Divide, Compress and Conquer: Querying XML via Partitioned Path-Based Compressed Data Blocks

Wilfred Ng Ho-Lam Lau
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong
{wilfred, lauhl}@cs.ust.hk

Aoying Zhou
Department of Computer Science and Engineering
Fudan University, Shanghai
ayzhou@fudan.edu.cn

Abstract

We propose a novel Partition Path-Based (PPB) grouping strategy to store compressed XML data in a stream of blocks. In addition, we employ a minimal indexing scheme called Block Statistic Signature (BSS) on the compressed data, which is a simple but effective technique to support evaluation of selection and aggregate XPath queries of the compressed data. We present a formal analysis and empirical study of these techniques. The BSS indexing is first extended into effective Cluster Statistic Signature (CSS) and Multiple-Cluster Statistic Signature (MSS) indexing by establishing more layers of indexes. We analyze how the response time is affected by various parameters involved in our compression strategy such as the data stream block size, the number of cluster layers, and the query selectivity. We also gain further insight about the compression and querying performance by studying the optimal block size in a stream, which leads to the minimum processing cost for queries. The cost model analysis provides a solid foundation for predicting the querying performance. Finally, we demonstrate that our PPB grouping and indexing strategies are not only efficient enough to support path-based selection and aggregate queries of the compressed XML data, but they also require relatively low computation time and storage space when compared with other state-of-the-art compression strategies.

Index Terms: Data Compression, Query processing, Cost Model, Markup Languages

1 Introduction

XML is, by nature verbose, since repeated tags and structures are needed to specify element items in the document. As a result, XML data are larger in size than the original data format.

For example, the file size of the Weblog obtained from [24] expands by three times when its log format is converted to XML. The *size problem* of XML documents hinders their practical usage, since the size substantially increases the costs of storing, processing and exchanging data.

In this paper, we propose a Partition Path-Based (PPB) data grouping strategy to address the size problem. The term "stream" used in our subsequent discussion originated from our XCQ compression methodology proposed in [12]. In a nut shell, given an XML document, we achieve the compression using a *DTD Tree and SAX Event Stream Parsing* (DSP) technique. This DSP algorithm takes as input the DTD tree and the SAX event stream created by the DTD Tree Building module and the SAX Parser module, respectively. The objectives of the DSP algorithm are to extract the structural information [18] from the input XML document that cannot be inferred from the DTD during the parsing process, and to group data elements in the document based on their corresponding tree paths in the DTD tree.

By structural information, we mean the information necessary to reconstruct the tree structure of the XML document. By data elements, we mean the attributes and PCDATA within the document. The output of the DSP algorithm is a stream of structural information, which we call the structure stream, and streams of XML data elements, which we call the data streams. The generated structure stream and data streams are compressed and packed into a single file. (cf. a full explanation of the DSP algorithm and a detailed example of generating the structure and data streams can be found in Sections 3.2 and 3.3 in [12].)

In Figure 1, we show that the structure stream derived from a given DTD is stored and compressed separately from the data streams. Each PPB data stream corresponds to a particular path structure in the DTD labeled by a key, and all data elements in the data stream are the elements that match the path structure. In addition, each PPB data stream is partitioned into its set of data blocks with a pre-defined block size, which helps to increase the overall compression ratio (cf. [9, 10, 16, 17]). A data block in a PPB data stream is able to be compressed or decompressed as an individual unit. This partitioning strategy allows us to access the data in a compressed document by decompressing only those data blocks that contain the data elements relevant to the imposed query. We term this a *partial decompression* strategy.

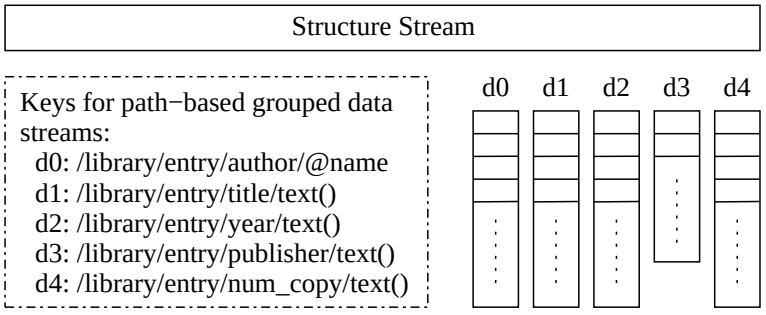


Figure 1: A Conceptual Diagram of Partitioned Path-Based Data Grouping

For example, if the block size has n records per block, the first batch of n records in the

stream, d_0, are packed in the first data block while the next batch of n records are packed in the subsequent block. As such, each data block in the data stream contains a certain number of elements in the order listed in their corresponding data streams, which are under the same tree path in the DTD tree as shown in Figure 2, where `name` is an *attribute node*; `author`, `title`, `year`, `publisher` and `num_copy` are *element nodes*; and `paper`, `course_notes` and `book` are *empty* elements.

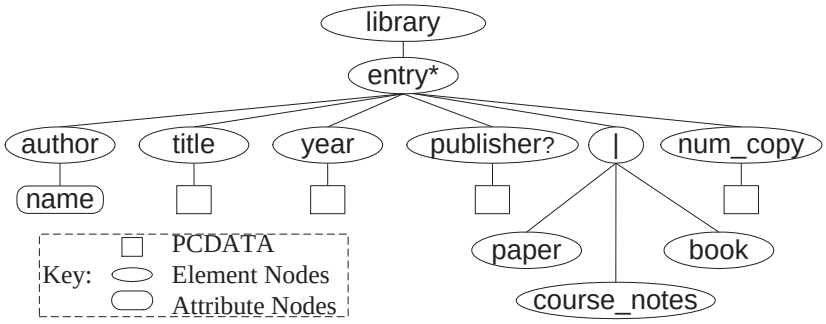


Figure 2: A Library DTD XML Tree

The data blocks in the data streams are compressed individually using Gzip [23]. The low-level compressor can be replaced by another generic one such as Bzip2 library as reported in [12], which could gain a better compression ratio at the expense of compression time. The underlying idea of query evaluation is that, by utilizing the structure stream, we only need to decompress *on the fly* the data streams that are relevant to the query evaluation. This *partial decompression* accessing strategy significantly reduces the decompression delay and storage overhead and is similar in spirit to some recent XML compressors [21, 15, 4, 1]. We do not present the query evaluation algorithms and the technical details of implementation for our query processor as they are described in another submitted paper.

Some preliminary ideas about the query evaluation strategy were highlighted in our earlier study [12], in which the PPB engine was developed to support the logical operators of core XPath queries: "*not*", "*or*" and "*and*"; along with the comparison operators: "$=$", "$\neq$", "$>$", and "$<$". The PPB engine also supports "*CONTAINS*" (i.e., for a substring) and "*STARTS-WITH*" for strings. We call this class of path-based queries collectively the *selection queries*. The PPB engine also implements the core library functions: "*COUNT*", "*SUM*", "*AVG*", "*MIN*" and "*MAX*", which are standard aggregation operators. These aggregation operators can be embodied in the core XPath queries as predicates. We call this class of path-based queries collectively the *aggregate queries*.

For the sake of clarity, we only illustrate how and why PPB is useful to tackle single path expressions that consist of at most one occurrence of a *descendant axis* "$//$", since such expressions are fundamental to current XML query languages such as XPath and XQuery [28, 29]. Nevertheless, the efficient support of complex path expressions (i.e., more than one descendant axis or having branch expressions in a path) over XML data can be naturally

extended with some decomposition and execution plan. For example, a complex path expression, "$p_1[/p_2]//p_3$", where p_1, p_2 and p_3 are single path expressions, can be supported as follows. We can first decompose the expression into "p_1/p_2" and "$p_1//p_3$". Then the query processor will first retrieve those results that satisfy p_1 and evaluate p_3 among the descendants of the result from p_1 as the first set of immediate answers. The second set of immediate answers is only an execution of a single path expression "p_1/p_2". The final result can be obtained by performing the set intersection or some sophisticated join techniques as illustrated in existing XML query evaluation techniques [1, 2, 15]. In principle, it would also be straightforward to modify the PPB engine to handle IDREF or IDREFS in DTDs by building a DTD graph rather than a DTD tree in the structure stream. When parsing a document against the DTD graph, the graph would still be traversed in depth-first order in order to determine the correct data stream.

