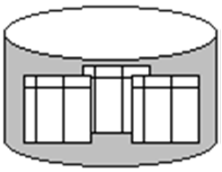
QUERY EXECUTION: How to Implement Relational Operations?



Introduction

- We've covered the basic underlying storage, buffering, indexing and sorting technology
  - Now we can move on to query processing
- Relational operators can be composed
  - Each relational operator takes as input and returns a relation
  - Optimize queries by composing operations in different ways
- Some database operations are EXPENSIVE
- Can greatly improve performance by being "smart"
  - e.g., can speed up 1,000,000x over naïve approach
- Main weapons are:
  - clever implementation techniques for operators
  - exploiting "equivalencies" of relational operators
  - using statistics and cost models to choose among these

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation
  - *Projection* (π) Deletes unwanted columns from relation
  - *Join* (⋈) Allows us to combine two relations
  - *Set-difference* (—) Tuples in Rel. 1, but not in Rel. 2
  - *Union* (∪) Tuples in Rel. 1 and in Rel. 2
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY

Evaluation of Relational Operators

- Techniques to implement operators
  - Iteration
  - Indexing
  - Partitioning
- Access paths denote alternative *algorithms* + *data structures* used to retrieve tuples from a relation
  - File scan
  - Binary search
  - Index and matching selection condition
- Selectivity of access paths
  - Number of pages retrieved: index + data pages
  - Most selective access path minimize retrieval cost

Schema for Examples

```
Sailors (sid:integer, sname:string, rating:integer, age:real)
Reserves (sid:integer, bid:integer, day:dates, rname:string)
```

- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages

Simple Selections

```
SELECT  *
FROM    Reserves R
WHERE   R.sid >= 35
```

- Algebraic form $\sigma_{R.attr\ op\ value}(R)$
- Question: how best to perform? Depends on:
  - what indexes/access paths are available
  - what is the expected size of the result (in terms of number of tuples and/or number of pages)
- Size of result (cardinality) approximated as

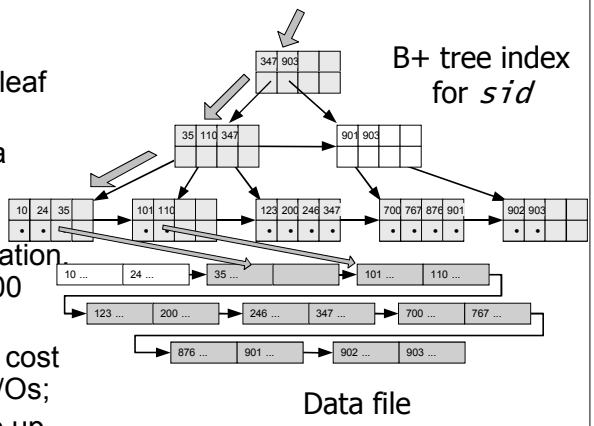
  *size of R * reduction factor*
  - "reduction factor" is usually called selectivity
  - estimate selectivity is based on statistics

Simple Selections

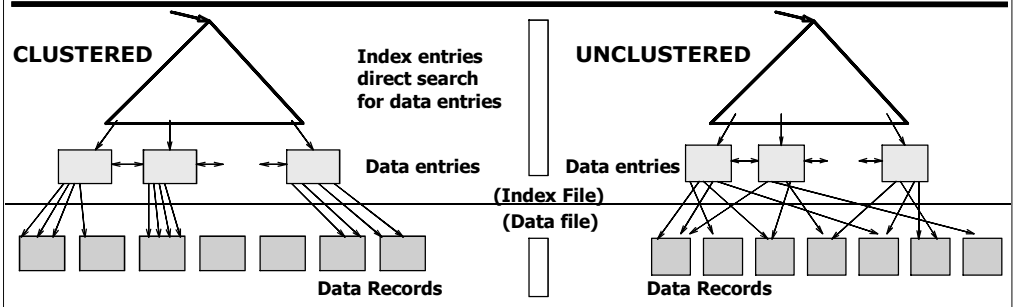
- With no index, unsorted:
  - Must essentially scan all data pages of the relation to find the rows satisfying selection condition: cost is M (#pages in R)
  - For "Reserves" = 1000 I/Os
- With no index, sorted:
  - Use binary search (O ($\log_2$M)) to locate first data page containing row in which `attr = value`
  - Scan further to get all rows satisfying `attr op value`: cost is O ($\log_2$M) + ⌈selectivity*#pages⌉
  - For "Reserves" = 10 I/Os + ⌈selectivity * 1000⌉
- With an index on selection attribute:
  - Use index to find first index entry that points to a qualifying tuple of R
  - Scan leaf pages of index to retrieve all data records in which key value satisfies selection condition
  - Retrieve corresponding data records
  - Cost?

Selections using Index

- Cost depends on #qualifying tuples, and clustering
  - Cost:
    - 2-3 I/Os to get starting leaf page (typically small)
    - + cost of retrieving data records (could be large w/o clustering)
  - In example "Reserves" relation, if 10% of tuples qualify (100 pages, 10000 tuples)
    - With a clustered index, cost is little more than 100 I/Os;
    - If unclustered, could be up to 10000 I/Os!
      - Unless you get fancy…



Selections using Unclustered Indexes

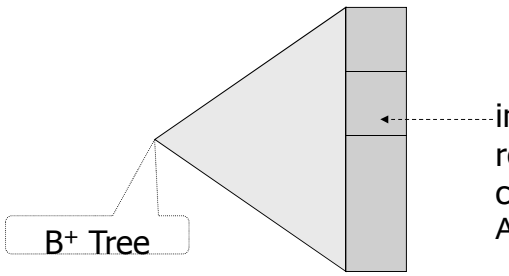


- Important refinement for unclustered indexes:
  - ❶Find qualifying tuples
  - ❷Sort the *rid*'s of the data records to be retrieved
  - ❸Fetch *rids* in order
    - This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering)

Computing Selection $\sigma_{(attr\ \ op\ \ value)}$ using Index

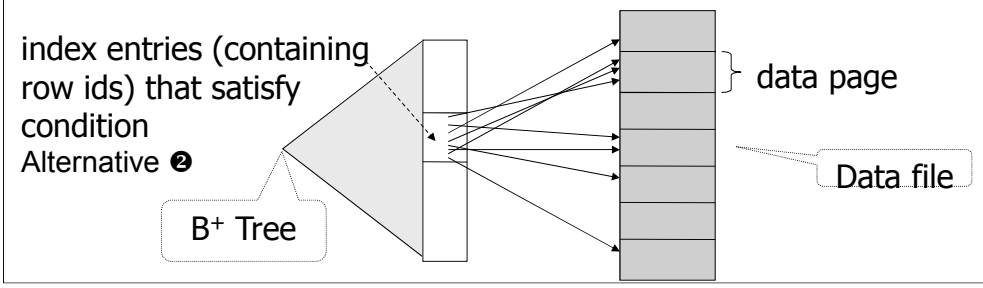
- Clustered B^+ tree index on `attr` (for equality or range search):
  - Locate first index entry corresponding to a data record in which `attr=value` Cost = depth of tree
  - Data records satisfying condition packed in sequence in successive data pages; scan those pages

    Cost: number of pages occupied by qualifying data records



Computing Selection $\sigma_{(attr\ \ op\ \ value)}$ using Index

- Unclustered B+ tree index on `attr` (for equality or range search):
  - Locate first index entry corresponding to a data record in which `attr=value`
    - Cost = depth of tree
  - Index entries with pointers to data records satisfying condition are packed in sequence in successive index pages
    - Scan entries & sort *rid*s to identify pages with qualifying data records
      - Each page with at least one such record must be fetched once
    - Cost: number of data records that satisfy selection condition

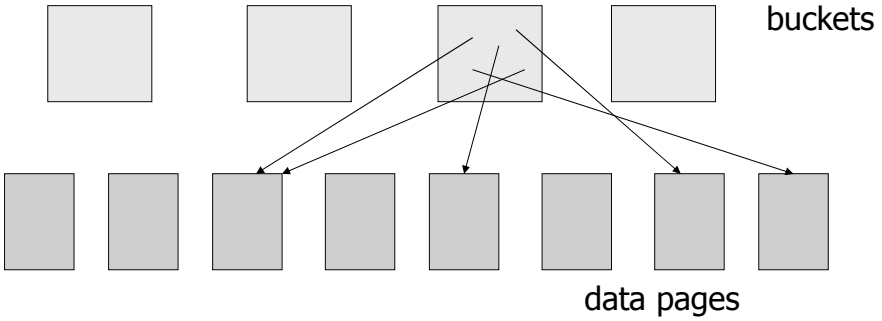


Computing Selection $\sigma_{(attr\ \ =\ \ value)}$ using Index

- Hash index on attr (for equality search only):
  - Hash on *value* Cost
    - ≈ 1.2 (primary) - 2.2 (secondary) – typical average cost of hashing (>1 due to possible overflow chains)
    - Finds the (unique) bucket containing all index entries satisfying selection condition
    - Clustered index – all qualifying data records packed in the bucket (a few pages)
      - Cost: number of pages occupies by the bucket
    - Unclustered index – sort *rid*s in the index entries to identify pages with qualifying data records
      - Each page with at least one such record must be fetched once
      - Cost: number of data records in bucket

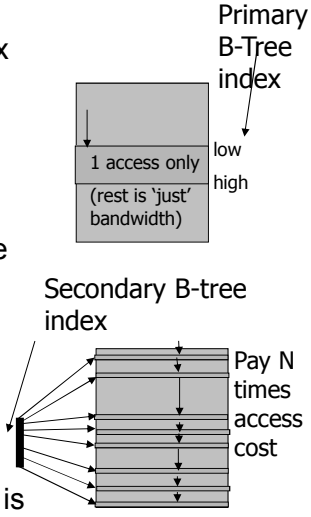
Computing Selection $\sigma_{(attr = value)}$ using Index

- Unclustered hash index on attr (for equality search)



Choosing Indices

- DBMSs may allow user to specify
  - Type (hash, B+ tree) and search key of index
  - Whether or not it should be clustered
- Using information about the frequency and type of queries and size of tables, designer can use cost estimates to choose appropriate indices
- Several commercial systems have tools that suggest indices
  - Simplifies job, but index suggestions must be verified
- Examples
  - If a frequently executed query that involves selection or a join and has a large result set, use a clustered B+ tree index
  - If a frequently executed query is an equality search and has a small result set, an unclustered hash index is best
    - Since only one clustered index on a table is possible, choosing unclustered allows a different index to be clustered



General Selection Conditions

```
(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3
```

- Such selection conditions are first converted to Conjunctive Normal Form (CNF):
  - (day<8/9/94 OR bid=5 OR sid=3) AND (rname='Paul' OR bid=5 OR sid=3)
- We only discuss the case with no ORs
  - a conjunction of terms of the form `attr` **op** `value`

Index Matching

- Hash index match selection condition if
  - Condition has conjunctive terms of the form
    `attribute = value`
  - for each attribute in the index search key
- Example:
  - Hash index on search key *<rname, bid, sid>*
  - Can retrieve tuples that satisfy condition
    `rname='Joe' AND bid=5 AND sid=3`
  - Cannot retrieve all tuples that satisfy condition
    `rname='Joe' AND bid=5`
  - or conditions on other attributes

Index Matching

- B-Tree index match selection condition if
  - Condition has conjunctive terms of the form
    `attribute` **`op`** `value`
  - for each attribute in a *prefix* of index search key
- Example:
  - B+ tree index on search key *<rname, bid, sid>*
  - Can retrieve tuples that satisfy conditions
    `rname='Joe' AND bid=5 AND sid=3`
    `rname='Joe' AND bid=5`
  - Cannot retrieve tuples that satisfy condition
    `sid= 5 AND bid=3`

17

Two Approaches to General Selections

- First approach:
  - ❶Find the most selective access path
  - ❷Retrieve tuples using it, and
  - ❸Apply any remaining terms that don't match the index
- *Most selective access path:*
  - An index or file scan that we estimate will require the fewest page I/Os
- Terms that match this index reduce the number of tuples *retrieved*;
  - other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched

18

Most Selective Index - Example

- Example: `day<8/9/94 AND bid=5 AND sid=3`
  - A B+ tree index on *day* can be used;
    - then, `bid=5 AND sid=3` must be checked for each retrieved tuple
  - Similarly, a hash index on <*bid, sid*> could be used;
    - then, `day<8/9/94` must be checked
  - How about a B+tree on <*rname,day*> ?
  - How about a B+tree on <*day, rname*> ?
  - How about a Hash index on <*day, rname*> ?

