Data Modeling and Databases

Ch 9: Query Processing - Algorithms

Gustavo Alonso
Systems Group
Department of Computer Science
ETH Zürich
Application

SQL

Server (Network, App Buffers, Security)

Query Processor

Data Manager (Indexes, Records)

Storage Manager (Pages)

Transactions (Locking, Logging)

Metadata Mgmt (Schema, Stats)

get/put

Storage System (disks, SSDs, SAN, ...)

{tuples}

{tuples}

block
Query Processor

SQL

Parser

Rewrite

Optimizer

CodeGen

{tuples}

Runtime System

Plan

Interpreter

Query Graph Model

QGM

QGM++
SQL -> Relational Algebra

Relational Algebra

\[ \Pi_{A_1, \ldots, A_n} \left( \sigma_P \left( R_1 \times \ldots \times R_k \right) \right) \]

Diagram:
- \( \sigma_P \)
- \( R_k \)
- \( R_3 \)
- \( R_2 \)
- \( R_1 \)
Runtime System

- Three approaches
  - A. Compile query into machine code
  - B. Compile query into relational algebra and interpret that
  - C. Hybrid: e.g., compile predicates into machine code

- What to do?
  - A: better performance
  - B: easier debugging, better portability
  - Today: Move towards compilation

- Query Interpreter
  - provide implementation for each algebra operator
  - define interface between operators
Algorithms for Rel. Algebra

- **Table Access**
  - scan (load each page at a time)
  - index scan (if index available)

- **Sorting**
  - Two-phase external sorting

- **Joins**
  - (Block) nested-loops
  - Index nested-loops
  - Sort-Merge
  - Hashing (many variants)

- **Group-by (~ self-join)**
  - Sorting
  - Hashing
Basic concepts

- **Query Selectivity**: number of tuples in the result versus number of tuples in the table (or the input)

- **Attribute Cardinality**: how many distinct values are there for the attribute
  - Gender: male/female = 2
  - Semester: [1 .. 12] = 12
  - Departments: (at ETHZ, 17)
  - Cities: hundreds to thousands
  - Streets: ...
  - Primary key = N, N = number of tuples in the table

- **Skew**: probability distribution of the values an attribute takes
Scans

- The most basic of all operations
  - Iterate over every tuple in a table
  - Match tuple against predicate

- Sounds expensive but ...
  - Selectivity
  - I/O
  - Block/page sizes

- Performance of scans is predictable
Examples:

Q1: SELECT count(Legi) as Number_of_Students
    FROM Student

Q2: SELECT Name
    FROM Student
    WHERE Legi = 26120

<table>
<thead>
<tr>
<th>Legi</th>
<th>Name</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>24002</td>
<td>Gerber</td>
<td>18</td>
</tr>
<tr>
<td>25403</td>
<td>Zollinger</td>
<td>12</td>
</tr>
<tr>
<td>26120</td>
<td>Frey</td>
<td>10</td>
</tr>
<tr>
<td>26830</td>
<td>Küng</td>
<td>8</td>
</tr>
<tr>
<td>27550</td>
<td>Fehr</td>
<td>6</td>
</tr>
<tr>
<td>28106</td>
<td>Lustenberger</td>
<td>3</td>
</tr>
<tr>
<td>29120</td>
<td>Schweizer</td>
<td>2</td>
</tr>
<tr>
<td>29555</td>
<td>Meier</td>
<td>2</td>
</tr>
</tbody>
</table>
Indexes

- The way to avoid a full table scan is to build an index over some attributes
What if we want the tuples to be ordered? – (B-Tree, if lucky)
Tips on indexes

- Where do you want an index:
  - On primary keys
  - On foreign keys
  - On attributes with high cardinality
  - Attributes frequently used in joins or as conditions
  - ...

- Warning: indexes do not come for free
  - Space
  - Maintenance
  - Not very useful for low selectivity operations
Selectivity and cardinality

- The selectivity of an index over an attribute can be calculated based on the cardinality of the attribute:
  - Table students, 20000 records
  - Index over gender (male/female)
    - 2/20000*100% = 0.01 %
  - Index over department (17 possibilities)
    - 17/20000*100% = 0.085 %
  - Index over Legi
    - 20000/20000*100% = 100 %
Examples of no free lunch

Q1: SELECT count(Legi) as Number_of_Students
    FROM Student
    WHERE Semester > 3

Q2: SELECT Name
    FROM Student
    WHERE Legi < 24000 and Legi > 27000

If I were a query optimizer
Index range scan

- If an index is available, many queries can be answered by scanning the index:
  - Range queries
  - Exist or Not Exist

