Provisional Application for United States Patent

TITLE: UBiquitous linear sorted order indeXing engine (UBX)
INVENTOR(S): Lawrence John Thoman, John Chwanshao Wang
USPTO Patent Application Number: 62748132

BACKGROUND

In the time of big data there is a need to find and access the data rapidly.

When performing ad hoc queries, many existing analytical tools use a combination of indexing of some frequently used fields and linear table scan on non-indexed fields.

Traditional indexes take a relatively long time to generate and a relatively long time to use. Often the index for a field is bigger than the actual data field that is indexed thus commonly leading to a situation where only a few key fields of a data record are indexed because of space constraints.

BRIEF SUMMARY OF THE INVENTION

There is a need for a way to generate an index very fast. The index also needs to be fast to use.

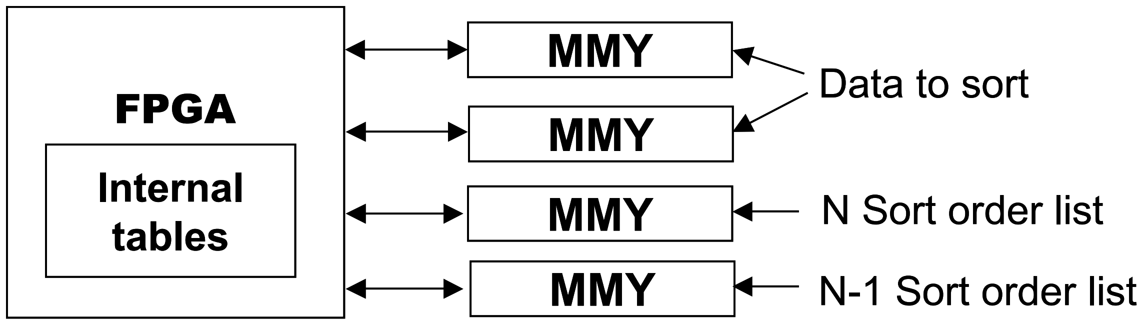
There are 2 parts of the index problem that are addressed by this implementation
The first part is rapid generation of the index the second part is the generation of an index of a form that can be used to rapidly access the data.

One of the common problems encountered in using big data is finding all the records that have a particular value or range of values in a key field. Commonly this is needed to select a group of records for further filtering or to sum up the values of another field of the record.

An important part of this index creation is creating a list of record numbers in the sorted order index table
.
Another part is creating and using a format and mechanism that make it possible to use this sorted order index table in an efficient manner.

To this end we have devised a sort method that generates the sorted order list by looking at each character in the field to be sorted once and only once. Thus the sort time is linear. Where "t" is the time to sort one record. The time to sort n records is t(n) not t(n log n)
In addition our sort method does all the record keeping for the sort at the same time as the character is being read from the memory.

This is accomplished by implementing a hardware design in a Field Programmable Gate Array (FPGA) with several different memory spaces



Logical example of the index tables generated by the implementation.

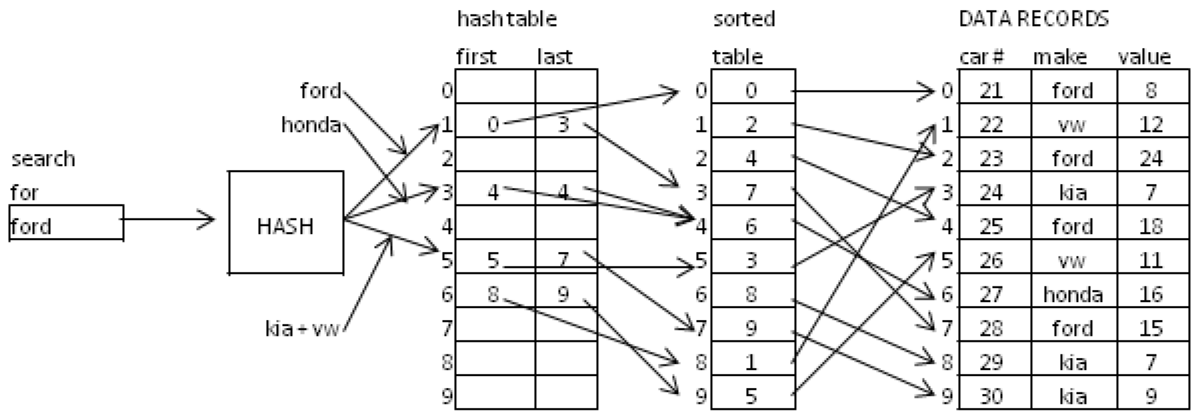
For each field indexed there are 2 tables, a sorted order list of record numbers and a hash table.

