CS411
Database Systems

8: Query Processing

Context

- Query: SELECT … FROM … WHERE …
- Query -> Logical query plan

Select a1, …, an
From R1, …, Rk
Where C

$$\Pi_{a1,\ldots,an}(\sigma_{C}(R1 \quad R2 \quad \ldots \quad Rk))$$

- Logical query plan -> Physical query plan

- Today's agenda: operations in the physical query plan

Logical v.s. Physical Operators

- Logical operators
  - *what* they do
  - e.g., union, selection, project, join, grouping
- Physical operators
  - *how* they do it
  - e.g., nested loop join, sort-merge join, hash join, index join
  - In other words, physical operators are particular implementations of relational algebra operations
  - Physical operators also pertain to non RA operations, such as "scanning" a table.

Getting started: Scanning Tables

- Read the entire contents of a relation R (or all tuples that satisfy a criterion)

- Table-scan: R is stored in some area of secondary storage (disk), in blocks. These blocks are known to the system. Can get all blocks one by one.

- Index scan: if we have index to find the blocks.
  – This can be particularly useful for getting tuples that satisfy a predicate

Sorting while scanning tables

- May want to sort the tuples as we read them: "Sort-scan"
- Why?
  - ORDER BY in query
  - Some RA operations are implemented using sort
- How?
  - If indexed, then trivial
  - If fits in main memory, then table-scan or index-scan and sort in memory
  - If too large, "multiway merge sort" (see later)

Physical operators and costs

- Logical query plan has RA operators
- Physical query plan has physical operators
- Often, we have to make choices about which physical operators to use.
- For this, we need to estimate the "cost" of each physical operator.

Cost Parameters (or "statistics")

- Estimating the cost:
  - Important in optimization (next lecture)
  - Compute disk I/O cost only
  - We compute the cost to *read* the arguments of the operator
  - We don't compute the cost to *write* the result

- Cost parameters
  - M = number of blocks that fit in main memory
  - B(R) = number of blocks needed to hold R
  - T(R) = number of tuples in R
  - V(R,a) = number of distinct values of the attribute a

Cost of Scan operators

- Cost parameters
  - M = number of blocks that fit in main memory
  - B(R) = number of blocks needed to hold R
  - T(R) = number of tuples in R
  - V(R,a) = number of distinct values of the attribute a

- Table scan: cost = B(R)

- Index scan: cost = B(R) + #blocks of the index

  $\approx$ B(R)

Sorting

- Two pass "multi-way merge sort"
- Have M main memory blocks available for use
- Step 1:
  - Read M blocks at a time, sort, write
  - Result: have runs of length M on disk
- Step 2:
  - Merge M-1 at a time, write to disk
  - Result: have runs of length $M(M-1) \approx M^2$
- Cost: $3B(R)$, Assumption: $B(R) \leq M^2$

Cost of the Scan Operator

- Table scan: B(R); Sort-scan: 3B(R)
- Index scan: B(R); Sort-scan: B(R) or 3B(R)

- Unclustered relation: we have assumed so far that all tuples of R are "clustered", i.e., stored in ~ B blocks. If tuples of R are interspersed with tuples of other relations, then cost:
  - T(R); to sort: T(R) + 2B(R)

The iterator model for implementing operators

- Each (physical) operation is implemented by 3 functions:
  - Open: sets up the data structures and performs initializations
  - GetNext: returns the the next tuple of the result.
  - Close: ends the operations. Cleans up the data structures.
- Enables pipelining!
  - As opposed to: execute each operator in entirety, store its results on disk or in main memory
  - Many operators can be “active” simultaneously
- Not always possible (or meaningful):
  - E.g., “sort scan”.

Overview of operator implementations

- Operator algorithms mostly of one of these types:
  - Sorting-based
  - Hash-based
  - Index-based
- Another classification of operator algorithms
  - One pass: reading the data only once from disk. Typically, at least one of the arguments must fit in memory.
  - Two pass: Data need not fit in memory, but is not "too large".
  - Multipass: No limit on data size.

One pass algorithms

One pass algorithms

- For:
  - “tuple at a time”, unary operations: selection, projection. (Do not require entire relation in memory at once.)
  - “full relation”, unary operations: grouping, duplicate-elimination. (Require all or most tuples in memory at once.) So one-pass methods are limited to scenarios where entire argument (relation) fits in memory
  - “full relation”, binary operations: everything else. (union, intersection, difference, joins, products). One-pass methods are limited to scenarios where at least one argument fits in memory.

One-pass Algorithms

Selection σ(R), projection Π(R)

- Both are <u>*tuple-at-a-Time*</u> operations

Input buffer → Unary operator → Output buffer

Note: available number of buffers (memory blocks) not an issue. M >= 1.