The main contributions of this paper are as follows. First, we clarify how the evaluation of an important class of XPath queries (selection and aggregate queries) is affected by various parameters involved in the compression strategy, such as the data stream block size, the number of cluster layers, the cost of scanning the indexes, the cost of decompressing a block, and the query selectivity. Second, we establish a solid foundation on which to optimize the block size in a stream, which leads to the minimum processing cost for a query. Finally, we verify our results by carrying out an extensive experimental study to test the performance of the PPB engine. Our empirical result presented in this paper demonstrates that our PPB engine prototype performs well when compared with two existing compressors, XMill and XGrind, and, importantly, our PPB cost model predicts the trend of the query response time correctly. To the best of our knowledge, our PPB cost model is the first analytical model of the cost of XML compression.

The remainder of this paper is organized as follows. The rest of this section is devoted to a description of related work. Section 2 presents the Block Statistic Signature (BSS) indexing scheme, which is adopted in the PPB system to aid in partial decompression. Section 3 establishes the cost model for the PPB data grouping and presents a preliminary analysis of the optimal query processing cost. Section 4 studies the impacts of varying the selectivity and block size on the processing cost for path-based selection and aggregate queries. Section 5 improves BSS by introducing a level of clustering of the blocks and the Cluster Statistic Signature (CSS) indexing scheme in a data stream. We then further generalize CSS by considering cases when there is more than one level of clustering, which we term the Multiple-cluster Statistic Signature (MSS). Section 6 uses real XML datasets to verify various predictions from our model. Finally, we offer concluding remarks in Section 7.

1.1 Related Work

The motivation for supporting direct querying compressed XML documents is in related to studies on traditional data compression [19, 5, 30]. We are able to save in bandwidth and achieve more efficient query processing over compressed data, due to the fact that more information can be carried by a given data size. In addition, compression techniques are particularly important in dealing with the verbosity of XML files. Most well-known XML-conscious compression technologies have been recently proposed (cf. see our recent survey in [13]). These technologies are able to achieve good compression performance. They include XMill and Millau [10, 20]. However, these systems are *solely* optimized for better compression but do not support querying of the compressed data. The more recently proposed XML compressors, such as XGrind [21], XPress [15], skeleton compression [2], XCQ [12], XQzip[4], and XQuec [1] (all these compressors except XGrind do not have released code) are able to support querying. However, their foundations are not sufficiently adequate from the analytical point of view. In particular, various underlying cost factors involved in processing queries over compressed data have not yet been introduced in these compressors. It is worth mentioning that XGrind and XPress are able to encode XML data items individually and to preserve the XML structure. Thus, both compressors possess the desirable feature that their querying operations can be executed without fully decompressing the document. However, the compression ratio and time performance of XMill is much better than XGrind and XPress (see Figures 12 and 13 in [21]).

2 BSS Indexing in PPB Data Streams

In this section, we describe the Block Statistic Signature (BSS) indexing scheme used in the PPB engine. The indexing scheme is designed for supporting data values from numerical or string domains and is minimal in the sense that it requires very small amounts of storage space and time resources in the PPB engine. This indexing scheme simplifies *signature file indexing* approaches [7, 11, 6]. Like *projection signature indexing* in [6], BSS indexing is used to index block-oriented compressed data.

Definition 2.1 (BSS Index and Value Range) Assume that there are n blocks. The *Block Statistic Signature* (BSS) index of an ith block is given by $B_i = \langle s_i^\beta, b_i^\beta \rangle$, where b_i^β represents the data items in the compressed block and the BSS index value, $s_i^\beta = \langle min(b_i^\beta), max(b_i^\beta), sum(b_i^\beta), count(b_i^\beta) \rangle$. We define the *value range* for a BSS indexed block, B_i, denoted as l_i^β, as $\langle min(b_i^\beta), max(b_i^\beta) \rangle$. From now on, we use more handy notation such as s_β, b_β and l_β when there is no ambiguity in the block labelling.

Figure 3 depicts the underlying idea of the BSS indexing scheme. In the PPB engine, a *statistical signature* is generated for each compressed data block for a document. The signature summarizes the data values inside a particular block. When a query is being evaluated,

the compressed data blocks in the corresponding data streams are accessed by the PPB engine. A *filtering process* is carried out by the PPB engine as follows. Before a data block is fetched from the disk, the PPB engine consults the corresponding BSS index and ignores those data blocks that do not contain the required record(s). To do this, the PPB engine checks the BSS signature of the data block and decides whether the value range of that block overlaps with the value range specified in the *interval query*. If these two ranges overlap, which means that the data block *may* contain the required record(s), then the data block is fetched and decompressed for evaluation. If the two ranges do not overlap, the block does *not* contain the required records, and in this case the data block is not fetched.

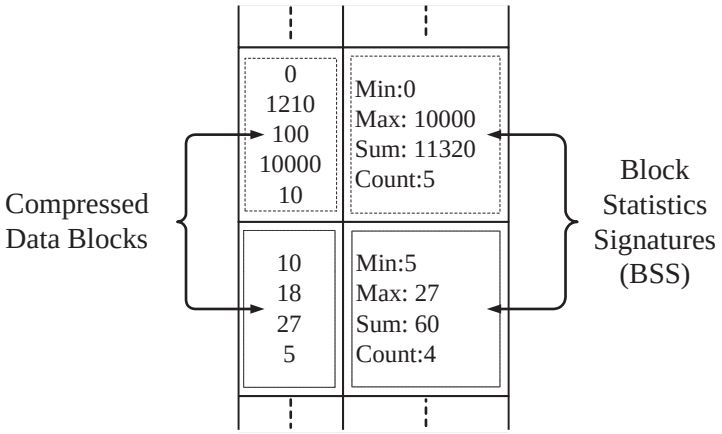


Figure 3: BSS Indexes in a PPB Data Stream

If more statistical information is stored in the block signature, we may have the benefit of having better knowledge about the compressed data block content, which helps in the block scanning and filtering processes during query evaluation. However, this additional information requires additional resources to generate, scan and store the block signatures, which will have a negative effect on the compression time, the query response time and the compression ratio. The BSS indexing scheme requires low computation overhead. It is easy to compute and generate the indexes in the compression phase and it is quick to scan the indexes during query processing. The operations of generating and scanning a signature can be done in linear time, $O(n)$. In the case of *signature generating*, *n* is the number of elements in the data stream. In the case of *signature scanning*, *n* is the number of compressed data blocks in the PPB data stream. We consider the following *selection query* of a compressed XML document that conforms to the DTD given in Figure 2 (we skip the full path of the queries for simplicity,):

(Q_1): entry[author/@name="Lam" and publisher ="X Ltd."].

This query, Q_1, selects those XML fragments that satisfy the two given predicates. We define the *selectivity* as the percentage of XML fragments (e.g., the entry fragments in Figure 2) returned in the query result compared to the total number of XML fragments in the

compressed document. Two data streams, d_0 and d_3, are involved during the process of query evaluation. The PPB engine first returns the XML fragments that have matched "entry" elements. Assuming there are m records satisfying the first predicate, the PPB engine then needs to find the blocks in d_3 that contain the corresponding m "publisher" records to evaluate the second predicate. The structure stream directs the engine to these two streams. The query predicates can be *Exact Matching* (equality comparison only) or *Range Matching* (inequality comparison involved).

Consider also the following *aggregate query* that finds the sum of (paper, book or course notes) copies based on the data elements that satisfy some specified predicates inside a *single* data stream. The query aims at finding aggregate information related to a particular path.