We are not storing the actual value of the field being indexed in the index (as is done in a traditional index) instead we are using the actual data in the record.

When we are using the index to find the records we need, we want to minimize the number of times we read the actual field value to find the correct first record and last record for the value we are looking for.

For this purpose we create a hash table that that contains pointers into the sorted order list This table is accessed by creating a hash of the value we are looking for and using that as a pointer into the hash table.

Each row of the hash table contains 2 pointers into the sorted order list one to the first occurrence of a value and one to the last occurrence.
Unused hash table entries are populated with NAN (not a number = some number that is too large to be a pointer)



Blanks are NAN

Example
To find all the records for “ford”:
“ford” hashes to row 1 in the hash table the first and last values point to the 0 and 3 rows in the sorted table. Check the 0 record if it contains “ford” then records we want are records 0,2,4,7, the rows indicated by rows 0 though 3 of the sorted table, There is no need to check each record to see if it is “ford”

To find all the records for “vw”:
“vw” hashes to row 5 in the hash table the first and last values point to the 5 and 3 rows in the sorted table Check the # 3 record, if it contains “vw” we have found what we want, however it has “ford” which is not a match so try again with the next row in the hash table, this time it points to row 8 in the sorted table which points to record 1 which contains “vw” so we have found what we want.

In general:

To find the correct records, hash the value into a hash table address and get the “first” value at that location and use it as the address of the sorted order list which points to the actual data record , if the data value does not match try the next hash table entry, repeat until match found or NAN found.
If NAN is found than an exact match is not found.

If the task is to find a range try to find a match for the other end of the range. If successful we have found the point in the sorted order list from which we can find all the values in the range by stepping through the sorted order list. If unsuccessful use a binary search on the sorted order index table.

