Physical Data Organization and Indexing
Introduction to Databases
CompSci 316 Fall 2017

Announcements (Wed., Mar. 8)
• Homework #3
  • follow piazza posts for updates after each lecture
• Project
  • Comments on milestone-1 tomorrow
  • Keep working on it

Today:
• Finish B+ tree and index
• Start query processing

Index

Recall: What are indexes for?
• Given a value (search key), locate the record(s) with this value, or range search
  • SELECT * FROM R WHERE A = value;
  • SELECT * FROM R, S WHERE R.A = S.B;
  • SELECT * FROM R WHERE A > value;

• Search key ≠ key in a relation (unique attributes)
  • “Key” is highly overloaded in databases

• Recap: index structure on whiteboard

Recall: Index classification
• Dense vs. Sparse
• Clustered vs. unclustered
• Primary vs. Secondary
• Tree-based vs. Hash-based
  • we will only do tree indexes in 316
Recall: B*-tree

- A hierarchy of nodes with intervals
- Balanced (more or less): good performance guarantee
- Disk-based: one node per block; large fan-out

Recall: Sample B*-tree nodes

Max fan-out: 4

Recall: B*-tree balancing properties

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

<table>
<thead>
<tr>
<th>Max # pointers</th>
<th>Max # keys</th>
<th>Min # active pointers</th>
<th>Min # keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-leaf</td>
<td>( f )</td>
<td>( f - 1 )</td>
<td>( \lceil f/2 \rceil - 1 )</td>
</tr>
<tr>
<td>Root</td>
<td>( f )</td>
<td>( f - 1 )</td>
<td>2</td>
</tr>
<tr>
<td>Leaf</td>
<td>( f )</td>
<td>( f - 1 )</td>
<td>( \lceil f/2 \rceil )</td>
</tr>
</tbody>
</table>

Lookups

- SELECT * FROM \( R \) WHERE \( k = 179 \);
- SELECT * FROM \( R \) WHERE \( k = 32 \);

Range query

- SELECT * FROM \( R \) WHERE \( k > 32 \) AND \( k < 179 \);

Insertion

- Insert a record with search key value 32

End of lecture 14
Another insertion example

• Insert a record with search key value 152

Node splitting

Oops, that node becomes full!

Oops, node is already full!

Max fan-out: 4

More node splitting

In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)

• Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level

Deletion

• Delete a record with search key value 130

Stealing from a sibling

Remember to fix the key in the least common ancestor of the affected nodes

Another deletion example

• Delete a record with search key value 179

Cannot steal from siblings
Then coalesce (merge) with a sibling!
Coalescing

- Deletion can “propagate” all the way up to the root of the tree (not illustrated here)
- When the root becomes empty, the tree “shrinks” by one level

Performance analysis

- How many I/O’s are required for each operation?
  - \( h \), the height of the tree (more or less)
  - Plus one or two to manipulate actual records
  - Plus \( O(h) \) for reorganization (rare if \( f \) is large)
  - Minus one if we cache the root in memory
- How big is \( h \)?
  - Roughly \( \log_{56789} N \), where \( N \) is the number of records
  - \( B^+\)-tree properties guarantee that fan-out is least \( f/2 \) for all non-root nodes
  - Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
- A 4-level \( B^+\)-tree is enough for “typical” tables

B^+-tree in practice

- Complex reorganization for deletion often is not implemented (e.g., Oracle)
  - Leave nodes less than half full and periodically reorganize
- Most commercial DBMS use \( B^+\)-tree instead of hashing-based indexes because \( B^+\)-tree handles range queries

The Halloween Problem

- Story from the early days of System R...
  UPDATE Payroll
  SET salary = salary * 1.1
  WHERE salary <= 25000;
- There is a \( B^+\)-tree index on Payroll(salary)
- The update never stopped until all employees earned 25k (why?)
- Solutions?

