Evaluation of relational operators and query optimization

Kathleen Durant PhD

CS 3200

Introduction

- In network and hierarchical DBMSs, low-level procedural query language is generally embedded in high-level programming language.
- Programmer's responsibility to select most appropriate execution strategy.
- With declarative languages such as SQL, user specifies what data is required rather than how it is to be retrieved.
- Relieves user of knowing what constitutes good execution strategy.

Introduction

- **Also gives DBMS more control over system performance.**

- **Two main techniques for query optimization:**
  - **heuristic rules that order operations in a query;**
  - **comparing different strategies based on relative costs, and selecting one that minimizes resource usage.**

- **Disk access tends to be dominant cost in query processing for centralized DBMS.**

Query Processing

Activities involved in retrieving data from the database.

- **Aims of QP:**
  - **transform query written in high-level language (e.g. SQL), into correct and efficient execution strategy expressed in low-level language (implementing RA);**
  - **execute strategy to retrieve required data.**

Why is it important?

- Now that *we* know about the benefits of indexes, how does the *DBMS* know when to use them?
- An SQL query can be implemented in many ways, but which one is best?
  - Perform selection before or after join etc.
  - Many ways of physically implementing a join (or other relational operator), how to choose the right one?
- The DBMS does this automatically, but we need to understand it to know what performance to expect

Query Evaluation

- SQL query is implemented by a query plan
  - Tree of relational operators
    - Each internal node operates on its children
    - Can choose different operator implementations
- Two main issues in query optimization:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan.
  - Practically: Avoid worst plans!

Query Optimization

Activity of choosing an efficient execution strategy for processing query.

- As there are many equivalent transformations of same high-level query, aim of QO is to choose one that minimizes resource usage.
- Generally, reduce total execution time of query.
- May also reduce response time of query.
- Problem computationally intractable with large number of relations, so strategy adopted is reduced to finding near optimum solution.

Query to Query Plan

Find all Managers who work at a London branch.

```
SELECT *
FROM Staff s, Branch b
WHERE s.branchNo = b.branchNo AND
(s.position = 'Manager' AND b.city = 'London');
```

Different Strategies

- **Three equivalent RA queries are:**

(1) $\sigma_{(position='Manager') \wedge (city='London') \wedge (Staff.branchNo=Branch.branchNo)} (Staff \times Branch)$

(2) $\sigma_{(position='Manager') \wedge (city='London')}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch)$

(3) $(\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'} (Branch))$

Phases of Query Processing

- **QP has four main phases:**
  - **decomposition (consisting of parsing and validation);**
  - **optimization;**
  - **code generation;**
  - **execution.**

Analysis

- **Analyze query lexically and syntactically using compiler techniques.**
- **Verify relations and attributes exist.**
- **Verify operations are appropriate for object type.**

