Query Processing

Query Processing Overview

1. clever implementation techniques for operators
2. exploiting 'equivalencies' of relational operators
3. using **cost models** to choose among alternatives

Workflow: query -> query parser (check rightness) -> query optimizer

EXAMPLE:

- Sailors (S):
  $N = NPages(S) = 500, p_S = NTuplesPP(S) = 80, NTuples(S) = 500 * 80 = 40000$
- Reserves (R):
  $M = NPages(R) = 1000, p_R = NTuplesPP(R) = 100, NTuples(R) = 100000$
- Index (I): $RF = 0.1, NPages(I) = 50$

Selection

depends on

1. available indexes/access paths
2. expected size of the result

Estimate result size (Reduction Factor)

$Size\ of\ result \approx size\ of\ relation \times \prod(reduction\ factors)$

Reduction Factor (Selectivity): estimated portion of the qualified relation, given **predicates (AKA Conditions)**; from optimizer; e.g. 0.3, 0.05...

Alternatives for Selections

(cost = I/O, related to times we visit pages)

1. **no index, unsorted (Heap Scan):**

$$Cost = NPages(R)$$

2. **no index, sorted:**

$$Cost = log_2(NPages(R)) + (RF * NPages(R))$$

3. **Clustered index (sorted):**

$$Cost = (NPages(I) + NPages(R)) * RF$$

(Go through the index; then go though data entries one by one; finally retrieve data)

1. **Unclustered index (unsorted):**

$$Cost = (NPages(I) + NTuples(R)) * RF$$

2. **multiple predicates/conditions:**

only RFs of **matching predicates (primary conjuncts)** (AKA part of the prefix) will be used

Selection Approach

1. find the cheapest access path = least estimated page I/O
2. retrieve tuples, reduced by predicates matching
3. select over other predicates **on-the-fly**

Projection

1. Sort-based Projection

pipeline:

1. Scan R, extract only the needed attributes
2. Sort the result set using EXTERNAL SORT
3. Remove adjacent duplicates

EXTERNAL SORT: AKA multiple merge sort; $NPasses$ depends on memory buffer B, Will give us

$$SortingCost = 2 * NPasses * ReadProjectedPages$$

WriteProjectedPages: write pages with projected attributes to disk

$$WriteProjectedPages = NPages(R) * PF (= ReadProjectedPages)$$

PF (Projection Factor): remaining ratio w.r.t. all attributes; e.g. 0.25, 0.1...

$$\begin{aligned} Cost = ReadTable+ &= NPages(R)+ \\ WriteProjectedPages+ &\quad NPages(R) * PF+ \\ SortingCost+ &\quad 2 * NPasses * NPages(R) * PF+ \\ ReadProjectedPages &\quad NPages(R) * PF \end{aligned}$$

$$Cost = NPages(R) * (1 + 2 * PF + 2 * NPasses * PF)???$$

2. Hash-based Projection

pipeline:

1. Scan R, extract only the needed attributes
2. **Partition Phase:** partitioning data into B partitions with $h1$ hash function
3. **Duplicate Elimination Phase:** Load each partition, hash it with another hash function ($h2$) and eliminate duplicates

$$\begin{aligned} Cost = ReadTable+ &= NPages(R)+ \\ WriteProjectedPages+ &\quad NPages(R) * PF+ \\ ReadProjectedPages &\quad NPages(R) * PF \end{aligned}$$

$$Cost = NPages(R) * (1 + 2 * PF)???$$

Join

For $R \times S$, **Outer (Left)** = R, **Inner (Right)** = S

> So in our cases: $NPages(S) = NPages(Inner), NPages(R) = Npages(Outer)$.

Cost metrics are the same as selection and projection, i.e., Number of total I/O (pages)

$$A \times B == B \times A$$

$$A \times (B \times C) == (A \times B) \times C$$

1. Nested Loops Join

divided into Simple Nested Loops Join (SNLJ), Page-oriented Nested Loops Join (PNLJ) and Block Nested Loops Join (BNLJ).

1.1 Simple Nested Loops Join (SNLJ)

For each tuple in the outer relation R, we scan the entire inner relation S

$$\begin{aligned} Cost(SNLJ) =& NPages(Outer) + \\ & NTuples(Outer) * NPages(Inner) \end{aligned}$$

In [ ]:
```
for r in R:
    for s in S:
        if r == s:
            result.append((r,s))
```

1.2 Page-oriented Nested Loops Join (PNLJ)

Same as 1.1, except that based on pages.