(Q_2): SUM(//num_copy $\geq$2).

This query, Q_2, selects only those elements in the data stream, d_4. The PPB engine needs to find the data elements that satisfy the predicate "num_copy $\geq$2" and then return the sum of the num_copy. The BSS index in d_4 is scanned in the filtering process and those data blocks that contain the data elements are fetched from the disk. This helps the PPB engine to avoid fetching the compressed data blocks that do not contain relevant records or the statistical parameters in the signature can be used directly to evaluate the query.

3 PPB Data Grouping Analysis

In this section, we study the process of checking data elements in the PPB data grouping against the selection predicate of a given query. This paves the way to study the optimization of the PPB engine to run selective and aggregative queries.

We begin by discussing the cost of running a selection query that has a simple *range-matching*, open-interval predicate, such as entry[num_copy $>$ 1]. We then extend the basic result to other queries. We now assume that the PPB data stream is finite with N data elements, each of which has a fixed and uniform probability, P, of being selected by the query predicate and there is only a single predicate that selects data elements from the range l_p.

Definition 3.1 (BSS Hit and Missed Blocks) We compare l_i^β of a compressed block, B_i, in a BSS indexed PPB data stream, $\langle B_1, \ldots, B_n \rangle$, with l_p in a query and check if these two intervals overlap (i.e., when $l_i^\beta \cap l_p \neq \emptyset$). If they overlap, we say that a ***BSS hit*** occurs at block B_i (or simply B_i is a *hit block*). A ***BSS miss*** occurs at block B_i (or simply B_i is a *missed block*) if they do not overlap. A block is decompressed for further comparison of elements with l_p iff it is hit.

We make the following assumptions to establish cost equations related to query processing.

1. The cost of scanning the value range of a block, B (which includes checking l_β against l_p), denoted as C_s, is relatively small compared with the cost of decompressing the block.

2. We assume that C_s is constant and independent of the block size. We also assume that the average cost of decompressing a data element in a block, denoted as C_d, is also constant and independent of the block size.

3. Elements in a block have the same inherent probability, P, of satisfying the predicate specified in a given query.

The first assumption is needed because decompressing a block is a costly operation as the whole block is fetched from the disk into the main memory. The glossary used in our model is given in Table 1.

Notation	**Definition**
l_p	The range specified in a query predicate.
π	The hitting probability of a compressed block in a PPB data stream.
N	Number of data elements in a data stream.
P	Probability that a data element in a stream satisfies the predicate.
K	Number of data elements in a block of a data stream.
L	Number of blocks in a data stream specified by a PPB path.
X	Number of blocks being hit (i.e., l_β and l_p overlap in these blocks).
C_q	Total cost of processing a query q.
C_s	Cost of scanning a BSS range of a block.
C_d	Average cost of decompressing a data element.
Y	Unit processing cost of processing a query.

Table 1: Notation Used in the Cost Processing Analysis

As N data elements with the same semantics in a PPB data stream are divided into L compressed data blocks, we have $K = \frac{N}{L}$ data elements in each block. The probability that all the elements in a block do *not* fall within l_p is equal to $(1-P)^K$, where P is the selectivity of the predicate. It follows that the probability that some blocks are hit is equal to $(1-(1-P)^K)$. Let us call $\pi = 1-(1-P)^K$ the *hitting probability* of a compressed data block in a PPB data stream (or simply the *block hitting probability*). The distribution that X blocks are hit satisfies a *binomial distribution* as follows.

The probability that X blocks are hit in a stream $= C_{L,X} \cdot \pi^X (1-\pi)^{L-X}$, where $C_{L,X} = \frac{L!}{X!(L-X)!}$ and $X \in \{0, 1, \ldots, L\}$ for some positive integer L.

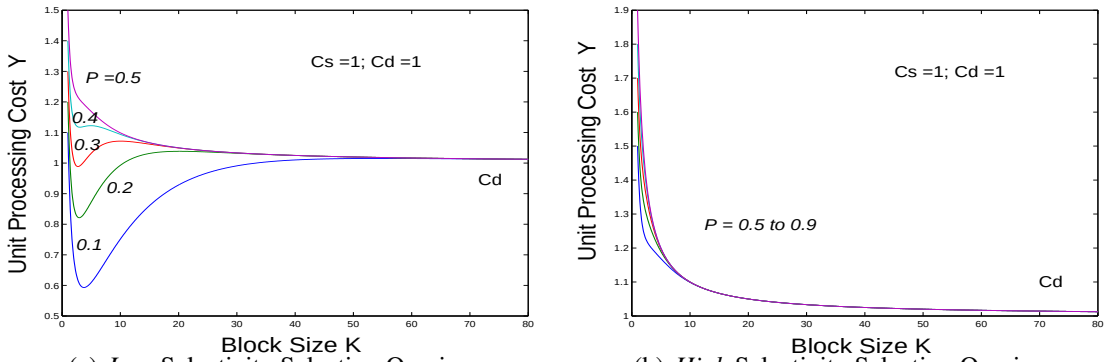
As the *mean* of the distribution is equal to πL, we have the expected number of hit blocks, $E(X) = \pi L$, which are decompressed for further checking of the data elements against l_p.

We ignore the cost of checking the decompressed items that are already fetched into the main memory. Thus, the total cost of processing a query in the data streams is equal to the sum of the costs of scanning the BSS indexes in a data stream and decompressing X blocks (i.e., $L \cdot C_s + K \cdot X \cdot C_d$). The expected cost of running the selection query is given by $E(C_q) = L \cdot C_s + K \cdot E(X) \cdot C_d$. Let us call the expression $Y_q = \frac{E(C_q)}{N}$ (or simply Y if q is clear in the context) the *unit processing cost with respect to the query, q*.

We now give the following formula that expresses Y in terms of the block size, K, and the selectivity, P:

$$Y = \frac{C_s}{K} + \left(1 - (1-P)^K\right) \cdot C_d. \tag{1}$$

In Figures 4(a) and 4(b), we sketch the unit processing cost function against the block size under different selectivity values, in which we assume $C_s = C_d = 1$ in plotting the charts for the sake of easy illustration. These charts help to give further insights for developing our cost model. Note that if the block size is large, then there are virtually no missed blocks (i.e., we need to decompress the *whole* document) and thus the unit processing cost function converges to the unit decompressing cost, C_d (i.e., as shown in Figure 4(a) $K \to \infty, Y \to C_d$ in our case). We can see that the cost function can take three values, depending on the selectivity values. First, the curves for high selectivity values in Figure 4(b) are generally decreasing (e.g., $P \geq 0.5$) and, in these cases, the larger the block size, the lower the unit processing cost. There is also comparatively little difference in these curves. Second, the curves for low selectivity values have a local minimum (e.g., $P = 0.4$) but the value may still be higher than the asymptotic value, C_d. Third, the curves for sufficiently low selectivity values have a global minimum (e.g., $P = 0.2$) (i.e., the minimum value is smaller than C_d).



(a) *Low* Selectivity Selection Queries (b) *High* Selectivity Selection Queries

Figure 4: Cost Analysis of the Unit Processing Cost for Selection Queries

We formally show that there exists an optimal query processing cost if and only if the selectivity value is within a certain threshold. The theorem helps to clarify the unit cost processing function shown in Figures 4(a) and 4(b).

Theorem 3.1 The following statements are true for the unit processing cost, Y.

