Applied Databases

4. Relational Query Optimisation.

November 10, 2010

Coverage

Reading: R&G 12.

The figures and example are taken from this chapter.

We'll consider implementation of the following operations:

- Join ($\bowtie$)
- Selection (σ)
- Projection (π)
- Union ($\cup$)
- Difference ($\setminus$)
- Aggregation (`GROUP-BY` queries)

Union and difference are closely related to *join*, which is by far the most interesting operation.

Two programs (on different data models)

OQL:

```
SELECT   e.Name, e.Dept.DName
FROM     Employees e
WHERE    e.Age > 40
```

SQL:

```
SELECT   e.Name, d.DName
FROM     Employees e, Departments d
WHERE    e.Age > 40
AND      e.DeptId = d.DeptId
```

Which is more efficient?

Sometimes it is better to do a join than a set of derefences. It may pay to "optimize" the OQL query to an SQL query!!

An example for relational operator optimization.

- Table R: p_R tuples/page, M pages ($= Mp_R$) tuples.
- Table S: p_S tuples/page, N pages ($= Mp_S$) tuples

Some plausible numerical values. When pages are 4000 bytes long, R tuples are 40 bytes, and S tuples are 50 bytes long.

tuples/page	# pages	# tuples
$p_R = 100$	$M = 1000$	$100,000$
$p_S = 80$	$N = 500$	$40,000$

Equality join on one column

```
SELECT   *
FROM     R, S
WHERE    R.i = S.i
```

Assume i is a key for S and is a foreign key in R, thus Np_S tuples in output

Simple nested loop join:

```
for each tuple r ∈ R
    for each tuple s ∈ S
        if r_i = s_i then add r, s to output
```

Cost (of I/O) $= M + p_R MN = 1000 + 100 \times 1000 \times 500 = 50,001,000$. This is the cost of reading all the pages of R plus the cost of reading all the pages of S for each *tuple* in R.

Page-oriented Nested Loop Join

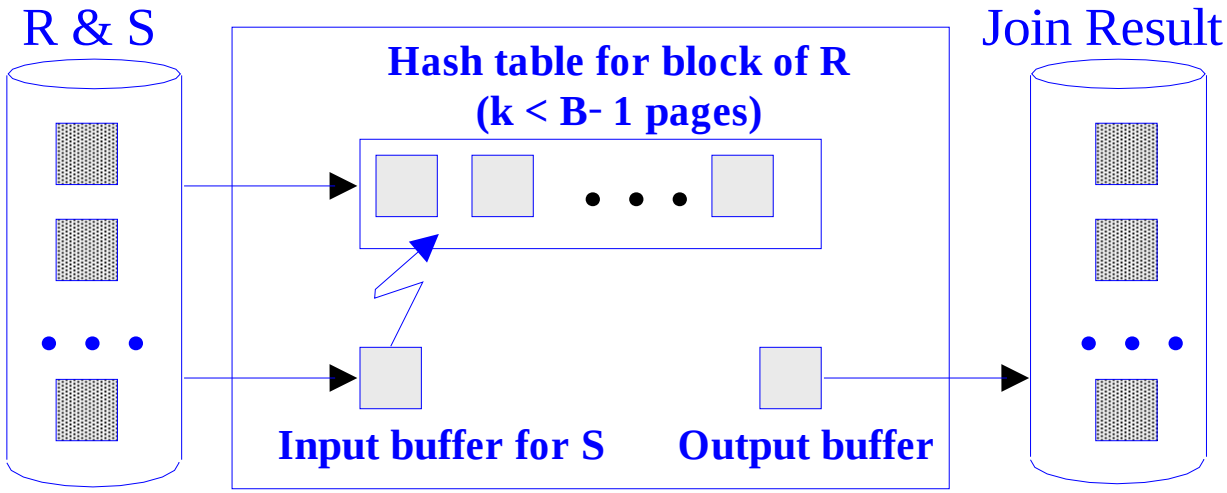
```
for each page B_R of R
  for each page B_S of S
    for each tuple r ∈ B_R
      for each tuple s ∈ B_S
        if r_i = s_i then add r, s to output
```

Cost is now $M + MN = 1000 + 1000 \times 500 = 501,000$ (Read all pages of R + all pages of S for every page of R

If we interchange the "inner" and "outer" tables, we get $N+NM = 500+1000\times500 = 500,500$

Block Nested Loop Join

Extension of page-oriented nested loop join. We read as many pages of the outer table into the buffer pool as we can (a "block" of pages).