- Cost: B(R), assuming …
  - clustered relation on disk,
  - no index available for the selection attributes
- Note: R could be coming from another operation, or could have an index available for use

One-pass Algorithms

Duplicate elimination $\delta(R)$

- Need to keep a "dictionary" in memory:
  - Unique tuples seen so far
  - balanced search tree, hash table, etc.
- Memory requirement: $M >= B(\delta(R))$
  - We need to estimate $B(\delta(R))$ in advance, when planning whether to use this algorithm. Significant penalties if we underestimated!
- Cost: $B(R)$, again assuming clustered relation on disk, no index use.

One-pass Algorithms

Grouping: $\gamma_{\text{city, sum(price)}}$ (R)

- SELECT city, SUM(price) FROM R GROUP BY city
- Need to keep a dictionary in memory
- Each entry in the dictionary is: (city, sum(price))
- What if "SUM" was replaced with "AVG" ?
- Memory requirement: number of cities fits in memory
- Cost: B(R), i.e., whatever it takes to read the blocks
- Note: Not ideally suited for the "iterator" model (pipelined production of output tuples). Why?

One-pass Algorithms

Binary operations: $R \cap S$, $R \cup S$, $R - S$

- Bag union: Trivial.
  - Read R, block by block, copy each tuple to output.
    Read S, block by block, copy each tuple to output.
- Memory requirement: M >= 1.
- Cost: B(R)+B(S)

One-pass Algorithms

- Other binary operations:
  - Read the smaller relation, store in memory.
  - Build some data structure so that tuples can be accessed and inserted efficiently. (Hash table or B-tree)
  - Read the other relation, block by block, go through each tuple and decide whether to output or not.

- Memory requirement: M >= min(B(R), B(S)).
- Cost: B(R)+B(S)

One-pass Algorithms

- Set Union:
  - Read the smaller relation (say S) into memory .
  - Build a search structure whose search key is entire tuple.
  - Read R, block by block. Once a block is loaded, for each tuple *t* in that block, see if *t* is in S; if not, copy *t* to the output block.

- Memory requirement: M >= min(B(R), B(S)).
- Cost: B(R)+B(S)

One-pass Algorithms

- Set Intersection:
  - Read the smaller relation (say S) into memory .
  - Build a search structure whose search key is entire tuple.
  - Read R, block by block. Once a block is loaded, for each tuple *t* in that block, see if *t* is in S; if so, copy *t* to the output block.

- Memory requirement: M >= min(B(R), B(S)).
- Cost: B(R)+B(S)

One-pass Algorithms

- Set Difference:
  - Not commutative (R – S is not the same as S – R)
  - Read the smaller relation (say S) into memory .
  - Build a search structure whose search key is entire tuple.
  - Read R, block by block. Once a block is loaded, for each tuple *t* in that block, see if *t* is in S; if not, copy *t* to the output block. This computes R – S.
  - To compute S – R: for each tuple *t* in R, see if *t* is in S; if so, delete *t* from the copy of S in memory. At the end, output every tuple in S (in memory).

One-pass Algorithms

- Bag intersection, Bag Difference, Product, Join:
  - See text.

Nested Loop Joins ("one and a half pass" algorithms)

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- R=outer relation, S=inner relation

<u>for</u> each tuple r in R <u>do</u>
  <u>for</u> each tuple s in S <u>do</u>
    <u>if</u> r and s join <u>then</u> output (r,s)

- M >= 2
- Cost: T(R) T(S)
- Fits the iterator model

Nested Loop Joins

- Block-based Nested Loop Join

```
for each (M-1) blocks bs of S do
   for each block br of R do
        for each tuple s in bs do
             for each tuple r in br do
                  if r and s join then output(r,s)
```

Nested Loop Joins

Nested Loop Joins

- Block-based Nested Loop Join

- Cost:
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-1) times, and each time need to read R: costs B(S)B(R)/(M-1)
  - Total cost: B(S) + B(S)B(R)/(M-1)

- Notice: it is better to iterate over the smaller relation first– i.e., S.

Summary so far

Operators	Approximate M required	Disk I/O (Cost)
σ, π	1	B
γ, δ	B	B
∩, U, –, x, Join	Min (B(R), B(S))	B(R) + B(S)
Join	Any M >= 2	B(R)B(S)/M

Two pass algorithms

Two-Pass Algorithms Based on Sorting

Duplicate elimination $\delta(R)$

- Simple idea: like sorting, but include no duplicates

- Step 1: sort runs of size M, write
  - Cost: 2B(R)
- Step 2: merge M-1 runs,

  but **include each tuple only once**
  - Cost: B(R)

- Total cost: 3B(R), Assumption: $B(R) <= M^2$
- Compare with one-pass: cost B(R); Assumption $B(R) <= M$

Q: What can sorting help? And, how?