1. Y has a *local* minimum if and only if $P \leq 1 - e^{-\frac{4C_d}{C_s}e^{-2}}$.

2. Y has a *global* minimum if and only if $P \leq 1 - e^{-\frac{C_d}{C_s}e^{-1}}$.

Proof. We let $x = (1 - P)$ where $0 \leq x \leq 1$ and we have the unit processing cost function as follows:

$$Y = \frac{C_s}{K} + (1 - x^K)C_d. \tag{2}$$

In order to establish the result in Part 1, we proceed to find the extreme points of Y by differentiating it with respect to K. We then set the derivative to zero and thus have the equation:

$$\begin{aligned} \frac{dY}{dK} &= -\frac{C_s}{K^2} - x^K C_d \ln x \\ &= 0. \end{aligned} \tag{3}$$

It follows from Equation 3 that we have

$$K^2 x^K = \frac{-C_s}{C_d \ln x}. \tag{4}$$

Note that the parameter K exists only on the right-hand side. In order to solve Equation 3, we define a function on K by $F(K) = K^2 x^K$ and find the maximum of F first. By solving the equation $\frac{dF}{dK} = 2Kx^K + K^2 x^K lnx = 0$, we obtain the (unique) extreme value of F at $K = K_F = \frac{-2}{\ln x}$. We conclude that K_F is at the global maximum point from the evidence $\frac{d^2F}{dK^2} = -2x^K < 0$ for all K. Therefore, there exists a root for x if and only if the following equation holds:

$$K_F^2 x^{K_F} \geq \frac{-C_s}{C_d \ln x}. \tag{5}$$

The intuition behind Equation 5 is that, given α and x, the horizontal line (with respect to K) $F' = \frac{-\alpha}{\ln x}$ cuts the curve F if and only if the line is lower than or equal to the maximum value of F. When the line is higher than $F(K_F)$, there is no intersection between F and F' and therefore no solution for Equation 3. By substituting $K_F = \frac{-2}{\ln x}$ into Equation 5, we then have $x \geq e^{-\frac{4C_d}{C_s}e^{-2}}$. It follows that $P \leq 1 - e^{\frac{-4C_d}{C_s}e^{-2}}$.

In order to prove Part 2, we need to compare the minimum value of Y in Equation 2 with the asymptotic value of Y when K is large (i.e., $Y_\infty = C_d$). We define

$$Y_{local\_min} = \frac{C_s}{K^*} + (1 - x^{K^*})C_d, \tag{6}$$

where K^* is the minimum point corresponding to Y_{local_min}.

If $Y_{local\_min} \leq C_d$, then Y_{local_min} is in fact the global minimum of Y occurring at some finite K value (cf. Figure 4(a)).

We therefore have the condition $Y_{local\_min} \leq C_d$. By Equation 6, we further simplify this condition and obtain the following equation:

$$\frac{C_s}{K^*} \leq C_d x^{K^*}. \tag{7}$$

Bear in mind that K^* is the root of Equation 3. By using Equations 4 and 7, we have the following condition:

$$-K^* \ln x \leq 1. \tag{8}$$

From Equation 8, we substitute K^* in Equation 4 and obtain the inequality condition $\ln\left(\frac{-C_s \ln x}{C_d}\right) \geq -1$. After simplifying the logarithmic terms, it follows that $P \leq 1 - e^{-\frac{C_d}{C_s}e^{-1}}$, which is a stricter condition than the counterpart of the local minimum, since we have $4 > e$. □

As illustrated in Figure 4(a), Y has the local minimum and the smaller the selectivity, the lower the local minimum. The local minimum is in fact the global one if it is smaller than C_d. This can be proved by using Equations 6 and 7 as follows:

$$\frac{dY_{local\_min}}{dP} = C_d(K^*(1-P)^{K^*-1}) > 0. \tag{9}$$

It follows from Equation 2 that when $K \to \infty$, we have $\frac{C_s}{K} + (1 - x^K)C_d \to C_d$, which means that Y can be brought arbitrarily close to C_d by choosing a sufficiently large K value. On the other hand, we find the expression of K that gives rise to the optimal Y for a sufficiently small K value in Corollary 3.2.

Corollary 3.2 Y has a global minimum at $K = \sqrt{\frac{C_s}{PC_d}}$ when P is sufficiently small.
Proof. From Theorem 3.1, it follows that the global minimum value of Y exists if P is sufficiently small. In this case, we have the approximation $(1 - x^K) \approx KP$. Using this approximation, we reduce Equation 1 to the following equation:

$$Y \approx \left(\frac{C_s}{K} + KPC_d\right). \tag{10}$$

We differentiate Equation 10 with respect to K, by which we are able to find the optimal Y value. So we have the following equation:

$$\frac{dY}{dK} = PC_d - \frac{C_s}{K^2} = 0. \tag{11}$$

Thus, Y has the extreme value at $K \approx \sqrt{\frac{C_s}{PC_d}}$, which is a global minimum, since $\frac{d^2Y}{dK^2} = \frac{2C_s}{K^3} > 0$ for all K. □

To summarize, we present three main findings that involve the unit processing cost, the block size, and the scanning and the unit decompressing cost in our analysis.

1. For low selectivity queries: if the selectivity satisfies $P \leq P_{local\_min} = 1 - e^{-\frac{4C_d}{C_s}e^{-2}}$ as shown in Part 1, Theorem 3.1, then the optimal unit processing cost, Y, occurs, in the case $K \approx \sqrt{\frac{C_s}{PC_d}}$ as shown in Corollary 3.2.

2. For high selectivity queries: if the selectivity satisfies $P > P_{global\_min} = 1 - e^{-\frac{C_d}{C_s}e^{-1}}$ as shown in Part 2, Theorem 3.1, the optimal unit processing cost, Y, approaches C_d as $K \rightarrow \infty$, as shown in Figure 4(b). In practice, this means that the larger the block size, the more efficient the query processing.

3. For sufficiently large K, the unit processing cost, Y, of a selection query approaches the unit decompressing cost, C_d, irrespective of the P values.

4 Processing Interval Queries

In this section, we discuss the hitting probabilities of selection and aggregate queries. We then present an analysis of their unit processing costs.

4.1 Modified Block Hitting Probability

We now consider the block hitting probability used in selection and aggregate queries with open or closed intervals as selective predicates.

Selection Queries. We classify the selection query intervals on data values specified by the selective predicate into two categories.

1. Open Interval Queries. The interval specified by the selection predicate has an open end. For example, the following query, which seeks the "byte count" in an compressed XMLised Weblog document [24], has an open interval used in the selection predicate:

   (Q_3): element[byteCount $>$ 1000].

2. Closed Interval Queries. The interval specified by the selection predicate has two closed ends. For example, the following query, which seeks the byte counts in an compressed XMLised Weblog document [24], has a closed interval used in the selection predicate.

   (Q_4): element[2000 $>$ byteCount $>$ 1000].

In Section 3 we simply assumed that hitting a block in a stream occurs when l_p and l_β overlap, which is given by the block hitting probability $\pi = 1 - (1 - P)^K$. The parameter π is essential for us to deduce the expected number of blocks to be decompressed in a stream. However, we have not tackled two issues. The first is if π is applicable to the open and closed intervals; the second is if π is applicable to the selection and aggregation queries.

We now consider the case of an open interval predicate. There are two cases of overlapping between l_β and l_p, as shown as Cases B and C in Figure 5. It is clear that these two cases happen if and only if there exist some data elements in a block falling in the interval l_p. In other words, we only need to *exclude* the probability of the event that no data element

falls in l_p (i.e., reject Case A), which is equal to $(1-P)^K$. Thus, our assumption of using $\pi = 1-(1-P)^K$ in Equation 1 is justified.

