Evaluation of Relational Operations

SS Chung
Cost Metric

- Query Processing Cost
  = Disk I/O Cost + CPU Computation Cost
- Disk I/O Cost
  = Disk Access Time + Data Transfer Time
- Disk Access Time
  = Seek Time + Latency
Disk I/O for Read/Write

- Unit for Disk I/O for Read/Write:
  - One Buffer for Read – One Block Size
  - One Buffer for Write – One Block Size

- Size of Block (Sector in Disk)
  - Traditionally 512 bytes
  - Modern OS uses $8 \times 512 = 4096$ bytes
Access Time

- Disk Access Time = Seek Time + Latency
- Seek Time: ~ 8ms in average
  the time it takes the head assembly on the actuator arm to travel to the track of the disk where the data will be read or written
- Rotational Latency: ~ 4ms in average
  the delay waiting for the rotation of the disk to bring the required disk sector under the read-write head.
Data Transfer Rate

- This rate depends on the track location, so it will be higher for data on the outer tracks (where there are more data sectors) and lower toward the inner tracks.
- **Internal rate**
  - moving data between the disk surface and the controller on the drive.
- **External rate: 60 MB/sec in average**
  - moving data between the controller on the drive and the host system.
Cost Metric for Query Processing Cost

- Query Processing Cost
  - $= \text{Disk I/O Cost} + \text{CPU Computation Cost}$
- CPU Cost is ignorable compared to I/O Cost
- Disk I/O Cost
  - $= \text{Disk Access Time} + \text{Data Transfer Time}$
- Data Transfer Time is ignorable compared to Disk Access Time
- Disk Access Time
  - $= \text{Seek Time} + \text{Latency}$
Query Processing Cost

- Query Processing Cost
- = Disk I/O Cost
- = # of Disk I/O * Disk Access Time
- = # of Disk I/O * (8 ms + 4 ms)
- = Total # of Disk Block access needed * 12ms
SSD: Solid State Disk/Drive

- Electrically, mechanically and software compatible with a conventional (magnetic) hard disk
- Storage medium is not magnetic (like a hard disk) or optical (like a CD) but solid state semiconductor such as battery backed RAM, EPROM or other electrically erasable RAM like chip such as flash.
- The SSD access time does not depend on a read/write interface head synchronising with a data sector on a rotating disk
- Greater physical resilience to physical vibration, shock and extreme temperature fluctuations. SSDs are also immune to strong magnetic fields
- Fast Access Time: 10x - 40x Faster Read, 5x Faster Write
SSD vs HDD

- A data storage device that uses solid-state memory to store persistent data with the intention of providing access in the same manner of a traditional block I/O hard disk drive.

- Traditional magnetic disks such as hard disk drives (HDDs) or floppy disk, which are electromechanical devices containing spinning disks and movable read/write heads.

- SSDs use microchips that retain data in non-volatile memory chips and contain no moving parts.

- Compared to electromechanical HDDs, SSDs are typically less susceptible to physical shock, are silent, have lower access time and latency, but are more expensive per gigabyte (GB).

- SSDs use the same interface as hard disk drives, thus easily replacing them in most applications.

- Most SSDs use NAND-based flash memory, which retains memory even without power. SSDs using volatile random-access memory (RAM) also exist for situations that require even faster access.
DBMS Parallel Architecture

- Parsing Engine (PE)
  - SQL Parser & Optimizer
  - Query Step Dispatcher
  - Session Manager
  - Input Data Conversion

Message Subsystem

AMP
- R3 R8 R11
AMP
- R1 R6 R4
AMP
- R7 R2 R22
AMP
- R12 R9 R5
Optimizer Input and Output

Resolver Tree

OPTIMIZER

Statement list that specifies "best" (join) plan
Learning Objectives

- In a typical major DBMS, statistics are automatically collected.
- Given collected statistics, estimate a predicate’s output size/selectivity.
- For each relational operator, describe the major algorithms, their optimizations, their pros and cons, and their costs
  - Selection
  - Projection
  - Equijoins
  - General Joins
  - Set Operators
Motivation/Review

- Assume Sailors has an index on age.
- Does the optimal plan for this query use the index?

```sql
SELECT *
FROM Sailors S
WHERE S.age < 31
```

- Moral: In order to choose the optimal plan we need to know:
  - selectivity of the predicate
  - size of the output.
Collecting Statistics

- **What are statistics**
  - Table sizes, index sizes, ranges of values, etc.

- **Where are statistics kept? In the system catalog.**

- **Why collect statistics?**
  - Statistics are needed to determine selectivity of predicates and sizes of outputs of operators.
  - Data in the previous bullet is needed to calculate costs of plans.
  - Data in the previous bullet is needed by the optimizer to find the optimal plan.

- **How often are statistics collected?**
  - Typically done when 10% of the data has been updated.
    - Can be overridden manually
    - UPDATE STATISTICS, ANALYZE
  - Typically done by sampling, if table is large

- **Which tables/columns are monitored?**
  - Typically all tables/columns are monitored
Sailors Example

- What statistics would you collect to find the number of rows satisfying:
  - Age < 31?
  - Rank =4 ?