B^+-tree versus ISAM

- ISAM is more static; \( B^+\)-tree is more dynamic
- ISAM can be more compact (at least initially)
  - Fewer levels and I/O’s than \( B^+\)-tree
- Overtime, ISAM may not be balanced
  - Cannot provide guaranteed performance as \( B^+\)-tree does

B^+-tree versus B-tree

- B-tree: why not store records (or record pointers) in non-leaf nodes?
  - These records can be accessed with fewer I/O’s
- Problems?
  - Storing more data in a node decreases fan-out and increases \( h \)
  - Records in leaves require more I/O’s to access
  - Vast majority of the records live in leaves!
B+ tree vs. Hash-based indexes

- Extensible hashing, linear hashing, etc.
- Can only handle "=" in join or selection
- Cannot handle range predicates >, ≥, <, ≤

Beyond ISAM, B-, and B+-trees, and hash

- Other tree-based indexes: R-trees and variants, GiST, etc.
  - How about binary tree?

- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.

Overview

- Many different ways of processing the same query
  - Scan? Sort? Hash? Use an index?
  - All have different performance characteristics and/or make different assumptions about data
- Best choice depends on the situation
  - Implement all alternatives
  - Let the query optimizer choose at run-time

Notation

- Relations: $R, S$
- Tuples: $r, s$
- Number of tuples: $|R|, |S|
- Number of disk blocks: $B(R), B(S)$
- Number of memory blocks available: $M$
- Cost metric
  - Number of I/O’s
  - Memory requirement

Scanning-based algorithms
Table scan

• Scan table \( R \) and process the query
  • Selection over \( R \)
  • Projection of \( R \) without duplicate elimination
• I/O’s: \( B(R) \)
  • Trick for selection: stop early if it is a lookup by key
• Memory requirement: 2
• Not counting the cost of writing the result out
  • Same for any algorithm!
  • Maybe not needed—results may be pipelined into another operator

Nested-loop join

\[ R \bowtie_p S \]

• For each block of \( R \), and for each \( r \) in the block:
  • For each block of \( S \), and for each \( s \) in the block:
  • Output \( rs \) if \( p \) evaluates to true over \( r \) and \( s \)
• I/O’s: \( B(R) + |R| \cdot B(S) \)
• Memory requirement: 3

Improvement: block-based nested-loop join

• For each block of \( R \), for each block of \( S \):
  • Output \( rs \) if \( p \) evaluates to true over \( r \) and \( s \)
• I/O’s: \( B(R) + B(R) \cdot B(S) \)
• Memory requirement: same as before

More improvements

• Stop early if the key of the inner table is being matched
• Make use of available memory
  • Stuff memory with as much of \( R \) as possible, stream \( S \) by, and join every \( S \) tuple with all \( R \) tuples in memory
• I/O’s: \( B(R) + \frac{|R|}{M} \cdot B(S) \)
  • Or, roughly \( B(R) \cdot B(S)/M \)
• Memory requirement: \( M \) (as much as possible)
• Which table would you pick as the outer?

Sorting-based algorithms

http://en.wikipedia.org/wiki/Mail_sorter#mediaviewer/File:Mail_sorting,1951.jpg

External merge sort

Remember (internal-memory) merge sort?
Problem: sort \( R \), but \( R \) does not fit in memory
• Pass 0: read \( M \) blocks of \( R \) at a time, sort them, and write out a level-0 run
• Pass 1: merge \( (M - 1) \) level-0 runs at a time, and write out a level-1 run
• Pass 2: merge \( (M - 1) \) level-1 runs at a time, and write out a level-2 run
  ...
• Final pass produces one sorted run

Toy example

• 3 memory blocks available; each holds one number
• Input: 1, 7, 4, 5, 2, 8, 3, 6, 9
• Pass 0
  • 1, 7, 4 \rightarrow 1, 4, 7
  • 5, 2, 8 \rightarrow 2, 5, 8
  • 9, 6, 3 \rightarrow 3, 6, 9
• Pass 1
  • 1, 4, 7 + 2, 5, 8 \rightarrow 1, 2, 4, 5, 7, 8
  • 3, 6, 9
• Pass 2 (final)
  • 1, 2, 4, 5, 7, 8 + 3, 6, 9 \rightarrow 1, 2, 3, 4, 5, 6, 7, 8, 9
Analysis

- **Pass 0**: read $M$ blocks of $R$ at a time, sort them, and write out a level-0 run
  - There are $\frac{K}{M}$ level-0 sorted runs
- **Pass $i$**: merge $(M - 1)$ level-$(i - 1)$ runs at a time, and write out a level-$i$ run
  - $(M - 1)$ memory blocks for input, 1 to buffer output
  - # of level-$i$ runs = \[ \frac{\text{# of level-} (i-1) \text{ runs}}{M-1} \]
- Final pass produces one sorted run

