Abstract

One of the techniques to estimate selectivities of query predicates is by generating equidepth partitions for a relation and maintaining equidepth histograms. This paper presents a new technique to generate equidepth partitions for a secondary memory resident relation using a linear median find algorithm. A variation of this technique, which involves sorting of the individual blocks, is also proposed and extended to generate arbitrary number of partitions. The proposed techniques are compared with an external sorting based partitioning approach. Finally, an overall strategy to select the partitioning algorithm with possibly dynamic allocation of buffers is proposed.

1 Introduction

Data analysis is used widely as a technique to study trends and patterns in several areas of research. Research in social sciences, demographics, management of natural resources, financial markets, stocks, insurance and actuarial sciences, telecommunications, engineering applications, weather prediction and market surveys are some of the more common areas where massive data analyses is one of the primary tools of research. Depending on the need of the application, the retrieval and interpretation of data differs widely. For example, in case of financial markets, one needs to know the average increase in the price of shares of the top 50 companies that are being traded on the market. In many cases, one tends to study a small portion of the entire data set. Such data analyses usually requires partitioning the data set to study subsets of the data.

Partitioning the data on key values of the tuples can be done in two ways. *Equiwidth partitioning*, where the entire range of the data set is divided into equal parts and the data set is partitioned over these sub-ranges. Here there is no control over the number of data items in each of the partitions and this could result in empty or sparsely filled partitions. The second method is known as *equidepth partitioning* where the entire data set is partitioned into a specified number of partitions such that the number of data items in each partition is the same. Here, one does not have any control over the sub-ranges of the partitions and for dense data sets, the partition boundaries could be very closely spaced.

One other useful application of partitioning the data is in the processing of queries to retrieve information from the database. The importance of accurate selectivity estimation for efficient query processing and optimization has been extensively researched and documented [Shapiro84, Mackert86]. Based on the selectivities of query predicates, it is possible to select an appropriate access path. B-Trees [Comer79], KDB Trees [Robinson81] and Grid Files [Nievergelt84] are some of the commonly used multi-dimensional index structures. Index structures are useful for accessing the data during tuple retrieval but do not directly yield an estimate of the query selectivities. The System R optimizer [Selinger79] used simple statistics such as the minimum and maximum values for tuple key values to estimate selectivity factors. Generally, a linear interpolation has been used which yields good estimates only if the key values are uniformly distributed. In general, for distributions other than uniform, histograms are used for estimating selectivity factors. It has been shown [Shapiro84] that equiwidth histograms are prone to substantial errors in the estimates if the key values are not uniformly distributed whereas equidepth histograms yield good selectivity estimates in such cases [Muralikrishna88].

One way of generating equidepth histograms is by partitioning a sorted relation into equal sized buckets [Muralikrishna88]. Sampling the database and using the sorted sample to build the equidepth histograms was also proposed in case of large databases. This sampling technique, compared to other sampling techniques, yields the most accurate selectivity estimates, by far. The major drawback of a sorting based equidepth partitioning technique is that the cost of sorting a large relation on a system with limited main memory can be prohibitively expensive. In this re-
port, we present a new technique for generating equidepth partitions using the median find algorithm. Sorting of the entire relation is not required. We also present a variation of this technique which only requires sorting the individual blocks of the relation. This is then adapted to generate any arbitrary number of partitions. The proposed technique is shown to perform appreciably better when the number of partitions required is small compared to the number of input blocks of the relation.

It is sufficient to generate a small number of partitions to get an estimate of the selectivity. Maintaining and dynamically updating a large number of histograms is expensive. For these reasons, it is felt that the proposed partitioning techniques are useful. One other possible application of the equidepth partitions is to perform the join operation of relations. Both the relations are partitioned on the join column. The Join operation then requires accessing only the overlapping partitions of the two relations. The associated costs of the Join operation and of maintaining the partitions as a storage structure remains to be studied.

Section 2 deals with the problem of equidepth partitioning that we address in this paper. Section 3 discusses the environment of operation and the unit operations used in the partitioning technique and their associated costs. In section 4, the various stages of the partitioning technique are discussed. The proposed techniques and the sorting based partitioning are evaluated for the I/O and CPU costs, in Section 5 followed by a comparative performance evaluation in Section 6.

2 Problem Definition

Before we discuss the actual problem at hand, we introduce some relevant terminology. We use the terms data, key value and tuple interchangeably. We define two data blocks, $B_1$ and $B_2$, with $n$ tuples each as being disjoint blocks if we can identify a key value, $m$, such that $(\forall r \in B_1)$ ($r \leq m$) and $(\forall s \in B_2)$ ($s \geq m$). The key value, $m$, is the median of the two blocks $B_1$ and $B_2$. This can be analogously extended to 2 sets of $n$ blocks being mutually disjoint. If the two sets of blocks are identified as $S_1$ and $S_2$, every tuple in $S_1$ is $\leq m$ and every tuple in $S_2$ is $\geq m$. Here, again, the key value, $m$, is the median of the 2 sets of blocks $S_1$ and $S_2$.

