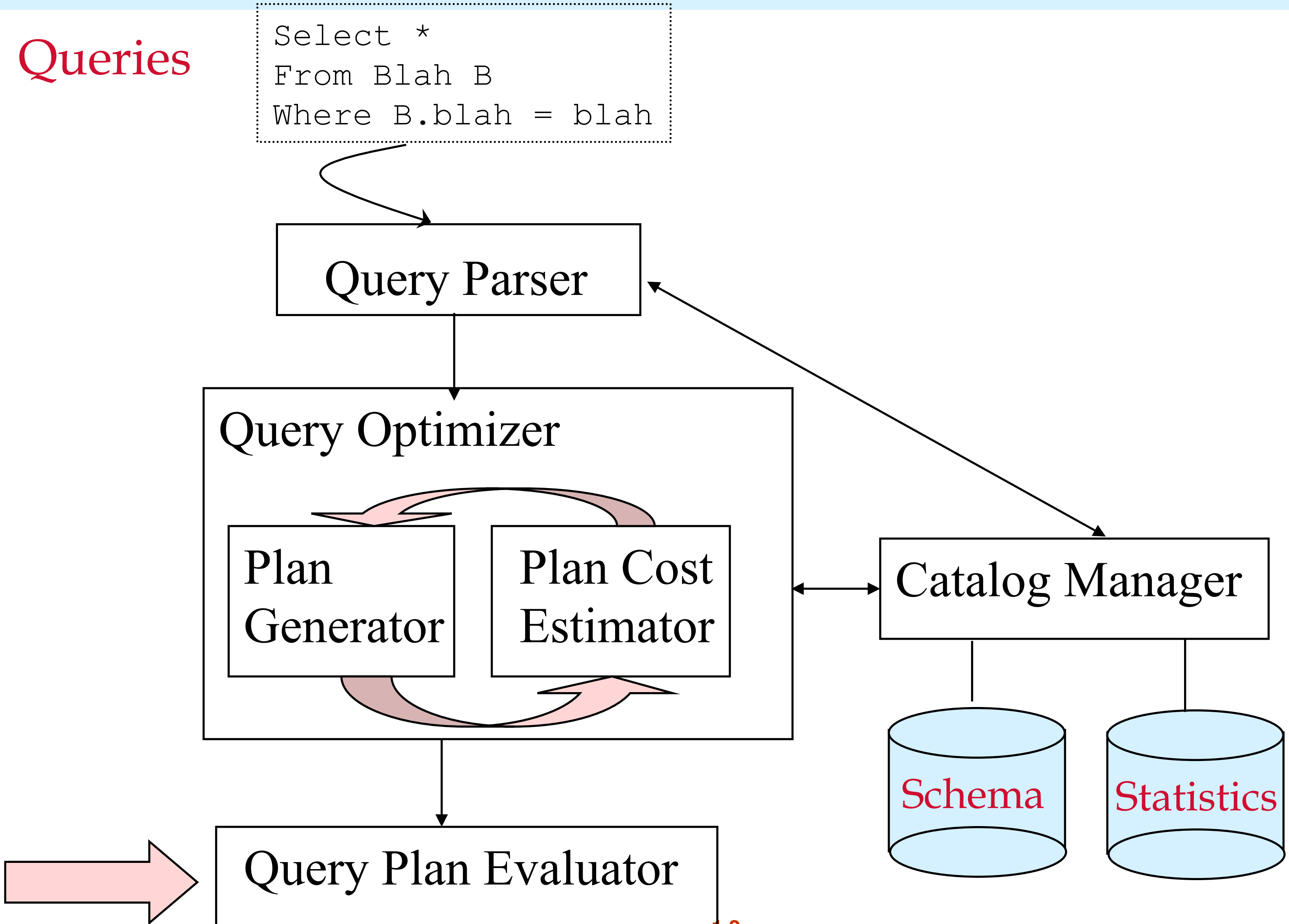
CAS CS 460/660
Introduction to Database Systems

Query Evaluation II

Cost-based Query Sub-System



Review - 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* ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* (-) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
  - *Also: Aggregation* (SUM, MIN, etc.) and GROUP BY
- Since each op returns a relation, ops can be *composed* ! After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Selection (filter) Operators

Schema for Examples

Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: date</u>, *rname*: string)

- Similar to old schema; *rname* added for variation.