Analysis - Example

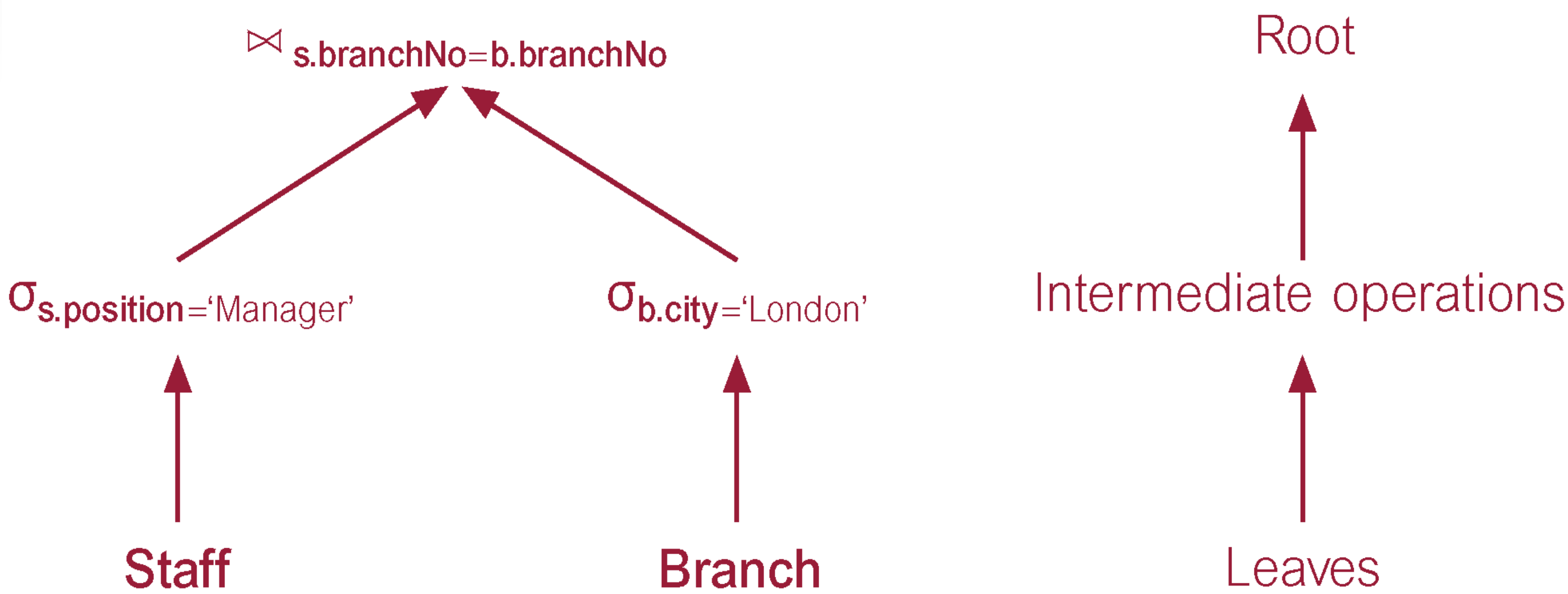
```
SELECT staffNumber
FROM Staff
WHERE position > 10;
```

- This query would be rejected on two grounds:
  - staffNumber is not defined for Staff relation (should be staffNo).
  - Comparison '>10' is incompatible with type position, which is variable character string.
- Rejection due: properly structured SQL statement, incorrect field name, type incompatibility etc.

Analysis

- **Finally, query transformed into some internal representation more suitable for processing.**
- **Some kind of query tree is typically chosen, constructed as follows:**
  - **Leaf node created for each base relation.**
  - **Non-leaf node created for each intermediate relation produced by RA operation.**
  - **Root of tree represents query result.**
  - **Sequence is directed from leaves to root.**

Relational Algebra Tree



Tree of relational operators

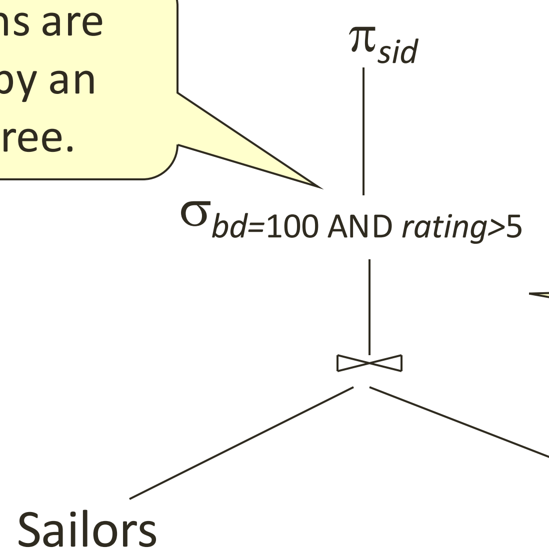
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

Reserves (*sid*: integer, *bid*: integer, *day*: date, *rname*: string)

```
SELECT sid
FROM Sailors NATURAL JOIN Reserves
    WHERE bid = 100 AND rating > 5;
```

$\pi_{sid}(\sigma_{bid=100 \text{ AND } rating>5}(\text{Sailors} \bowtie \text{Reserves}))$