Equidepth partitioning Problem: We now formally state the EDP problem. Given $N = 2^n$ blocks of unsorted data and $M$ blocks of main memory buffers, partition the data into $p = 2^i, 1 \leq i \leq n$ equal sized partitions, $P_1, P_2, \cdots, P_p$ such that, $-1 \leq |P_i| - |P_j| \leq 1, 1 \leq i, j \leq p$. Here, $|P_i|$ is the cardinality of partition $p_i$. If $i = 1$ the problem reduces to finding the median of the entire data set. The median can be used to generate 2 disjoint sets of $N/2$ blocks each. The $N/2$ blocks in each of the two partitions may not be disjoint themselves. If $i = 2$, we get the quartile data values which separate the $N$ blocks into 4 disjoint partitions of $N/4$ blocks each, and so on. The input data set is thus, partitioned into $2^i$ partitions of size $2^{n-i}$ blocks, where $1 \leq i < n$, referred to as Partial partitioning. In the limit, we wish to generate $N = 2^n$ partitions with each partition being one disk block, referred to as Complete partitioning.

3 Environment of Operation

Consider $N$ blocks of data resident on secondary storage. Disk block sizes typically range from 512-8K bytes. Main memory available to a process is of the order of a few megabytes. Let the available main memory be $M$ disk blocks. We consider the case where the size of the data set far exceeds the size of the available main memory. To determine the true median of the $N$ blocks, we have to read the entire data set in several passes. The algorithms presented in the following sections process parts of the entire data set which can fit into the available main memory at one time. Henceforth, the $M$ blocks of data that can be read into the main memory at one time will be referred to as a chunk of data. Let $N_T$ be the number of data items in one disk blocks.

3.1 Basic Operations

Given $M$ blocks of main memory we consider two basic operations - (i) Sorting $M$ blocks of data at a time and (ii) Finding the median of $M$ blocks of data. Any one of several internal sorting algorithms can be used. We employ an efficient algorithm to find the median of $N_T$ tuples [Blum72]. The construction is as described in [Knuth73]. This algorithm will be referred to, henceforth, as MedFind. It has been shown in [Knuth73] that the total number of comparisons to find the median of $N_T$ elements is at most $15N_T - 163$ for $32 \leq N_T \leq 2^{10}$. Thus, the selection of the median can be done in "linear time". MedFind returns the median of the input data. Also, the input array is separated into two halves with all data in the lower half $\leq$ the median and all data in the upper half $\geq$ the median. If $M > 1$ the unit operations become the internal sorting and median find of a chunk of data, respectively.

3.2 Cost Model

We consider both CPU and I/O costs to evaluate the algorithms proposed forthwith. If we use a comparison based sorting algorithm, such as Heapsort [Floyd64] or Quicksort [Hoare62], the unit operation is a comparison of two key values followed by a swap operation, if necessary. This is also the unit operation for the median find algorithm presented earlier. We use the number of (comparison + swap) operations as a measure of the CPU cost. I/O cost is measured as the number of disk block accesses for reads and writes. On an average, the time to seek and read a block from disk is equal to the time required to seek and write
the block to disk. The I/O cost gives the number of block reads and writes for an algorithm.

Costs for MedFind: For median find, Schonage has shown that the asymptotic number of comparison and swap operations to find the median of \( N \) tuples is \( 3N \) [Knuth73]. The I/O cost is 1 read and 1 write operation. Thus,

\[
\text{CPU Cost} = 3N \quad \text{and} \quad \text{I/O Cost} = 2
\]

MedFind also separates the data into two disjoint halves by comparing all \( N \) tuples against the median value and a swap if necessary. We refer to the total CPU cost of scanning and comparing \( N \) tuples as \( \text{SCAN}_1 \). Thus,

\[
\text{CPU Cost} = \text{CPU MF}_1 = 4N = 4 \text{SCAN}_1
\]

\[
\text{I/O Cost} = \text{I/O MF}_1 = 2
\]