(assume pages of 4000 bytes each)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. (100K reseravtions)
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages. (40K sailors)

Simple Selections

```
SELECT  *
FROM    Reserves R
WHERE   R.date > '1/1/2015'
```

- Of the 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 approximated as

  *size of R * reduction factor*

  - "reduction factor" is usually called *selectivity*.
  - estimate of reduction factors is based on statistics – we will discuss shortly.

Alternatives for Simple Selections

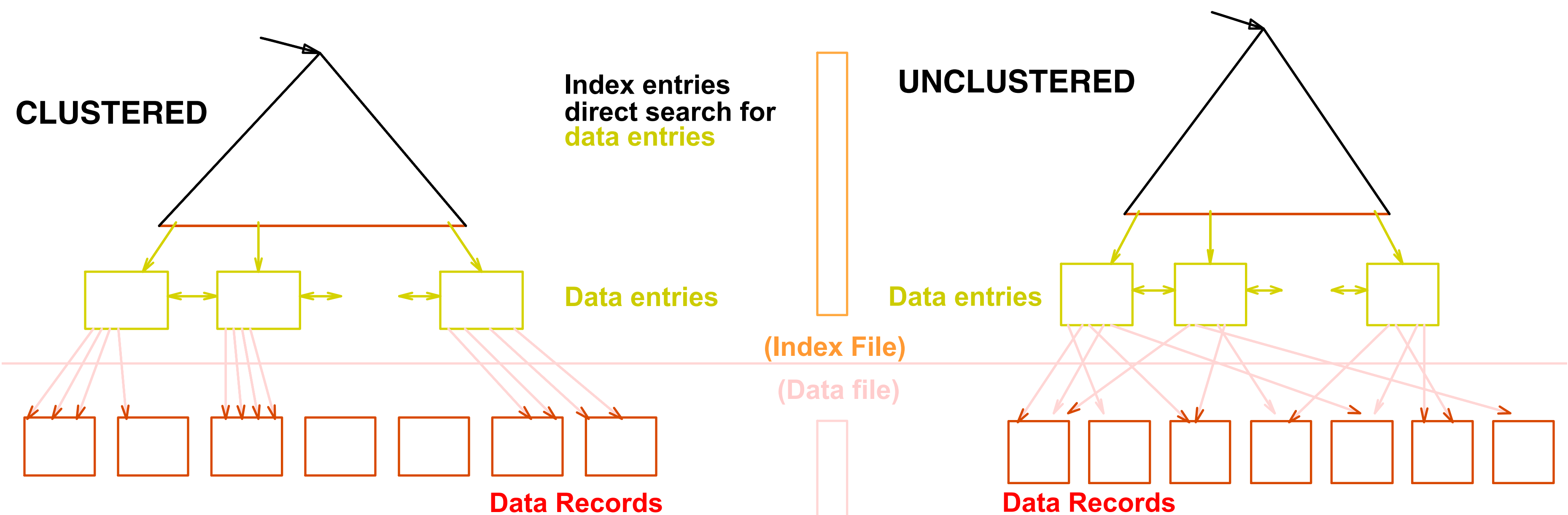
- With no index, unsorted:
  - Must essentially scan the whole relation
  - cost is M (#pages in R). For "Reserves" = 1000 I/Os.
- With no index, sorted on day:
  - cost of binary search + number of pages containing results.
  - For reserves = 10 I/Os + ⌈selectivity*1000⌉
- With an index on selection attribute:
  - Use index to find qualifying data entries,
  - then retrieve corresponding data records.
  - (Hash index useful only for equality selections.)

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost:
    - finding qualifying data entries (typically small)
    - plus cost of retrieving records (could be large w/o clustering).
  - In example "Reserves" relation, if 10% of tuples qualify (result size estimate: 100 pages, 10000 tuples).
    - With a *clustered* index, cost is little more than 100 I/Os;
    - if *unclustered*, could be more than 10000 I/Os! unless…

Selections using Index (cont)

- *Important refinement for unclustered indexes*:
  1. Find qualifying data entries.
  2. Sort the rid's of the data records to be retrieved.
  3. Fetch 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).



