B+ Trees: An IO-Aware Index Structure
“If you don’t find it in the index, look very carefully through the entire catalog”

- Sears, Roebuck and Co., Consumers Guide, 1897
Today’s Lecture

1. [Moved from 12-3]: External Merge Sort & Sorting Optimizations

2. Indexes: Motivations & Basics

3. B+ Trees
1. External Merge Sort
What you will learn about in this section

1. External merge sort

2. External merge sort on larger files

3. Optimizations for sorting
Recap: External Merge Algorithm

• Suppose we want to merge two **sorted** files both much larger than main memory (i.e. the buffer)

• We can use the **external merge algorithm** to merge files of **arbitrary length** in \(2^*(N+M)\) IO operations with only **3 buffer pages**!

Our first example of an “IO aware” algorithm / cost model
External Merge Sort
Why are Sort Algorithms Important?

• Data requested from DB in sorted order is **extremely common**
  • e.g., find students in increasing GPA order

• **Why not just use quicksort in main memory??**
  • What about if we need to sort 1TB of data with 1GB of RAM...

A classic problem in computer science!
More reasons to sort...

• Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)

• Sorting is first step in *bulk loading* B+ tree index.

• *Sort-merge* join algorithm involves sorting
Do people care?

http://sortbenchmark.org

Sort benchmark bears his name
So how do we sort big files?

1. Split into chunks small enough to sort in memory (“runs”)

2. **Merge** pairs (or groups) of runs using the external merge algorithm

3. **Keep merging** the resulting runs (each time = a “pass”) until left with one sorted file!
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

1. Split into chunks small enough to **sort in memory**
External Merge Sort Algorithm

Example:
• 3 Buffer pages
• 6-page file

Orange file = unsorted

1. Split into chunks small enough to sort in memory
Example:
- 3 Buffer pages
- 6-page file

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

1. Split into chunks small enough to **sort in memory**
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Each sorted file is a called a run

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

2. Now just run the **external merge** algorithm & we’re done!
Calculating IO Cost

For 3 buffer pages, 6 page file:

1. Split into two 3-page files and sort in memory
   1. $1 \text{ R} + 1 \text{ W}$ for each file $= 2 \times (3 + 3) = 12 \text{ IO operations}$

2. Merge each pair of sorted chunks using the external merge algorithm
   1. $= 2 \times (3 + 3) = 12 \text{ IO operations}$

3. Total cost $= 24 \text{ IO}$
Running External Merge Sort on Larger Files

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer...

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer... and sort

Assume we still only have 3 buffer pages (Buffer not pictured)

Call each of these sorted files a run
Running External Merge Sort on Larger Files

2. Now merge pairs of (sorted) files... the resulting files will be sorted!

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

3. And repeat...

Assume we still only have 3 buffer pages (Buffer not pictured)

Call each of these steps a pass
Running External Merge Sort on Larger Files

4. And repeat!
Simplified 3-page Buffer Version

Assume for simplicity that we split an N-page file into N single-page *runs* and sort these; then:

- First pass: Merge $N/2$ pairs of runs each of length 1 page
- Second pass: Merge $N/4$ pairs of runs each of length 2 pages
- In general, for N pages, we do $\lfloor \log_2 N \rfloor$ passes
  - +1 for the initial split & sort
- Each pass involves reading in & writing out all the pages = $2N$ IO

→ $2N*(\lfloor \log_2 N \rfloor + 1)$ total IO cost!
Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. **Increase length of initial runs.** Sort B+1 at a time!
   At the beginning, we can split the N pages into runs of length B+1 and sort these in memory

**IO Cost:**

Starting with runs of length 1

Starting with runs of length \(B+1\)
Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

2. Perform a B-way merge.

On each pass, we can merge groups of B runs at a time (vs. merging pairs of runs)!

IO Cost:

\[ 2N(\lfloor \log_2 N \rfloor + 1) \rightarrow 2N\left(\left\lfloor \frac{N}{B + 1} \right\rfloor + 1\right) \rightarrow 2N\left(\left\lfloor \log_B \frac{N}{B + 1} \right\rfloor + 1\right) \]

Starting with runs of length 1
Starting with runs of length \( B+1 \)
Performing \( B \)-way merges
Repacking
Repacking for even longer initial runs

• With B+1 buffer pages, we can now start with \textit{B+1-length initial runs} (and use \textit{B-way merges}) to get \(2N\left(\log_B \frac{N}{B+1}\right) + 1\) IO cost...

• Can we reduce this cost more by getting even longer initial runs?

• Use \textit{repacking} - produce longer initial runs by “merging” in buffer as we sort at initial stage
Repacking Example: 3 page buffer

• Start with unsorted single input file, and load 2 pages
Repacking Example: 3 page buffer

- Take the minimum two values, and put in output page

Also keep track of max (last) value in current run…
Repacking Example: 3 page buffer

• Next, **repack**
Repacking Example: 3 page buffer

- Next, **repack**, then load another page and continue!
Repacking Example: 3 page buffer

- Now, however, **the smallest values are less than the largest (last) in the sorted run**...

We call these values *frozen* because we can’t add them to this run...
Repacking Example: 3 page buffer

• Now, however, **the smallest values are less than the largest (last) in the sorted run**...

![Diagram showing disk and main memory with buffer and data sets.](image-url)
Repacking Example: 3 page buffer

• Now, however, the smallest values are less than the largest (last) in the sorted run…
Repacking Example: 3 page buffer

- Now, however, **the smallest values are less than the largest (last) in the sorted run**...
Repacking Example: 3 page buffer

• Once *all buffer pages have a frozen value*, or input file is empty, start new run with the frozen values
Repacking Example: 3 page buffer

- Once *all buffer pages have a frozen value*, or input file is empty, start new run with the frozen values
Repacking

• Note that, for buffer with B+1 pages:
  • If input file is sorted → nothing is frozen → we get a single run!
  • If input file is reverse sorted (worst case) → everything is frozen → we get runs of length B+1

• In general, with repacking we do no worse than without it!