In the algorithms presented in the following sections, we study the impact of varying the number of available input buffers. In general, if \( 2^k \) input buffers are available, the unit operations will be internal sorting and median find operations on \( 2^k \) input blocks. For internal sorting of \( 2^k \) blocks of data,

\[
\text{I/O SORT}_p = [2, 2^k]
\]

\[
\text{CPU SORT}_p = [K_S 2^{k \cdot \lfloor \log(N \cdot 2^k) \rfloor}] \text{ SCAN}_1
\]

where, \( K_S \) is the average value of the sorting constant. \( K_S \) is typically 12 for Quicksort [Knuth73]. For median find on \( 2^k \) input blocks, we have

\[
\text{I/O MF}_p = 2, 2^k
\]

\[
\text{CPU MF}_p = [4, 2^k] \text{ SCAN}_1
\]

4 Equidepth partitioning algorithms

In this section, we present a family of algorithms to generate equidepth partitions for a disk based data set. We now describe the algorithm to find the median of 2 blocks of input data with 1 available input buffer by a recursive application of algorithm MedFind.

4.1 Median of 2 Blocks

Consider two blocks of unsorted data, \( B_1 \) and \( B_2 \) which are not disjoint initially and 1 available buffer. Using MedFind, find the medians \( m_1 \) and \( m_2 \) of blocks \( B_1 \) and \( B_2 \), respectively. MedFind also separates the blocks into disjoint half-blocks separated by the block median. A typical distribution of the data in \( B_1 \) and \( B_2 \) is shown in Figure 1. Rearrange the blocks such that \( m_1 \leq m_2 \). Let the lower and upper bounds of the data in the two rearranged blocks be \( [l_1, u_1] \) and \( [l_2, u_2] \) respectively. We need not know the exact values of these bounds but do know their position relative to the block medians. Identify the half-block

\[
[l_1, m_1] \quad \text{as LOWER}_1 \quad \text{and} \quad [m_1, u_1] \quad \text{as UPPER}_1.
\]

Similarly, for block \( B_2 \) the median of \( B_1 \) and \( B_2 \) lies in the half blocks \( \text{UPPER}_2 \) and \( \text{LOWER}_2 \). In Figure 1 above, \( \text{BMED} \) represents the expected location of the block median of these two half-blocks. As a final step, read in the \( 2^k \) blocks, \( \text{UPPER}_1 \) and \( \text{LOWER}_2 \) into the input buffers. A final call to MedFind yields the value of \( \text{BMED} \) which is the true median of blocks \( B_1 \) and \( B_2 \).

The resulting structure of the 2 input blocks is as shown in Figure 2. We now consider the problem of identifying the median of \( N \) blocks of data using 1 buffer.

4.2 True Median of \( N \) blocks

Consider \( N = 2^k \) blocks of unsorted data which are not mutually disjoint. We now describe a procedure, algorithm FILTER, which will be used to reduce the number of input blocks that have to be processed to \( N/2 \). By repeated applications of this procedure, the size of the input set of blocks that need to be processed will be reduced to 1 block. The median of this block will be the true median of the input set of \( N \) blocks. Filtering: Algorithm FILTER

1. Find the block median of each of the \( N \) input blocks using the unit MedFind operation.

2. Sort the list of block medians in a non-decreasing order and rearrange the \( N \) input blocks in this order. Let the blocks in this ordered sequence be \( \{B_1, B_2, \ldots, B_N\} \) with the corresponding block medians being \( \{m_1, m_2, \ldots, m_N\} \). Represent block \( i \) by the triplet \( (L_i, m_i, U_i) \) which is the lower half block, block median and the

---

**Figure 1: Relative position of 2 input blocks**

**Figure 2: Resulting data distribution after identifying the true median of 2 blocks**
Figure 3: Partially processed sequence of N input blocks.

upper half block, respectively. The sequence of blocks at the end of step 2 is shown in Figure 3.

3. Consider block B₁ and Bₙ. As m₁ ≤ mₙ, every tuple in U₁ which is > m₁ and there is a corresponding tuple in Uₙ which is ≥ mₙ. Therefore, the true median of the 2 blocks B₁ and Bₙ lies among the elements of U₁ and Uₙ. By extension of this argument [Blum72], the true median of the N blocks lies among the elements of (U₁, U₂, ..., U₁ₙ/₂) and (L₁ₙ/₂₊₁, L₁ₙ/₂₊₂, ..., Lₙ). The upper halves of the blocks numbered (1, 2, ..., 2ⁿ⁻¹) and the lower halves of blocks numbered (2ⁿ⁻₁+1, 2ⁿ⁻₁+2, ..., 2ⁿ) have to be processed further to locate the true median of the N input blocks.

4. Thus, one pass of FILTER identifies N half blocks in which the median can lie and N half blocks in which it cannot. Fully processed half blocks are removed from the list of input blocks.

5. For i = 1...N/2, half block Uᵢ is merged with half block Lᵢ₂, to form an output list of N/2 blocks. This becomes the input list for the next pass of FILTER. Figure 4 shows the blocks after 1 pass of FILTER.