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*).
- A B-tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - Index on *<a, b, c>* matches *a=5 AND b= 3*, but not *b=3*.
- For Hash index, must have all attributes in search key

Two Approaches to General Selections

- First approach: Find the *most selective access path*, retrieve tuples using it, and apply 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.

Most Selective Index - Example

- Consider *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 2 or more matching indexes (w/ Alternatives (2) or (3) for data entries):
  - Get sets of rids of data records using each matching index.
  - Then *intersect* these sets of rids.
  - Retrieve the records and apply any remaining terms.
- Consider *day<8/9/94 AND bid=5 AND sid=3*. With a B+ tree index on *day* and an index on *sid*, we can retrieve rids of records satisfying *day<8/9/94* using the first, rids of recs satisfying *sid=3* using the second, intersect, retrieve records and check *bid=5*.
  - Note: commercial systems use various tricks to do this:
    - bit maps, bloom filters, index joins

Join Operators

Join Operators

- Joins are a very common query operation.
- Joins can be very expensive:

  Consider an inner join of R and S each with 1M records. Q: How many tuples in the answer?

  (cross product in worst case, 0 in the best(?))

- Many join algorithms have been developed

- Can have very different join costs.

Equality Joins With One Join Column

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

- In algebra: $R \bowtie S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
- Assume:
  - M = 1000 pages in R, p_R =100 tuples per page.
  - N = 500 pages in S, p_S = 80 tuples per page.
  - In our examples, R is Reserves and S is Sailors.
- *Cost metric* : # of I/Os. We will ignore output costs.
- We will consider more complex join conditions later.

Simple Nested Loops Join

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

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
- How much does this Cost?
- $(p_R * M) * N + M$ = 100,000*500 + 1000 I/Os. (about 50M I/Os!!)
  - At 10ms/IO, Total: ???
- What if smaller relation (S) was outer?
- $(p_s * N) * M + N$ = 40,000*1000 + 500 I/Os. (better…. ☺ 40M I/Os)
- Prohibitively expensive…

Q: What is cost if one relation can fit entirely in memory?
M+N = 1500 I/Os!!!!!