Intersection of Rids

- Second approach: if we have two or more matching indexes :
  - ❶Get sets of rids of data records using each matching index
  - ❷Then intersect these sets of rids
  - ❸Retrieve the records and apply any remaining terms
- Example: `day<8/9/94 AND bid=5 AND sid=3`
  - With a B+ tree index on *day* and a hash index on *sid*, we can retrieve rids of records satisfying *day<8/9/94* using the first, rids of records satisfying *sid=3* using the second, then intersect, retrieve records and check *bid=5*
  - Note: commercial systems use various tricks to do this:
    - bit maps, bloom filters, index joins

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
  - A bitmap is simply an array of bits
- Records in a relation are assumed to be numbered sequentially
  - Given a number *n* (from 0) it must be easy to retrieve record *n*
- Applied on attributes with a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Bitmap indices generally very small compared with relation size

Bitmap Indices

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*name*	*gender*	*address*	*income-level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices

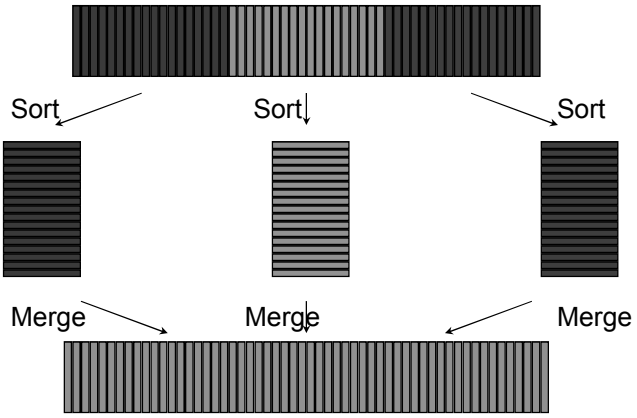
- Queries are answered using bitmap operations
  - *Intersection* (and)
  - *Union* (or)
  - *Complementation* (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve qualifying data records
    - Counting number of qualifying data records is even faster
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A=v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A=v*) :
    - intersect above result with (NOT *bitmap-A-Null*)

Projection

```
SELECT DISTINCT
       R.sid, R.bid
FROM   Reserves R
```

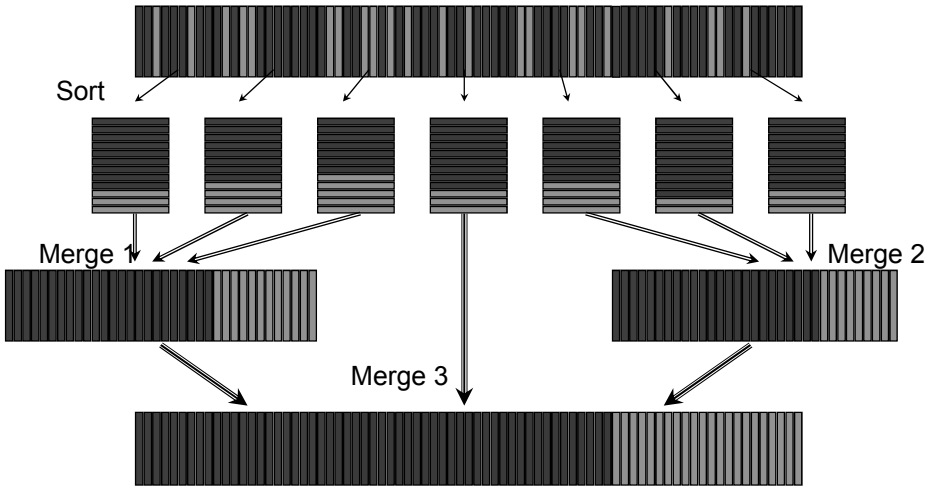
- Algebraic form $\pi_{R.attr\,1, R.attr\,2, \ldots}(R)$
- To implement projection
  - Remove unwanted attributes
  - Eliminate any duplicate tuples
- Sort-based projection
- Hash-based projection

Big Sorts



●One-pass sort: The data has been read, sorted, and dumped to disc in chunks, then re-read once to be merged into order, and dumped again

Huge Sorts



●Multi-pass sort: After sorting the data in chunks, we can't re-read the top of every chunk simultaneously, so we have multiple merge passes

Sort Based Projection

❶ Scan R, to produce a set of tuples containing only the desired attributes (why do this 1st?)
- Cost M I/Os to scan R +
- Cost T I/Os to write temp relation R',
  where M is # pages of R and T is # pages of R'

❷ Sort tuples using combination of attributes as key
- Cost = O (T log T)

❸ Scan sorted result, compare adjacent tuples and discard duplicates
- Cost = T

27

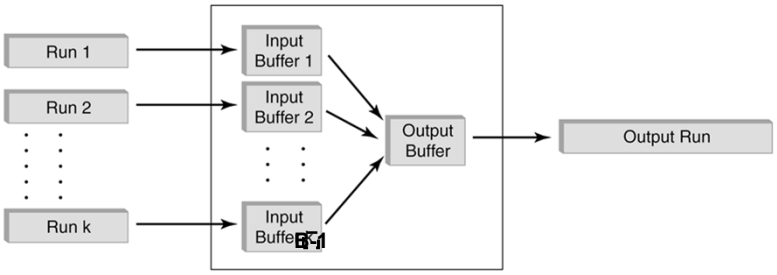
Example: Sort-based Proj. on "Reserves"

- Step 1:
  - Scan "Reserves" with 1000 I/Os
  - If a tuple in R' is 10 bytes (i.e., *size ratio* 0.25) we need 250 I/Os to write out R'
- Step 2:
  - Given 20 buffer pages, sort R' in two passes at a cost of 2*2*250 I/Os
    - # passes = $\lceil \log_{20-1} \lceil 250/20 \rceil \rceil + 1 = 2$
    - per pass = 2 * 250
- Step 3:
  - 250 I/Os to scan for duplicates
- Total cost: 1000 + 250 + 1000 + 250 = 2500 I/Os

28

Sort Based Projection Improvements

- Project out unwanted fields during Pass 0 of sorting
  - Read in B pages of R and write out (2)B internally sorted pages (runs) of temporary relation
  - Tuples in *runs* are smaller than input tuples
    - *Size ratio* depends on number and size of fields dropped
- Eliminate duplicates during the merging passes
  - Number of result tuples smaller than input
  - Difference depends on number of duplicates (very first merging pass)



29

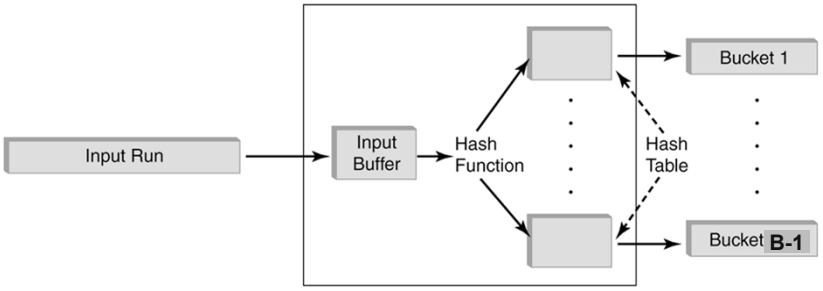
Example: Tournament Sort-based Proj. on "Reserves"

- First pass:
  - Scan "Reserves" with 1000 I/Os
  - Write out R' with 250 I/Os
    - With 20 buffer pages, the 250 pages are written out as 7 internally sorted *runs*, each (except the last) about 40 pages (instead of 250/20 = 13 *runs* of 20 pages)
- Second pass:
  - Read the runs at a cost of 250 I/Os and merge them
- Total cost: 1000 + 250 + 250 = 1500 I/Os

30

Hash-Based Projection

- Partitioning
  - One input buffer page and B-1 output buffer pages
  - Read R using one input buffer
  - For each tuple do
    - Discard unwanted fields
    - Apply hash function **h** to combination of remaining fields
    - Output tuple to one of the B-1 output buffer pages



31

Hash-Based Projection

- Duplicate elimination
  - For each partition produced in the first phase
    - Read in one page at a time
    - Build an in-memory hash table by applying a hash function **h2** (≠**h**) to each tuple
    - If a new tuple hashes to the same value as some existing tuple, check if new tuple is a duplicate
    - Discard duplicates as they are detected
  - After entire partition has been read in, write tuples in hash table to result file
  - Clear in-memory hash table for next partition

32

Hash-Based Projection Cost

- Partitioning
  - Read R = M I/Os
  - Write R' = T I/Os
- Duplicate elimination
  - Read in partitions = T I/Os
- Total cost = M + 2T I/Os
- Example: "Reserve" relation
  - assuming partitions fit in memory
    - i.e. B >= $\sqrt{T}$ T is #of pages of projected tuples
  - read 1000 pages and write out partitions of projected tuples
    - 250 pages
  - do duplicates elimination on each partition
    - total 250 page reads
  - Total : = 1000 + 2*250 = 1500 I/O

Remarks on Projections

- Sort-based projection is the standard approach
  - Better if we have many duplicates or if the distribution of (hash) values in very non-uniform (data skew)
  - Result is sorted
  - Same tricks apply to GROUP BY/Aggregation
- Index on relation contains all wanted attributes in its search key
  - Retrieve the key values from the index without accessing the actual relation: index only scan
  - Apply projection techniques this smaller set of pages
- Ordered tree index contains all wanted attributes as prefix of search key, even better:
  - Retrieve data entries in order (index-only scan)
  - Discard unwanted fields
  - Compare adjacent tuples to check for duplicates
- Commercial Systems
  - Informix uses hashing, IBM DB2, Oracle 8 and Sybase ASE uses sorting. Microsoft SQL Server and Sybase ASIQ implement both hash-based and sort-based algorithms