4.3 Algorithm TRUE_MEDIAN

Given an input of N = 2ⁿ blocks, TRUE_MEDIAN repeatedly calls FILTER, with the input size of blocks being halved on successive calls. The input to the nth pass of FILTER is a pair of blocks, the output being a single block. A final call to MedFind yields the block median which is also the true median of the original set of N blocks input to TRUE_MEDIAN. At the end of lg N calls to FILTER, TRUE_MEDIAN terminates by identifying the true median of the N blocks. Thus, each pass of TRUE_MEDIAN partitions one input set of 2ⁿ blocks into 2 disjoint sets of 2ⁿ⁻¹ blocks each. This partitioning will be represented as (1, 2ⁿ⁻¹) ≃ (2, 2ⁿ⁻²)...

Multiple Partitions for N input blocks: EDPMF We now address the original problem of generating multiple partitions. Given an input set of N = 2ⁿ blocks, we wish to generate p = 2ⁱ disjoint partitions, for some i ≤ n. We partition the given set of N blocks successively into sets of blocks of sizes N/2, N/4 etc. This can be represented as:

(2ⁿ⁻¹, 2ⁿ⁻²) ≃ (2⁻¹, 2⁻²)...

Algorithm PARTITION is a recursive implementation which partitions the input set of N blocks as required. The input to the first pass of PARTITION is the set of N blocks which is partitioned into 2 sets of N/2 blocks each. On the second pass, the input is 2 sets of N/2 blocks each which are partitioned into 4 sets of N/4 blocks each. If 2ⁱ partitions are required, PARTITION terminates when the size of each final partition is 2ⁿ⁻ⁱ.

Enhanced Partitioning Algorithm: EDPSMF In EDPMF the median find operation performed processes all the elements in each block and no useful information about the range of the data in a block is retained to be used for subsequent median finds on the same block. Initially, each of the N blocks is sorted using a standard internal sorting algorithm. Algorithms TRUE_MEDIAN and PARTITION remain unchanged. As the input blocks are sorted, the median of 2 half blocks is obtained directly by a simple scan...
merge of the sorted half blocks. Also, the blocks remain sorted for processing in subsequent passes over the data. As against a total cost of \(4N_T\) comparisons for regular MedFind partitioning, initially sorting all the input blocks reduces the number of comparisons required for successive MedFind operation to only \(N_T\) operations. We analyze the impact of this offset of costs in Section 5.

**Sorting Based Partitioning:** It is clear that an external sorting technique is a viable solution to the problem of determining the required number of partition boundaries. The advantage of this technique, **EDFSORT**, is that there is no restriction on the number of partitions that can be generated. External sorting can be used to generate any number of partitions, for arbitrary \(p\) whereas algorithms **EDPMF** and **EDPSMF**, as described so far, can only be used to generate \(p = 2^j\) disjoint partitions, for arbitrary \(j\). We will later describe techniques to extend **EDPMF** and **EDPSMF** to generate arbitrary number of partitions, where \(2^{j-1} \leq p \leq 2^j\), with some additional I/O and CPU costs.

### 4.4 Partitioning with \(2^k\) buffers

We now discuss the partitioning algorithms when \(2^k\), \(k > 0\), input buffers are available. With \(2^k\) buffers available, upto \(2^k\) blocks of data can be processed in memory at one time. The unit median find operation, MedFind, is appropriately modified to find the median of a chunk \(2^k\) blocks of data at one time. Knowing the chunk median, the input blocks are separated into 2 disjoint half chunks of \(2^{k-1}\) blocks each. The separated half chunks are then written back to disk.

The filtering process remains essentially the same as before but for the fact that the number of the input data chunks is reduced by a factor of \(2\). The number of chunks input to the \(i\)th call of **TRUE-MEDIAN** is \(2^n-k-i-1\). **FILTER** terminates after \(n-k\) calls to **TRUE-MEDIAN** after identifying the true median of the \(2^n-k\) input chunks. Algorithm **PARTITION** remains essentially unchanged. Partitions are generated recursively. If \(2^k\) partitions are required and \(1 \leq i \leq n-k\), **PARTITION** terminates after \(i\) recursive calls when the size of each partition is \(2^n-i\) blocks.

If \(i > n-k\), **PARTITION** terminates after \(n-k\) recursive calls which generate \(2^n-k\) partitions each of size \(2^k\) blocks. Then, each of the partitions can be directly read into the main memory for further partitioning.

### 5 Cost Analysis

In this section we discuss the performance of the different partition algorithms and the associated I/O and CPU costs.