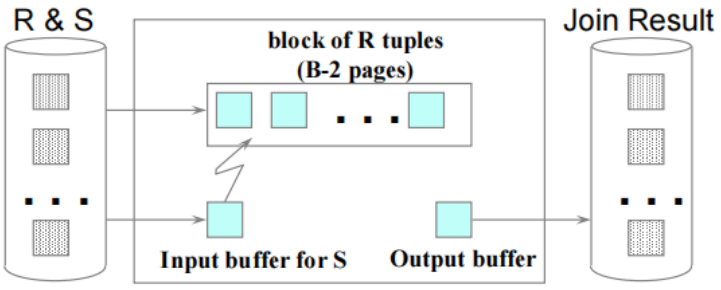
$$\begin{aligned} Cost(PNLJ) &= NPages(Outer) + \\ & NPages(Outer) * NPages(Inner) \\ &= NPages(Outer) * (1 + NPages(Inner)) \end{aligned}$$

In [ ]:
```
for page_R in R:
    for page_S in S:
        for r in page_R:
            for s in page_S:
                if r == s:
                    result.append((r,s))
```

1.3 Block Nested Loops Join (BNLJ)

Page-oriented NL doesn't exploit extra memory buffers; BNLJ reduces the passes

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



$$NBlocks(Outer) = \frac{NPages(Outer)}{B-2}$$

$$\begin{aligned} Cost(BNLJ) &= NPages(Outer) + \\ & NBlocks(Outer) * NPages(Inner) \\ &= NPages(Outer) * \left[1 + \frac{NPages(Inner)}{B-2}\right] \end{aligned}$$

2. Sort-merge Join

Sort R and S on the **join column**, then scan them to do a merge (on join column), and output result tuples; $NPasses$ will be given

Good for cases:

1. One or both inputs are already sorted on join attribute(s)
2. output is required to be sorted on join attribute(s).

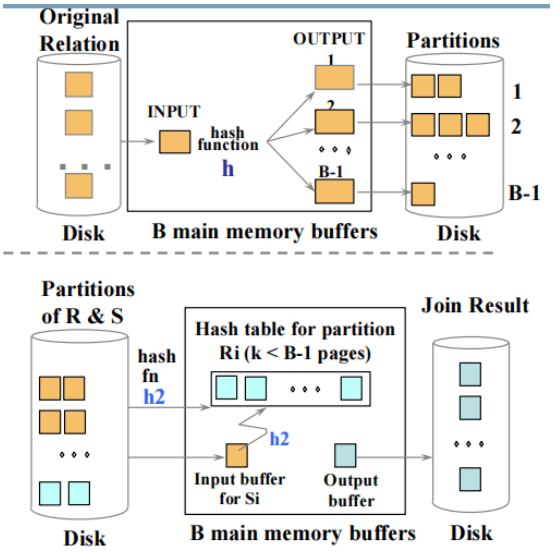
$$Sort(R) = EXTERNAL\_SORT(R) = 2 * NPasses * NPages(R)$$

$$\begin{aligned} Cost(SMJ) = \ & Sort(Outer) + Sort(Inner) \\ & + NPages(Outer) + NPages(Inner) \\ = \ & (2 * NPasses + 1) * (NPages(Outer) + NPages(Inner)) \end{aligned}$$

3. Hash Join

Partition Phase: both relations using hash function h: R tuples in partition l will only match S tuples in partition l; read+write both relations

Matching Phase: Read in a partition of R, hash it using h2 (<> h!). Scan matching partition of S, probe hash table for matches; read both relations



$$\begin{aligned} Cost(HJ) = \ & 2 * NPages(Outer) + 2 * NPages(Inner) && [partition] \\ & + NPages(Outer) + NPages(Inner) && [matching] \\ = \ & 3 * NPages(Outer) + 3 * NPages(Inner) \end{aligned}$$

General Join

I. Equalities over several attributes

- Nested-Loops Join: nothing changes
- Sort-Merge/Hash Join: sort/partition on combination of n join columns

II. Inequality conditions

- Block NL might be the best join method
- Hash Join, Sort Merge Join not applicable (haven't implemented)