Query Processing: A Systems View

CPS 216
Advanced Database Systems

Announcements (March 1)

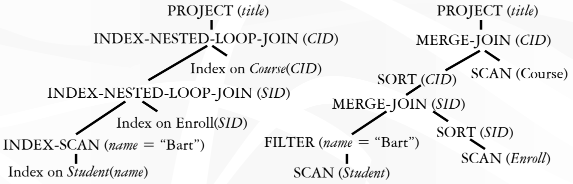
- Reading assignment due Wednesday
  - Buffer management
- Homework #2 due this Thursday
- Course project proposal due in one week
- Midterm next Thursday in class
  - Open book, open notes

Physical (execution) plan

- A complex query may involve multiple tables and various query processing processing algorithms
  - E.g., table scan, index nested-loop join, sort-merge join, hash-based duplicate elimination…
- A physical plan for a query tells the DBMS query processor how to execute the query
  - A tree of physical plan operators
  - Each operator implements a query processing algorithm
  - Each operator accepts a number of input tables/streams and produces a single output table/stream

Examples of physical plans

```
SELECT Course.title
FROM Student, Enroll, Course
WHERE Student.name = 'Bart'
AND Student.SID = Enroll.SID AND Enroll.CID = Course.CID;
```



PROJECT (*title*)
INDEX-NESTED-LOOP-JOIN (*CID*)
Index on *Course*(*CID*)
INDEX-NESTED-LOOP-JOIN (*SID*)
Index on Enroll(*SID*)
INDEX-SCAN (*name* = "Bart")
Index on *Student*(*name*)

PROJECT (*title*)
MERGE-JOIN (*CID*)
SORT (*CID*)
SCAN (Course)
MERGE-JOIN (*SID*)
FILTER (*name* = "Bart")
SORT (*SID*)
SCAN (*Student*)
SCAN (*Enroll*)

- ❖ Many physical plans for a single query
  - Equivalent results, but different costs and assumptions!
  - ☞DBMS query optimizer picks the "best" possible physical plan

Physical plan execution

- ❖ How are intermediate results passed from child operators to parent operators?
  - Temporary files
    - Compute the tree bottom-up
    - Children write intermediate results to temporary files
    - Parents read temporary files
  - Iterators
    - Do not materialize intermediate results
    - Children pipeline their results to parents

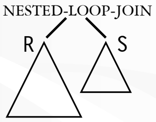
Iterator interface

- ❖ Every physical operator maintains its own execution state and implements the following methods:
  - `open()`: Initialize state and get ready for processing
  - `getNext()`: Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  - `close()`: Clean up

An iterator for table scan

- `open()`
  - Allocate a block of memory
- `getNext()`
  - If no block of *R* has been read yet, read the first block from the disk and return the first tuple in the block (or the null pointer if *R* is empty)
  - If there is no more tuple left in the current block, read the next block of *R* from the disk and return the first tuple in the block (or the null pointer if there are no more blocks in *R*)
  - Otherwise, return the next tuple in the memory block
- `close()`
  - Deallocate the block of memory

An iterator for nested-loop join



R: An iterator for the left subtree

S: An iterator for the right subtree

- `open()`

```
R.open(); S.open(); r = R.getNext();
```

- `getNext()`

```
do {
  s = S.getNext();
  if (s == null) {
    S.close(); S.open(); s = S.getNext(); if (s == null) return null;
    r = R.getNext(); if (r == null) return null;
  }
} until (r joins with s);
return rs;
```

- `close()`

```
R.close(); S.close();
```

An iterator for 2-pass merge sort

- `open()`
  - Allocate a number of memory blocks for sorting
  - Call `open()` on child iterator
- `getNext()`
  - If called for the first time
    - Call `getNext()` on child to fill all blocks, sort the tuples, and output a run
    - Repeat until `getNext()` on child returns null
    - Read one block from each run into memory, and initialize pointers to point to the beginning tuple of each block
  - Return the smallest tuple and advance the corresponding pointer; if a block is exhausted bring in the next block in the same run
- `close()`
  - Call `close()` on child
  - Deallocate sorting memory and delete temporary runs

Blocking vs. non-blocking iterators

- A blocking iterator must call `getNext()` exhaustively (or nearly exhaustively) on its children before returning its first output tuple
  - Examples:
- A non-blocking iterator expects to make only a few `getNext()` calls on its children before returning its first (or next) output tuple
  - Examples:

Execution of an iterator tree

- Call `root.open()`
- Call `root.getNext()` repeatedly until it returns null
- Call `root.close()`

☞ Requests go down the tree

☞ Intermediate result tuples go up the tree

☞ No intermediate files are needed

Memory management for DBMS

- DBMS operations require main memory
  - While data resides on disk, it is manipulated in memory
  - Sometimes the more memory the better, e.g., sort
- One approach: let each operation pre-allocate some amount of "private" memory and manage it explicitly
  - Not very flexible
  - Limits sharing and reuse
- Alternative approach: use a buffer manager
  - Responsible for reading/writing data blocks from/to disk as needed
  - Higher-level code can be written without worrying about whether data is in memory or not

Buffer manager basics

- Buffer pool: a global pool of frames (main-memory blocks)
  - ☞Some systems use separate pools for different objects (e.g., tables and indexes) and for different operations (e.g., sorting and others)
- Higher-level code can pin and unpin a frame
  - Pin: I need to work on this frame in memory
  - Unpin: I no longer need this frame
  - A completely unpinned frame is a candidate for replacement
  - ☞In some systems you can hate a frame (i.e., suggesting it for replacement)
- A frame becomes dirty when it is modified
  - Only dirty frames need to be written back to disk
  - ☞Related to transaction processing

Standard OS replacement policies

- Example
  - Current buffer pool: 0, 1, 2
  - Past requests: 0, 1, 2
  - Incoming requests: 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, …
  - ☞Which frame to replace?
- Optimal: replace the frame that will not be used for the longest time (2)
- Random (0, 1, or 2 with equal probability)
- LRU: least recently used (0)
- LRU approximation: clock, aging
- MRU: most recently used (2)

Problems with OS buffer management

Stonebraker. "Operating System Support for Database Management." *CACM*, 1981.

- Performance problems
  - Getting a page from the OS to user space is usually a system call (process switch) and copy
- Replacement policy
  - LRU, clock, etc. often ineffective
  - DBMS knows access pattern in advance and therefore should dictate policy → major OS/DBMS distinction
- Prefetch policy
  - DBMS knows of multiple "orders" for a set of records; OS only knows physical order
- Crash recovery
  - DBMS needs more control

Next

Chou and DeWitt. “An Evaluation of Buffer Management Strategies for Relational Database Systems.” *VLDB* 1985.

- ❖ Old algorithms
  - Domain separation algorithm
  - “New” algorithm
  - Hot set algorithm
- ❖ Query locality set model
- ❖ DBMIN algorithm

Domain separation algorithm

- ❖ Split work/memory into domains; LRU within each domain; borrow from other domains when out of frames
  - Example: one domain for each level of the B^+-tree
- ❖ Limitations
  - Assignment of pages to domains is static, and ignores how pages are used
    - Example: A data page is accessed only once in a scan, but the same data page is accessed many times in a NLJ
  - Does not differentiate relative importance between types of pages
    - Example: An index page is more important than a data page
  - Memory allocation is based on data rather queries → need orthogonal load control to prevent thrashing

---

The “new” algorithm

- ☞ Observations based on the reference patterns of queries
  - Priority is not a property of a data page, but of a relation
  - Each relation needs a “working set”
- ❖ Divide buffer pool into chunks, one per relation
- ❖ Prioritize relations according to how often their pages are reused
- ❖ Replace a frame from the least reused relation and add it to the chunk of the referenced relation
- ❖ Each active relation is guaranteed with one frame
- ❖ MRU within each chunk (seems arbitrary)
- ❖ Simulations look good; implementation did not beat LRU

Hot set algorithm

☞ Exploit query behavior more!

- A set of pages that are accessed over and over form a hot set
  - "Hot points" in the graph of buffer size vs. number of page faults
  - Example: For nested-loop join $R \bowtie S$, size of hot set is $B(S) + 1$ (under LRU)
- Each query is given enough memory for its hot set
- Admission control: Do not let a query into the system unless its hot set fits in memory
- Replacement: LRU within each hot set (seems arbitrary)
- Derivation of hot set assumes LRU, which may be suboptimal
  - Example: What is better for nested-loop join?