#### 5.1 Algorithm EDPMF

We analyze the costs of determining the true median of the \(N = 2^n\) input blocks. We present the results for the case of 1 available buffer and then for the case of \(2^k\) available buffers.

**1 available buffer:** Given a partition of \(N\) blocks, generate 2 disjoint partitions of size \(N/2\) blocks each, i.e. we wish to perform \((1,N) \Rightarrow (2,N/2)\). The unit I/O operation is R disk access of a block. The unit CPU operation is MedFind on 1 block of data represented as \(MF_1\). (DAs represent disk accesses.)

<table>
<thead>
<tr>
<th>Step</th>
<th>(MF_1)'s</th>
<th>DAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read in (N) blocks, (1) block at a time.</td>
<td>(N)</td>
<td>(N)</td>
</tr>
<tr>
<td>MedFind of each block</td>
<td>(N)</td>
<td>(N)</td>
</tr>
<tr>
<td>Write out (N) blocks to disk</td>
<td>(N)</td>
<td>(N)</td>
</tr>
</tbody>
</table>

\[ i = N/2 \]

while \((i \geq 1)\) {

- \(i \) blocks:
  - Read in 2 half blocks at a time \((2 \cdot i)\)
  - MedFind of 1 block \(i\)
  - Write out 2 half blocks to disk \((2 \cdot i)\)

\[ (i = i/2) \]

From above,

\[ C_{I/O} = 6N - 4\], disk accesses.

\[ C_{CPU} = 2N - 1, MF_1\] operations.

**Generating \(2^k\) partitions:** The process of recursively generating \(2^k\) partitions can be represented by the sequence

\[(1,N) \Rightarrow (2,N/2) \Rightarrow \cdots \Rightarrow (2^k, N/2^k)\]

Here, \(1 \leq i \leq n - 1\). The associated costs are:

\[ C_{I/O} = 6Ni - 2^{i+2} + 4\] disk accesses.

\[ C_{CPU} = 2Ni - 2^i + 1\] \(MF_1\) operations.

For complete partitioning,

\[ C_{I/O} = 6N \log_2 N - 4N + 4\]

\[ C_{CPU} = 2N \log_2 N - N + 1\]

**\(2^k\) available buffers:** Given 1 partition of \(N = 2^n\) blocks generate 2 disjoint partitions of size \(N/2\) blocks each with \(2^k\) available buffers of main memory, i.e. we wish to perform \((1,N) \Rightarrow (2,N/2)\) with \(2^k\) buffers. The partitioning algorithm operates on chunks of size \(2^k\) blocks each. The unit I/O operation is disk access of a block. The unit CPU operation is MedFind on \(2^k\) blocks of data, represented as \(MF_{2^k}\), which also includes the separation of the chunk into 2 disjoint half chunks. (DAs represent disk accesses.)
above,

\[ C_{I/O} = 4N - 2.2^k \] disk accesses.
\[ C_{CPU} = 2^{n+1-k} - 1 \] \(M_{F_{2^k}}\) operations.

**Generating 2^i partitions:** PARTITION is used to recursively generate \(2^i\) partitions, where \(1 \leq i \leq n - k\). When \(i = n - k\), each partition size is \(2^k\) blocks and further processing can be done in main memory.

\[ C_{I/O} = 4N_i - 2.2^{k+i} + 2.2^k \] disk accesses.
\[ C_{CPU} = [N_i/2^{k-1} - 2^i + 1] \] \(M_{F_{2^k}}\) operations.

For complete partitioning, \(i = n\), initially \(2^{n-k}\) partitions of size \(2^k\) blocks each are generated. Then each of these partitions is read into main memory and completely partitioned. The associated costs are:

\[ C_{I/O} = 4N(n - k) + 2.2^k \]
\[ C_{CPU} = \left[ \frac{2N(n - k)}{2^k} - 2^{n-k} + 1 \right] \] \(M_{F_{2^k}}\) +
\[ \left[ 2^{n-k} \right] \] \(M_{F_{2^k}}\)

where, \(M_{F_{2^k}}\) is complete partitioning of chunk of \(2^k\) blocks using available \(2^k\) buffers.

### 5.2 Algorithm EDPSMFSF

For every pass of TRUE.MEDIAN, all the input blocks are partially sorted. The number of passes made over the data remain the same as for EDPMF. So, the I/O costs remain the same. An additional initial cost of sorting the \(N\) input blocks is incurred which is offset by the reduction in the cost for the MedFind operation implemented as a simple scan merge operation. As both EDPMF and EDPSMFSF are basically identical, the number of MedFind operations required in EDPMF is equal to the number of scan merge operations required in EDPSMFSF.