Note that we can make each block a hash table to speed up finding matching tuples.

I/O cost of block nested loop join

Cost: Scan of outer table + # outer blocks $\times$ scan of inner table

(# outer blocks = $\lceil$# pages of outer / blocksize$\rceil$)

- With R as outer, and blocksize=100:
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, we scan S; 10×500 I/Os.
  - TOTAL: $6,000$ I/Os
- S as outer, and blocksize=100:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan R; $5 * 1000$ I/Os.
  - TOTAL: $5,500$ I/Os

Index joins

```
for each tuple r ∈ R
    for each tuple s with key r_i in index for S
        add r, s to output
```

Suppose average cost of lookup in index for S is L.

Cost of join is # of pages in R plus one lookup for each tuple in R. $M + MLp_R = 1000 + 100,000L$

If $L = 3$, this is $301,000$

The *minimum* value for L is 1 (when tuples of S are stored directly in hash-table buckets.)
Total is then $101,000$

Sort-merge join

- Sort both R and S on join column
- "Merge" sorted tables

Cost of sorting. Under reasonable assumptions about the size of a buffer pool, external memory sorting on 1000 pages can be done in two passes. Each pass requires us to read and write the file.

Note. Sorting $m = 100,000$ tuples requires $m \log_2 m \approx 1,700,000$ main memory comparisons. The I/O time in this example dominates.

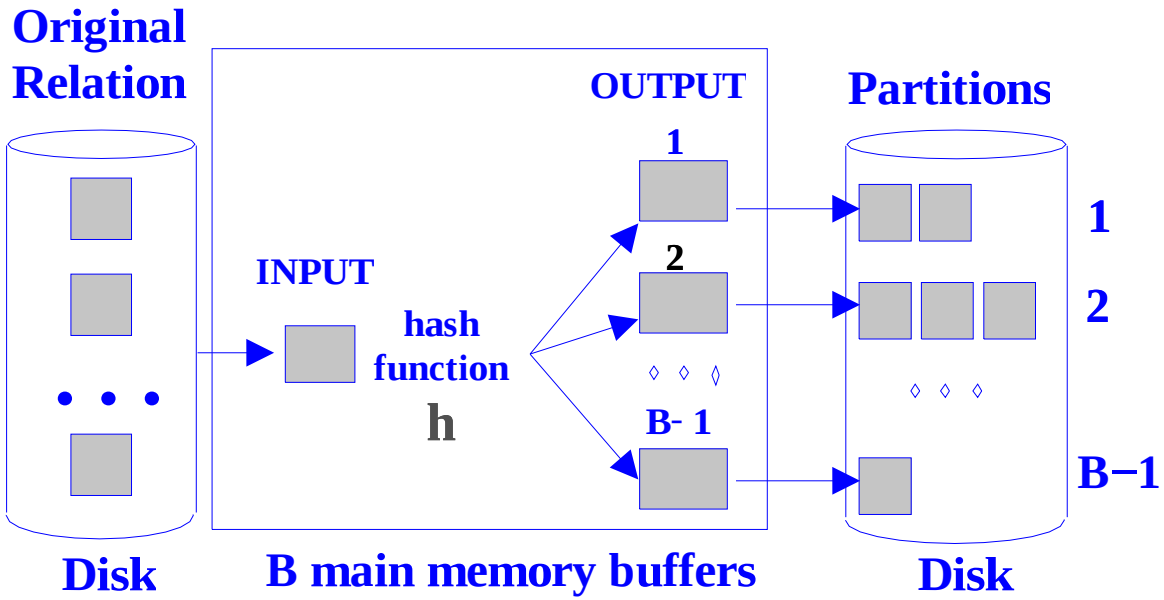
Cost of sorting R and S is $4 \times 1,000 + 4 \times 500 = 6,000$