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
```

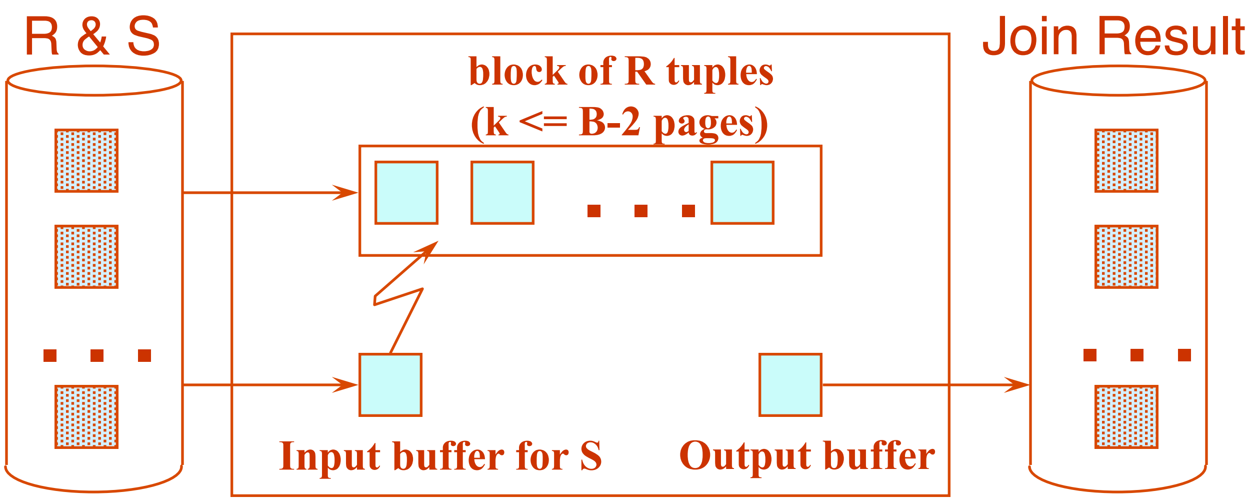
- For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.

- What is the cost of this approach?

- M*N + M= 1000*500 + 1000 = 501000
  - If smaller relation (S) is outer, cost = 500*1000 + 500 = 500500

Block Nested Loops Join

- Page-oriented NL doesn't exploit extra 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 pages to hold ``block'' of outer R.
- For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost:

  Scan of outer + #outer blocks * scan of inner

  - #outer blocks = ceiling(#pages of outer/blocksize)
- With Reserves (R) as outer, and 100 pages/Block (B=102):
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks (B=102)*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - Total cost: 10*500+1000 = 6000 I/Os
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total cost: 5 * 1000 + 500 = 5500 I/Os. (Much better ☺)
  - We may be able to do even better for different B!

Index Nested Loops Join

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

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + (($M*p_R$) * cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree.
- Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
- Clustered index: 1 I/O per page of matching S tuples.
- Unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 1000+100*1000*2.2
- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered. Assume clustered.
  - Totals: 500 + 80*500*2.2 = 88.5K I/Os!!! (not so good here ☹)
  - Other scenarios may be better though.

Sort-Merge Join $(R \bowtie S)$

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
- Particularly useful if
  - one or both inputs are already sorted on join attribute(s)
  - output is required to be sorted on join attributes(s)
- "Merge" phase can require some back tracking if duplicate values appear in join column
- 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.)