**1 available buffer:** \((1, N) \leadsto (2, N/2)\).

\[ C_{I/O} = 6N - 4 \]
\[ C_{CPU} = [N] \text{SORT}_1 + [N - 1] \text{SCAN}_1 \]

**Generating 2^i partitions:** Consider the partitioning \((1, N/2) \Rightarrow (2, N/4)\). The median of the \(N/2\) blocks can be found in \([N/2] \text{SCAN}_1\) operations. The actual partitioning requires \([N/2 - 1] \text{SCAN}_1\) operations. The total number of operations required to find the true median of \(N/2\) blocks is \([N - 1] \text{SCAN}_1\) operations. Therefore, the total costs to generate \(2^i\) partitions are

\[ C_{I/O} = 6Ni - 2^{i+2} + 4 \]
\[ C_{CPU} = [N] \text{SORT}_1 + [2Ni - 2^i + 1] \text{SCAN}_1 \]

\(2^k\) available buffers: \((1, N) \leadsto (2, N/2)\)

\[ C_{I/O} = 4N - 2.2^k \]
\[ C_{CPU} = [N/2^k] \text{SORT}_{2^k} + [N - 2^k] \text{SCAN}_1 \]

**Generating 2^i partitions:** For stage 2 onwards, the chunk median of \(2^k\) blocks is found by scan merge of previously sorted half chunks. Consider the partitioning \((1, N/2) \Rightarrow (2, N/4)\). The median of the \((N/2)/2^k\) chunks requires \(N/2\) \text{SCAN}_1\) operations. This is followed by \((N/2 - 2^k)\) \text{SCAN}_1\) operations to find the median of the \(N/2\) blocks. Thus, the total CPU cost of determining the true median of \(N/2\) blocks, and for all following true median find operations, is

\[ C_{CPU} = [N/2] \text{SCAN}_1 + [N/2 - 2^k] \text{SCAN}_1 \]
\[ = [N - 2^k] \text{SCAN}_1 \]

For 2^i partitions, we have:

\[ C_{I/O} = 4Ni - 2.2^{k+i} + 2.2^k \]
\[ C_{CPU} = [N/2^k] \text{SORT}_{2^k} + [2Ni - 2^{i+k} + 2^k - N] \text{SCAN}_1 \]

For complete partitioning, we first create \(2^{n-k}\) partitions of size \(2^k\) blocks each. Each of these partitions is read into main memory. Each partition is made up of 2 sorted half chunks of \(2^{i-1}\) blocks each. A simple scan merge completely sorts the \(2^k\) blocks in the partition. In fact, when all the partitions are sorted in this way, the entire set of \(N\) blocks becomes sorted and we get sorted partitions.

\[ C_{I/O} = 2N\lg N - 2Nk + 2^k \]
\[ C_{CPU} = [N/2^k] \text{SORT}_{2^k} + [2N\lg N - 2Nk + 2^k - N] \text{SCAN}_1 \]

### 5.3 Algorithm EDPSORT

This technique is basically an implementation of m-way external sort merge algorithm. To implement this algorithm we need at least 2 input buffers and 1 output buffer. We only consider the case when \(2^k\) input buffers are available. The associated costs are:

\[ C_{I/O} = 2N + 2(N/k) \lg N \]
\[ C_{CPU} = [N/2^k] \text{SORT}_{2^k} + [(N/k) \lg N] \text{SCAN}_1 \]

Table 1 and Table 2 give an overall picture of the I/O and CPU costs incurred for the different partitioning techniques proposed.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$p$</th>
<th>$2^k = 1$</th>
<th>$2$</th>
<th>$2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDPMF</td>
<td>2</td>
<td>$6N - 4$</td>
<td>$4N - 4$</td>
<td>$4N - 2^{k+1}$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>2</td>
<td>$6N - 4$</td>
<td>$4N - 4$</td>
<td>$4N - 2^{k+1}$</td>
</tr>
<tr>
<td>EDPMF</td>
<td>$2^i$</td>
<td>$6Ni - 2^{i+2} + 4$</td>
<td>$4Ni - 2^{i+2} + 4$</td>
<td>$4Ni - 2^{2k+1} + 2^{2i+1}$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>$2^i$</td>
<td>$6Ni - 2^{i+2} + 4$</td>
<td>$4Ni - 2^{i+2} + 4$</td>
<td>$4Ni - 2^{2k+1} + 2^{2i+1}$</td>
</tr>
<tr>
<td>EDPMF</td>
<td>$N$</td>
<td>$6N \lg N - 4N + 4$</td>
<td>$4N \lg N - 4N + 4$</td>
<td>$4N \lg N - 4Nk + 2^{k+1}$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>$N$</td>
<td>$6N \lg N - 4N + 4$</td>
<td>$4N \lg N - 4N + 4$</td>
<td>$4N \lg N - 4Nk + 2^{k+1}$</td>
</tr>
<tr>
<td>EDPSORT</td>
<td>$N$</td>
<td>$-2.2N + 2.1N \lg N$</td>
<td>$-2.2N + 2.(N/k) \lg N$</td>
<td>$-2.2N + 2.(N/k) \lg N$</td>
</tr>
</tbody>
</table>