Cost of Merge $= 1,500$

Total: $7,500$

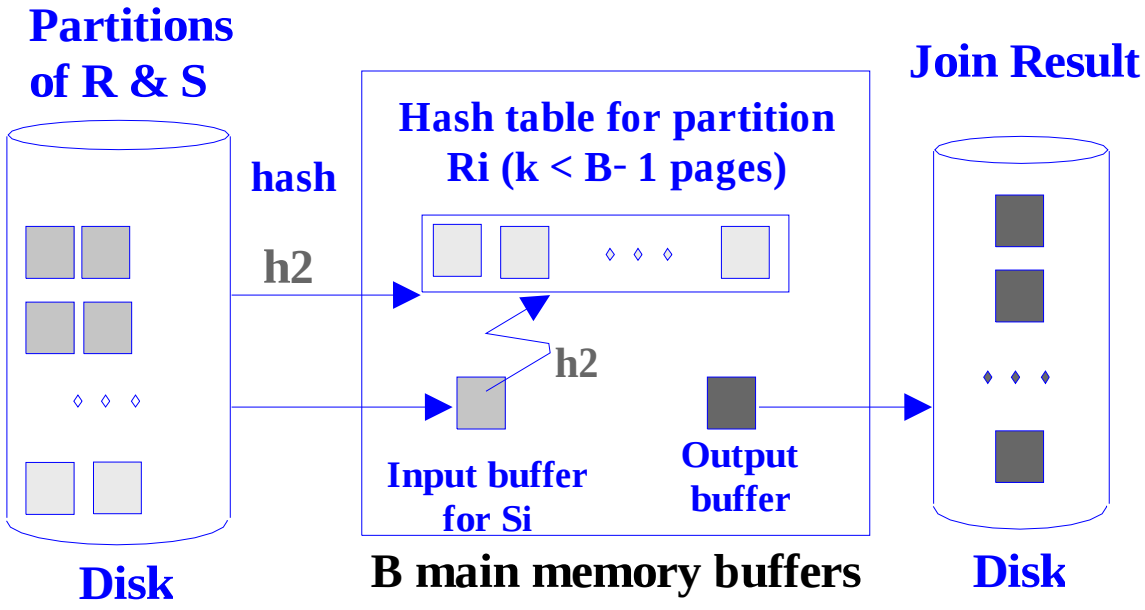
Hash join – phase 1

Partition both relations using hash function h applied to the join column values: R tuples in partition i will only match S tuples in partition i.



Hash join – phase 2

We do a block-nested join on each pair of partitions: read in a partition of R, hash it using h_2 (*must be different from* h) then scan matching partition of S and search for matches.



Observations on Hash-Join

- The number of partitions is determined by the number of buffer pages, $B - 1$
- We want each partition (of R) to fit into the buffer memory, the partitions should have less than $B - 2$ pages.
- Thus each table should occupy less than $\approx B^2$ pages.
- Can apply the technique recursively if needed.

Cost of hash-join = 3 passes through both tables, $3(M + N) = 4,500$!!

Hash-join is parallelisable.

General Join Conditions

- Equi-join on several fields. Straightforward generalisation of join on one field. Can make use of index on all or any of the join columns.
- Inequality joins, e.g., $R \bowtie_{R.i<S.i} S$. Hash joins inapplicable. Variations on sort-merge may work. E.g. for "approximate" joins such as
  $R \bowtie_{S.i-C<R.i \ \wedge \ R.i<S.i+C} S$.
  Indexing may work for clustered tree index.
  Block nested loop join is a good bet.

Selection

Single column selections, e.g. $\sigma_{i=C}(R)$ and $\sigma_{i>C}(R)$.

- No index on i – scan whole table.
- Index on i
  - Hash and tree index OK for equality.
  - Tree index may be useful for $\sigma_{i>C}(R)$, especially if index is clustered.
    If tree index is unclustered:
    * Retrieve qualifying rid's.
    * Sort these rid's.
    * Retrieve from data pages with a "merge".

Selection on "complex" predicates

General problem, can indexes do better than a complete scan of the table. Look for special cases.

- Range queries, $\sigma_{A<i \wedge i<B}(R)$. Use tree index.
- Conjunction of conditions, e.g. $\sigma_{i=A \wedge j=B}(R)$
  - Index on (i, j) – Good!
  - Index on i. Obtain tuples and then perform $\sigma_{j=B}$.
  - Separate indexes on i and j. Obtain and sort rid's from each index. Intersect (use merge) the sets of rid's.

Projection

Only interesting case is a "real" projection, `SELECT DISTINCT` $R.i$, $R.j$ `FROM` R.

- Eliminate duplicates by sorting
  - Eliminate unwanted fields at first pass of sort.