Joins

```
SELECT *
FROM   Reserves R1, Sailors S1
WHERE  R1.sid=S1.sid
```

- Algebraic form: R ⋈ S
  - Joins are very common
  - Joins are very expensive
- Join operation can be implemented by
  - Cross-product e.g,. R × S
  - Followed by selections and projections
- Inefficient
  - Cross-product much larger than result of a join
- Note: join is associative and commutative

35

Joins

- Techniques to implement join
  - Iteration
  - Simple Nested Loops
  - Block Nested Loops
- Indexing
  - Index Nested Loops
- Partition
  - Sort Merge Join
  - Hash Join
- Factors affecting performance
  - Available buffer pages
  - Choice of inner vs. outer relation
  - Join Selectivity factor

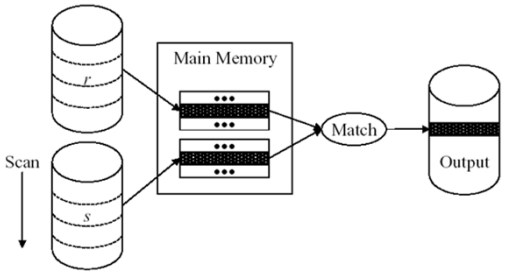
- Assume
  - M pages in R, p_R tuples per page
  - N pages in S, p_S tuples per page
- Cost metric : # of I/Os
  - We will ignore output costs
- If R is "Reserves" and S is "Sailors"
  - M = 1000, p_R = 100
  - N = 500, p_S = 80
- Focus on natural joins with one join column
  - We will consider more complex join conditions later

36

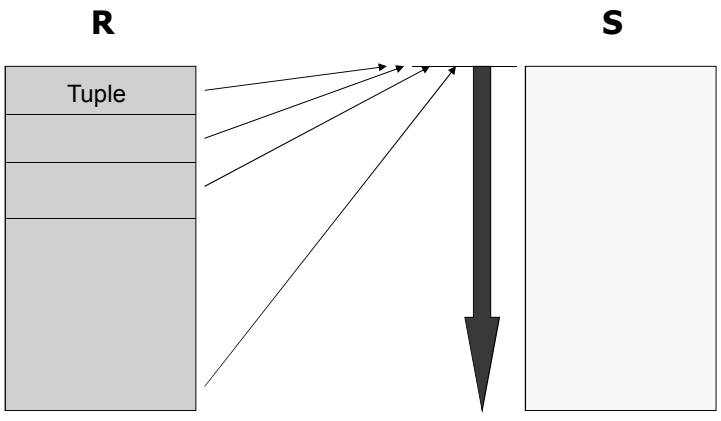
Simple Nested Loops Join

```
foreach tuple r in R do
     foreach tuple s in S do
          if ri == sj  then add <r, s> to result
```

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S
- Total Cost = $(p_R * M) * N + M$ I/Os where $(p_R * M)$ the # of R tuples
  - in our example = 100*1000*500 + 1000 I/Os
  - At 10ms/IO, Total: ???
- What assumptions are being made here?
- What if smaller relation (S) was outer?
- What is the cost if one relation can fit entirely in memory?



Simple Nested Loop Join



Page-Oriented Nested Loops Join

```
foreach page b_R in R do
    foreach page b_S in S do
        foreach tuple r in b_R do
            foreach tuple s in b_S do
                if r_i == s_j then add <r, s> to result
```

- For each *page* of R, get each *page* of S, and write out matching pairs of tuples *<r, s>*, where ***r*** is in R-page and ***s*** is in S-page
  - Improvement: in the worst case, each block in inner relation ***s*** is read only once for each *page* in the outer relation (instead of once for each *tuple*)
- Total Cost = M*N + M I/Os
  - in our example = 1000*500 + 1000 I/Os
- Choose smaller relation to be outer relation
  - If smaller relation (S) is outer, Total cost = 500*1000 + 500

39

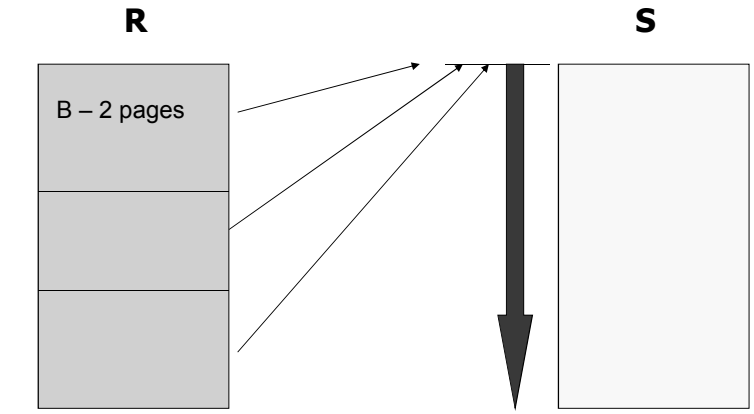
"Block" Nested Loops Join

- Page-oriented nested loop (NL) doesn't exploit extra page buffers
- Alternative approach:
  - use one page as an input buffer for scanning the *inner* S,
  - one page as the output buffer, and
  - use all remaining buffer pages (B-2) to hold ``block'' (think "chunk") of *outer* R
- For each matching tuple ***r*** in R-chunk, ***s*** in S-page, add *<r, s>* to result. Then read next R-chunk, scan S, etc.

```
for each block of B-2 pages of R do
  for each page of S do
     {for all matching in-memory tuples
                    r in R-block and s in S-page
         add <r, s> to result}
```

40

"Block" Nested Loop Join



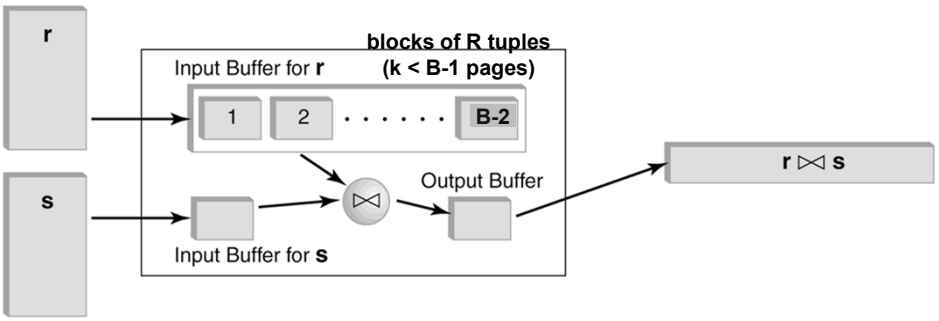
⌈M / (B – 2)⌉ blocks

1 scan per R block

N pages per scan

"Block" Nested Loops Join

- To find matching pairs of tuples efficiently
  - Build a main-memory hash table for R
- Cost: *#outer blocks* * Scan of inner + Scan of outer
  - *#outer blocks* = ⌈ # of pages of outer/ block size ⌉
  - Total cost = M / (B-2) * N + M



Examples of Block Nested Loops

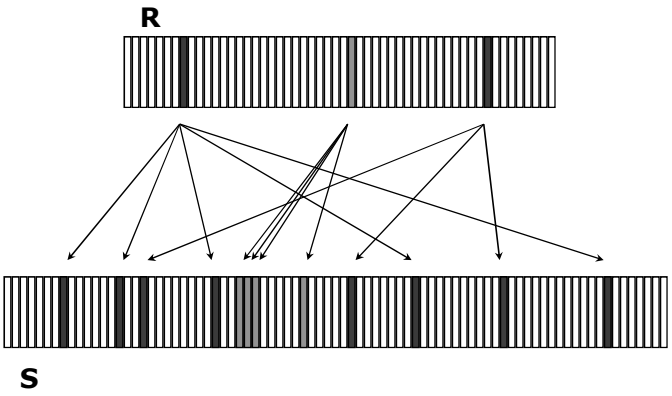
- With 100 buffer pages of "Reserves" (R) as outer:
  - Cost of scanning R is 1000 I/Os; a total of 10-page *blocks*
  - Per chunk of R, we scan "Sailors" (S); 10*500 I/Os
  - Total: 10*500 + 1000 = 6000 IOs
  - If space for just 90 buffer pages of R, we would scan S 12 *times*
- With 100-page *block* of "Sailors" (S) as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 *blocks*
  - Per block of S, we scan "Reserves" (R); 5*1000 I/Os
  - Total: 5*1000 + 500 = 5500 IOs
- If you consider seeks, it may be best to divide buffers evenly between R and S
  - Disk arm "jogs" between read of S and write of output
  - If output is not going to disk, this is not an issue

Index Nested Loops Join

```
foreach tuple r in R do
     foreach tuple s in S where ri == sj do
          add <r, s> to result
```

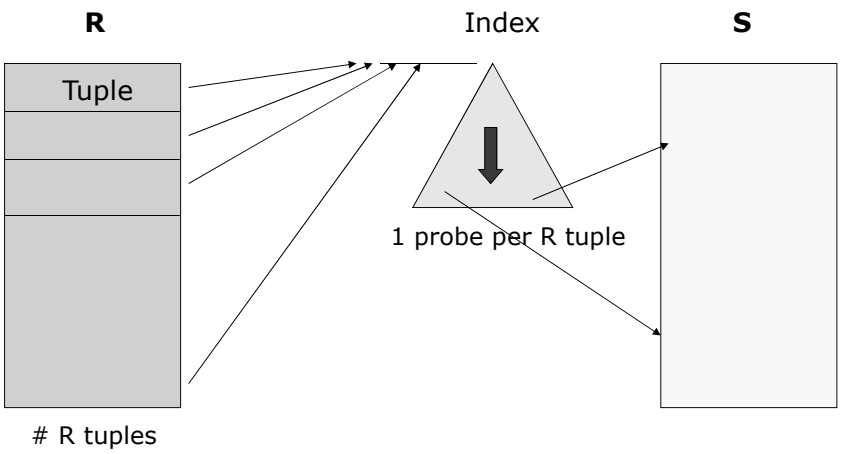
- If there is an index on the join attribute of one relation
  - Make indexed relation the inner relation (say S)
  - Compare each tuple in R with tuples in the same partition (same value in join attribute)
  - Does not enumerate cross-product of R and S
- Cost to scan R = M I/Os
- Cost to retrieve matching S tuples depends on
  - Index
    - Hash index: 1.2 (primary) -2.2 (secondary) I/Os to find appropriate bucket (probing)
    - B+ tree: 2-4 I/Os to find appropriate leaf
  - Number of matching tuples
    - Clustered index: 1 I/O per outer tuple **r**
    - Unclustered index: up to 1 I/O per matching **s** tuple

Index Nested Loops Join



- The basic cost of the nested loop join is visible in the picture
  - We do three indexed access into **S**, but need the three driving rows from **R** first