Table 1: Summary of I/O costs for different partitioning techniques

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDPMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPMF</td>
<td>$2^k$</td>
</tr>
<tr>
<td>EDPSMF</td>
<td>$2^k$</td>
</tr>
</tbody>
</table>

Table 2: Summary of CPU costs for different partitioning techniques
6 Performance Evaluation

We consider two distinct scenarios of generating $2^i$ partitions - with 1 input buffer and with $2^k$ input buffers. The algorithms are compared to determine the range of applicability for the factors involved. In some cases, the limiting conditions are not analytically tractable and we present general qualitative results on applicability of the different algorithms. The details of the following results are presented in [Gurajada90].

Comparison of I/O Costs: We compare the disk accesses required in EDPMF and EDPSORT to generate $2^i$ partitions with $2^k$ available buffers. We check for the condition on $i$ such that $I/O_{EDPMF} > I/O_{EDPSORT}$. We need to satisfy $(N/k)\log N > (2i - 1)N - 2^k(2^i - 1)$. If $i = 1$, $I/O_{EDPMF} < I/O_{EDPSORT}$. It is shown in [Gurajada90] that as $i \rightarrow n$, $I/O_{EDPMF} > I/O_{EDPSORT}$.

Depending on the values of $N$ and $k$, we can obtain explicit values of the range for $i$ in which EDPMF or EDPSORT is suited. Figures 5 and 6 are plots of the difference of the above inequality, for $N = 2^{10}$ and $N = 2^{15}$, respectively, plotted versus $k$ for a family of $i$. It can be observed from the curves that as $i \rightarrow n$, providing any number of buffers to EDPMF will not improve the performance over EDPSORT. For the case of complete partitioning, as $k \leq n - 1$, the I/O cost for EDPSORT is always lower.

Comparison of CPU costs: We first compare the CPU costs of generating $2^i$ partitions with $2^k$ buffers for EDPMF and EDPSMF. The condition for $CPU_{EDPMF} > CPU_{EDPSMF}$ is $3[2N-2^{i+k}+2^k]+N > K_N/N!g(2^kN_T)$. We show in [Gurajada90] that for the range $1 \leq i \leq n - k$ and with typical values for $K_N$ and $N_T$, the inequality holds true for small values of $N$. Figures 7 and 8 show the variation of the difference of the two expressions above as a function of $k$ over a family of values for $i$. It is clear that for large data sets and with $i \rightarrow n$, the CPU cost for EDPSMF is lower than that for EDPMF.
the required number of buffers are allocated. We now compare the cost of generating $2^i$ partitions with $2^k$ buffers for EDPSORT v/s EDPSMF. We wish to check for the condition on $i$ such that $CPU_{EDPSORT} > CPU_{EDPSMF}$ which is, $N/k \lg N > N(2i - 1) - 2^k(2^i - 1)$. This condition is identical to the condition on comparison of $I/O$ costs of EDPSORT v/s EDPSMF. It is interesting to note here that the difference in the number of $SCAN_1$ operations and $I/O$ operations for EDPSMF is exactly $N$ which is the number of initial sort operations performed. The conditions on $i$ for this inequality to be satisfied have been derived in [Gurjada90] and the result follows from before.

6.1 Summary of Performance Evaluation

- EDPSORT is cheapest technique of partitioning from $I/O$ and CPU costs when complete partitioning is required.
- $I/O$ costs for EDPMF and EDPSMF is lower than that for EDPSORT when the number of partitions required is small compared to the total number of disk blocks.
- For small number of partitions, the CPU costs are $EDPMF < EDPSMF < EDPSORT$.
- For large data sets and large number of partitions, the CPU costs are $EDPSORT < EDPSMF < EDPMF$.

The range over which these relations change can be determined from the values of $N$, $i$ and $k$.