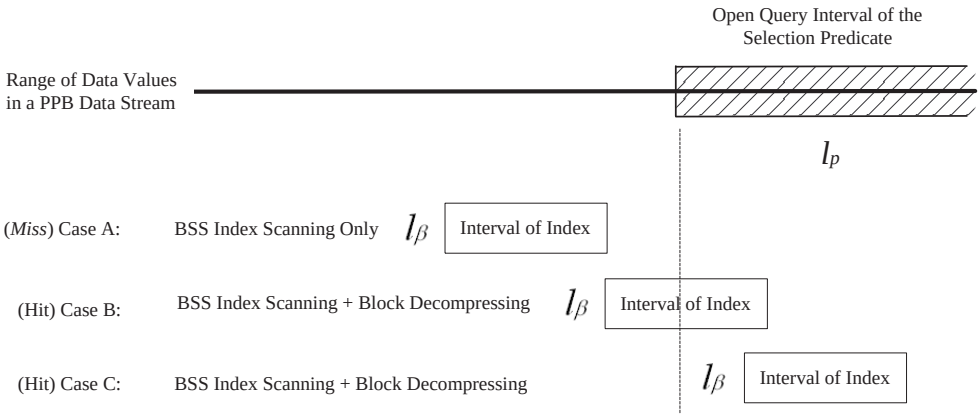


Figure 5: Overlapping BSS Index and Open Interval Selection Query

However, there are three cases of overlapping, Cases B, C and D, in the case of closed interval queries as shown in Figure 6. Notably, in Case D, it may happen that all the data elements in a block are not actually in the range of the interval l_p, but the block is still a hit. This is because, even when l_p and l_β overlap, we still cannot determine whether the data elements fall in l_p, l_a or l_b. In other words, the hitting parameter π does not reflect the outcomes in Case D.

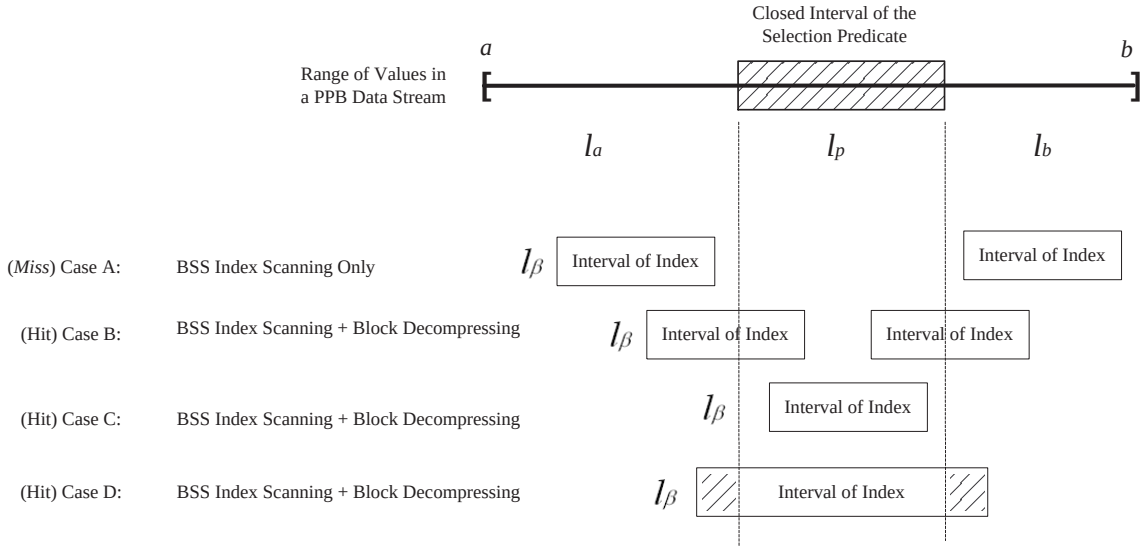


Figure 6: Overlapping BSS Index and Closed Query Intervals for Selection Queries

In order to modify the expression of π to π^*, we take into account the possibility that some data elements may fall in l_a or l_b in Case D as follows:

π^* = the probability of having at least one element in the interval of l_p (i.e., related to Cases B and C and (part of) D) + the probability of having at least one element at l_a but others at l_b (i.e., related to (part of) Case D) + the probability of having at least one element at l_b but others at l_a (i.e., related to (part of) Case D).

We assume that the elements in a PPB data stream are distributed with the probabilities P_a and P_b in the intervals of l_a and l_b, respectively. Thus, we have the following equation:

$$\pi^* = (1 - (1 - P)^K) + (1 - P)^K(1 - (1 - P_a)^K)(1 - (1 - P_b)^K). \quad (12)$$

Note that $P + P_a + P_b = 1$ in Equation 12. The probability values of P, P_a and P_b depend on the data distribution in the range $[a, b]$. If we assume that the probability for the distribution of the data elements is uniform in the range, then we have the simple ratio for the probabilities, $P_a = (1 - P)(\frac{l_a}{l_a+l_b})$ and $P_b = (1 - P)(\frac{l_b}{l_a+l_b})$.

Aggregate Queries. There is one fundamental difference between aggregate and selection queries in estimating the processing cost, C_q. It may not be necessary to decompress a hit block in a PPB data stream. For example, consider the following aggregate queries, which are modified from the selection queries Q_3 and Q_4:

(Q_5): element[COUNT(byteCount $>$ 1000)].

(Q_6): element[COUNT(2000 $>$ byteCount $>$ 1000)].

To process the above queries, we do not need to decompress the block if the comparison satisfies the condition that l_β is contained in l_p. We have to exclude the decompressing cost arising from Case C. We now modify the expression of π to π^* in order to take account of this savings in decompression.

In the case of an open interval query, we have the following hitting probability:

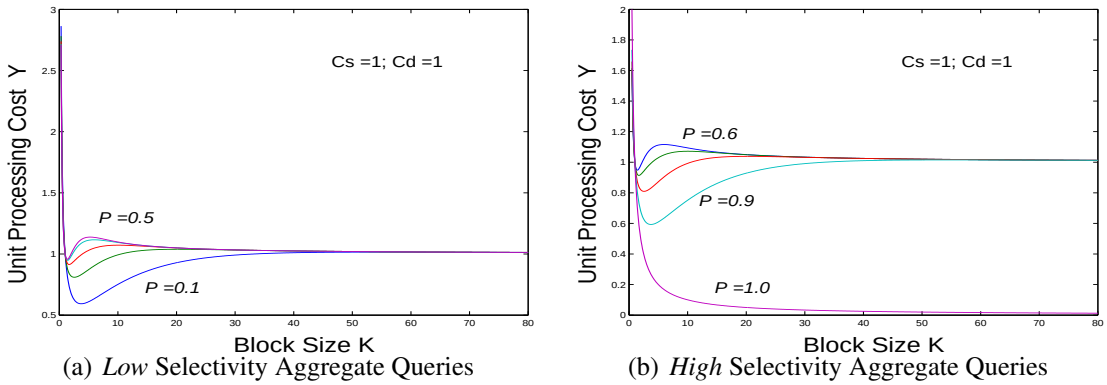
π^* = the probability of having at least one element in the interval of l_p (i.e., Cases B and C) $-$ the probability of having all elements in l_p (i.e., reject Case C).

Thus, we have the following equation:

$$\pi^* = (1 - (1 - P)^K) - P^K. \quad (13)$$

From Equation 13, we show in Figures 7(a) and 7(b) the unit processing cost of aggregate queries against block sizes when the selectivity values are low and high, respectively. The low selectivity case in Figure 7(a) is similar to its counterpart in the selection queries. However, the high selectivity case in Figure 7(b) also exhibits the minimum unit processing cost at a certain block size, which is in contrast to what we observe in Figure 4(b).

In the case of a closed interval query, we have the following hitting probability:



(a) *Low* Selectivity Aggregate Queries (b) *High* Selectivity Aggregate Queries

Figure 7: Cost Analysis of the Unit Processing Cost for Aggregate Queries

π^* = the probability of having at least one element in the interval of l_p (i.e., Cases B and C) $-$ the probability of having all elements in l_p (i.e., reject Case C) $+$ the probability of having at least one element at l_a but others at l_b (i.e., Case D) $+$ the probability of having at least one element at l_b but others at l_a (i.e., Case D).

Thus, we have the following equation:

$$\pi^* = (1-(1-P)^K) - P^K + (1-P)^K(1-(1-P_a)^K)(1-(1-\pi_b)^K). \quad (14)$$

We summarize the modified block hitting probabilities to be used in the unit cost equations in Table 2 for different query interval cases.