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: Sort S +Sort R + (M+N)
  - The cost of merging: usually M+N,
    - worst case is M*N (but very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(*BNL cost: 2500 to 16500 I/Os*)

Cost of Sort-Merge

- For B = 35
  - Sort-Merge:
    - sort R: in two passes=> 4M = 4000
    - sort S: in two passes => 4N = 2000
    - merge: M+N (hopefully…) => 1500
    - Total: 7500
  - Block Nested Loop:
    - celing(N/B-2)*M+N = 16*1000+500 = 16500
  - Sort-Merge Better for B=35!!!!
- For B = 300
  - Sort-Merge: the same: 7500
  - BNLJ:
    - celing(N/B-2)*M+N = 2*1000+500 = 2500
  - Here BNLJ is better!!!!

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.
  - Pass 0 as before, but apply to both R then S before merge.
  - If B > $\sqrt{L}$, where *L* is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - In "Merge" phase: Allocate 1 page per run of each relation, and `merge' while checking the join condition
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os for B=300.
- In practice, the I/O cost of sort-merge join, like the cost of external sorting, is *linear*.

Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.
- Repeated access patterns interact with buffer replacement policy.
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops? Sort-Merge Join?

Hash-Join

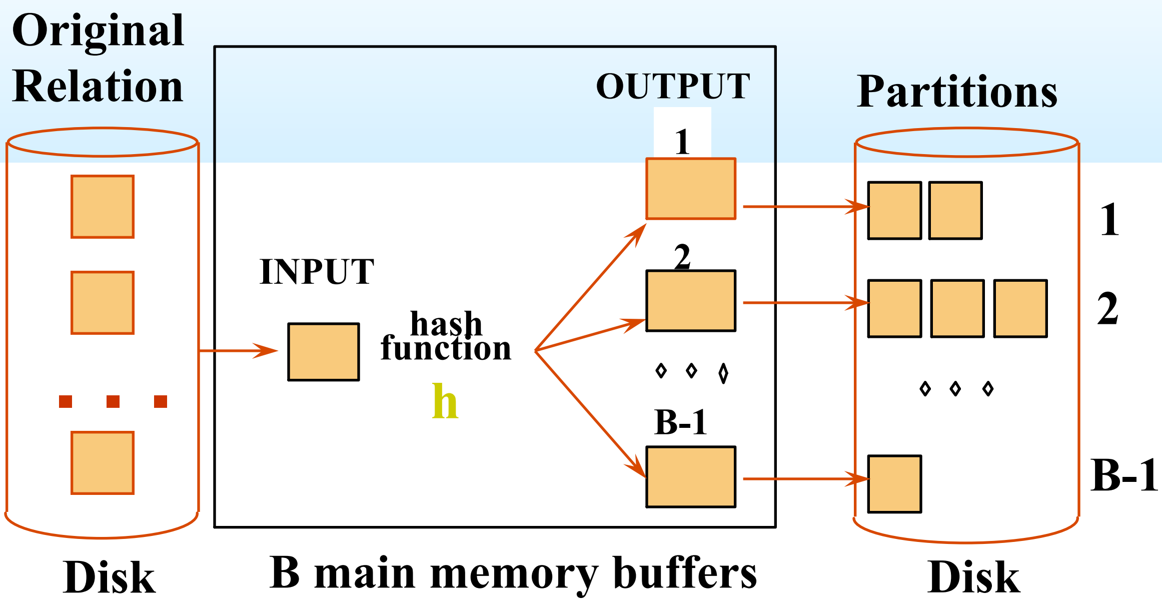
- Partition both relations on the join attributes using hash function h.
- R tuples in partition R_i will only match S tuples in partition S_i.

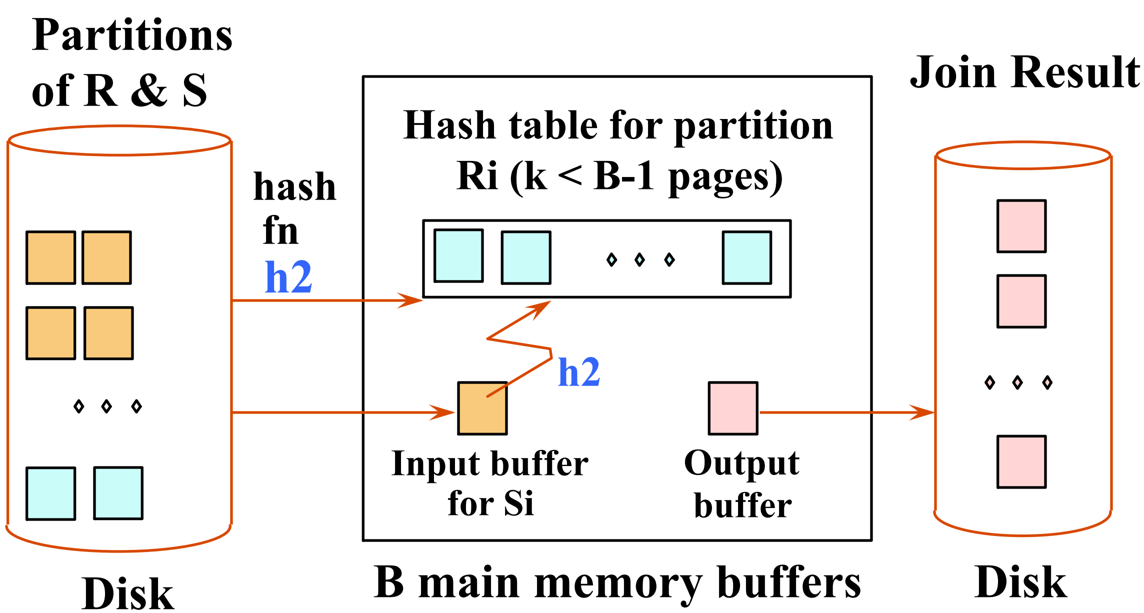
For i= 1 to #partitions {

Read in partition R_i and hash it using h2 (not h).

Scan partition S_i and probe hash table for matches.

}





Observations on Hash-Join

- #partitions k < B, and B-1 > size of smaller partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  k= B-1, and M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$

- Since we build an in-memory hash table to speed up the matching of tuples in the second phase, a little more memory is needed.

- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior if relation sizes differ greatly (e.g., if one reln fits in memory). Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Set Operations

- Intersection and cross-product special cases of join.
- Union (Distinct) and Except similar; we' ll do union.

- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0 for *both* relations.

- Hash based approach to union:
  - Partition R and S using hash function *h*.
  - For each S-partition, build in-memory hash table (using *h2*), scan corr. R-partition and add tuples to table while discarding duplicates.

General Join Conditions

- Equalities over several attributes

  (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):

  - For Index NL, 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 columns.

- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, 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 NL quite likely to be the best join method here.

Review

- Implementation of Relational Operations as Iterators
  - Focus largely on External algorithms (sorting/hashing)
- Choices depend on indexes, memory, stats,…
- Joins
  - Blocked nested loops:
    - simple, exploits extra memory
  - Indexed nested loops:
    - best if 1 rel small and one indexed
  - Sort/Merge Join
    - good with small amount of memory, bad with duplicates
  - Hash Join
    - fast (if enough memory), bad with skewed data
    - Relatively easy to parallelize

Aggregation Operators

Schema for Examples

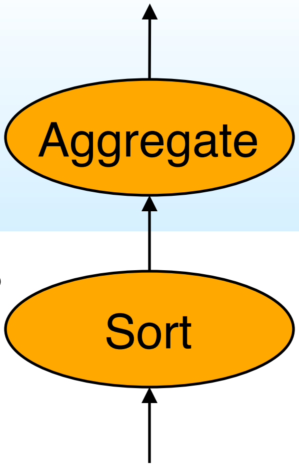
Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. So, M = 1000, p_R = 100.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - So, N = 500, p_S = 80.

Aggregate Operations (AVG, MIN, etc.)

- Without grouping:
  - In general, requires scanning the relation.
  - Given a tree index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.
- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Better: combine sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given a 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.

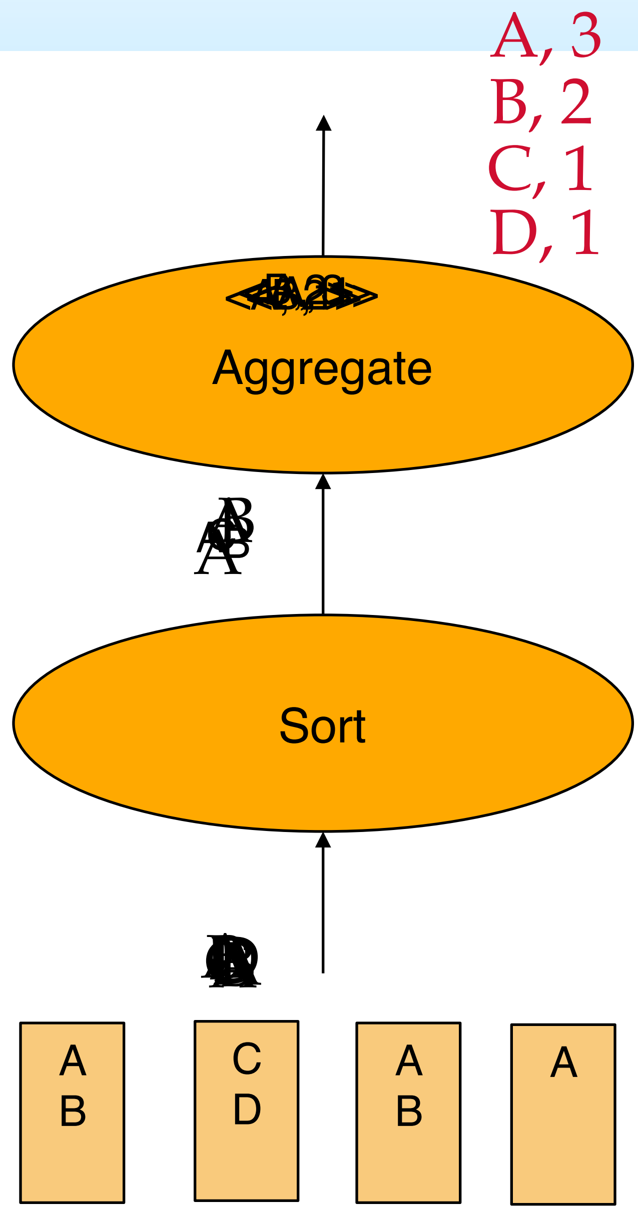
Sort GROUP BY: Naïve Solution



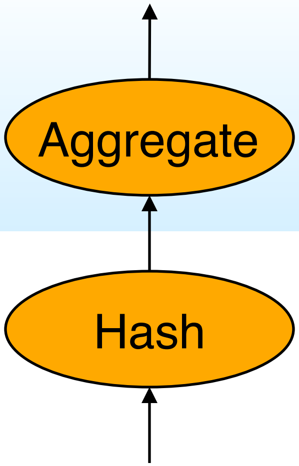
- The Sort iterator naturally permutes its input so that all tuples are output in sequence
- The Aggregate iterator keeps running info ("transition values" or "transVals") on agg functions in the SELECT list, per group: Example transVals:
  - For COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVERAGE it keeps `sum-so-far` *and* `count-so-far`
- As soon as the Aggregate iterator sees a tuple from a new group:
  1. It produces an output for the old group based on the agg function

     E.g. for AVERAGE it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple's info

Sort GROUP BY: Naïve Solution



Hash GROUP BY: Naïve Solution (similar to the Sort GROUPBY)



- The Hash iterator permutes its input so that all tuples are output in groups.
- The Aggregate iterator keeps running info ("transition values" or "transVals") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps `count-so-far`
  - For SUM, it keeps `sum-so-far`
  - For AVERAGE it keeps `sum-so-far` *and* `count-so-far`
- When the Aggregate iterator sees a tuple from a new group:
  1. It produces an output for the old group based on the agg function

     E.g. for AVERAGE it returns (`sum-so-far/count-so-far`)
  2. It resets its running info.
  3. It updates the running info with the new tuple's info

External Hashing

- Partition:

Each group will

be in a single

disk-based partition file. But those files have many groups inter-mixed.

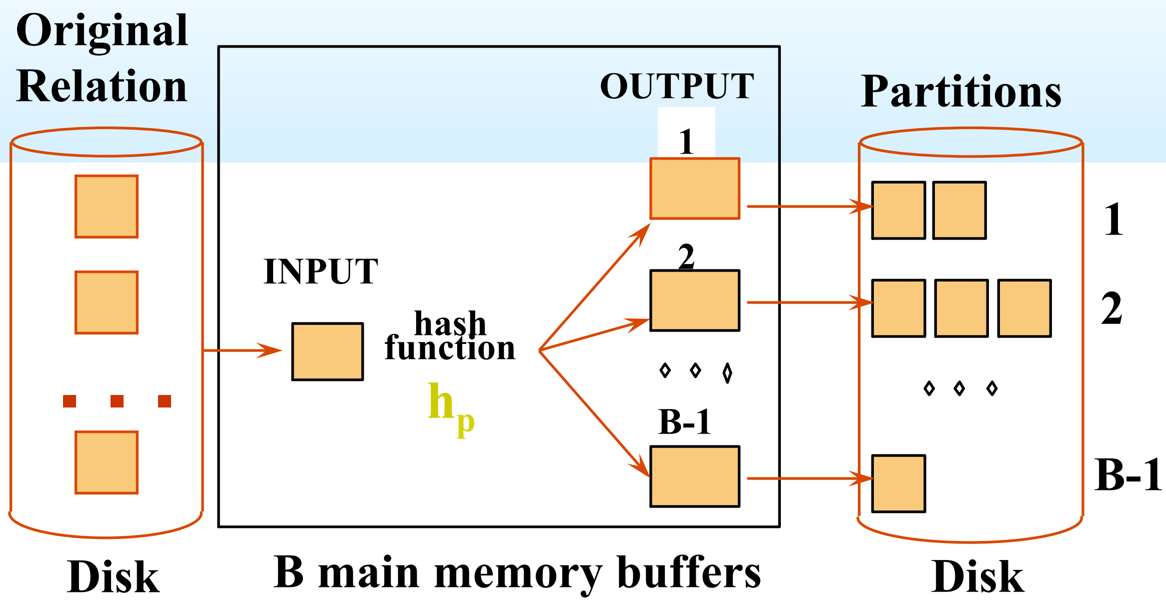
- Rehash:

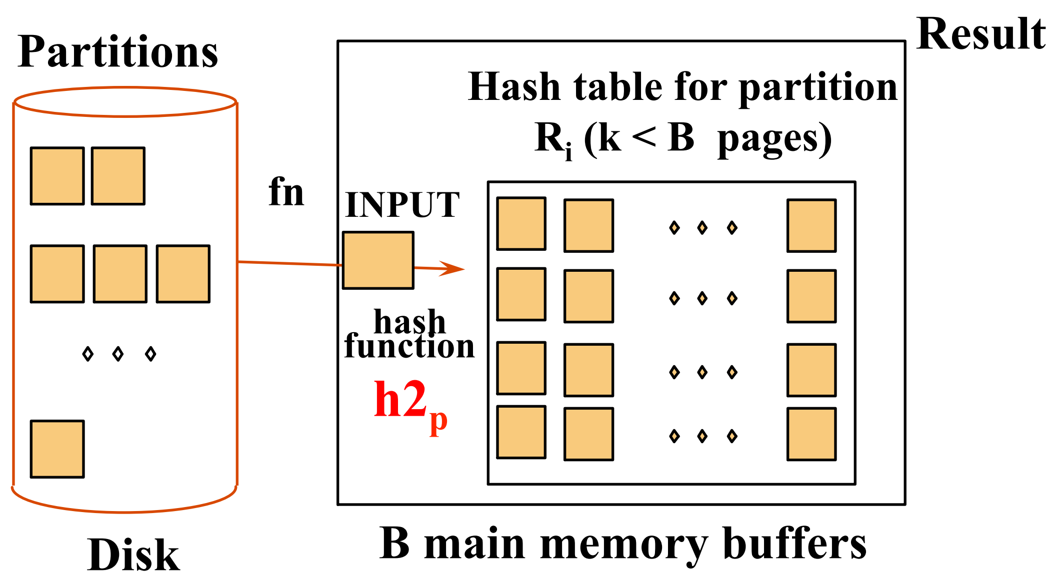
For Each Partition i:

hash i into an in-memory hash table

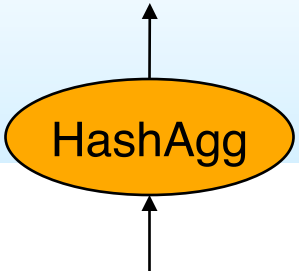
Return results until

records exhuasted then i++





We Can Do Better!



- Put summarization into the hashing process