6.2 Recommendations

SUPER-PART: The different partitioning algorithms can be incorporated as part of an overall strategy, SUPER-PART, to choose the appropriate partitioning technique. The number of input blocks, $N$, and the number of partitions required, $2^i$, will be user selected. SUPER-PART chooses the number of buffers to allocate to the task of partitioning. Buffer allocation can be made statically or dynamically during the execution of the different partitioning algorithms. Dynamic allocation of buffers can ensure full utilization. State Diagram: Figure 9 represents the possible paths that can be chosen to generate the required number of partitions, $N$ and $i$ are input to SUPER-PART which evaluates the $I/O$ and CPU costs for the different options by varying the number of input buffers in terms of $k$. Each state in the system is represented by the quadruplet $(2^n, 2^{n-z}, i, k)$ where,

$$
\begin{align*}
2^n & = \text{Total number of input blocks} \\
2^z & = \text{Number of input partitions} \\
2^{n-z} & = \text{Number of blocks in each partition} \\
2^i & = \text{Required number of partitions} \\
2^k & = \text{Number of buffers available}
\end{align*}
$$

The partitioning algorithms selected will be partly disk based and partly main memory based.

6.3 Generating arbitrary $p$ partitions

In this section we propose approaches to solve the problem of generating $2^{i-1} \leq p \leq 2^i$ partitions. Using EDPMF or EDPSMF: To the $N$ input blocks, add $[N - (N/p).2^{i-1}]$
dummy data blocks with \(+\infty\) key values. The total number of blocks becomes \(2N - (N/p)2^{i-1}\) which are then partitioned into \(2^i\) partitions. The first true MedFind generates 2 partitions of size \(N - (N/p)2^{i-2}\) blocks each. The first partition is further processed recursively to generate \(2^{i-1}\) partitions. All of these partitions are stored. The second partition will contribute a further \(2^{i-1}\) partitions of which only \(p - 2^{i-1}\) partitions are required. The remaining are dummy partitions which can be discarded. In fact, the algorithm can be optimized to skip the partitioning of any dummy blocks of data. In this way, with some additional processing cost, the required number of \(p\) partitions can be generated. One of EDPMF and EDPSMF can be used as the partitioning algorithm. The additional cost involved can be up to two times the cost of generating \(2^{i-1}\) partitions. This needs to be further studied and analyzed.

Using EDPSMF: Another approach requires the use of EDPSMF only. For the given \(N\) blocks, generate \(2^{i-1}\) partitions, which are partially sorted. From these, we need to generate \(p = 2^{i-1}\) additional partitions. These are created by decreasing the size of the \(2^{i-1}\) partitions by shifting the partition boundaries. The size of each partition has to be reduced by \(N/p - 2^{i-1}\) tuples which can be done by a merge sort of the blocks in each partition. If \(i - 1 = k\), the partition sizes can be reduced as required with a total of \(N\) SCAN operations and \(N\) disk accesses. In other cases, the costs have to be further analyzed as the shifting of the boundary requires \(i - 1\) passes over each partition. The additional benefit of this approach is that all the \(p\) partitions that result are completely sorted.

7 Conclusions

In this report, we have studied the problem of generating equidepth partitions of a secondary memory resident data set. EDPs are useful for data analysis on subsets of the entire data set. EDPs are also required to generate equidepth histograms which can then be used to estimate the selectivities of query predicates. It has to be noted that using histograms to estimate selectivities are applicable to static data sets. To update and maintain the partitions and the histograms on a dynamic basis requires extra overhead. We have presented two new partitioning techniques based on the median find algorithm. An extensive evaluation of the CPU and I/O costs was performed and compared with an external sorting based partitioning approach. The proposed techniques, EDPMF and EDPSMF, are cheaper than EDPSORT for generating small number of partitions for a data set which is large in comparison to the size of available main memory. The proposed methods do not require any sampling of the data set and accurate histograms can be generated from the partitions. We also proposed an overall strategy, SUPER-PART, which can be used to select the least cost partitioning technique based on the number of partitions required and available buffers, along with dynamic buffer allocation. The algorithms EDPMF and EDPSMF were extended to permit the generation of arbitrary \(p\) number of partitions.

It is also possible to parallelize the operation of the partitioning algorithms. At last, two kinds of parallelization are possible. One, the recursive partitioning of the input blocks can be done in parallel. Secondly, the median find operation of a set of blocks can be parallelized. The cost analysis presented earlier simply considers the I/O and CPU costs independently. In reality, some degree of load balancing can be performed. Load balancing can also be used to set a limit on the maximum size of blocks input to the parallel finding algorithm. This could be another way to control the number of buffers that are allocated to the algorithms. The effect of parallel execution of the algorithms along with load balancing requires further detailed analysis.

References

Comer79 Comer, Douglas, "The Ubiquitous B-Tree", Computing Surveys, 11,2 (June 1979), 121-137.
Olken89 Olken, F and Rotem D., "Random sampling from B* Trees", Proc. 15th International Conf. on VLDB, (1989), 269-278.