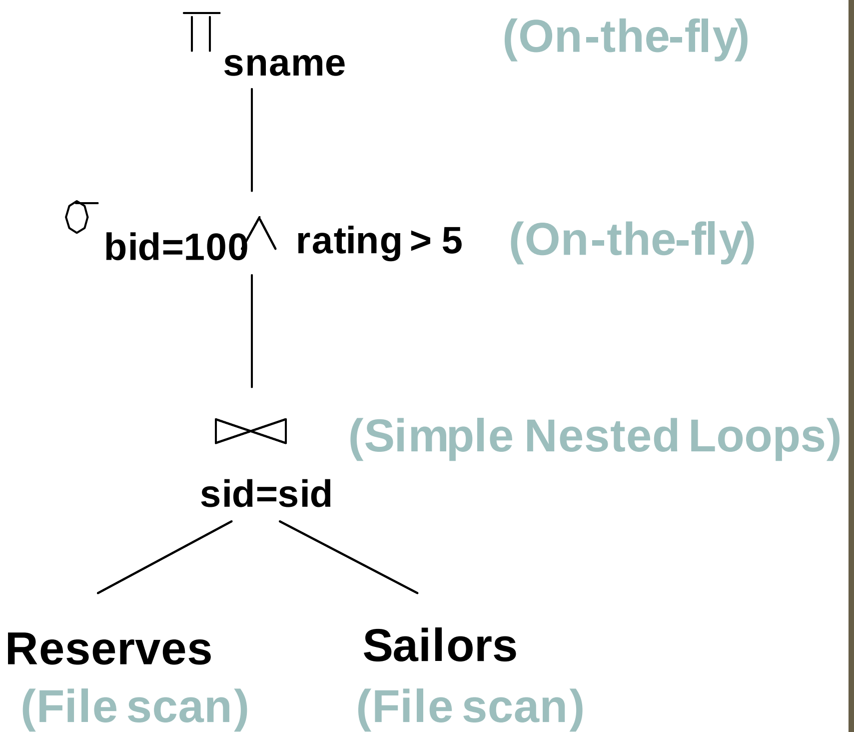
RA expressions are represented by an expression tree.



An algorithm is chosen for each node in the expression tree.

Query Evaluation Plan

- *Query evaluation plan* is an extended RA tree, with additional annotations:
  - *access method* for each relation;
  - *implementation method* for each relational operator.
- Cost: 500+500*1000 I/Os
- Misses several opportunities:
  - Selections could have been `pushed' earlier.
  - No use is made of any available indexes.
  - More efficient join algorithm…



Pipelined Evaluation

- *Materialization*: Output of an *op* is saved in a temporary relation for uses (multiple scans) by the next op.
- *Pipelining*: No need to create a temporary relation. Avoid the cost of writing it out and reading it back. Can occur in two cases:
  - *Unary operator*: when the input is pipelined into it, the operator is applied *on-the-fly*, e.g. selection on-the-fly, project on-the-fly.
  - *Binary operator*: e.g., the outer relation in indexed nested loops join.

Iterator Interface for Execution

- A query plan, i.e., a tree of relational ops, is executed by calling operators in some (possibly interleaved) order.
- *Iterator Interface* for simple query execution:
  - Each operator typically implemented using a uniform interface: *open*, *get_next*, and *close*.
  - Query execution starts top-down (*pull-based*). When an operator is `pulled' for the next output tuples, it
    1. `pulls' on its inputs (opens each child node if not yet, gets next from each input, and closes an input if it is exhausted),
    2. computes its own results.
- *Encapsulation*
  - Encapsulated in the operator-specific code: access methods, join algorithms, and materialization vs. pipelining…
  - Transparent to the query executer.

Approaches to Query Evaluation

- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

** Watch for these techniques as we discuss query evaluation during this lecture*

Statistics and Information Schema

- Need information about the relations and indexes involved. Catalog typically contains:
  - #tuples (NTuples) and #pages (NPages) for each relation.
  - #distinct key values (NKeys), INPages index pages, and low/high key values (ILow/IHigh) for each index.
  - Index height (IHeight) for each tree index.
  - Catalog data stored in tables; can be queried
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; costs are approximate anyway, so slight inconsistency expected.
- More detailed information (e.g., histograms of the values in some field) sometimes stored.