Query	**Selection**	**Aggregate**
Open Interval	$\pi = (1-(1-P)^K)$	Equation 13
Closed Interval	Equation 12	Equation 14

Table 2: Modified Block Hitting Probabilities

4.2 Processing Cost and Selectivity

For the sake of clarity, we first transform the unit processing cost equation for selection queries with an open interval from Equation 1 as follows,

$$Y = H_K - (1-P)^K C_d, \quad (15)$$

where $H_K = \left(\frac{C_s}{K} + C_d\right)$.

We differentiate Y with respect to P ($0 \leq P \leq 1$). At the extreme cases of $P = 0$ and 1, we have $Y = (H_K - C_d)$ and H_K, respectively. This result implies that our partitioning strategy favors low selectivity queries and the processing cost is bounded by H_k (when all the blocks are decompressed). The implication is reasonable, since in practice we do not

enjoy a processing cost benefit if we have to decompress too many blocks in a data stream, which happens in the cases of high selectivity selection queries. We show the unit processing cost function in relation to the selectivity of the selection queries in Figure 8(a).

We now study the relationship between the unit processing cost and selectivity for aggregation queries. The unit processing cost equation, in which we use the modified block hitting probability given in Equation 13, is:

$$Y = H_K - ((1-P)^K + P^K) \cdot C_d. \tag{16}$$

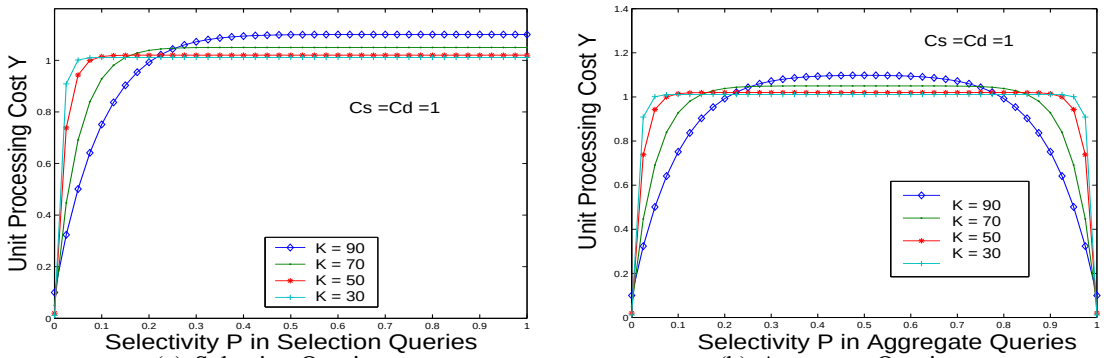
Similarly, we differentiate Y with respect to P and then set the corresponding derivative to zero. Now we have the following equation:

$$\frac{dY}{dP} = C_d \cdot K(1-P)^{K-1} - C_d K P^{K-1} = 0, \tag{17}$$

which gives the stationary point at $P = P_m = \frac{1}{2}$. Assuming that $K \gg 1$, which is reasonable in practice, we conclude that P_m is in fact a local maximum point, since we have

$$\left.\frac{d^2Y}{dP^2}\right|_{P=P_m} = -C_d \cdot K(1-P_m)^{K-2} - C_d K(K-1)P_m^{K-2} < 0. \tag{18}$$

By substituting P_m into Equation 16, we obtain the maximum value of $Y(P_m) = H_K - \frac{2C_d}{2^K}$. At the extreme cases of $P = 0$ and 1, we have the same value given by $Y = H_K - C_d$. We show the unit processing cost function Y in relation to the selectivity in Figure 8(b). The graph is remarkably different from that in Figure 8(a) in the region of high selectivity, since the cost decreases rapidly when the selectivity is high. The difference can be explained as follows. When a wider selectivity is given, then more data items are selected in a block; the chance that the l_p covers l_β becomes higher (i.e., the occurrence of Case C becomes more frequent than others). It is therefore less likely that the hit blocks are decompressed.



(a) Selection Queries (b) Aggregate Queries

Figure 8: Cost Analysis of the Unit Processing Cost against Selectivity

In a given workload, if the distribution of selectivities and queries can be estimated (e.g., from the past query log data), then similar differentiation can be straightforwardly applied to the following extended unit processing cost equation: $Y = w_1Y_1 + w_2Y_2 + \cdots + w_nY_n$, where w_i is the normalized weight and Y_i is the cost for the query having P_i.

5 A Block Clustering Scheme

In this section, we impose a layer of clusters on the PPB data streams; each cluster consists of a sequence of an equal number of data blocks. We then extend the BSS indexing scheme into the Cluster Statistics Signature (CSS) indexing scheme. We further extend the results obtained from studying the CSS indexing scheme into the general case of the Multiple-cluster Statistics Signature (MSS) indexing scheme.

5.1 The CSS Indexing Scheme

The CSS index of a given cluster, denoted by s_α, is computed from the BSS indexes (recall Definition 2.1) in the cluster.

Definition 5.1 (CSS Index and Value Range) Assume that there are n_j compressed blocks in a j-th cluster. The *Cluster Statistic Signature* (CSS) index is given by $C_j = \langle s_j^\alpha, c_j^\alpha \rangle$, where $c_j^\alpha = \{b_1, \ldots, b_{n_j}\}$ is a collection of n_j blocks of data items. The index value is given by $s_j^\alpha = \langle min_j^\alpha, max_j^\alpha, sum_j^\alpha, count_j^\alpha \rangle$. The parameters are $min_j^\alpha = Min(min(b_1), \ldots, min(b_{n_j}))$, $max_j^\alpha = Max(max(b_1), \ldots, max(b_{n_j}))$, $sum_j^\alpha = sum(b_1) + \cdots + sum(b_{n_j})$, and $count_j^\alpha = count(b_1) + \cdots + count(b_{n_j})$. Similar to l_β, we define the *value range* for a CSS index, l_j^α, as $\langle min_j^\alpha, max_j^\alpha \rangle$. We may use more handy notation such as s_α, b_α and l_α when no ambiguity arises from the cluster labelling.

The index checking of the compressed data with this new scheme is carried out as follows. In the first phase, we scan and check the value ranges of the CSS indexes. If there is an overlap between l_p and l_α (i.e., when the cluster is hit), then we use the BSS indexes to scan and check for all blocks that are contained in the hit cluster in the second phase. We depict the idea of imposing a CSS indexing scheme on a PPB data stream in Figure 9. We study the unit processing cost while including a level of clusters in the PPB engine. In Table 3, we extend the BSS notation in order to specify the parameters in the CSS layer.

Notation	Definition
π_c	Cluster hitting probability of a PBB data stream.
π_b	Block hitting probability in a hit cluster.
K_b	Number of data elements in a block of a data stream.
K_c	Number of data elements in a cluster.
L_b	Number of blocks in a cluster.
L_c	Number of clusters in a PBB data stream.
X_b	Number of hit blocks in a (hit) cluster (i.e., l_β and l_p overlap).
X_c	Number of hit clusters (i.e., l_α and l_p overlap in these clusters).

Table 3: Notation Used in the CSS Indexing Scheme

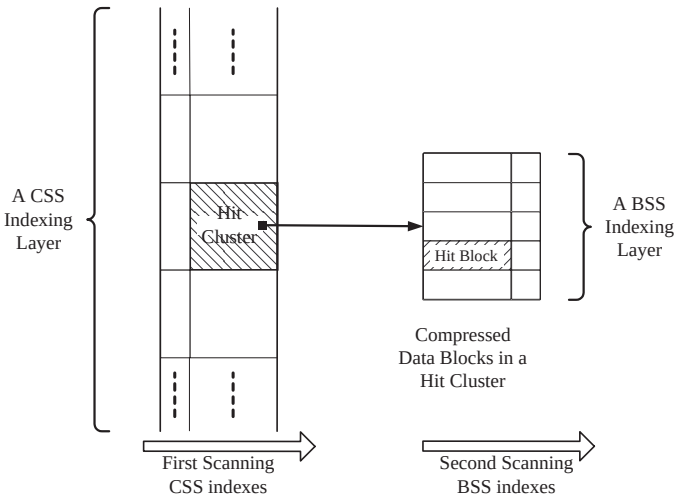


Figure 9: The CSS Indexing Scheme

Note that, as shown in Figure 9, X_c is equal to the number of clusters that has an overlap between l_α (the CSS index) and l_p in a PPB data stream, which is processed in the first phase, and that X_b is equal to the number of blocks that has an overlap between l_β (the BSS index) and l_p in a hit cluster, which is processed in the second phase. For simplicity, we assume that the cost of scanning a BSS index is equal to the cost of scanning a CSS index.

