CSE 530A

Query Implementation

Washington University
Fall 2013
Query Implementation

• Assume we have the table

    CREATE TABLE students (  
        sid BIGSERIAL PRIMARY KEY,  
        name TEXT NOT NULL,  
        year TEXT NOT NULL,  
        major TEXT  
    );

• What must the DBMS do to answer a query like

    SELECT * FROM students WHERE year = '2010';
Query Implementation

• If we assume
  – the table is not ordered by the \textit{year} field and
  – there are not any indexes on that field
• then we must do a table scan
  – A table scan scans the entire table looking for records that match the WHERE predicate
Query Implementation

• Since the time required for disk IO usually dominates all the other time, we typically measure cost as the number of page fetches required to execute a query
  – For our example, if we assume 10,000 records and 100 records a page then the query would require 100 page fetches
Query Implementation

• What if we could somehow keep the table in sorted order by the year field?
  – We could then do a binary search to find the first record with the value '2010' and then read the matching record sequentially
  • If we do the binary search by pages then the cost would be $\log_2 100 \approx 6.6$ plus the number of pages with matching records
Query Implementation

• But it is generally too cost prohibitive to keep a table in sorted order
  – For performance reasons, inserted and updated rows are added to the end of the table rather than in sorted order

• And what about searching on a different field?
Query Implementation

• If we know that most searches on a table will be by a particular field then we could try to keep the table ordered by that field
  – Many DBMSs have the ability to *cluster* a table
    • Reorders the table on disk by the specified field
    • An expensive operation that usually requires a lock on the table
    • A frequently modified table (inserts and updates) can quickly get out of order
    • Using a *fill factor* of less than 100% when clustering will cause the DBMS to leave space in the pages to use later to try to keep in order as long as possible
Query Implementation

What if the query was

```sql
SELECT * FROM students WHERE sid = 12345;
```

- **sid** is the primary key, which means there is an index on that field by default
  - If we assume a B+ tree index of order 101 (that is, between 50 and 100 keys per node) then
    - For 10,000 records, or 100 pages, the depth of the tree will be at most 3
    - So, 3 pages for the index lookups and 1 page for the record
      - Only one matching record since **sid** is a key
Query Implementation

• What if we created a B+ tree index on the year field?
  – 3 pages to find the first matching key in the index
  – Some number, say k, pages for matching index leaves
  – Some number of pages for the matching records
    • Best case: matching records are packed tightly in pages
    • Worst case: each matching record is on a different page
Multi-Key Indexes

• What if our queries often contain a conjunction?

```sql
SELECT * FROM students WHERE year = '2010' and major = 'CSE';
```

• We could create a multi-key index
  – Orders first by the first field and then by the second
  – Conceptually, can think of the index key as the concatenation of the fields
Multi-Key Indexes

• If we create an index on multiple fields, which field should we put first?
  – Index can only be used if the conjunction of terms contains the first index field
  – Generalizing, if an index is created on fields \((x, y, z)\) then it can be used for queries on
    • \(x\)
    • \(x\) and \(y\)
    • \(x\) and \(y\) and \(z\)
  – But not on
    • \(y\)
    • \(z\)
    • \(y\) and \(z\)
Disjunctions

• What if our query contains a disjunction?

SELECT * FROM students WHERE year = '2010' or major = 'CSE';

• Could use a table scan to find matching records
• If an index exists on year and another index exists on major then the results of scanning both indexes could be combined
  – But duplicates would somehow have to be eliminated
• If an index exists on one field but not the other then a table scan would need to be done anyway
  – So might as well not do the index scan
Sorting

• Sorting is often necessary in a DBMS
  – Explicitly requested order in query
  – Intermediate step in implementing joins (more on this later)
Sorting

- If the data set that needs sorting is small enough then it can be sorted in memory
  - What is "small enough"? Depends on amount of memory available
- In memory sorting is usually done using either quicksort or merge sort
### Sorting

<table>
<thead>
<tr>
<th></th>
<th>Average case time</th>
<th>Worst case time</th>
<th>Space (extra)</th>
</tr>
</thead>
<tbody>
<tr>
<td>quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

- Despite worst-case time of $O(n^2)$ quicksort is often preferred for in-memory sorting
  - Worst-case is rare
  - No extra space other than call stack
Merge Sort

- Merge sort is often learned as a recursive algorithm, but can also be done iteratively
  - Requires additional buffer of size $n$
  - Pass 1: start by merging pairs of values into ordered sets of size 2
  - Pass 2: merge pairs of size-2 sets into ordered sets of size 4
  - Repeat until single ordered set of size $n$ is reached
  - Each pass copies values from one buffer to the other
Merge Sort

Start: 4 2 8 1 5 6 3 7

Pass 1: 2 4 1 8 5 6 3 7

Pass 2: 1 2 4 8 3 5 6 7

Pass 3: 1 2 3 4 5 6 7 8
External Sorting

• What if the data set to sort is too large to fit in available memory?
  – Can use an *external merge sort*
External Sorting

• External merge sort
  – Assume we have \( n \) values to sort and three buffers of size \( b \) (where \( b < n \)).
  – Pass 1: read \( b \) values at a time from the set of \( n \), sort using an in-memory sort, write sorted set of \( b \) values to a temporary file.
    • Results in \( \frac{n}{b} \) files of size \( b \).
    • Can potentially do this in parallel using the three buffers.
  – Pass 2: read two of the sorted temp files of size \( b \) and merge into a sorted temp file of size \( 2b \). Repeat for each pair of size \( b \) temp files.
    • Use two of the size \( b \) buffers as input buffers and the other buffer as output buffer.
    • Note that the output buffer will need to be flushed halfway through.
    • Results in \( \frac{n}{2b} \) files of size \( 2b \).
  – Repeat until single sorted file of size \( n \) is reached.
Double Buffering

- On modern machines, disk IO can occur in parallel with CPU computation
- Can take advantage of this to avoid blocking and waiting for IO to complete by using *double buffering*
  - Consider the output buffer in pass 2
    - When the buffer is full it needs to be written to disk before more computation can be done. Need to block waiting for IO to complete before continuing to write to the output buffer. (Problem is worse in subsequent passes when more data needs to be output.)
    - If we used two output buffers then we could write to the second one while the first is being flushed to disk.
    - Could use double buffering on input as well.
- Requires twice the amount of buffer space (or reading/writing half the amount of data at a time)
Increasing Fan-In

- With the external merge sort we'd like to reduce the number of passes as much as possible
  - Each pass requires reading in and writing out the entire set of $n$ values
- We can reduce the number of passes by merging together more than two temp files at a time
  - Merging four files at a time instead of two requires half the number of passes
- Trade-off: increasing the number of files merged at a time requires increasing the number of input buffers, so less of each file can be read at a time
  - But since we're reading each file sequentially we can effectively stream the files, especially if using double buffering