• What if the file is already sorted?

• Engineer’s approximation: runs will have \(~2(B+1)~\) length

\[
\sim 2N\left(\log_B \frac{N}{2(B + 1)}\right) + 1
\]
Summary

• Basics of IO and buffer management.
  • See notebook for more fun! (Learn about sequential flooding)

• We introduced the IO cost model using sorting.
  • Saw how to do merges with few IOs,
  • Works better than main-memory sort algorithms.

• Described a few optimizations for sorting
2. Indexes
What you will learn about in this section

1. Indexes: Motivation
2. Indexes: Basics
3. ACTIVITY: Creating indexes
Index Motivation

• Suppose we want to search for people of a specific age

• **First idea:** Sort the records by age... we know how to do this fast!

• How many IO operations to search over \( N \) sorted records?
  • Simple scan: \( O(N) \)
  • Binary search: \( O(\log_2 N) \)

Could we get even cheaper search? E.g. go from \( \log_2 N \) to \( \log_{200} N \)?
Index Motivation

- What about if we want to **insert** a new person, but keep the list sorted?

- We would have to potentially shift $N$ records, requiring up to $\sim 2*N/P$ IO operations (where $P = \# \text{ of records per page}$)
  - We could leave some “slack” in the pages...

Could we get faster insertions?
Index Motivation

• What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?
  • We could keep multiple copies of the records, each sorted by one attribute set... this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We’ll create separate data structures called *indexes* to address all these points
Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) *just indexes*!

- A lot more is left to the user in NoSQL... one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!

- Sometimes use B+ Trees (covered next), sometimes hash indexes (not covered here)

Indexes are critical across all DBMS types
Indexes: High-level

• An index on a file speeds up selections on the search key fields for the index.
  • Search key properties
    • Any subset of fields
    • is not the same as key of a relation

• Example:

\texttt{Product(name, maker, price)}

On which attributes would you build indexes?
More precisely

• An index is a data structure mapping search keys to sets of rows in a database table
  
  • Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table

• An index can store the full rows it points to (primary index) or pointers to those rows (secondary index)

  • We’ll mainly consider secondary indexes
Operations on an Index

• **Search**: Quickly find all records which meet some *condition on the search key attributes*
  • More sophisticated variants as well. Why?

• **Insert / Remove** entries
  • Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance
Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row...

### Russian_Novels

<table>
<thead>
<tr>
<th>BID</th>
<th>Title</th>
<th>Author</th>
<th>Published</th>
<th>Full_text</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td><em>War and Peace</em></td>
<td>Tolstoy</td>
<td>1869</td>
<td>...</td>
</tr>
<tr>
<td>002</td>
<td><em>Crime and Punishment</em></td>
<td>Dostoyevsky</td>
<td>1866</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td><em>Anna Karenina</em></td>
<td>Tolstoy</td>
<td>1877</td>
<td>...</td>
</tr>
</tbody>
</table>

```sql
SELECT * 
FROM Russian_Novels 
WHERE Published > 1867
```
Conceptual Example

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?
Conceptual Example

**By_Yr_Index**

<table>
<thead>
<tr>
<th>Published</th>
<th>BID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1866</td>
<td>002</td>
</tr>
<tr>
<td>1869</td>
<td>001</td>
</tr>
<tr>
<td>1877</td>
<td>003</td>
</tr>
</tbody>
</table>

**By_Author_Title_Index**

<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
<th>BID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dostoyevsky</td>
<td>Crime and Punishment</td>
<td>002</td>
</tr>
<tr>
<td>Tolstoy</td>
<td>Anna Karenina</td>
<td>003</td>
</tr>
<tr>
<td>Tolstoy</td>
<td>War and Peace</td>
<td>001</td>
</tr>
</tbody>
</table>

**Russian_Novels**

<table>
<thead>
<tr>
<th>BID</th>
<th>Title</th>
<th>Author</th>
<th>Published</th>
<th>Full_text</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>War and Peace</td>
<td>Tolstoy</td>
<td>1869</td>
<td>...</td>
</tr>
<tr>
<td>002</td>
<td>Crime and Punishment</td>
<td>Dostoyevsky</td>
<td>1866</td>
<td>...</td>
</tr>
<tr>
<td>003</td>
<td>Anna Karenina</td>
<td>Tolstoy</td>
<td>1877</td>
<td>...</td>
</tr>
</tbody>
</table>

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures...
Covering Indexes

We say that an index is **covering** for a specific query if the index contains all the needed attributes—meaning the query can be answered using the index alone!

<table>
<thead>
<tr>
<th>By_Yr_Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Published</td>
</tr>
<tr>
<td>1866</td>
</tr>
<tr>
<td>1869</td>
</tr>
<tr>
<td>1877</td>
</tr>
</tbody>
</table>

The “needed” attributes are the union of those in the SELECT and WHERE clauses...

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```
High-level Categories of Index Types

• B-Trees (covered next)
  • Very good for range queries, sorted data
  • Some old databases only implemented B-Trees
  • We will look at a variant called B+ Trees

• Hash Tables (not covered)
  • There are variants of this basic structure to deal with IO
  • Called linear or extendible hashing - IO aware!

The data structures we present here are “IO aware”

Real difference between structures: costs of ops determines which index you pick and why
Activity-13.ipynb