We now construct the unit processing cost equation based on the following reasoning:

Total cost of processing a selection query
= *Cost of scanning the CSS indexes in a data stream*
+ *Cost of scanning the BSS indexes in all the hit clusters*
+ *Cost of decompressing all the hit blocks.*

Thus, we have the following cost equation for processing a query using the CSS indexing scheme: $C_q = L_c \cdot C_s + X_c \cdot L_b \cdot C_s + X_c \cdot X_b \cdot K_b \cdot C_d$.

We extend the notion of the hitting probability to the *cluster hitting probability* by using a similar argument leading to the notion of the block hitting probability, which is discussed in Section 4.1. We now simply state the cluster hitting probability and the block hitting probability in the following expressions, $\pi_c = 1 - (1 - P)^{K_c}$ and $\pi_b = 1 - (1 - P)^{K_b}$, respectively. We refer to the case of *open interval selection queries*. However, other cases of interval queries can be studied in a similar way, using the respective hitting probabilities given in Table 2.

We extend the concept of hit blocks to *hit clusters*. The number of hit clusters, X_c, is equal to the number of clusters that have a corresponding CSS value range overlapping with the predicate range of a query (i.e., $l_\alpha \cap l_p \neq \emptyset$). The distribution of the event that X_c is hit satisfies a binomial distribution, which is the first phase of scanning in Figure 9. Thus, the expected number of hit clusters is given by $E(X_c) = \pi_c \cdot L_c$. In the second phase, all the blocks (i.e., L_b blocks) in these hit clusters are scanned and their BSS indexes are checked.

The expected number of hit blocks in a hit cluster is given by $E(X_b) = \pi_b \cdot L_b$.

Based on the above reasoning, we formulate the expected cost equation, which can be viewed as an extension of Equation 1:

$$E(C_q) = L_c \cdot C_s + \pi_c \cdot L_b \cdot L_c \cdot C_s + \pi_b \pi_c L_b L_c \frac{N}{L_b L_c} C_d. \tag{19}$$

By using the formula $N = L_b L_c K_b$, we simplify Equation 19 and formulate the following unit processing cost equation:

$$Y = \frac{E(C_q)}{N} = \left(\frac{C_s}{K_c} + \frac{\pi_c C_s}{K_b} + \pi_b \pi_c C_d \right). \tag{20}$$

Theorem 5.1 Y has a local minimum value for sufficiently small selectivity values, where Y is given in Equation 20.

Proof. Consider the case of using small selectivity values in Equation 20 as follows: $\pi_b \approx K_b P$ and $\pi_c \approx K_c P$. We then obtain the following equation:

$$Y = \frac{E(C_q)}{N} = \left(\frac{C_s}{K_c} + \frac{C_s K_c P}{K_b} + C_d K_b K_c P^2 \right). \tag{21}$$

We now find the partial derivatives with respect to K_b and K_c and then set them to be zero, in order to obtain the stationary points of the unit processing cost function Y:

$$\begin{aligned} \frac{\partial Y}{\partial K_b} &= -\frac{K_c P C_s}{K_b^2} + K_c P^2 C_d, \\ \frac{\partial Y}{\partial K_c} &= -\frac{C_s}{K_c^2} + \frac{C_s P}{K_b} + K_b P^2 C_d. \end{aligned} \tag{22}$$

By inserting $\frac{\partial Y}{\partial K_b} = 0$ and $\frac{\partial Y}{\partial K_c} = 0$ into Equation 22, we have the following optimal choices when, respectively, $K_b = K_b^{min}$ and $K_c = K_c^{min}$:

$$K_b^{min} = \sqrt{\frac{C_s}{C_d P}}, \quad K_c^{min} = \left(\frac{C_s}{4 C_d P^3} \right)^{\frac{1}{4}}. \tag{23}$$

We proceed to prove the claim that (K_b^{min}, K_c^{min}) is a local minimum point of Y by considering the following sufficient conditions (cf. [14]): (1) $\frac{\partial^2 Y}{\partial K_b^2} > 0$ or $\frac{\partial^2 Y}{\partial K_c^2} > 0$ and (2) $\frac{\partial^2 Y}{\partial K_b^2} \frac{\partial^2 Y}{\partial K_c^2} - (\frac{\partial^2 Y}{\partial K_b \partial K_c})^2 > 0$.

From Equation 23, the second derivatives $\frac{\partial^2 Y}{\partial K_b^2}$ and $\frac{\partial^2 Y}{\partial K_c^2}$ are obtained as follows:

$$\begin{aligned} \frac{\partial^2 Y}{\partial K_b^2} &= \frac{2 C_s P K_c}{K_b^3}, \\ \frac{\partial^2 Y}{\partial K_c^2} &= \frac{2 C_s}{K_c^3}, \end{aligned} \tag{24}$$

which are always positive for all positive parameters. In addition, for sufficiently small P (ignoring P^2 and other higher power terms), we have

$$\begin{aligned} \frac{\partial^2 Y}{\partial K_b^2}\frac{\partial^2 Y}{\partial K_c^2} - \left(\frac{\partial^2 Y}{\partial K_b \partial K_c}\right)^2 &= \frac{4C_s^2 P K_c}{K_b^3 K_c^3} - \left(P^2 C_d - \frac{PC_s}{K_b^2}\right)^2 \\ &\approx P\left(\frac{4C_s^2 K_c}{K_b^3 K_c^3}\right), \end{aligned} \tag{25}$$

which is always positive. □

We now present in Figure 10 a three-dimensional plot of the unit processing cost, Y, in relation to the block and cluster sizes in the case of low selectivity.

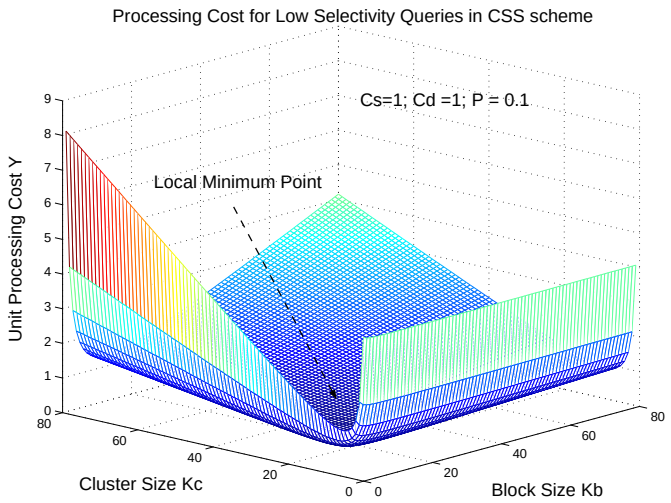


Figure 10: Unit Processing Cost in Relation to Block and Cluster Sizes in Low Selectivity Queries

Notably, there is a minimum point on the surface in Figure 10 from Theorem 5.1. In fact, we can easily deduce from Equation 23 that in the extreme case of $P \to 0$, we have $K_b^{min} \to \infty$ and $K_c^{min} \to \infty$. There are two further remarks concerning the minimum points given in Equation 23. First, the parameters K_b and K_c should be some positive integers. In addition, we assume that $L_c = \frac{K_c}{K_b}$, which must be an integer in order to give the whole number of clusters in a stream. In reality, K_c may not be an exact multiple of K_b and hence there may be a source of inaccuracy in our modelling. Second, it is interesting to see that the minimum point (K_b^{min}, K_c^{min}) depends only on selectivity P. (C_s and C_d are constant in a given configuration.)