Access Paths: Method for retrieval

- Access path = **way of retrieving tuples**:
  - File scan, or index that matches a selection (in the query)
  - Cost depends heavily on access path selected
- A tree index matches (a conjunction of) conditions that involve only attributes in a prefix of the search key.
- A hash index matches (a conjunction of) conditions that has a term attribute = value for every attribute in the search key of the index.
- Selection conditions are first converted to conjunctive normal form (CNF):
  - E.g., (day<8/9/94 OR bid=5 OR sid=3) AND (rname='Paul' OR bid=5 OR sid=3)

Matching an index

Search key <a, b, c>

1. a=5 and b= 3?
2. a > 5 and b < 3
3. b=3
4. a=7 and b=5 and c=4 and **d>4**
5. a=7 and c=5

Tree Index	Hash Index
1. Yes	1. No
2. Yes	2. No
3. No	3. No
4. Yes	4. Yes
5. Yes	5. No

Index matches (part of) a predicate if:
Conjunction of terms involving only attributes (no disjunctions)
Hash: only equality operation, predicate has all index attributes.
Tree: Attributes are a prefix of the search key, any ops.

Selectivity of access path

- Selectivity = #pages retrieved (index + data pages)
- Find the most selective access path, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - Most selective path – fewer I/O
  - Terms that match the index reduce the number of tuples retrieved
  - Other terms are used to discard some retrieved tuples, but do not affect number of tuples fetched.
  - Consider "day < 8/9/94 AND bid=5 AND sid=3".
    - Can use B+ tree index on day; then check bid=5 and sid=3 for each retrieved tuple
    - Could similarly use a hash index on <bid,sid>; then check day < 8/9/94

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* ($\bigcup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY
  - *Order By* Returns tuples in specified order.
- 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.

Relational Operators to Evaluate

- **Evaluation of joins**
- Evaluation of selections
- Evaluation of projections

Schema for Examples

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

- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page
  - 500 pages. ~40,000 tuples
- Reserves:
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages. ~100,000 tuples