- Selection?
- Projection?
- Set operations?
- Join?
- Duplicate elimination?
- Grouping?

Two-Pass Algorithms Based on Sorting

Grouping: $\gamma_{\text{city, sum(price)}}$ (R)

- Same as before: sort, then compute the sum (price) for each group
- Compute sum(price) during the merge phase.
- Total cost: 3B(R)
- Assumption: $B(R) <= M^2$

Two-Pass Algorithms Based on Sorting

Binary operations: R ∩ S, R U S, R – S

- Idea: sort R, sort S, then do the right thing
- A closer look:
  - Step 1: split R into runs of size M, then split S into runs of size M. Cost: 2B(R) + 2B(S)
  - Step 2: merge *all* x runs from R; merge all y runs from S; ouput a tuple on a case by case basis ($x + y \le M$)
- Total cost: 3B(R)+3B(S)
- Assumption: B(R)+B(S)<= M^2

Two-Pass Algorithms Based on Sorting

Join $R \bowtie S$. Let's recap what we've seen so far.

(a) min(B(R), B(S)) < M: Load smaller table to memory and load other table block by block. Cost: B(R)+B(S). This is the <u>one-pass algorithm</u>.

(b) Min(B(R), B(S)) > M: Load to memory (M-1) blocks of S; go over every block of R; repeat. Cost: B(R)B(S)/M. This is the <u>nested-loop join algorithm</u>.

Nested loop join is the only option we have if Min(B(R), B (S)) > M, but is too expensive (α B(R)B(S)). Why?

Two-Pass Algorithms Based on Sorting

Join R $\bowtie$ S

- Start by sorting both R and S on the join attribute:
  - Cost: 4B(R)+4B(S) (because need to write to disk)
- Read both relations in sorted order, match tuples
  - Cost: B(R)+B(S)
- Difficulty: many tuples in R may match many in S
  - If at least one set of tuples fits in M, we are OK
  - Otherwise need nested loop, higher cost
- Total cost: 5B(R)+5B(S)
- Assumption: $B(R) <= M^2$, $B(S) <= M^2$
- See Section 15.4.6.

Two pass algorithms based on hashing

Two Pass Algorithms Based on Hashing

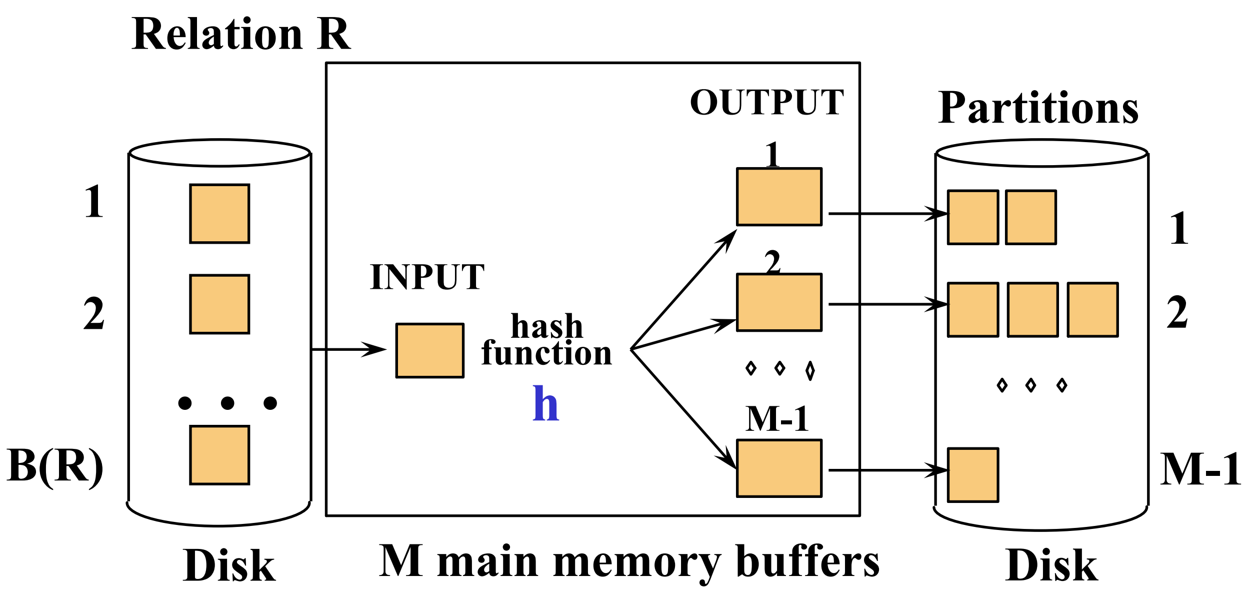
- Idea: partition a relation R into M roughly equal sized buckets, on disk
- For unary operations (e.g., duplicate elimination): All tuples that need to be considered together (in the operation) are in the same bucket.
- Can therefore do the operation by only looking at one bucket at a time.