- Clustered index:
  - The leaf nodes in the index contain the data instead of a pointer
  - Can only be one per table
  - Index access produces results directly

- Index seek vs index scan
On modern architectures

- Sequential access is faster than random access
  - Was (and is) true for disks
  - Also true for memory (because of caches)
- Fine trade-off between doing more reads but sequentially and doing less reads but random
  - Hardware and software prefetching
  - Hardware parallelism (SIMD, AVX) speeds up sequential scanning
Other forms of indexing

- Partitioning: dividing a table into smaller chunks
  - For parallel access (in disks)
  - For better fit in caches
  - To increase concurrency
  - To split hot-spots
  - ...

- Hash partitioning
- Range partitioning
- Interleaved range partitioning
Hash Index

Example of Hash Index

©Silberschatz, Korth and Sudarshan

D-INFK, ETH Zurich, Data Modeling and Databases
A glimpse into research

- Recent trend (in research):
  - Predictability (robustness) more important than raw performance
  - Indexes give you performance if:
    - We have the index
    - Queries hit the index
    - High selectivity
  - Maybe we are better off operating without indexes and running just scans?

- Ask me if you are interested ...
Recall that tuples in table are not sorted

Sorting is important:
- As a result (query requests results to be sorted)
- As an intermediate step
  - For joins (sort-merge join)
  - For group-by
  - For max, min, etc.

Sorting is expensive:
- In CPU (comparisons)
- In space (sorting not done in place with respect to the table)
External sort

- Why external sort?
  - Obvious: data does not fit in main memory (data and results!!)
  - Less obvious: many queries running at the same time sharing memory

- Two key parameters
  - N: number of pages of input
  - M: size of in memory buffer

- Behavior of algorithm determined by many parameters: I/O, CPU, I/O costs, caches, etc.
Two-phase External Sorting

- N size of the input in pages, M size of the buffer in pages

- Phase I: Create Runs
  1. Load allocated buffer space with tuples
  2. Sort tuples in buffer pool
  3. Write sorted tuples (run) to disk
  4. Goto Step 1 (create next run) until all tuples processed

- Phase II: Merge Runs
  - Use priority heap to merge tuples from runs

- Special cases
  - M >= N: no merge needed
  - M < sqrt(N): multiple merge phases necessary
External Sort

97
17
3
5
27
16
2
99
13
External Sort

load

97
17
3
5
27
16
2
99
13

97
17
3
External Sort

97
17
3
5
27
16
2
99
13

sort

3
17
97
External Sort

97
17
3
5
27
16
2
99
13

write

3
17
97

run
External Sort

load

97
17
3
5
27
16
2
99
13

3
17
97
External Sort

sort & write

97
17
3
5
27
16
2
99
13

5
16
27

3
17
97
5
16
27
External Sort
External Sort

End of Phase 1
External Sort

merge

3
5
2

3
17
97

5
16
27

2
13
99
External Sort

2

merge

3
5
2

3
17
97
5
16
27
2
13
99
External Sort

2
3

merge

3
5
13

3
17
97

5
16
27

2
13
99
External Sort

merge

2 3 5

17 5 13

3 17 97 5 16 27 2 13 99
External Sort

merge

2
3
5

17
16
13

3
17
97
5
16
27
2
13
99
External Sort

2
3
5
13

17
16
13

3
17
97

5
16
27

2
13
99
One pass vs. multi-pass sort

- Previous algorithm is a one-pass algorithm (every data item is read once and written once)

- However:
  - If there are many runs, I/O overhead is too high (we need to bring too many runs to memory)
  - Merge step cannot be parallelized
Multi-way Merge (N = 7; M = 2)

input file

65  43  47  89  52  13  8

1-page runs

56  34  47  89  25  13  8

2-page runs

34  47  12  8

56  89  35

4-page runs

34  12

45  35

67  8

89

7-page run

1  2  3  3  4  4  5  5  6  7  8  8  9

Pass 0

Pass 1

Pass 2

Pass 3
Analysis

- N: size of table in pages
- M: size of (available) main memory in pages
- IO Cost
  - $O(N)$: if $M \geq \sqrt{N}$
    - $2 \times N$: if $M = N$
    - $4 \times N$: if $N > M \geq \sqrt{N}$
  - $O(N \log_M N)$: if $M < \sqrt{N}$
    - Base of logarithm: in $O$ notation not relevant, but constants matter
- CPU Cost ($M \geq \sqrt{N}$)
  - Phase 1 (create $N/M$ runs of length $M$): $O(N \times \log_2 M)$
  - Phase 2 (merge $N$ tuples): $O(N \times \log_2 N/M)$
Sorting Summary