Performance of external merge sort

- Number of passes: \( \log_{\frac{M-1}{M}} \left( \frac{K}{M} \right) + 1 \)
- I/O’s
  - Multiply by \( 2 \cdot B(R) \): each pass reads the entire relation once and writes it once
  - Subtract \( B(R) \) for the final pass
  - Roughly, this is \( O(B(R) \times \log_{\frac{M-1}{M}}(B(R)) \)
- Memory requirement: \( M \) (as much as possible)

Some tricks for sorting

- Double buffering
  - Allocate an additional block for each run
  - Overlap I/O with processing
  - Trade-off: smaller fan-in (more passes)
- Blocked I/O
  - Instead of reading/writing one disk block at a time, read/write a bunch (“cluster”)
  - More sequential I/O’s
  - Trade-off: larger cluster \( \rightarrow \) smaller fan-in (more passes)

Example of merge join

\[
\begin{align*}
R: & \quad s: \quad R \bowtie_{R.A=S.B} S: \\
 & r_1. A = 1 & s_1. B = 1 & \quad r_1s_1 \\
 & r_2. A = 3 & s_2. B = 2 & \quad r_2s_3 \\
 & r_3. A = 3 & s_3. B = 3 & \quad r_2s_4 \\
 & r_4. A = 5 & s_4. B = 3 & \quad r_3s_3 \\
 & r_5. A = 7 & s_5. B = 8 & \quad r_3s_4 \\
 & r_6. A = 7 & & \quad r_4s_5 \\
 & r_7. A = 8 & &
\end{align*}
\]

Sort-merge join

\[ R \bowtie_{R.A=S.B} S \]

- Sort $R$ and $S$ by their join attributes; then merge $r, s$ = the first tuples in sorted $R$ and $S$
  - Repeat until one of $R$ and $S$ is exhausted:
    - If $r. A > s. B$ then $s$ = next tuple in $S$
    - Else if $r. A < s. B$ then $r$ = next tuple in $R$
    - Else output all matching tuples, and $r, s$ = next in $R$ and $S$

- I/O’s: sorting + \( 2B(R) + 2B(S) \)
  - In most cases (e.g., join of key and foreign key)
  - Worst case is \( B(R) \cdot B(S) \): everything joins

Optimization of SMJ

- Idea: combine join with the (last) merge phase of merge sort
- **Sort**: produce sorted runs for $R$ and $S$ such that there are fewer than $M$ of them total
- **Merge and join**: merge the runs of $R$, merge the runs of $S$, and merge join the result streams as they are generated:

![Optimization of SMJ Diagram](image)
Performance of SMJ

- If SMJ completes in two passes:
  - I/O's: \(3 \cdot (B(R) + B(S))\)
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: \(M > \frac{B(R) + B(S)}{M}\)
- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce the number of runs as necessary before final merge and join

Other sort-based algorithms

- Union (set), difference, intersection
- More or less like SMJ
- Duplication elimination
  - External merge sort
    - Eliminate duplicates in sort and merge
  - Grouping and aggregation
    - External merge sort, by group-by columns
      - Trick: produce "partial" aggregate values in each run, and combine them during merge
        - This trick doesn't always work though
        - Examples: SUM(DISTINCT ...), MEDIAN(...)

Hashing-based algorithms

Hash join

\(R \bowtie_{R.A=S.B} S\)

- Main idea
  - Partition \(R\) and \(S\) by hashing their join attributes, and then consider corresponding partitions of \(R\) and \(S\)
  - If \(r.A\) and \(s.B\) get hashed to different partitions, they don't join

Partitioning phase

- Partition \(R\) and \(S\) according to the same hash function on their join attributes

Probing phase

- Read in each partition of \(R\), stream in the corresponding partition of \(S\), join
  - Typically build a hash table for the partition of \(R\)
    - Not the same hash function used for partition, of course!
Performance of (two-pass) hash join

- If hash join completes in two passes:
  - I/O’s: $3 - (B(R) + B(S))$
  - Memory requirement:
    - In the probing phase, we should have enough memory to fit one partition of $R$: $M - 1 > \frac{B(R)}{M - 1}$
    - We can always pick $R$ to be the smaller relation, so:
      $M > \sqrt[3]{B(R)} + 1$

Generalizing for larger inputs

- What if a partition is too large for memory?
  - Read it back in and partition it again!
    - See the duality in multi-pass merge sort here?