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R, Sailors S
WHERE  R.sid = S.sid
```

- In algebra: $R \bowtie S$, natural join, common operation
  - R X S is large; R X S followed by a selection is inefficient.
  - Must be carefully optimized.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
- *Cost metric*: # of I/Os. Ignore output cost in analysis.

Page-Oriented Nested Loops Join

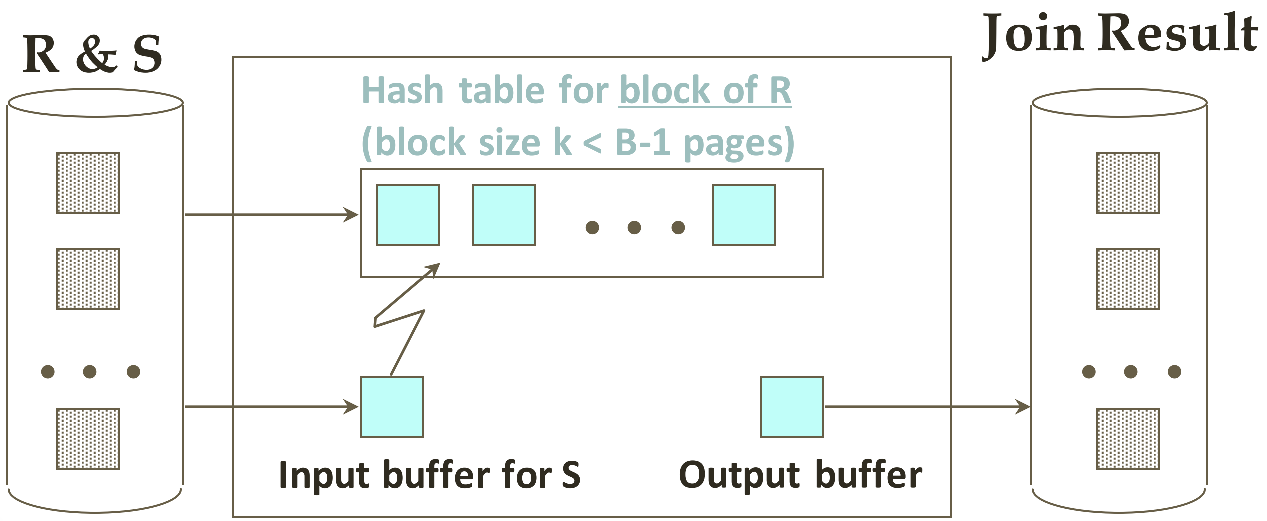
- How can we improve Simple Nested Loop Join?
- For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: M + M * N = 1000 + 1000*500 = 501,000 I/Os.
  - If each I/O takes 10 ms, the join will take 1.4 hours.
- Which relation should be the *outer*?
  - The ***smaller*** relation (S) should be the **outer**:
    cost = 500 + 500*1000 = 500,500 I/Os.
- How many buffers do we need?

Block Nested Loops Join

- How can we utilize additional buffer pages?
  - If the smaller relation fits in memory, use it as outer, read the inner only once.
  - Otherwise, read a big chunk of it each time, resulting in reduced # times of reading the inner.
- Block Nested Loops Join:
  - Take the <u>smaller</u> relation, say R, as <u>outer</u>, the other as inner.
  - Buffer allocation: one buffer for scanning the inner S, one buffer for output, all remaining buffers for holding a ``block'' of outer R.

Block Nested Loops Join Diagram

foreach block in R do
    build a hash table on R-block
    foreach S page
        for each matching tuple r in R-block, s in S-page do
            add <r, s> to result



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 depends on clustering.
  - Clustered index: 1 I/O (typical).
  - Unclustered: up to 1 I/O per matching S tuple.

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

- Sort R and S on join column using external sorting.
- Merge R and S on join column, output result tuples.

  Repeat until either R or S is finished:
  - *Scanning*:
    - Advance scan of R until current R-tuple >=current S tuple,
    - Advance scan of S until current S-tuple>=current R tuple;
    - Do this until current R tuple = current S tuple.
  - *Matching*:
    - Match all R tuples and S tuples with same value; output <r, s> for all pairs of such tuples.
- Data access patterns for R and S?

R is scanned once, each S partition scanned once per matching R tuple

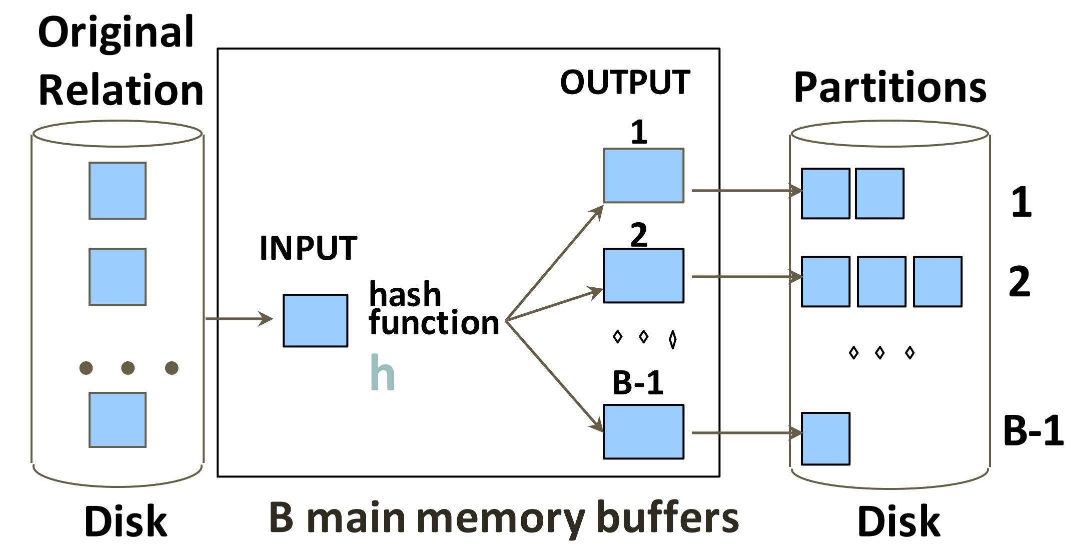
Refinement of Sort-Merge Join

- *Idea*:
  - *Sorting* of R and S has respective merging phases
  - *Join* of R and S also has a merging phase
  - Combine all these merging phases!
- Two-pass algorithm for sort-merge join:
  - Pass 0: sort subfiles of R, S individually
  - Pass 1: merge sorted runs of R, merge sorted runs of S, and merge the resulting R and S files as they are generated by checking the join condition.

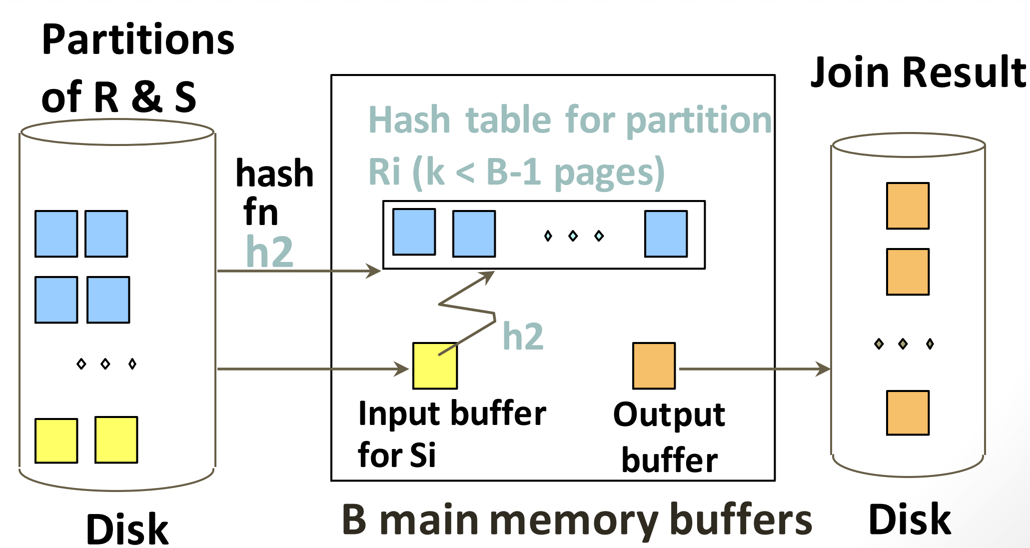
Hash-Join

- *Idea*: Partition both R and S using a hash function such that R tuples will only match S tuples in partition i.

- Partitioning: Partition both relations using hash fn **h**: Ri tuples will only match with Si tuples.



- Probing: Read in partition i of R, build hash table on Ri using **h2 (<> h!)**. Scan partition i of S, search for matches.



Hash Join Memory Requirement

- Partitioning: # partitions in memory ≤ B-1,

  Probing: size of largest partition (to fit in memory) ≤ B-2.
  - A little more memory is needed to build hash table, but ignored here.
- Assuming uniformly sized partitions, L = min(M, N):
  - L / (B-1) < (B-2) → B > $\sqrt{L}$
  - Hash-join works if the smaller relation satisfies above size restriction
- What if hash fn h does not partition uniformly and one or more R partitions does not fit in memory?
  - Can apply hash-join technique recursively to do the join of this R-partition with the corresponding S-partition.

Cost of Hash-Join

- Partitioning reads+writes both relations; 2(M+N).
  Probing reads both relations; M+N I/Os.
  Total cost = 3(M+N).