- In Intro DB we assumed data was uniformly distributed.
Relational Operations

- **Selection** (\(\sigma\)) Selects a subset of rows from relation.
- **Projection** (\(\pi\)) Deletes unwanted columns from relation.
- **Join** (\(\bowtie\)) Allows us to combine two relations.
- **Set-difference** (\(\setminus\)) Tuples in reln. 1, but not in reln. 2.
- **Union** (\(\cup\)) Tuples in reln. 1 and in reln. 2.
- **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.
Schema for Examples

Sailors (\textit{sid}: integer, \textit{sname}: string, \textit{rating}: integer, \textit{age}: real)
Reserves (\textit{sid}: integer, \textit{bid}: integer, \textit{day}: dates, \textit{rname}: string)

- **Specs:**
  - page size, block size, table size
  - \# of tuples(rows) per page = size of page / length of a tuple
  - table size in page = size of table / size of page

Example: 1 page = 4096 byte = 8sector, 1 sector = 512 byte

- **Reserves:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.

- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
This tiny size is inefficient, and the operating systems long ago abandoned it for (in almost all cases) a 4096 unit of storage called a "page" or "allocation unit". Physically, each page is simply a collection of 8 sectors on the disk.

The largest number that can be stored in 32 bits is 4 gig. If this number is a sector number, that means that the most sectors you can address on a hard drive with a 32 bit number is 2 terabytes. Disks are now available that hold 2 terabytes of data, and bigger disks are in the pipeline. So for these large disks, a new sector size is a requirement.

An increasingly large number of modern disks are coming with a physical sector size of 4096 bytes. They can pretend to have 512 byte sectors for compatibility, but this is inefficient.

- 1 page = 8 sector = 4096 byte
- 1 sector = 512 byte
- Average Access time = 4 ms
- These days: 1 sector = 4096 byte
- 1 page = 8 sectors * 4096 byte
Equality Joins With One Join Column

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

- In algebra: $R \bowtie S$ is Common! Must be carefully optimized. If $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.

- Size of Table R:
  - $M$ (size of $R$/page size) pages in $R$
  - $p_R$ tuples per page (page size/length of a tuple in $R = 100$)

- Size of Table S:
  - $N$ (size of $S$/page size) pages in $S$
  - $p_S$ tuples per page (page size/length of a tuple in $S = 80$)

- *Cost metric: # of I/Os.* We will ignore output costs.
**Selection** R.age = 25 \((R)\)

- **Selection with No Index and Unsorted Data**
  Scan entire table \(R\) doing comparison:
  \(M\) (page) I/O : \(O(M)\)

- **Selection with No Index and Sorted Data**
  Scan with Binary Search doing comparison:
  \(\log_2 M\) (page) I/O: \(O(\log_2 M)\)
**Projection** $R$.attr1, $R$.attr2 ($R$)

- Select Distinct $R$.attr1, $R$.attr2
  
  From Reserve $R$

1. Remove unwanted Attributes
2. Eliminate duplicates from temp1 in step 1

- Algorithm Steps:
  1. Scan $R$ producing a set of tuples with only attr1, attr2 as temp1:
     
     $M$ pages I/O to scan $R$ + $T$ pages I/O to write temp1
  2. Sort temp1 : $T \log T$
  3. Scan sorted temp1 doing comparison to discard duplicates: $T$ pages I/O

- Total I/O Cost: $(M+T) + T \log T + T \approx O(M) + O(M \log M) + O(M)\n  
  = O(M \log M)$

  where $T$ is # of pages of temp1 with 10 byte for a tuple in temp1
Simple Nested Loops Join

For each tuple r in R do
   for each 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. (tuple at a time)
  - Cost: M (to scan R) + (p_R * M) times * N (to scan S)
    = 1000 + 100*1000*500 I/Os

- Page-oriented Nested Loops join (page at a time): 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 (to scan R) + M times *N = 1000 + 1000*500 I/Os
  - If smaller relation (S) is outer T, Cost = 500 + 500*1000 I/Os
Index Nested Loops Join

For each tuple r in R do
   for each 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: \( M + (M * p_R) * \text{Cost of finding matching S tuples} \)

- For each R tuple,
  - Cost of probing S index:
    - 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 index: up to 1 I/O per matching S tuple.
Examples of Index Nested Loops

- **Hash-index 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 (hash table entry)
    - + 1 I/O to get (the exactly one the hash index points to) matching Sailors tuple.
  - Total: \((220,000 + 1000)\) I/Os

- **Hash-index 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
    - + 1 I/O cost of retrieving matching Reserves tuples.
  - Total: \(40000*2.2 + 500 = 88000 + 500\) I/Os if one matching
    - \((40000*3.7 + 500\) I/Os if ave 2.5 unclustered matching)
  
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.
Block Nested Loops Join

- 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 (smaller).
  - 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.
  - Usually good when enough memory for hash table, buffers for outer block
Examples of Block Nested Loops

- **Cost**: Scan of outer + \( \# \text{outer blocks} \times \text{Scan of inner} \)
  - \( \# \text{outer blocks} = \left\lceil \frac{\# \text{of pages of outer}}{\text{blocksize}} \right\rceil \)

  - With Reserves (R) as outer, and 1000 pages of R, block size 100 pages:
    - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
    - Per block of R, we scan Sailors (S); 10*500 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.

- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.
Examples of Block Nested Loops

- **Cost**: Scan of outer + #outer blocks * Scan of inner
  - #outer blocks = \[ \lceil \text{# of pages of outer} / \text{blocksize} \rceil \]
    - M + N when smaller fits into block memory
    - M + M/(B-2) * N if not

- With Reserves (R) as outer, and 1000 pages of R, block size 100 pages:
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks.
  - Per block of R, we scan Sailors (S); 10*500 I/Os. Total = 1000 + 10*500
  - 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 = 500 + 5*1000

- With **sequential reads** considered, analysis changes: may be best divided differently among R and S.
Sort-Merge Join \((R \bowtie S)_{i=j}\)

- Sort R and S on the join column - Sort Phase
- Scan them to do a "merge" (on join col.), and output result tuples - Merge Phase:
  - Advance scan of R until current R-tuple \(\geq\) current S tuple, then advance scan of S until current S-tuple \(\geq\) current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (current R group) and all S tuples with same value in Sj (current S group) match; output \(<r, s>\) for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; Each S group (duplicates) is scanned once per matching R tuple.
- More duplicates in R, each S group scanned more. If join is on the key of R, no multiple scan for each S group.
- Multiple scans of an S group are likely to find needed pages in buffer.
Example of Sort-Merge Join

- Cost: $(M \log M) + (N \log N) + (M+N)$
  - The cost of scanning, $M+N$, could be $M*N$ (very unlikely!)
  - Good when one relation is already sorted on join column and/or clustered index on join column

- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

Compared to BNL cost: 2500 to 15000 I/Os
**Refinement of Sort-Merge Join**

- We can combine the merging phases of the *sorting* of R and S with the merging phase for matching for the join.
  - With $B > \sqrt{L}$ 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 $< B/2$.
  - Allocate 1 page per run of each relation, and `merge` while checking the join condition.
  - **Cost:** read and write each relation for sorting in Pass 1: $2(M+N) + \text{read each relation in (only) merging in Pass 2 : } (M+N)$
    
    $$= 3(M + N) \text{ (plus writing of result tuples).}$$
  - In example, cost goes down from 7500 to 4500 I/Os.
Hash-Join

- **Partitioning Phase:** Partition both relations using hash fn $h$ on join col: $R$ tuples in partition $i$ will only match $S$ tuples in partition $i$. $k$ output buffers + 1 input buffer

- **Probing Phase:**
  Read in a partition of $R$ ($R_i$), build hash table using $h2(ri)$ ($<> h1$). Scan the matching partition of $S$ ($S_i$) probing with $h2(si)$, search for matches. Write the matches
Observations on Hash-Join

- Memory requirement:
  - #partitions $k < B-1$: $k$ output buffers + 1 input buffer
  - $B-2 >$ size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing $k$ with minimizing partition size to fit into memory $B-2$, we get:
    - $k = B-1$, and $M/(B-1) + 2 < B$, i.e., $B$ must be $> \sqrt{M}$

- If we build an in-memory hash table to speed up the matching of tuples, 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.  
  Total Cost: $3(M + N)$

- In our running example, this is a total of 4500 I/Os.

- 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 on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.
General Join Conditions

- Equalities over several attributes (e.g., \( R.sid = S.sid \) \( \text{AND} \) \( R.rname = S.sname \)):
  - For Index NL, build index on \(<\text{sid, sname}>\) (if S is inner); or use existing indexes on \text{sid} or \text{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 (only for =).
  - Block NL quite likely to be the best join method here.
Set Operation: UNION and EXCEPT

- They are similar; we’ll do UNION.
- The hard part is removing duplicates (as in Project).

Sorting based approach to Union:
- Sort both relations (on a key).
- Merge sorted relations, discarding duplicates.
  - *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, scan corresponding R-partition and add tuples to table while discarding duplicates.
14.6 Aggregates w/o grouping

- **Example**
  - `SELECT AVE(S.age) FROM SAILORS S`

- **In general, requires scanning the relation maintaining running info on aggregating col.**
  - What is the cost for this query? O(N)

- **Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.**
Aggregates with grouping

- **Example**
  - SELECT S.rating, AVE(S.age)
    FROM SAILORS S
    GROUP BY S.rating

- **Algorithm based on sorting for grouping (partitioning)**
  - Assuming 2-pass sort, Cost: Cost for Sort S + Cost for Scan S for aggregation
  - Refined: If combining aggregation in sorting phase: Cost for sorting S

- **Algorithm based on hashing for grouping (partitioning)**
  - Required enough memory to fit hash table is needed
  - Cost : Cost of scanning S : O(N) for building hash table for partitioning (grouping) and maintaining running aggregating info per group

- **Conclusion**: Hash is best

- If there is an appropriate index, use index-only alg.