Index Nested Loops Join



Examples of Index Nested Loops

- Hash-based index on ***sid*** of "Sailors"
  - 1.2 I/Os to retrieve the appropriate page of index
  - At most one matching tuple since ***sid*** is the key of "Sailors"
  - Cost to scan "Reserves" = 1000 I/Os
  - Each "Reserves" tuple takes 1.2 I/Os to get data entry in index + 1 I/O to get matching "Sailors" tuple
  - 1000*100 "Reserves" tuples takes 220000 I/Os
  - Total cost = 221000 I/Os

Examples of Index Nested Loops

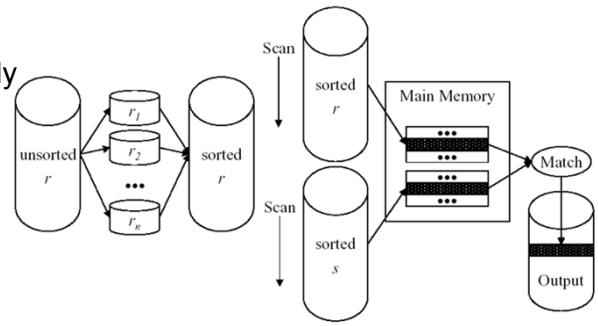
- Hash-based index on ***sid*** of "Reserves"
  - Cost to scan "Sailors" = 500 page I/Os
  - Each "Sailors" tuple can match 0 or more "Reserves" tuples
  - Assume uniform distribution
    - Average of 2.5 reservations per sailor (100000 / 40000)
    - Cost to retrieve reservation of a sailor is 1 or 2.5 I/Os depending on whether the index is clustered
  - 1 "Sailors" tuple takes 1.2 I/Os to find index page with data entries + 1 to 2.5 I/Os to retrieve matching "Reserves" tuples
  - 80*500 "Sailors" tuples (40000) take 88000 to 148000 I/Os
  - Total cost range from 88500 to 148500 I/Os

Examples of Index Nested Loops

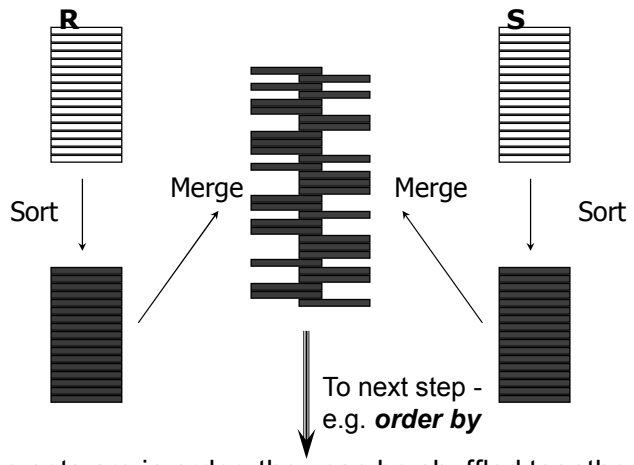
- B+-tree index on ***sid*** of "Sailors" (as inner):
  - Scan "Reserves": 1000 page I/Os, 100*1000 tuples
  - For each "Reserves" tuple: 2 I/Os to find index page with data entries + 1 I/O to get (the exactly one) matching "Sailors" tuple (clustered or unclustered)
- B+-tree index on ***sid*** of "Reserves" (as inner):
  - Scan "Sailors": 500 page I/Os, 80*500 tuples
  - For each "Sailors" tuple: 2 I/Os to find index page with data entries + cost of retrieving matching "Reserves" tuples
    - Assuming uniform distribution, 2.5 reservations per sailor (100000/ 40000) the cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered

Sort-Merge Join

- (External) Sort R and S on the join attribute
  - Group tuples with same value together
- Merge the sorted relations on the join attribute
  - Scan R and S for qualifying tuples
  - Compare R tuples in a partition with only the S tuples in the same partition
- Useful if
  - One or both inputs already sorted on join attribute(s)
  - Output should be sorted on join attribute(s)



Sort-Merge Join ($R \bowtie_{i=j} S$)



- Once the two sets are in order, they can be shuffled together
  - The shuffling can be quick - the sorting may be the most expensive operation

Sort-Merge Join ($R \bowtie_{i=j} S$)

- Merging step
  - Scan start at first tuple in each relation
  - Advance scan of R until `current R tuple > current S tuple` w.r.t. value in join attribute
  - Advance scan of S until `current S tuple > current R tuple`
  - Alternate between such advances until `current R tuple = current S tuple`
    - All R tuples with same value in `Ri` (current R group) and all S tuples with same value in `Sj` (current S group) match
    - Output <`r, s`> for all pairs of such tuples
      - Like a mini nested loop
  - Resume scan of R and S
- R is scanned once; each S group is scanned once per matching R tuple
  - Multiple scans of an S group will probably find needed pages in buffer

Merge Step of a Sort-Merge Join

Input: *relation* **r** *sorted on attribute* A;
relation **s** *sorted on attribute* B
Output: $\mathbf{r} \bowtie_{A=B} \mathbf{s}$

```
Result := {}                                  // initialize Result
t_r := getFirst(r)                            // get first tuple
t_s := getFirst(s)
while !eof(r) and !eof(s) do{
     while !eof(r) && t_r[A] < t_s[B] do
          t_r = getNext(r)                    // get next tuple
     while !eof(s) and t_r[A] > t_s[B] do
          t_s = getNext(s)
     if t_r[A] = t_s[B] = c then              // for some const c
          Result := (σ_{A=c}(r) × σ_{B=c}(s)) ∪ Result;
}
return Result;
```

Selections advance tr and ts

53

Example of Sort-Merge Join

R:	S:	$R \bowtie_{R.A = S.B} S$:
⇨ $r_1.A = 1$	⇨ $s_1.B = 1$	$r_1 s_1$
⇨ $r_2.A = 3$	⇨ $s_2.B = 2$	$r_2 s_3$
$r_3.A = 3$	⇨ $s_3.B = 3$	$r_2 s_4$
⇨ $r_4.A = 5$	$s_4.B = 3$	$r_3 s_3$
⇨ $r_5.A = 7$	⇨ $s_5.B = 8$	$r_3 s_4$
$r_6.A = 7$		$r_7 s_5$
⇨ $r_7.A = 8$		

54

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

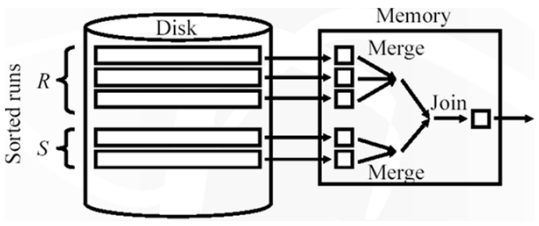
- Cost to sort R = 2 * M* (# merge passes + 1) I/Os
- Cost to sort S = 2 * N * (#merge passes + 1) I/Os
- Cost to merge both = M+N I/Os

- With 35, 100 or 300 buffer pages, both “Reserves” and “Sailors” can be sorted in 2 *passes*;
  total join cost = 4 * 1000 + 4 * 500 + 1000 + 500 = 7500 I/Os
  - BNL cost: 2500 to 15000 I/Os

55

Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the *merging* required for the join
  - Sort: produce sorted *runs* of maximum size *L* for *R* and *S*
    - Allocate 1 page buffer per *run* of each relation
    - With B > $\sqrt{L}$, where *L* is the size of the larger relation, using the tournament sorting that produces on average *runs* of size 2B in Pass 0, *#runs* of each relation is < B/2
  - Merge and join: merge the runs of *R*, merge the runs of *S*, and merge the result streams as they are generated!



56

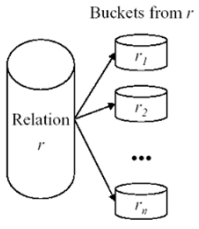
Refined Sort-Merge Join Cost

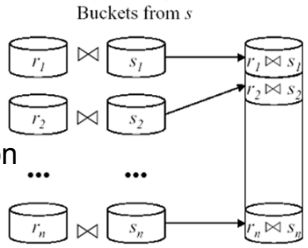
- Sort-Merge Join has a cost of 3(M+N) I/Os
  - read + write each relation in Pass 0 +
  - read each relation in (only one) merging pass
  - (+ writing of result tuples)
- Memory Requirement:
  - To be able to merge in one pass, we should have enough memory to accommodate one page from each *run*:
    B > M / B + N / B => B > $\sqrt{M+N}$ => B > $\sqrt{L}$
    where *L* is the size of the larger relation
- In our example, cost goes down from 7500 to 4500 (3 * 1500) I/Os
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear* (very few passes)

57

Hash-Join

- Usually done with static number of hash buckets
  - Alternatives use *directories*, are more complex:
    - Extensible, Linear hashing
  - Generally have fairly long overflow chains
- "Classic" (In-memory) Hash join
  - Efficient when memory can hold one of the two relations
- Simple hash-based join
  - Efficient when memory is large
  - Too many I/O operations with small memory
- GRACE hash-based join
  - Separate partitioning phase, parallel execution
  - Avoid bucket overflow
- Hybrid hash-based join
  - Combines Simple and Grace hash-join
  - Better memory usage



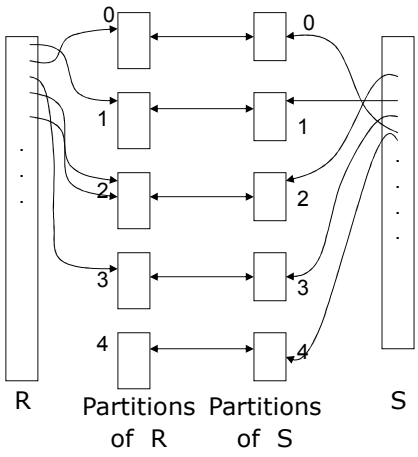


Join corresponding *r* and s buckets

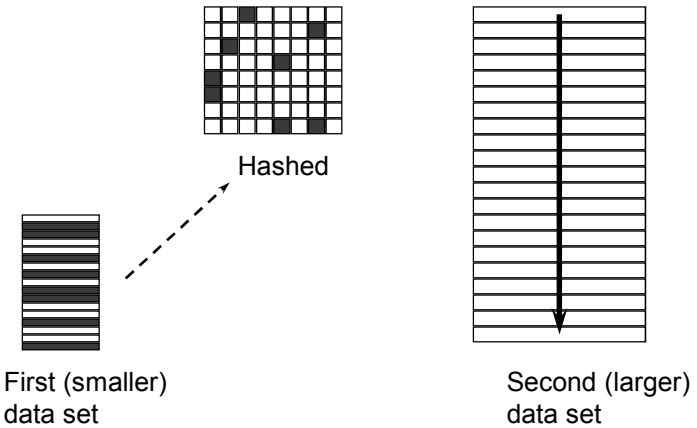
58

Hash-Join Basic Concepts

- Partitioning (Building) phase
  - Use same hash function **h** to hash relations R and S on their join attributes to the same hash file
- Probing (Matching) phase
  - Compares R (build input) tuples in partition **i** with S tuples (probe input) in same partition **i**, then join
  - While applying a hash function to two attributes, if the hash values are different, then the two attribute values can not be equal
- Best when the hash table fits in main memory!