- Complexity: $N \times \log(N)$ theory is right, but
  - DB people care about CPU and IO complexity
  - Constants matter!
  - Buffer allocation matters! Many concurrent queries?
  - More main memory can hurt performance!
- Main memory is large. Do two-way sort because...
  - Parallelize sorting on different machines/cores
  - Or many concurrent sorts on same machine
  - But more than 2-ways very rare in practice
- Huge amount of research in sorting with many optimizations (cache locality, favoring sequential access, hardware optimizations, SIMD/AVX, etc.)
Joins

- One of the most common operations
- Can be very expensive (→ views, materialized)
- Performance affected by:
  - Actual join algorithm
  - Relative table sizes
  - Number of tables to join
  - Order in which joins are performed
  - Selectivity
  - Predicates in the query
  - Memory hierarchy
  - Indexes
Nested loop join

- Actually, two nested scans
- While there are tuples in R
  - Get a tuple from R
  - Compare with all tuples in S (scan S for matches)
  - Output if match
- Complexity is $O(|R| \times |S|)$, i.e., $O(N^2)$
- Sounds expensive but still used in practice (makes sense if, e.g., S is sorted, join is on an index attribute)
- Optimization: block nested loop join (get several tuples from R in one go, hash and then compare with S) = less scans of S
Sort Merge Join
Canonical Hash Join

1. Build phase

2. Probe phase

.hash table

k

match

hash(key)

R

bucket 0

bucket 1

bucket n-1

S

Complexity: $O(|R| + |S|)$, i.e., $O(N)$

Easy to parallelize
(Grace) Hash Join

Function: hash_join \( (R, S, \alpha = \beta) \)

1. foreach record \( r \in R \) do
   2. append \( r \) to partition \( R_{h(r.\alpha)} \)

3. foreach record \( s \in S \) do
   4. append \( s \) to partition \( S_{h(s.\beta)} \)

6. foreach partition \( i \in 1, \ldots, n \) do
   7. build hash table \( H \) for \( R_i \), using hash function \( h' \)
   8. foreach block in \( S_i \) do
      9. foreach record \( s \) in current \( S_i \)-block do
         10. probe \( H \) and append matching tuples to result ;
Grace Hash Join

Relation $R$ \( h \)

Relation $S$ \( h \)

Partition 1 ($R_1$ and $S_1$)

Partition 2 ($R_2$ and $S_2$)

Partition 3 ($R_3$ and $S_3$)

\vdots

Partition $n$ ($R_n$ and $S_n$)

\[ R_i \Join S_j = \emptyset \text{ for all } i \neq j \]
**Partitioned Hash Join (Shatdal et al. 1994)**

- **Idea:** Partition input into disjoint chunks of cache size
  - No more cache misses during the join

---

**Problem:** $p$ can be too large!

"Cache conscious algorithms for relational query processing", Shatdal et al, VLDB ‘94
Problem: Hardware limits fan-out, i.e. $T = \#\text{TLB-entries}$ (typically $64\text{-}512$)

Solution: Do the partitioning in multiple passes!

**TLB & Cache** efficiency compensates multiple read/write passes

Parallel Radix Join
Parallelizing the Partitioning: Pass - 1

Local Histograms

Global Histogram
& Prefix Sum

Each thread scatters out its tuples based on the prefix sum

Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs, VLDB ‘09
Parallel Radix Join
Parallelizing the Partitioning: *Pass - (2 .. i)*

Each thread individually partitions sub-relations from pass-1

**Thread-2**

**Thread-4**

**Thread-N**
Sorting vs. Hashing

- Both techniques can be used for joins, group-by, ... 
  - binary and unary matching problems

- Same asymptotic complexity: $O(N \log N)$
  - In both IO and CPU
  - Hashing has lower constants for CPU complexity
  - IO behavior is almost identical

- Merging (Sort) vs. Partitioning (Hash)
  - Merging done *afterwards*; Partitioning done *before*
  - Partitioning depends on good statistics to get right

- Sorting more robust. Hashing better in average case!
Group-by

- Several options:
  - Hash on the group-by attribute, aggregate on hash collisions
  - Sort on the group-by attribute, then aggregate the sorted ranges
  - Choice depend on several factors (like existence of an index)

- Today: complex algorithms to exploit parallelism in multicore (like radix joins)