  - During the ReHash phase, don't store tuples, store pairs of the form <GroupVals, TransVals>
  - When we want to insert a new tuple into the hash table
    - If we find a matching GroupVals, just update the TransVals appropriately
    - Else insert a new <GroupVals,TransVals> pair
- What's the benefit?
  - Q: How many pairs will we have to maintain in the rehash phase?
  - A: Number of **distinct values** of GroupVals columns
    - Not the number of tuples!!
  - Also probably "narrower" than the tuples

Projection (DupElim)

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

- Issue is removing duplicates.
- Basic approach is to use sorting
  - 1. Scan R, extract only the needed attrs (why do this 1st?)
  - 2. Sort the resulting set
  - 3. Remove adjacent duplicates
  - Cost: Reserves with size ratio 0.25 = 250 pages. With 20 buffer pages can sort in 2 passes, so
    1000 +250 + 2 * 2 * 250 + 250 = 2500 I/Os
- Can improve by modifying external sort algorithm:
  - Modify Pass 0 of external sort to eliminate unwanted fields.
  - Modify merging passes to eliminate duplicates.
  - Cost: for above case: read 1000 pages, write out 250 in runs of 20 pages, merge runs = 1000 + 250 +250 = 1500.

DupElim & Indexes

- If an index on the relation contains all wanted attributes in its search key, can do *index-only* scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as *prefix* of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.
- Same tricks apply to GROUP BY/Aggregation

Summary of Query Evaluation

- *Queries are composed of a few basic operators*;
  - The implementation of these operators can be carefully tuned (and it is important to do this!).
  - Operators are "plug-and-play" due to the *Iterator* model.
- Many alternative implementation techniques for each operator; no universally superior technique for most.
- Must consider alternatives for each operation in a query and choose best one based on statistics, etc.
- This is part of the broader task of Query Optimization, which we will cover next!