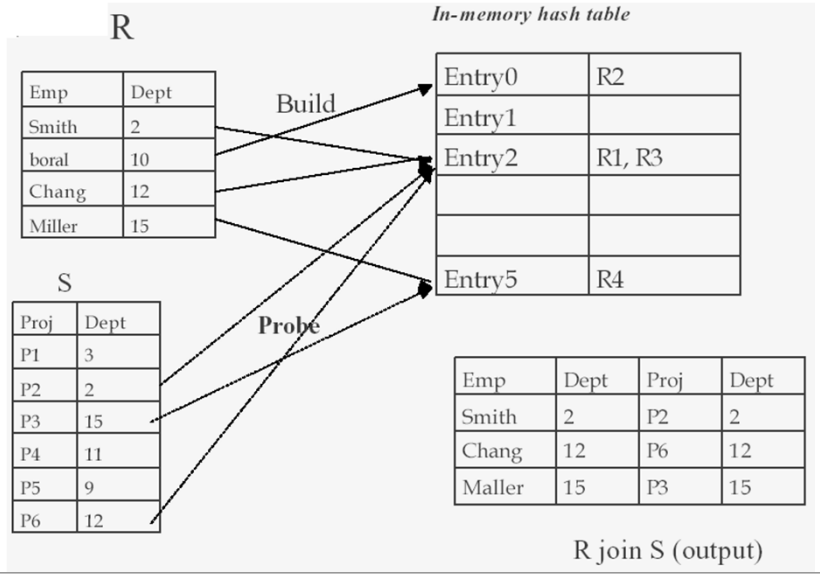


Hash-Join: "Classic"



- The first table is hashed in memory (build), the second table is used to probe the hash table for matches
  - In simple cases the cost is easy to calculate

Hash-Join: "Classic"



Cost of Hash-Join: "Classic"

```
Read entire inner relation into hash
  table (join attributes as key)
For each tuple from outer, look up in
  hash table & join
```

- Build phase
  - Read R once and construct in-memory hash table
  - I/Os: M (# of pages of R)
- Probe phase
  - Read all of S tuples and search for matching tuples
  - I/Os: N (# of pages of S)
- Total Cost: M+N I/Os if we have enough memory to hold the inner relation in memory
  - How do you choose the relation for Build and for Probe?

Hash-Join: Simple

- What if we do not have enough memory?
  - Use whatever memory is available as one bucket and write the rest to disk
- Simple Hash Join
  - Choose hash function *h1()* so that each bucket of R fits in memory
  - Scan R; keep the contents of first bucket in memory; write out the rest on a new file
    - hash the contents of first bucket, using *h2(),* again, in memory
  - Scan S; for each tuple probe, or write-out
  - Repeat this process until the entire join is performed
- Simple vs. "Classic" Hash Join
  - Identical to "classic" if R fits in memory: special case, with one bucket
  - Performs well if R almost fits in memory: say, half of R fits
  - But poorly otherwise
    - introduces too many I/O operations (passes) when the memory is not too large!

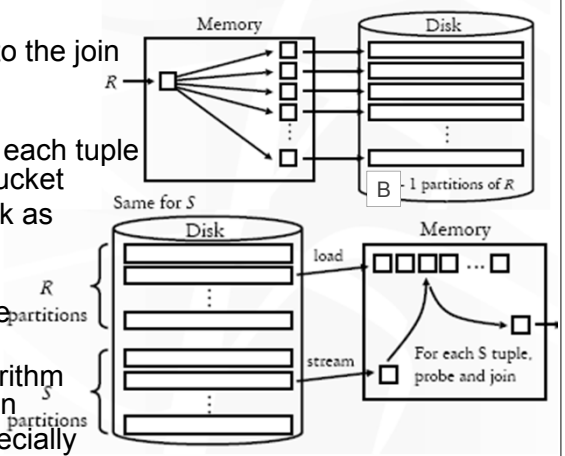
Cost of Hash-Join: Simple

```
for each logical bucket j -- h1()
 for each record r in R
   if r is in bucket j then
     insert r into the hash table;-- h2()
   else write r into Rtmp
 for each record s in S
   if s is in bucket j then
     probe the hash table;          -- h2()
   else write s into Stmp
```

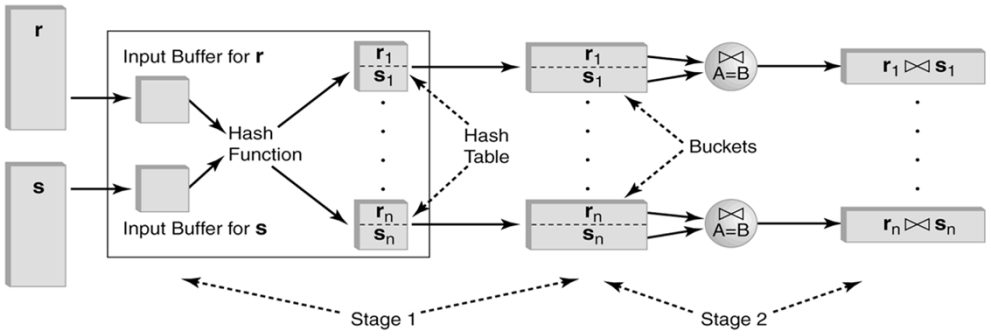
- Simple Hash-Join algorithm combines the partitioning work and probing work into each iteration of the loop
- Assuming that the hash function divide relations uniformly into k buckets
  - each bucket `j` requires one more pass over R and `j` passes over S
    - `j` buckets of R read; `j-1` written; similarly for S
  - The process terminates whenever either the *Rtmp* or *Stmp* is empty
  - The cost is k * (M+N) I/O's
    - We read and write buckets of each relation k times!

Hash-Join: GRACE

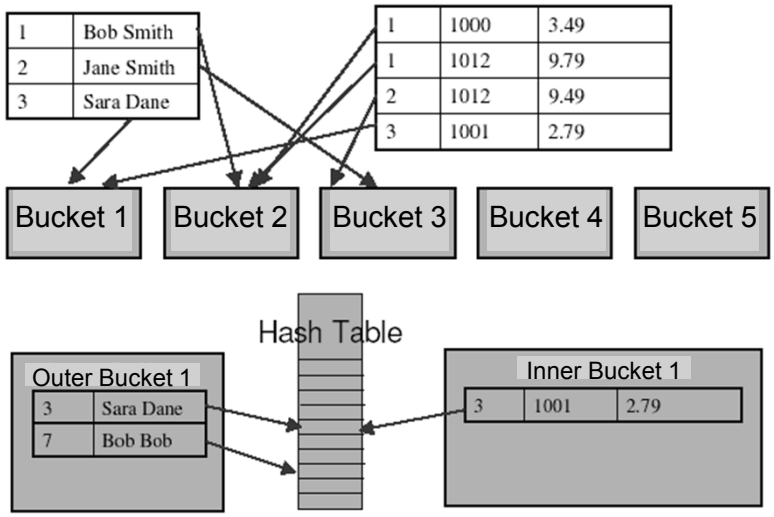
- Partitioning phase:
  - Apply a hash function h(x) to the join attributes of both R and S
    - Assume B-1 buckets
  - According to the hash value, each tuple is put into a corresponding bucket
    - Write these buckets to disk as separate files (physical)
- Joining phase:
  - Get one partition of R and the corresponding partition of S
  - Apply the "classic" hash algorithm using a different hash function
- Better than simple hashing, especially when little memory is available
  - Twice as much work when memory is large
  - Scan once for partitioning, once for hash/join



Hash-Join: GRACE

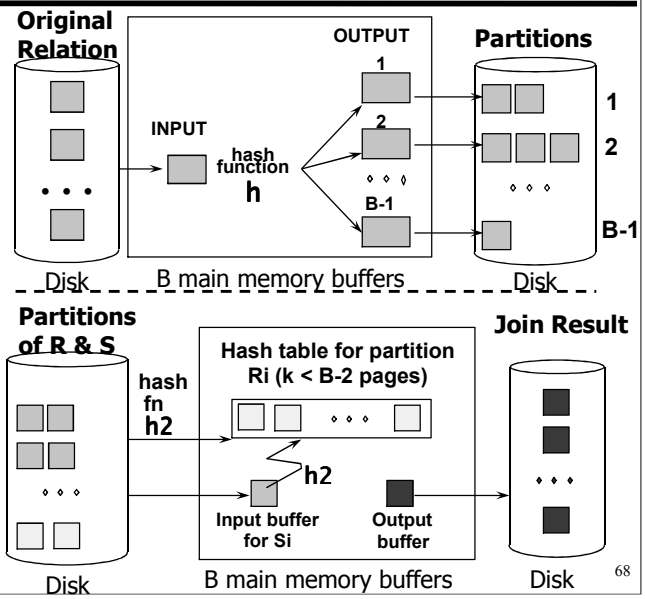


Hash-Join: GRACE



Hash-Join: GRACE

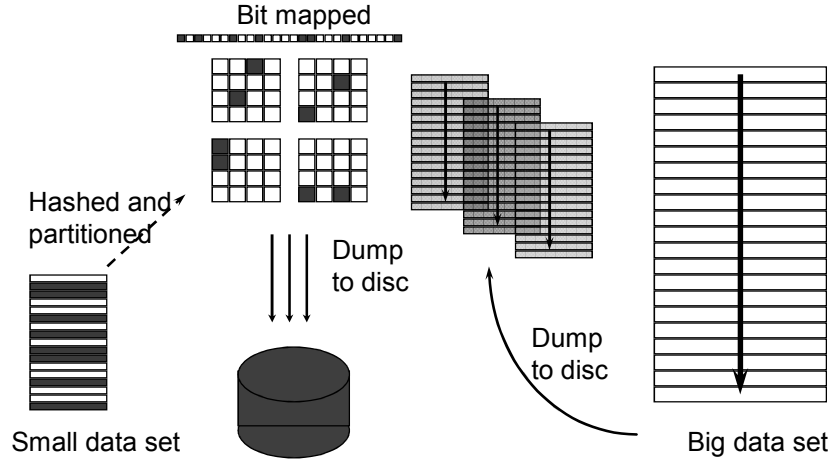
- Partition both relations using hash function **h1**: R tuples in partition **i** will only match S tuples in partition **I**
- Read in a partition of R, hash it using **h2** (<> **h1**!)
- Scan matching partition of S, and probe hash table for matches