  - In our running example, a total of 4,500 I/Os using hash join, less than 1 min (compared to 1.4 hours w. Page Nested Loop Join).
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3(M+N) I/Os.
  - Hash Join superior if relation sizes differ greatly
  - Hash Join is shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index Nested Loop, build index on *<sid, sname>* (if S is inner); or use existing indexes on *sid* or *sname* and check the other join condition on the fly.
  - 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 Nested Loop, need B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins (clustered index is much preferred).
  - Hash Join, Sort Merge Join not applicable.
  - Block Nested Loop quite likely to be a winner here.

Relational Operators to Evaluate

- Evaluation of joins
- **Evaluation of selections**
- Evaluation of projections

Using an Index for **Selections**

- Cost depends on # qualifying tuples, and *clustering*.
  - Cost of finding data entries (often small) + cost of retrieving records (could be large w/o clustering).
  - For *gpa* > 3.0, if 10% of tuples qualify (100 pages, 10,000 tuples), cost ≈ 100 I/Os with a clustered index; otherwise, up to 10,000 I/Os!
- 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.

Each data page is looked at just once, although # of such pages likely to be higher than with clustering.

Approach 1 to General Selections

- (1) Find the *most selective access path*, *retrieve* tuples using it, and (2) apply any remaining terms that don't match the index *on the fly*.
  - *Most selective access path:* An index or file scan that is expected to require the smallest # 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 I/O cost.
  - 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.
    - A hash index on *<bid, sid>* could be used; *day<8/9/94* must then be checked on the fly.

Approach 2: Intersection of Rids

- If we have 2 or more matching secondary indexes:
  - Get sets of rids of data records using each matching index.
  - *Intersect* these sets of rids.
  - Retrieve the records and apply any remaining terms.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can:
    - retrieve rids of records satisfying *day<8/9/94* using the first, rids of records satisfying *sid=3* using the second,
    - intersect these rids,
    - retrieve records and check *bid=5.*

Relational Operators to Evaluate

- Evaluation of joins
- Evaluation of selections
- **Evaluation of projections**

The Projection Operation

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

- Projection consists of two steps:
  - Remove unwanted attributes (i.e., those not specified in the projection).
  - Eliminate any duplicate tuples that are produced, if DISTINCT is specified.
- Algorithms: single relation sorting and hashing based on all remaining attributes.

Discussion of Projection

- Sort-based approach is the standard; better handling of skew and result is sorted.
- If an index on the relation contains all wanted attributes in its search key, can do *index-only* scan.
  - Apply projection techniques to index entries (much smaller!)
- If a 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.
  - E.g. projection on <sid, age>, search key on <sid, age, rating>.

Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
  - *Cost of lookup = Height(I)+1 for a B+ tree, ≈ 1.2 for hash index.*
  - *Cost of record retrieval = 1*
- Clustered index I matching one or more selections:
  - *Cost of lookup + (INPages'(I)+NPages(R)) * product of RF's of matching selections. (Treat INPages' as the number of leaf pages in the index.)*
- Non-clustered index I matching one or more selections:
  - *Cost of lookup + (INPages'(I)+NTuples(R)) * product of RF's of matching selections.*
- Sequential scan of file:
  - *NPages(R).*
- May add extra costs for GROUP BY and duplicate elimination in projection (if a query says DISTINCT).

Example

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating=8
```