If the projection fields are entirely contained within an index, e.g., we have an index on (i, j), we can obtain results from index only.

Other set operations

- Intersection is a special case of join.
- Union and difference are similar. Again, the problem is eliminating duplicates.
  - Sorting based approach:
    - Sort both relations (on combination of all attributes).
    - Scan sorted relations and merge them.
  - Hash based approach to:
    - Partition R and S using hash function h.
    - For each S-partition, build in-memory hash table (using $h2$), scan corresponding R-partition and add tuples to table while discarding duplicates.

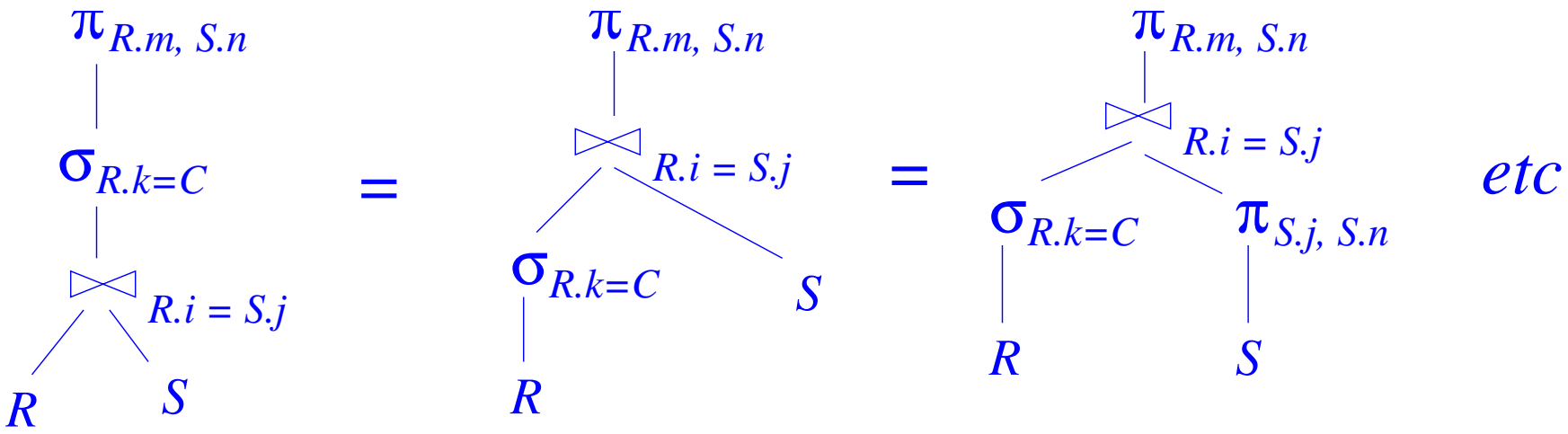
Query optimization – brief notes

We have examined how to evaluate each relational operation efficiently. How do we evaluate a whole query?

We make use of the rich set of *rewriting rules* for relational algebra. Examples:

- *Join re-ordering*. $R \bowtie S \bowtie T = R \bowtie S \bowtie T$. Do the most "selective" join first.
- *Pushing selection through ....* $\sigma_{R.i=C}(R \bowtie S) = \sigma_{R.i=C}(R) \bowtie S$.

Problem: the search space of expressions can be very large. In practice query optimizers explore a small subset of the possibilities.



Cost-based optimization

Whether or not a particular rewriting is a good idea depends on the statistics of the data. Consider pushing selection through joins. Is it a good idea in the following queries?

```
SELECT   E.Name, D.DName
FROM     Employee E, Department D
WHERE    E. DeptId = D.DeptId
AND      D.Address = "KB"
AND      E. Age < 20
```

```
SELECT   E.Name, D.DName
FROM     Employee E, Department D
WHERE    E. DeptId = D.DeptId
AND      D.Address = "KB"
AND      E. Age < 60
```

Keeping statistics of tables such as the *selectivity* of an attribute is needed to decide when a rewriting is useful. This compounds the problem of finding the optimum.

Relational Query Optimisation – Review

- Joins
  - Page-oriented joins
  - Block-nested loop joins
  - Index-based joins
  - Sort-merge joins
  - Hash joins
- Using indexes in selections.
- Projection – elimination of duplicates
- Union and Difference
- Query rewriting, why algebraic identities are useful
- Cost-based optimization.