Hash-Join: GRACE

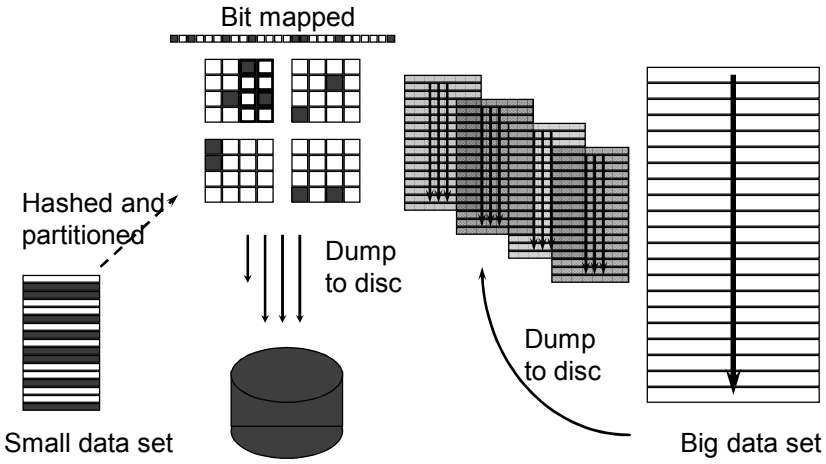
```
for each tuple r in R
  apply hash function to the join attributes of r;
  put r into the appropriate bucket R[i]}
for each tuple s in S
  apply hash function to the join attributes of s;
  put r into the appropriate bucket S[i]}
for each bucket i < B-1
  /* using a different hash function h2*/
  build the hash table for R[i];
  for each tuple s in S[i]
     apply the hash function h2 to the join
   attributes of S;
     use s to probe the hash table;
     output any matches to the result relation;
```


Hash-Join: GRACE



- If the smaller data set cannot be hashed in memory, it partitioned, mapped, and partly dumped to disc
  - The larger data set is partitioned in the same way

Hash-Join: GRACE



- And if things go really wrong (bad statistics) Oracle uses partitions which are too large - and the probe (secondary) partitions are re-read many times

Workload of Hash-Join: GRACE

```
bucketID = X mod 4
Join on R.X = S.X
```

R ⋈ S = R0 ⋈ S0 +
R1 ⋈ S1 +
R2 ⋈ S2 +
R3 ⋈ S3

Nested-loop join considers all slots

Hash join considers only those along the diagonal

R \ S	0	1	2	3
0	X X X X X X X X X			
1		X X X X X X X X X		
2			X X X X X X X X X	
3				X X X X X X X X X

Observations on Hash-Join: GRACE

- Given B buffer pages, the maximum # of partitions is B-1
  - B-1 > size of largest partition of R to be held in memory for the probing phase
- Assuming uniformly sized partitions, the size of each R partition is M/(B-1)
- The number of pages in the (in-memory) hash table built during the probing phase is *f**M/(B-1) where *f* is the fudge factor
  - *f* ≈ 1.2 is used to capture the (small) increase in size between the partition and a hash table for the partition
- During the probing phase, in addition to the hash table for the R partition, we require a buffer page for scanning the S partition, and an output buffer
  - Therefore, we require B > f*M/(B-1) +2
- Approximately, we need B > $\sqrt{M}$ for the Grace hash join to perform well
  - We can always pick R to be the smaller relation, so:
    B must be > $\sqrt{\min(M, N)}$

Cost of Hash-Join: GRACE

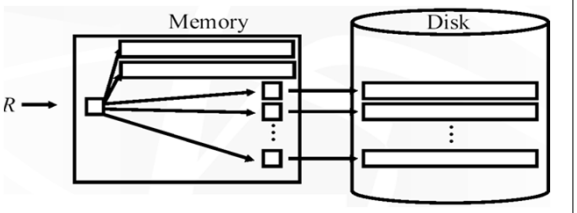
- Grace hash join
  - Assume hash function *h1()* to partition R into $\sqrt{M}$ subsets
  - Scan R, then S, placing into output buffer
    - When full, flush buffer to disk
  - For each R buffer, read from disk, make hash table (use *h2()*)
  - Read S buffer, lookup each tuple in hash, output on match
- Assume each partition fits into memory
  - Cost for partitioning phase: Scan R and S once and write them out once 2(M+N) I/Os
  - Cost for joining phase: Scan each partition once M+N I/Os
  - Total cost = 3(M+N) I/Os
- In our example joining “Reserves” and “Sailors” relation costs 3(1000+500) = 4500 I/Os

Utilizing Extra Memory

- Suppose we are partitioning R (and S) into k partitions where B > f * M / k , i.e. we can build an in-memory hash table for each partition
  - The partitioning phase needs k +1 buffers, which leaves us with some extra buffer space of B−(k +1) pages
- If this extra space is large enough to hold one partition, i.e., B−(k+1) >= f * M / k, we can collect the entire first partition of R in memory during the partitioning phase and need not write it to disk
- Similarly, during the partitioning of S, we can avoid storing its first partition on disk and rather immediately probe the tuples in S's first partition against the in-memory first partition of R and write out results
  - At the end of the partitioning phase for S, we are already done with joining the first partitions
- The savings obtained result from not having to write out and read back in the first partitions of R and S
  - This version of hash join is called Hybrid Hash Join

Hybrid Hash-Join

- What if there is extra memory available?
  - Use it to avoid writing/ rereading partitions of both *R* and *S*!



- Hybrid Hash-Join
  - Joining phase of one of the partitions is included during the partitioning phase
  - Useful when memory sizes are relatively large, and the build input is bigger than memory

Works as Grace hash join, but R_1 (can be big) is kept in the main memory, as a hash table and partitions R_2, ..., R_M get to disk

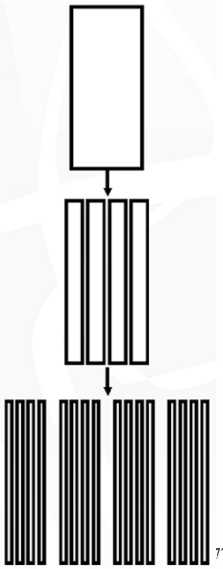
When partitioning S, tuples of the first bucket S_1 can be joined directly with R_1

Finally, join R_i & S_i for i=2,...,k

Savings: no need to write R_1 and S_1 to disk or read back to memory

Partition Overflow

- If the hash function does not partition uniformly, one or more R partitions may not fit in memory
- Two possible strategies to handle partition overflow:
  - Overflow prevention (prevent from happening)
  - Overflow resolution (handle overflow when it occurs)
- Case 1, overflow on disk: an *R* partition is larger than memory size
  - Solution (a) small partitions first and combine before join;
  - Solution (b) recursive partition
- Case 2, overflow in memory: the in-memory hash table of *R* becomes too large
  - Solution: revise the partitioning scheme and keep a smaller partition in memory
- See the duality in multi-pass merge sort here?



Duality of Sort and Hash

- Divide-and-conquer paradigm
  - Sorting: physical division, logical combination
  - Hashing: logical division, physical combination
- Handling very large inputs
  - Sorting: multi-level merge
  - Hashing: recursive partitioning
- I/O patterns
  - Sorting: sequential write, random read (merge)
  - Hashing: random write, sequential read (partition)

Duality of Sort and Hash

Table 7. Duality of Sort- and Hash-Based Algorithms

Aspect	Sorting	Hashing
In-memory algorithm	Quicksort	Classic Hash
Divide-and-conquer paradigm	Physical division, logical combination	Logical division, physical combination
Large inputs	Single-level merge	Partitioning
I/O Patterns	Sequential write, random read	Random write, sequential read
Temporary files accessed simultaneously	Fan-in	Fan-out
I/O Optimizations	Read-ahead, forecasting Double-buffering, striping merge output	Write-behind Double-buffering, striping partitioning input
Very large inputs	Multi-level merge Merge levels Nonoptimal final fan-in	Recursive partitioning Recursion depth Nonoptimal hash table size
Optimizations	Merge optimizations	Bucket tuning
Better use of memory	Reverse runs & LRU Replacement selection ?	Hybrid hashing ? Single input in memory
Aggregation and duplicate removal	Aggregation in replacement selection	Aggregation in hash table
Algorithm phases	Run generation, intermediate and final merge	Initial and intermediate partitioning, in-memory (hybrid) hashing
Resource sharing	Eager merging Lazy merging	Depth-first partitioning Breadth-first partitioning
Partitioning skew and effectiveness	Merging run files of different sizes	Uneven output file sizes
"Item value"	log (run size)	log (build partition size/original build input size)
Bit vector filtering	For both inputs and on each merge level?	For both inputs and on each recursion level
Interesting orderings, multiple joins	Multiple merge-joins without sorting intermediate results	N-ary partitioning and joins
Interesting orderings: grouping/aggregation followed by join	Sorted grouping on foreign key useful for subsequent join	Grouping while building the hash table in hash join
Interesting orderings in index structures	B-trees feeding into a merge-join	Merging in hash value order

79

Hash-Join vs. Sort-Merge Join

- Both have a cost of 3(M+N) I/Os
  - Assuming two-pass Sort-Merge join
- Memory requirement: hash join is lower

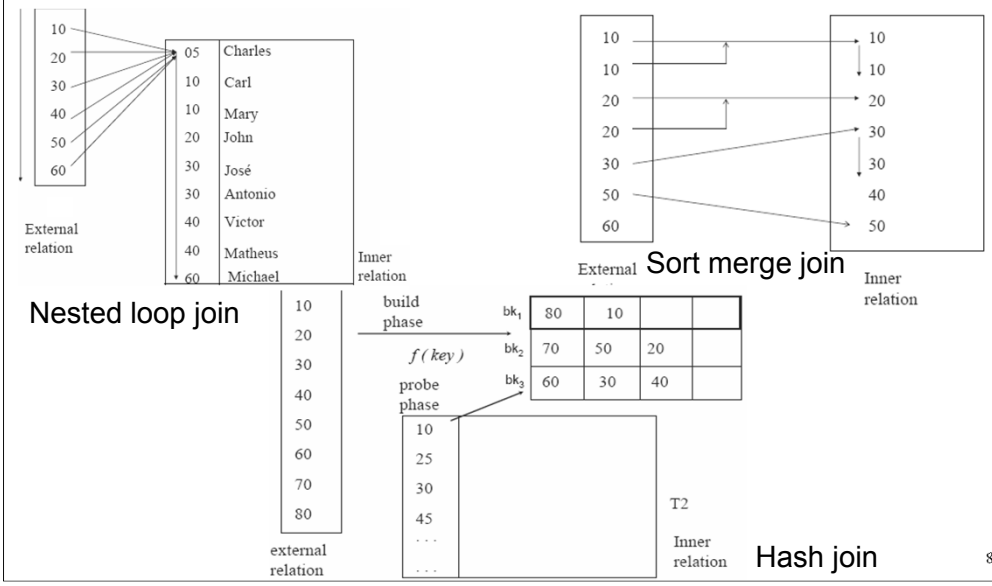
  $\sqrt{\min(M,N)} < \sqrt{M+N}$
  - Hash join wins on this count if relation sizes differ greatly
  - Also, Hash Join shown to be highly parallelizable
- Other factors
  - Hash join performance depends on the quality of the hash
    - Might not get evenly sized buckets
  - Sort-Merge join can be adapted for inequality (!=) join predicates
  - Sort-Merge join wins if R and/or S are already sorted
  - Sort-Merge join wins if the result needs to be in sorted order
  - Sort-Merge join less sensitive to *data skew*