Query locality set model

- Observations
  - DBMS supports a limited set of operations
  - Reference patterns are regular and predictable
  - Reference patterns can be decomposed into simple patterns
- Reference pattern classification
  - Sequential
  - Random
  - Hierarchical

Sequential reference patterns

- Straight sequential: read something sequentially once
  - Example: selection on unordered table
  - ☞Each page is only touched once, so just buffer one page
- Clustered sequential: repeatedly read a "chunk" sequentially
  - Example: merge join; rows with the same join column value are scanned multiple times
  - ☞Keep all pages in the chunk in buffer
- Looping sequential: repeatedly read something sequentially
  - Example: nested-loop join
  - ☞Keep as many pages as possible in buffer, with MRU replacement

Random reference patterns

- ❖ Independent random: truly random accesses
  - ▪ Example: index scan through a non-clustered (e.g., secondary) index yields random data page access
  - ☞The larger the buffer the better?
- ❖ Clustered random: random accesses that happen to demonstrate some locality
  - ▪ Example: in an index nested-loop join, inner index is non-clustered and non-unique, while outer table is clustered and non-unique
  - ☞Try to keep in buffer data pages of the inner table accessed in one cluster

---

Hierarchical reference patterns

- ❖ Example: operations on tree indexes
- ❖ Straight hierarchical: regular root-to-leaf traversal
- ❖ Hierarchical with straight sequential: traversal followed by straight sequential on leaves
- ❖ Hierarchical with clustered sequential: traversal followed by clustered sequential on leaves
- ❖ Looping hierarchical: repeatedly traverse an index
  - ▪ Example: index nested-loop join
  - ☞Keep the root index page in buffer

DBMIN algorithm

- ❖ Associate a chunk of memory with each file instance (each table in `FROM`)
  - ▪ This chunk is called the file instance's locality set
  - ▪ Instances of the same table may share buffered pages
  - ▪ But each locality set has its own replacement policy
    - ☞Based on how query processing uses each relation (finally!)
    - ☞No single policy for all pages accessed by a query
    - ☞No single policy for all pages in a table
- ❖ Estimate locality set sizes by examining the query plan and database statistics
- ❖ Admission control: a query is allowed to run if its locality sets fit in free frames

DBMIN algorithm (cont’d)

- Locality sets: each “owns” a set of pages, up to a limit *l*
- Global free list: set of “orphan” pages
- Global table: allow sharing among concurrent queries
- Query *q* requests page *p*
  - If *p* is in memory and in *q*’s locality set
    - Just update usage statistics of *p*
  - If *p* is in memory and in some other query’s locality set
    - Just make *p* available to *q*; no further action is required
  - If *p* is in memory and in the global free list
    - Add *p* to *q*’s locality set; if *q*’s locality set exceeds its size limit, replace a page (release it back to the global free list)
  - If *p* is not in memory
    - Use a page from global free list to get *p* in; proceed as in the previous case

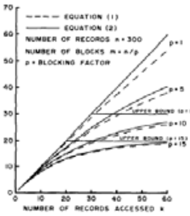
---

Locality sets for various ref. patterns

- Straight sequential
  - Size = 1
  - Just replace as needed
- Clustered sequential
  - Size = number of pages in the largest cluster
  - FIFO or LRU (assuming large enough size)
- Looping sequential
  - Size = number of pages in the table
  - MRU

Locality sets for more ref. patterns

- Independent random
  - Size = 1 (if odds of revisit is low), or *b* (expected number of block accessed by a given number *k* of random record accesses; Yao, 1977)
    - Use $(k - b) / b$ to choose between 1 and *b*
  - Replacement policy does not matter
- Clustered random
  - Size = number of blocks in the largest cluster (≈ number of tuples because of random access, or use Yao’s formula)
  - LRU or FIFO



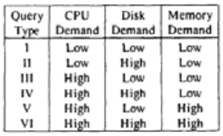
Locality sets for more ref. patterns

- Straight hierarchical, hierarchical/straight sequential: just like straight sequential
  - Size = 1
  - Just replace as needed
- Hierarchical/clustered sequential: like clustered sequential
  - Size = number of index pages in the largest cluster
  - FIFO or LRU
- Looping hierarchical
  - At each level of the index you have random access among pages
  - Use Yao's formula to figure out how many pages need to be accessed at each level
  - Size = sum over all levels that you choose to worry about
  - LIFO with 3-4 buffers should be okay

Simulation study

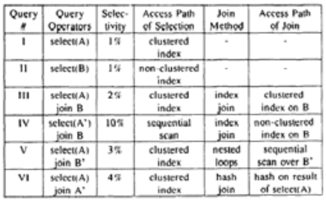
- Hybrid simulation model
  - Trace-driven simulation
    - Recorded from a real system (running Wisconsin Benchmark)
    - For each query, record its execution trace
      - Page read/write, file open/close, etc.
  - Distribution-driven simulation
    - Generated by some stochastic model
    - Synthesize the workload by merging query execution traces
- Simulator models CPU, memory, and one disk
- Performance metric: query throughput

