Review of Paper – 5: - Ramkumar Rajabaskaran (85241493)

This paper focuses on the Optimizations for Dynamic Inverted Index Maintenance for Information Retrieval systems. **Inverted Index** is an efficient data-structure that maps a word to a set of documents. There are different algorithms used in inverted indices such as Boolean, extended Boolean, proximity and relevance search algorithms. Since access to the inverted index is based on a single key, it is sorted or organized as a **Sorted Hash Table**. The paper also defines performance criteria such as **Block Update Speed (**The time required to update the documents), **Access Speed** (The time required to access the postings), **Index Size** (Amount of storage required for the inverted index), **Dynamics** (Ease at which index is incrementally updated and **Scalability** (How it scales with an increase in the corpus size).

B-Trees, which are a file-based data structure which are appropriate for the implementation of dynamically modified inverted indices. In B-trees, the nodes are represented as disk-pages and algorithms exist to ensure that the insertion, examination and deletion of entries are in $\mathbf{O (\log_b N),}$ where b is the branching factor of the B-Tree and N is the number of entries contained in the B-Tree. This proves particularly useful in cases where there are **1 Million entries** and **b=100**. This means that the depth of the B-Tree is **3.** That is, we can access any page in **3-disk accesses**. This may be improved by retaining the pages in the cache. If we have 10000 pages in the cache, we need only one disk operation. Other strategies such as LRU cache have similar performance characteristics.

The characteristics of B-Tree Indexing are addressed next. Thus, the **Block Update Time** is the number of disk reads required (as it will take most the time). Each entry is of the form (word, location) ordered first by word then location and the indexing of n words requires $\mathbf{n (\log_b N - \log_b C)}$ where C is the no. of cached pages. Removal can be accomplished at the same expense. The number of disk accesses can be reduced through caching of the upper nodes of the B-Tree. If this core memory is used as a **buffer**, many more disk accesses can be eliminated if the buffer is used for sorting and subsequently merging postings into an existing B-Tree page.

In the naïve indexing, there is redundancy where the instance requires a reference to the word itself. If instead the postings are decomposed into a word and set of locations, we can reduce redundancies. **Grouping** is shown to reduce the space requirements by 50%. A **grouped indexing** can be implemented by a variable length B-tree with F as a marginal frequency. (Word, (location)*). This tends to overflow and can be avoided by indexing as tuples of the form (word, F, pos) where position is in an auxiliary structure as a **heap file**, where a sequence of locations can be found. **Pulsing** is where the use of a **Heap** solves the **B-Tree overflow problem**, but the cost of access time is increased and the occurrence of overflow is less. Hence Pulsing is used here where a heap file is used when necessary using a threshold t. The updates are made only after t new instances of the word are seen and the t locations are pulsed to the heap. **Delta Encoding** is another technique mentioned where each location is stored as an integer and the difference in the integers is then encoded. Deletion is one application where random access to individual postings is desired. This leads to a sequence of much smaller numbers being stored. Hence space and time is optimized at the expense of this relatively rare operation.

My Observation: - For Information retrieval, the pages are to be stored in the memory and must be indexed for their relevance. One of the better methods to achieve this indexing is through **Inverted Indexing** where the data is stored as (Word, Frequency). This inverted index is then implemented as a **hash table** where all operations are only O (N log N). **Dynamic update** of this index is an important feature required in modern day corpora and the **B-Tree** is an efficient data structure to implement this. The Inverted index is implemented as a B-Tree, where the B-Tree has the advantage of having **a balanced storage** of pages where the **Depth is** $\mathbf{\text{Log}_B N}$ and hence reduces the number of disk accesses by a huge amount since the Disk access takes up most the time. **Merge and update** that uses a **buffer** to improve the speed, **pulsing and delta encoding** help in the space optimization apart from handling other problems such as overflowing(space) and time performance are handled using a heap and Caching the data in main memory**.** Thus, we can see the effectiveness of using a B-Tree in Inverted Indexing to be an efficient dynamic data structure for faster and compact information retrieval systems.