80

Remarks on Joins

- Hash join is very efficient but is only applicable to equijoin
  - Three hash-based algorithms are proposed, out of which the hybrid hash join is the best
    - "Simple" is good for large memories
    - Grace, for small memories
- Sort merge join performs better than nested loop when both relations are large
  - especially true if one or both relations are already sorted on the joining attributes
- Nested loop join performs well when one relation is large & one is small
  - A special case is when the smaller relation can be entirely held in main memory which implies that both relations need to be read in only once
  - When combined with the index on the joining attribute of the (larger) inner relation, excellent performance can yield

Putting it all Together



Impact of Buffering

- Effective use of the buffer pool is crucial for efficient implementations of a relational query engine
  - Several operators use the size of available buffer space as a parameter
  - When several operations are executed concurrently, estimating the number of available buffer pages is guesswork
- Keep the following in mind:
  - When several operators execute concurrently, they share the buffer pool
  - Using an unclustered index for accessing records makes finding a page in the buffer rather unlikely and dependent on (rather unpredictably!) the size of the buffer
  - Furthermore, each page access is likely to refer to a new page, therefore, the buffer pool fills quickly and we obtain a high level of I/O activity
  - If an operation has a repeated pattern of page accesses, a clever replacement policy and/or sufficient number of buffers can speed up the operation significantly

Buffering and Repeated Access Patterns

- Repeated access patterns interact with buffer replacement policy e.g.,
  - Simple nested loop join: for each outer tuple, scan all pages of the inner relation
    - With enough buffer pages to hold inner, replacement policy does not matter
    - Otherwise, LRU is worst (sequential flooding)
      - MRU gives best buffer utilization, the first B − 2 pages of the inner will always stay in the buffer (pinning a few pages is best)
  - Block nested loop join: for each block of the outer, scan all pages of the inner relation
    - Since only one unpinned page is available for the scan of the inner, the replacement policy makes no difference
  - Index nested loop join: for each tuple in the outer, use the index to find matching tuples in the inner relation
    - For duplicate values in the join attributes of the outer relation, we obtain repeated access patterns for the inner tuples and the index
      - The effect can be maximized by sorting the outer tuples on the join attributes

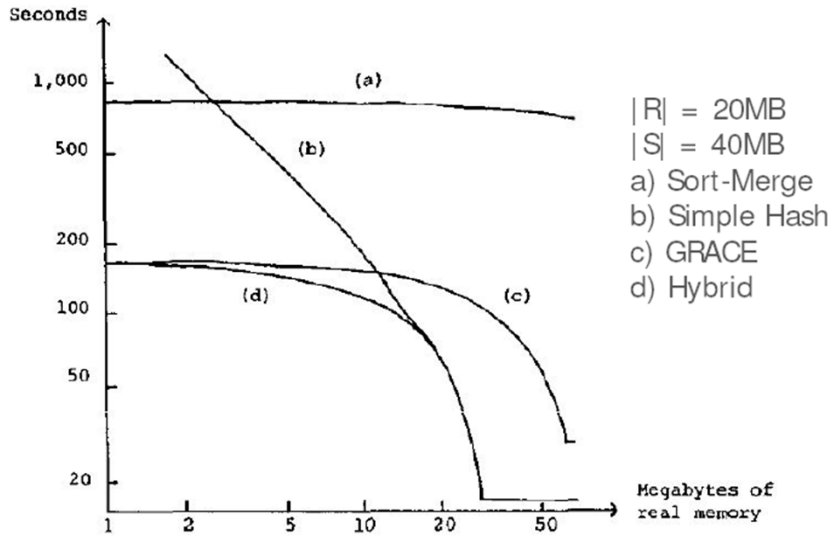
I/O Cost and Buffer Requirements for Join Algos

Join Algorithm	Buffer size B	Approx. Cost I/Os
Page-oriented Nested Loop Join	= N + 1	M+N
Block Nested Loop Join	>= 2	(M/(B-2))*N + M
Simple Sort Merge Join	> sqrt(N)	5*(M+N)
Improved Sort Merge Join	> sqrt(M+N)	3*(M+N)
Grace Hash Join	> sqrt(N)	3*(M+N)
Hybrid Hash Join	>> sqrt(N)	(3-2B/N)*(M+N)

- Cost to generate the output relation are not considered, i.e., the minimum required buffer size is 2 pages (one page for each of the two relations)
- The two relations to be joined are R and S, and M and N denote the number of pages of S and R, respectively
- Moreover, we assume N <= M and that R has a hashed index on the join attribute

85

Performance Notes on Join Algorithms



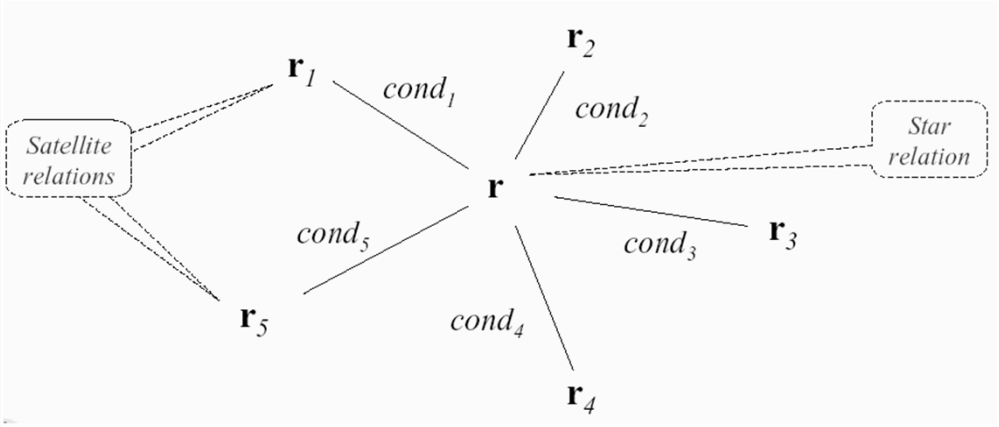
86

General Join Conditions

- Equalities over several attributes: e.g., *R.sid = S.sid* **AND** *R.rname = S.sname*
  - For Index Nested Loop Join, build index on <*sid, sname*> (if S is inner); or use existing indexes on *sid* or *sname*
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join attributes <*sid, rname*> and <*sid, sname*>
- Range conditions over attributes: e.g., *R.rname < S.sname*
  - For Index Nested Loop Join, need (clustered!) B+ tree index
    - Range probes on inner; # matches likely to be much higher than for equality joins
  - Hash Join, Sort Merge Join not applicable!
  - Block Nested Loop Join quite likely to be the best join method here

Star Joins

- **R** $\bowtie_{cond1}$ **R1** $\bowtie_{cond2}$ **R2** $\bowtie_{condn}$ **Rn**
  - Each $cond_i$ involves only the attributes of $\mathbf{R}_i$ and **R**



Computing Star Joins

- Use *join index*
  - Scan **R** and the join index { $<R, R_1, \ldots, R_n>$ } in one scan
    - which is a set of tuples of rids
  - Retrieve matching tuples in $R_1, \ldots, R_n$
  - Output result
- Use *bitmap indices*
  - Use one bitmapped join index, J_i , per each partial join $\mathbf{R} \bowtie_{\text{condi}} \mathbf{Ri}$
  - Recall: J_i is a set of *<v, bitmap>*, where v is an rid of a tuple in **Ri** and bitmap has 1 in k-th position iff k-th tuple of **R** joins with the tuple pointed to by v
  - ❶Scan J_i and logically OR all bitmaps
    - We get all rids in **R** that join with **Ri**
  - ❷Now logically AND the resulting bitmaps for $J_1, \ldots, J_n$
  - ❸Result: a subset of **R**, which contains all tuples that can possibly be in the star join
    - Rationale: only a few such tuples survive, so can use indexed loops

Other Join Operations: Outer Join

```
SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r ON s.sid=r.sid
```

sid	name	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96

s.name	s.sid	s.name	r.bid	s.name	s.sid	r.sid	r.bid
Dustin	22	Dustin	101	Dustin	22	22	101
Lubber	95	Bob	103				
Bob	31	Lubber					
Dustin	22	95	102				
Lubber	31	95	102				
Bob	95	95	102	Bob	95	95	102

Other Join Operations: Outer Join

- Outer join (⟕) can be computed either as
  - A join followed by addition of null-padded non-participating tuples
  - by modifying the join algorithms
  - Note: non participating tuples are those in $R - \Pi_R(R \bowtie S)$
- Modifying merge join to compute left outer-join R ⟕ S
  - During merging, for every tuple t_r from R that do not match any tuple in S, output t_r padded with nulls
- Modifying hash join to compute left outer-join R ⟕ S
  - If R is probe relation, output non-matching R tuples padded with nulls
  - If R is build relation, when probing keep track of which R tuples matched S tuples
    - At end of s_i output non-matched R tuples padded with nulls
- Right outer-join and full outer-join can be computed similarly

Other Join Operations: Semijoins

- Origin: Distributed DBMSs where transport cost dominates I/O-cost

$$\text{Semijoin } R \ltimes S := \pi_R(R \bowtie S) \subseteq R$$

- Idea: to compute the distributed join between two relations R, S stored on different nodes N_R,N_S (assuming we want the result on N_R; let the common attributes be J):
  - ❶Compute π_J (R) on N_R
  - ❷Send the result to N_S
  - ❸Compute π_J (R) ⋉ S on N_S
  - ❹Send the result to N_R
  - ❺Compute R ⋈ (π_J (R) ⋉ S) on N_R
  - N.B. Step ❸ computes the semijoin between S and R
- This algorithm is preferable over sending all of S to NR, if (C_{tr} denotes transport cost, depending on size of transferred data):

$$C_{tr}(\pi_J(R)) + C_{tr}(S \ltimes R) < C_{tr}(S)$$

Other Join Operations: Semijoins

- Let relations R and S be given as

R	A	B
	1	4
	1	5
	1	6
	2	4
	2	6
	3	7
	3	8
	3	9

S	B	C	D
	4	13	16
	4	14	16
	7	13	17
	10	14	16
	10	15	17
	11	15	16
	11	15	17
	12	15	16

- This yields

$\pi_B(R)$	B
	4
	5
	6
	7
	8
	9

$S \ltimes R$	B	C	D
	4	13	16
	4	14	16
	7	13	17

- Cost of Semijoin: C_{tr} = 6 + 9 = 15 whereas sending all of S has C_{tr} = 24

93

Set Operations

- Intersection and cross-product are special cases of join
- Union (Distinct) and Except similar; we'll do union
- Sorting based approach to union:
  - Sort both relations R and S (on combination of *all* attributes)
  - Scan sorted relations R and S in parallel and merge them, eliminating duplicates
  - *Alternative*: Merge runs from Pass 0 for *both* relations
- Hash based approach to union:
  - Partition R and S using hash function ***h1***
  - For each S-partition, build in-memory hash table (using ***h2***), scan corresponding R-partition and add tuples to table while discarding duplicates

94

Aggregate Operations (AVG, MIN, etc.)

```
SELECT AVG(S.age)
FROM Sailors S
GROUP BY S.rating
```

- Basic algorithm
  - Scan entire relation
    - Given B+-tree index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan
  - Maintain running information
    - SUM: Total of values retrieved
    - AVG: (Total, Count) of values retrieved
    - COUNT: Count of tuples retrieved
    - MIN: Smallest value retrieved
    - MAX: Largest value retrieved

Aggregate Operations (AVG, MIN, etc.)