- If we have an index on *rating* ($1 \le$ rating ≤ 10):
  - NTuples(R) /NKeys(I) = 40,000/10 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages'(I)+NPages(R)) = (1/10) * (50+500) pages retrieved, plus lookup cost.
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40,000) pages retrieved, plus lookup cost.
- If we have an index on *sid*:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.
- Doing a file scan:
  - We retrieve all file pages (500).

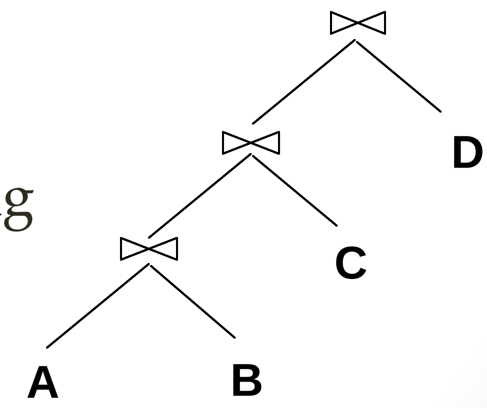
Queries Over Multiple Relations

- As the number of joins increases, the number of alternative plans grows rapidly.

❖ System R: (1) use *only left-deep join trees, where the* inner is a base relation, (2) avoid cartesian products.

- Allow *pipelined* plans; intermediate results not written to temporary files.
- Not all left-deep trees are fully pipelined!
  - Sort-Merge join (the sorting phase)
  - Two-phase hash join (the partitioning phase)

Left-deep



Cost Estimation for Multi-relation Plans

```
SELECT attribute list
FROM relation list
WHERE term1 AND ... AND termk
```

- Consider a query block:
- *Reduction factor (RF)* is associated with each *term*.
- *Max number tuples in result* = the product of the cardinalities of relations in the FROM clause.
- *Result cardinality* = max # tuples * product of all RF's.
- Multi-relation plans are built up by joining one new relation at a time.
  - Cost of join method, plus estimate of join cardinality gives us both cost estimate and result size estimate.

Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned.
- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Summary: Query plan

- Many 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 state (e.g., memory) and
  - statistics (table size, # tuples matching value k).
- This is part of the broader task of optimizing a query composed of several ops.

Summary: Optimization

- Query optimization is an important task in relational DBMS.
- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.