Two Pass Algorithms Based on Hashing

- Idea: partition a relation R into M roughly equal sized buckets, on disk
- For binary operations (e.g., intersection): All tuples that need to be considered together (in the operation) are in a pair of buckets with the same hash value.
- Can therefore do the operation by only looking at one pair of buckets at a time.

Two Pass Algorithms Based on Hashing

- How to partition a relation R into (M-1) roughly equal sized buckets (on disk)?
- Each bucket has size approx. B(R)/(M-1)



- Does each bucket fit in main memory ?
  - Yes if $B(R)/(M-1) <= M$, i.e. $B(R) <= M^2$

Hash Based Algorithms for δ

- Recall: $\delta(R)$ = duplicate elimination
- Step 1. Partition R into M-1 buckets.
  - Note: duplicate tuples must fall into same bucket
- Step 2. Apply δ to each bucket
  - May use one-pass duplicate elimination here (cost B(R))
  - This assumes each bucket fits in memory.
  - That is, $B(R)/(M-1) <= M$, or $B(R) <= M^2$
- Cost: 3B(R)
- Same costs and constraints as sorting-based join.

Hash Based Algorithms for γ

- Recall: $\gamma(R)$ = grouping and aggregation
- Step 1. Partition R into M-1 buckets; *use grouping attributes as hash function*.
- Step 2. Apply γ to each bucket
  - Note: all tuples of the same group will be in the same bucket.
- Cost: 3B(R)
- Assumption: $B(R) <= M^2$
- Same as sort-based join.

Two pass Hashing-based Join

- R $\bowtie$ S
- Recall the *one-pass hash-based join*:
  - Scan S into memory, build buckets in main memory
  - Then scan R and join
  - Assumed of course that the smaller table is smaller than the memory available.

Two pass Hashing-based Join

$R \bowtie S$

- Step 1:
  - Hash S into M buckets, using join attribute(s) as hash key
  - send all buckets to disk
- Step 2
  - Hash R into M buckets, using join attribute(s) as hash key
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets with the same bucket number. Use the one pass algorithm for this.
  - Works as long as for each bucket number *i*, either R_i or S_i fits in memory.
  - Roughly speaking: $\min(B(R), B(S)) <= M^2$
- Cost = $3(B(R)+B(S))$

Read 15.5.6 (not covered in lecture)

Sort-based vs Hash-based

- For sorting-based implementations of binary operations, size requirement was $B(R)+B(S)<=M^2$. For hashing-based implementation, requirement is $\min(B(R),B(S))<=M^2$.
  - Hashing wins!
- Output of sorting-based algorithms are in sorted order, which may be useful for subsequent operations.
  - Sorting wins!
- Hashing-based algorithms rely on buckets being of roughly equal size. This may be a problem, and may lead
  - Sorting wins!
- Other differences too. Read 15.5.7.

Index-based algorithms

Index Based Algorithms

- In a clustered index all tuples with the same value of the key are clustered on as few blocks as possible

DISK BLKS: | … a a a | | a a a a a | | a a …. |

Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- Clustered index on a: cost $B(R)/V(R,a)$
  - $V(R, a)$ was defined as the number of distinct values of the attribute a.
- Unclustered index on a: cost $T(R)/V(R,a)$

Index Based Selection

- Example: $B(R) = 2000$, $T(R) = 100{,}000$, $V(R, a) = 20$, compute the cost of $\sigma_{a=v}(R)$

- Cost of un-indexed selection:
  - If R is clustered: $B(R) = 2000$ I/Os
  - If R is unclustered: $T(R) = 100{,}000$ I/Os

- Cost of index based selection:
  - If index is clustered: $B(R)/V(R,a) = 100$
  - If index is unclustered: $T(R)/V(R,a) = 5000$

- Note: when $V(R,a)$ is small, then unclustered index is useless

Index Based Join

- $R \bowtie S$
- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S
- Assume R is clustered. Cost:
  – If index (on S) is clustered: $B(R) + T(R)B(S)/V(S,a)$
  – If index (on S) is unclustered: $B(R) + T(R)T(S)/V(S,a)$
- Looks useless (Example 15.12), but see text (paragraph following the example).

Index Based Join

- Assume both R and S have a sorted index (B+ tree) on the join attribute. (A clustering index.)
- Then perform a merge join (called zig-zag join)
  - This is only the last step of the "two pass sorting-based join" algorithm we saw previously.
- Cost: B(R) + B(S)