- Sort based approach to grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group
    - Can improve upon this by combining sorting and aggregate computation
  - Given B+-tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan
    - if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order
- Hash based approach to grouping:
  - Build a hash table on grouping attribute
  - Entries of the form <grouping-value, running information>
  - For each tuple scanned
    - Probe hash table to find entry for the group to which tuple belongs to
    - Update running information

Evaluation of Relational Expressions

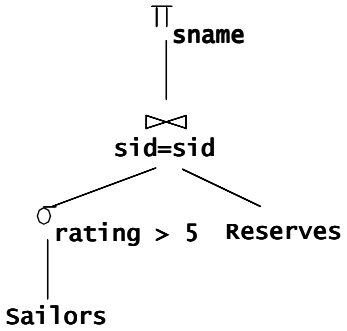
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree:
  - Materialization: Evaluate expression one operation at a time
    - Generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk and repeat
    - use temporary relations to hold intermediate results
  - Pipelining: Evaluate several operations simultaneously in a pipeline
    - pass on tuples to parent operations even as an operation is being executed
    - results are passed from one operation to next, no need for temp relations

Materialization

- Materialized evaluation:
  - Evaluate one operation at a time, starting at the lowest-level
  - Use intermediate results materialized into temporary relations to evaluate next-level operations
- Example: compute and store

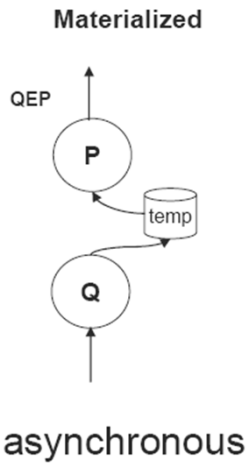
$$\sigma_{rating>5}(SAILORS)$$

then compute the store its join with "*Reserves",* and finally compute the projections on *sname*



Materialization

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: Use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

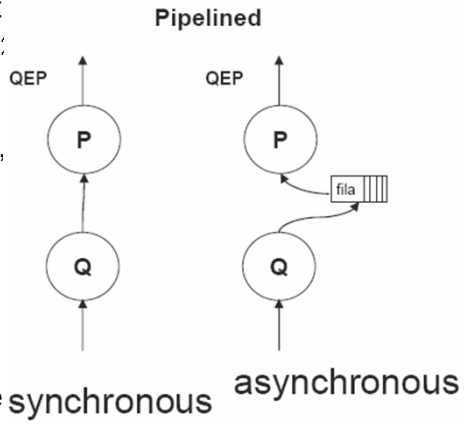


Pipelining

- Pipelined evaluation : Evaluate several operations simultaneously, passing the results of one operation on to the next
  - Much cheaper than materialization: no need to store a temporary relation to disk
- Example: in previous expression tree, don't store result of

$$\sigma_{rating>5}(SAILORS)$$

  - Instead, pass tuples directly to the join
  - Similarly, don't store result of join, pass tuples directly to projection
- Pipelining may not always be possible (more latter)
  - Streaming Input vs. Output

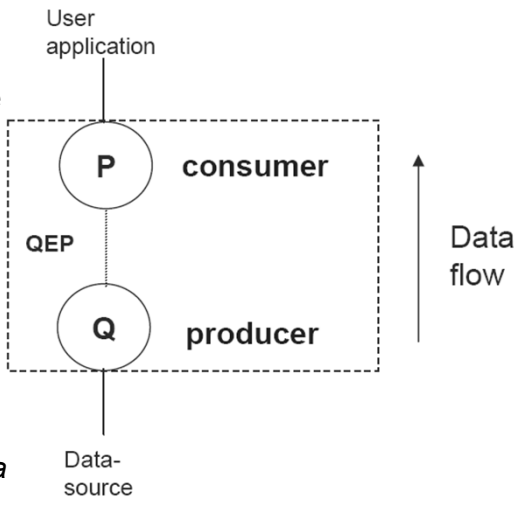


How to Implement Pipelining?

❶ Implement pipeline by constructing single, complex operation that combines operations that constitute pipeline
- ◆too complex, little code reuse

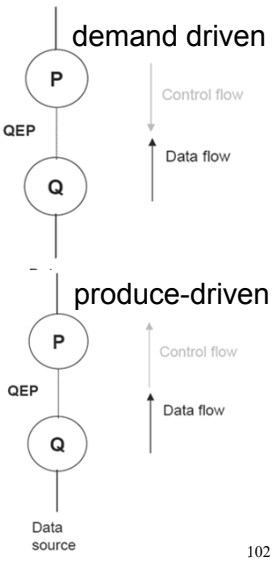
❷ Instead, model each operation as separate process (or thread), adjacent processes are connected via buffer

- Pipelines can be executed in two ways (control flow):
  - ◆Demand driven: "*pulling data up an operation tree from top*"
  - ◆Producer driven: "*pushing data up an operation tree from below*"



Pipelining Execution

- In demand driven (lazy evaluation)
  - ◆System repeatedly requests next tuple from top level operation
  - ◆Each operation requests next tuple from children operations as required, in order to output its next tuple
  - ◆In between calls, operation has to maintain "state" so it knows what to return next
- In produce-driven (eager pipelining)
  - ◆Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - ◆System schedules operations that have space in output buffer and can process more input tuples

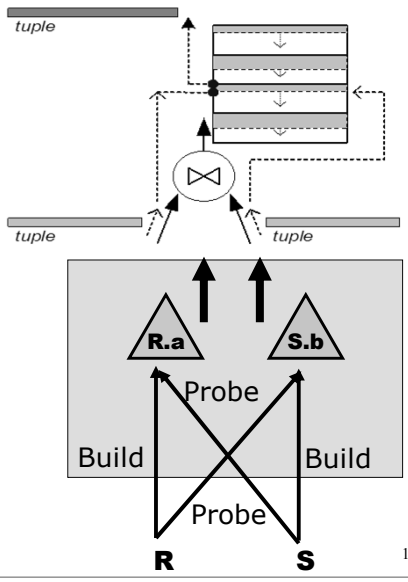


Pipelining vs. Materialization

- Many operations (or certain implementations of them) allow us to pipeline i.e., to accept one or both arguments in a stream without seeing the entire relation before starting
  - If not then the argument must be materialized (stored on disk if it is large) before beginning
- Some examples:
  - Projection, Selection allow pipelining
  - Intersection does not allow pipelining
  - Nested loop join allows the outer argument to be pipelined but not the inner
    - Index join allows pipelining of the non indexed argument
  - Sort or hash join allows pipelining of either argument but there are problems if both are
    - Sort join requires sharing of memory for sorting *runs* (subfiles)
    - Hash join requires buffers for *buckets* of both relations in memory

Pipelining vs. Materialization

- Some algorithms are not able to output results even as they get input tuples (e.g., sort, merge-join or hash-join)
  - In such cases intermediate results must be written to disk and then read back
- Algorithm variants are possible to generate (at least some) results on the fly, as input tuples are read in
  - Pipelined (symetric) hash join: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
    - When a new R_0 tuple is found, match it with existing S_0 tuples, output matches, and save it in R_0
    - Symmetrically for S_0 tuples



The Iterator Interface

- Once the evaluation plan is decided, it is executed by calling the operators in some (possibly interleaved) order
  - Each operator has one or more inputs
  - Passes result tuples to the next operator
  - Materialization is usually done at the input stage of an operator
- Every physical operator has a uniform *iterator* interface:
  - `open()`: Initialize state and get ready for processing
  - `get_next()`: Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  - `close()`: Clean up
- Internally an operator
  - maintains its own execution state
  - encapsulates materialization or on-the-fly processing
  - also encapsulates use of indexes

An Iterator for Table Scan

- `open()`
  - Allocate a block of memory
- `getNext()`
  - If no block of R has been read yet, read the first block from the disk and return the first tuple in the block (or the null pointer if R is empty)
  - If there is no more tuple left in the current block, read the next block of R from the disk and return the first tuple in the block (or the null pointer if there are no more blocks in R)
  - Otherwise, return the next tuple in the memory block
- `close()`
  - Desallocate the block of memory

An Iterator for Nested-loop Join

- R: An iterator for the outer expression
- S: An iterator for the inner expression
- open()
  - R.open(); S.open(); r = R.getNext();
- getNext()
  - do {s = S.getNext();
    if (s == null) {
    S.close(); S.open(); s = S.getNext();
    if (s == null) return null;
    r = R.getNext();
    if (r == null) return null;
    }
    } until (r joins with s);
  - return rs;
- close()
  - R.close(); S.close();

An Iterator for 2-pass Merge Sort

- open()
  - Allocate a number of memory blocks for sorting
  - Call open() on child iterator
- getNext()
  - If called for the first time
    - Call getNext() on child to fill all blocks, sort the tuples, and output a run
    - Repeat until getNext() on child returns null
    - Read one block from each run into memory, and initialize pointers to point to the beginning tuple of each block
  - Return the smallest tuple and advance the corresponding pointer;
    - if a block is exhausted bring in the next block in the same run
- close()
  - Call close() on child
  - Desallocate sorting memory and delete temporary runs

Blocking vs. non-blocking Iterators

- A blocking iterator must call `getNext()` exhaustively (or nearly exhaustively) on its children before returning its first output tuple
  - Examples: sort, aggregation
- A non-blocking iterator expects to make only a few `getNext()` calls on its children before returning its first (or next) output tuple
  - Examples: filter, merge join with sorted inputs

Execution of an Iterator tree

- Call `root.open()`
- Call `root.getNext()` repeatedly until it returns null
- Call `root.close()`

- Requests go down the tree
- Intermediate result tuples go up the tree
- No intermediate files are needed
  - But maybe useful if an iterator is opened many times
    - Example: complex inner iterator tree in a nested-loop join; "cache" its result in an intermediate file

Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*
  - The implementation of these operators can be carefully tuned to improve performances (and it is important to do this!)
  - Access paths are the alternative ways to retrieve tuples from a relation
  - Index matches selection condition if it can be used to only retrieve tuples that satisfy selection condition
  - Selectivity of an access path w.r.t a query is the total number of pages
- Many alternative implementation techniques for each operator; no universally superior technique for most operators
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc.
  - This is part of the broader task of optimizing a query composed of several operators

References

- Based on slides from:
  - R. Ramakrishnan and J. Gehrke
  - H. Garcia Molina
  - J. Hellerstein
  - C. Faloutsos
  - L. Mong Li
  - M. H. Scholl
  - A. Silberschatz, H. Korth and S. Sudarshan
  - P. Lewis, A. Bernstein and M. Kifer
  - J. Lewis: “How the CBO works” www.jlcomp.demon.co.uk