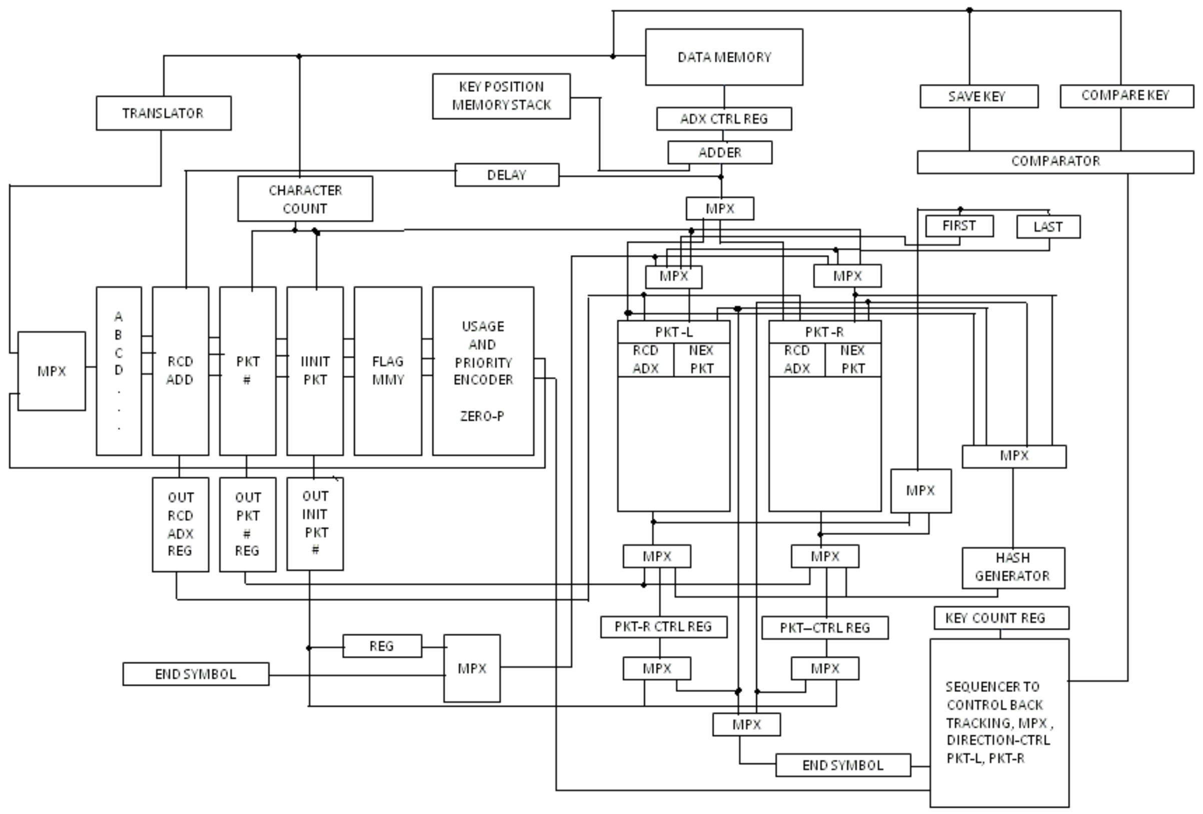
If we found both ends of the range we don’t need to look at the actual value of the data to process the records we can just access the rows in the sorted order index table that are between the “first” row of the low end of the range and the “last” row of the high end of the range.

The needed size of the hash table depends somewhat on the variety of the data being indexed. If the number of possible data values is known there are well established guidelines for figuring it out and the best kinds of hash schemes.

Implementation

The drawing shows the general hardware design low level block diagram to implement the above index method in linear time while doing all the time for record keeping is hidden by doing it at the same time as the data memory access. This is accomplished by having several separate memories and logic operating at the same time as the data memory access effectively trading space (more hardware) for time.

The entire hardware logic for this process is implemented in a FPGA and 4 connected banks of memory, DATA MEMORY, PKT-L, PKT-R



Creating the sorted order list (sorted list)

The present invention provides an improvement in sorting of data wherein two pocket memories are alternately employed as source and destination pocket memories for storing column lists of record addresses corresponding to character addresses in successive columns of records in data memory (DATA-MMY). The characters of successive stored records of data at the same number of characters from the start of each record are considered to form a column and successive columns from least significant character (LSC) to most significant character (MSC) in any field or part thereof are sorted by a predetermined character rank to form sort lists of record start addresses with each column being sorted in the order of addresses in the proceeding list.

Sorting of one column of characters is accomplished in one memory cycle per character and thus in general, there are as many memory cycles as there are characters in a column. By the use of multiple memories, the present invention provides for reading out of one memory while writing into another memory so that sorting of each character called out of DATA-MMY occurs in one memory cycle. Full pipeline operation is provided herein. Input data to be sorted contains information such as number of records (RCDCNT), offset key position numbers, character collating sequence, and record length (RL), or record end markers for records of different length, together with command words. The foregoing information is employed for initializing the system,

Virtual pockets are used to reduce required size of memory for sorting, by the provision of a character memory system to control pocket memories and in the provision of a translator having a controllable logic decision set for expanding the type of sorting or collating sequence from a character set to a wide variety of sets.