Hash join versus SMJ

(Assuming two-pass)

- I/O’s: same
- Memory requirement: hash join is lower
  - $\sqrt[3]{\min(B(R), B(S))} + 1 < \sqrt[3]{B(R)} + B(S)$
- Hash join wins when two relations have very different sizes
- Other factors
  - Hash join performance depends on the quality of the hash
  - Might not get evenly sized buckets
  - SMJ can be adapted for inequality join predicates
  - SMJ wins if $R$ and/or $S$ are already sorted
  - SMJ wins if the result needs to be in sorted order

What about nested-loop join?

- May be best if many tuples join
  - Example: non-equality joins that are not very selective
- Necessary for black-box predicates
  - Example: WHERE user_defined_pred(R.A, S.B)

Other hash-based algorithms

- Union (set), difference, intersection
  - More or less like hash join
- Duplicate elimination
  - Check for duplicates within each partition/bucket
- Grouping and aggregation
  - Apply the hash functions to the group-by columns
  - Tuples in the same group must end up in the same partition/bucket
  - Keep a running aggregate value for each group
    - May not always work

Duality of sort and hash

- Divide-and-conquer paradigm
  - Sorting: physical division, logical combination
  - Hashing: logical division, physical combination
- Handling very large inputs
  - Sorting: multi-level merge
  - Hashing: recursive partitioning
- I/O patterns
  - Sorting: sequential write, random read (merge)
  - Hashing: random write, sequential read (partition)
**Index-based algorithms**

**Selection using index**

- **Equality predicate:** \( \sigma_{A=p}(R) \)
  - Use an ISAM, B*-tree, or hash index on \( R(A) \)
- **Range predicate:** \( \sigma_{A > p}(R) \)
  - Use an ordered index (e.g., ISAM or B*-tree) on \( R(A) \)
  - Hash index is not applicable
- Indexes other than those on \( R(A) \) may be useful
  - Example: B*-tree index on \( R(A, B) \)
  - How about B*-tree index on \( R(B, A) \)?

**Index versus table scan**

**Situations where index clearly wins:**

- **Index-only queries** which do not require retrieving actual tuples
  - Example: \( \pi_A(\sigma_{A > p}(R)) \)
- Primary index clustered according to search key
  - One lookup leads to all result tuples in their entirety

**Index versus table scan (cont’d)**

**BUT(!):**

- Consider \( \sigma_{A=p}(R) \) and a secondary, non-clustered index on \( R(A) \)
  - Need to follow pointers to get the actual result tuples
  - Say that 20% of \( R \) satisfies \( A > v \)
  - Could happen even for equality predicates
  - I/O’s for index-based selection: lookup + 20% \( |R| \)
  - I/O’s for scan-based selection: \( B(R) \)
  - Table scan wins if a block contains more than 5 tuples!

**Index nested-loop join**

\( R \bowtie_{R.A=S.B} S \)

- **Idea:** use a value of \( R.A \) to probe the index on \( S.B \)
- For each block of \( R \), and for each \( r \) in the block:
  - Use the index on \( S(B) \) to retrieve \( s \) with \( s.B = r.A \)
  - Output \( rs \)
- I/O’s: \( B(R) + |R| \cdot (\text{index lookup}) \)
  - Typically, the cost of an index lookup is 2-4 I/O’s
  - Beats other join methods if \( |R| \) is not too big
  - Better pick \( R \) to be the smaller relation
- Memory requirement: \( 3 \)

**Zig-zag join using ordered indexes**

**R \bowtie_{R.A=S.B} S**

- **Idea:** use the ordering provided by the indexes on \( R(A) \) and \( S(B) \) to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
  - Possibly skipping many keys that don’t match
Summary of techniques

• Scan
  • Selection, duplicate-preserving projection, nested-loop join

• Sort
  • External merge sort, sort-merge join, union (set), difference, intersection, duplicate elimination, grouping and aggregation

• Hash
  • Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation

• Index
  • Selection, index nested-loop join, zig-zag join