/2).(1000+2000) = 6000*

Pointer-based Join

Conventional join algorithms set up R ↔ S connections via attribute values.

Join could be performed faster if direct connections already existed.

- *in OODBMSs, they generally already exist in the form of object references (oids)*
- *in RDBMSs, they could be introduced via extra rid attributes*

Such a modification to conventional RDBMS structure would be worthwhile:

- *if we know in advance what kind of joins will be required*
- *adding the extra rid attributes into tuples is feasible*

... Pointer-based Join

The basic idea for pointer-based join is:

```
for each tuple r in relation R {
    for each rid associated with r {
        fetch tuple s from S via rid
        add (r,s) to result relation
    }
}
```

Often, each R tuple is associated with only one rid, so the inner loop is not needed.

... Pointer-based Join

The advantage over value-based joins:

- *rather than find S tuples via value-based lookup (e.g. hashing, index)*
- *we find S tuples by direct fetch with rid (much faster per tuple)*
- *requires no assumption about sorted-ness of relations*
- *does not require large numbers of buffers*

The (potential) disadvantages:

- *every `fetch` goes to a different page of S*
  (this essentially returns us to the worst-case scenario for nested-loop join)
- *the join only works in "one direction" (from R to S)*
- *requires additional data for each different join type*
- *requires tuples to be larger ⇒ b_R is larger*

General Join Conditions

83/87

Above examples all used simple equijoin e.g. $Join_{[i=j]}(R,S)$.

For theta-join e.g $Join_{[i<j]}(R,S)$:

- *index nested loop join: need B+ tree index on inner relation*
- *sort-merge join can be adapted, but is not very effective*
- *hash join is inapplicable*
- *other methods are essentially unchanged*

... General Join Conditions

84/87

For multi-equality (pmr) join e.g. $Join_{[i=j \wedge k=l]}(R,S)$

- *index nested loop join:*
  - *build index on all join fields of inner relation*
  - *e.g. if S is inner, build index on (S.j,S.l)*
- *sort-merge join:*
  - *sort both relations on combined join fields*
  - *e.g. sort R on (R.i,R.k), sort S on (S.j,S.l)*
- *hash-join:*
  - *use multi-attribute hashing on combined join fields*
- *other methods are essentially unchanged*

Join Summary

85/87

No single join algorithm is superior in some overall sense.

Which algorithm 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*

Choosing the "best" join algorithm is critical because the cost difference between best and worst case can be very large.

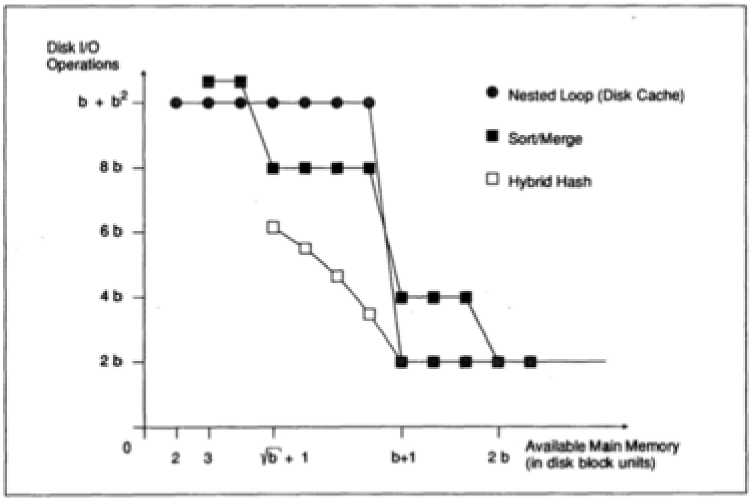
E.g. $Join_{[id=stude]}(Student,Enrolled)$: 3,000 ... 2,000,000

In some cases, it may be worth modifying access methods "on the fly" (e.g. add index) to enable an efficient join algorithm.

... Join Summary

86/87

Comparison of join costs *(from Zeller/Gray VLDB90, assumes $b_R = b_S = b$)*



Join in PostgreSQL

87/87

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)*

Produced: 17 Jul 2019