Workload



Query Type	CPU Demand	Disk Demand	Memory Demand
I	Low	Low	Low
II	Low	High	Low
III	High	Low	Low
IV	High	High	Low
V	High	Low	High
VI	High	High	High

Query Classification



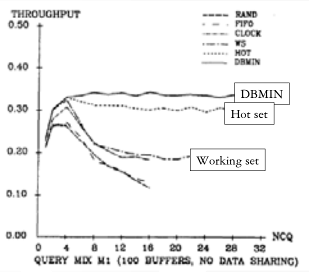
Query #	Query Operators	Selec-tivity	Access Path of Selection	Join Method	Access Path of Join
I	select(A)	1%	clustered index	-	-
II	select(B)	1%	non-clustered index	-	-
III	select(A) join B	2%	clustered index	index join	clustered index on B
IV	select(A') join B	10%	sequential scan	index join	non-clustered index on B
V	select(A) join B'	3%	clustered index	nested loops	sequential scan over B'
VI	select(A) join A'	4%	clustered index	hash join	hash on result of select(A)

A,B:10K tuples; A':1K tuples; B':300 tuples; 182 bytes per tuple.

Description of Base Queries

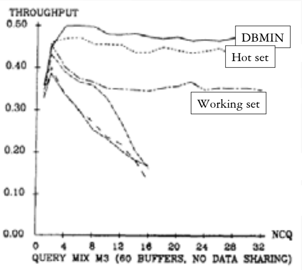
- Mix 1: all six types equally likely
- Mix 2: I and II together appear 50% of the time
- Mix 3: I and II together appear 75% of the time

Mix 1 (no data sharing)



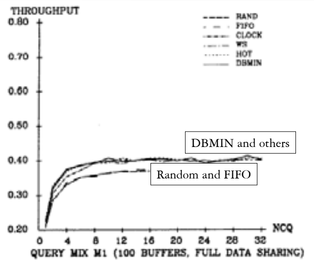
- ❖ Thrashing is evident for simple algorithms with no load control
- ❖ Working set (a popular OS choice) fails to capture join loops for queries with high memory demand (types V and VI)
  - It still functions (though suboptimally) with large number of current queries (NCQ)

Mix 3 (no data sharing)



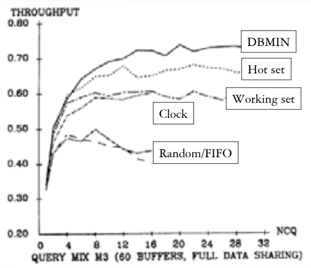
- ❖ Thrashing is still evident
- ❖ Working set fares better because mix 3 has more simple queries and fewer ones of types V and VI

Mix 1 (full data sharing)



- ❖ With full data sharing, locality is easier to capture
  - Performance improves across the board and the gap disappears
  - Random and FIFO do not capture locality as effectively as others

Mix 3 (full data sharing)

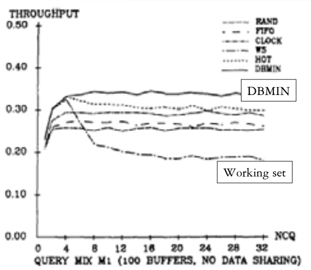


- ❖ Performance starts to diverge again
  - Mix 3 is dominated by lots of small queries, and locality becomes harder to capture

Feedback load control

- ❖ Mechanism to check resource usage in order to prevent system from overloading
- ❖ Rule of thumb: “50% rule”—keep the paging device busy half of the time
- ❖ Implementation
  - Estimator measures the utilization of device
  - Optimizer analyzes measurements and decides whether/what load adjustment is appropriate
  - Control switch activates/deactivates processes according to optimizer’s decisions

Mix 1 (load control, no data sharing)



- ❖ DBMIN still the best
- ❖ (Simple algorithms + load control) outperforms working set!
- ❖ Cons of feedback load control
  - Runtime overhead
  - Non-predictive
    - Only responds after undesirable condition occurs

Conclusion

- Same basic access patterns come up again and again in query processing
- Make buffer manager aware of these access patterns

☞ Look at the workload, not just the content

  - Contents can at best offer guesses at likely workloads