The sorting of very large volumes of electronically recorded data in a single sort operation tends to require extremely large memory storage. It is herein provided that the traditional physically separate pockets of pocket memories employed in forming the sort lists are replaced by virtual pockets by linking pocket position or location for repetitions of characters. Thus, the memory for each of two "pocket memories" need only have the capability of storing the number of records being sorted instead of the number of records being sorted times the number of characters of the sorting list. This saving of memory is quite significant for large sorting operations as of the order of many millions of records.

The method and system of the present invention incorporates the concept of virtual memories wherein the sorting lists are recorded or entered into alternate pocket memories that are required to accommodate only the same number of addresses as there are records being sorted. It will be appreciated that this concept very materially reduces the amount of memory required for any large sorting operation. With the traditional implementation of a pocket sort each pocket employed needs to accommodate storage of the same number of addresses as there are records being sorted. This is true because any one column of characters in the records being sorted may comprise only a single character.

The present invention provides for linking or chaining of repetitions of each character present by recording or storing not only a record start address, but also the next pocket location wherein the same character is stored.

Considering this virtual pocket concept somewhat further, it is noted that successive addresses from a source pocket memory are only read into the alternative pocket memory upon the second occurrence of a character. In order to accomplish this manner of writing into a pocket memory, there is herein provided a character memory which records, for each character, the identify thereof, the record start address, the current pocket number, the initial pocket number (INIT-PKT#), and a flag to identify recurrence of such character. Such information for each character is retained in character memory (C-MMY) until the second occurrence of a character, at which point the record start address of the first occurrence of such character is transferred to the receiving pocket memory together with the present pocket number of such character as the next pocket (NXT-P) location of the same character in the receiving pocket memory. In this manner, all of the pockets for any particular character are chained or linked together by means of the next pocket address associated with each occurrence of such character.

Proceeding further with the foregoing, it is noted that at the end of column sort, there will remain in C-MMY the record addresses of all characters which has only occurred once, and the record address of the last occurrence of each character that has occurred more than once. The method hereof then proceeds to backtrack from the last character memory position to successively transfer record start addresses, and next position numbers, into the receiving pocket memory with the highest ranking character having the record start address thereof associated with the next pocket location of the lowest ranking character. This then provides the start position for reading out the contents of the receiving pocket memory at the end of the sorting of the column of characters. Backtracking identifies the most significant character rank of the list of characters used and likewise the least significant rank which then identifies the start position of the next column sort.

Creating the hash table

The hash table is created by first initializing the hash table (in the other now unused pocket memory) with NAN values and setting the key count register to 1
Then using the sorted order list and starting with the first entry (lowest value) reading and saving the key value in both the key save register and the compare register. The record number of the sorted order list is saved in both in the first and last register.

The next sorted order list entry is read and the key value is saved in the compare register and compared to the current saved key value If they match the current sorted order list record number is saved in the last register and the process repeats until key value does not match.

When the key values don't match the first and last registers are copied to the hash table at the address generated by the hash system or in the first unused (NAN) location if the location is already populated, The compare register is copied to the current saved key register and the record number of the sorted order list is saved in both in the first and last register. The key count register is incremented. This process continues until the END SYMBOL is encountered.

At that point the hash table is complete

If it turns out that the count in the key count register is small this process can be rerun with a smaller hash table.

It is also possible to use the hash table to create a smaller hash table.

Claims

An indexing system that runs in Linear time that also does sorting using hardware to speed up the process.

A hardware system consisting of a FPGA and several memory banks that are all running at the same time.
Logic and the house keeping of the various that need to be kept track of to do the sort and index creation all run in parallel with the main memory that holds the data to be sorted. The result of which the access time for the main memory is the limiting time not all the record keeping needed for sorting.
A sorting system that accesses the data only one time so is linear in time not n log n, this is big advantage when sorting large numbers of records