5.2 The MSS Indexing Scheme

The clustering technique can be generalized to p layers of clusters that are arranged in a top-down manner. All the statistical parameters of the parent's cluster layer are computed from

the indexes of its children's cluster layers. To visualize this idea, we extend Figure 9 into Figure 11.

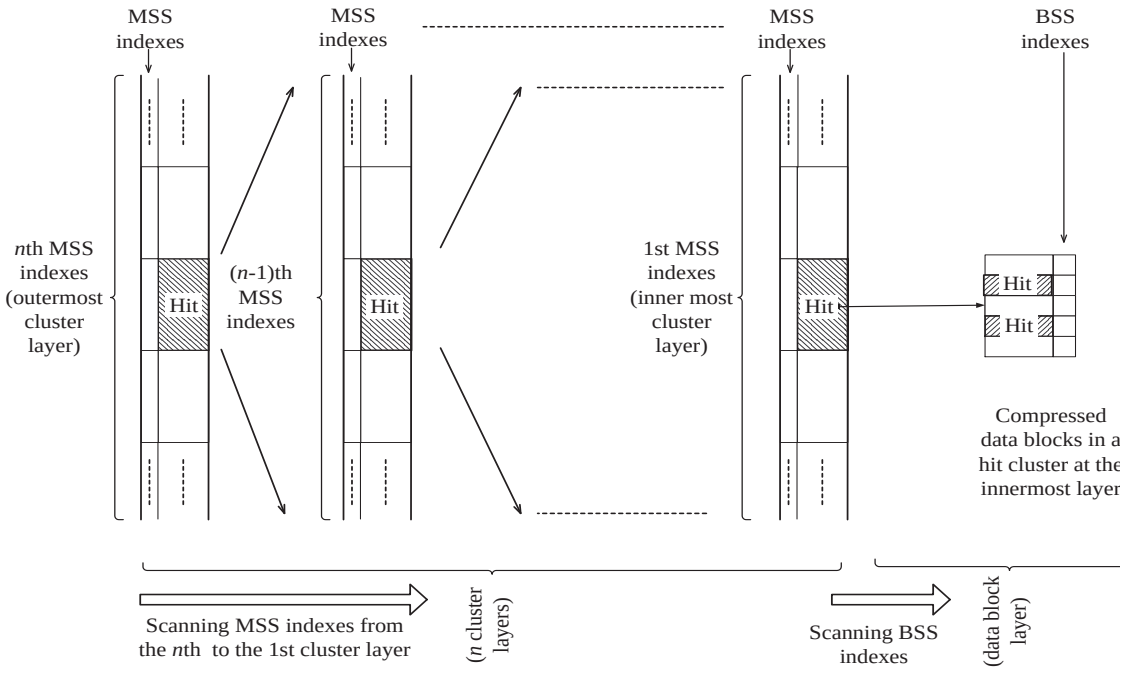


Figure 11: MSS Indexing Scheme

Let us call this indexing the Multiple-cluster Statistics Signature (MSS) indexing scheme. The scanning first starts at the outermost layer of clusters. Once we identify the hit clusters at this level, we then proceed to the scanning of the next level of clusters (i.e., C_j^{p-1} for $j \in \{1, \ldots, X_n\}$). The process repeats until it reaches the innermost cluster and finally the BSS indexes of the blocks (i.e., B_k for $j \in \{1, \ldots, X_b\}$ in the hit clusters are scanned and the hit blocks are collected for decompression).

Definition 5.2 (The MSS Index and Value Range) Assume that there are n_j clusters at level $(p-1)$ being contained in a j-th cluster at level p. The *Multiple-cluster Statistics Signature* (MSS) index at the pth level of clusters is given by $C_j^p = \langle s_j^p, c_j^p \rangle$, where $c_j^p = \{c_1^{p-1}, \ldots, c_{n_j}^{p-1}\}$ is a collection of the n_j clusters at the immediate lower level (i.e., at the $(p-1)$th level). The index value is given by $s_j^p = \langle min_j^{\alpha_p}, max_j^{\alpha_p}, sum_j^{\alpha_p}, count_j^{\alpha_p} \rangle$. The parameters are $min_j^{\alpha_p} = Min(min_1^{\alpha_{p-1}}, \ldots, min_{n_j}^{\alpha_{p-1}})$, $max_j^{\alpha_p} = Max(max_1^{\alpha_{p-1}}, \ldots, max_{n_j}^{\alpha_{p-1}})$, $sum_j^{\alpha_p} = sum_1^{\alpha_{p-1}} + \cdots + sum_{n_j}^{\alpha_{p-1}}$, and $count_j^{\alpha_p} = count_1^{\alpha_{p-1}} + \cdots + count_{n_j}^{\alpha_{p-1}}$. Similar to l_β, we define the *value range* for a MSS indexed cluster at the pth level, denoted as $l_j^{\alpha_p}$, as $\langle min_j^{\alpha_p}, max_j^{\alpha_p} \rangle$.

The searching of the hit clusters and blocks in the MSS indexing scheme is formally presented as the following pseudo-code algorithm, designated as SEARCH(C_j^p).

Algorithm 1 (SEARCH(C_j^p))

1. **begin**
2. Result:= $\emptyset$;
3. **if** $p = 0$ **do**
4. **let** $C_j^0 = \{B_{1,j}, \ldots, B_{n_j,j}\}$ where $B_{i,j} = \langle s_{i,j}^{\beta}, b_{i,j} \rangle$;
5. **for** $i = 1$ to n_j **do**
6. $scan(s_{i,j}^{\beta})$;
7. **if** $hit(B_{i,j}) =$ true **do** Result := Result $\cup \{b_{i,j}\}$;
8. **end for**
9. **else**
10. **let** $C_j^p = \{C_1^{p-1}, \ldots, C_{n_j}^{p-1}\}$ where $C_j^{p-1} = \langle s_{i,j}^{\alpha_{p-1}}, c_{i,j}^{p-1} \rangle$;
11. **for** $i = 1$ to n_j **do**
12. $scan(s_{i,j}^{\alpha_{p-1}})$;
13. **if** $hit(C_j^p) =$ true **do**
Result := Result $\cup$ SEARCH(C_j^{p-1});
14. **end for**
15. **return** Result;
16. **end.**

The notation in Algorithm 1 is heavy, since there is more than one cluster at different levels in general. For example, the term $s_{i,j}^{\beta}$ represents the BSS index value of the ith block in the jth cluster and the term $c_{i,j}^{p-1}$ represents the ith cluster at the $(p-1)$ level, which is a member in the jth cluster at the p level. In essence, Algorithm 1 performs scanning on MSS indexes from the outermost level in order to find the hit blocks in the innermost level, which is done in a *depth-first* manner. The complexity of Algorithm 1 is $O(mn)$, where m and n are the number of levels and blocks.

We now state the unit processing cost equation for the MSS indexing scheme, which is a generalized form of Equation 21. The parameters are naturally extended in a corresponding way, for example, K_i and π_i indicate the number of elements in a cluster and the cluster hitting probability at the ith MSS layer for $i \in \{1, \ldots, n\}$. When $i = 0$, $K_0 = K_b$, which represents the number of data elements in a block.

$$Y_n = \frac{C_s}{K_n} + \frac{K_n}{K_{n-1}} C_s P + \frac{K_n K_{n-1}}{K_{n-2}} C_s P^2 + \cdots + \frac{K_n K_{n-1} \cdots K_1}{K_0} C_s P^n + C_d K_n \cdots K_0 P^{n+1} \tag{26}$$

The following theorem is a generalization of Theorem 5.1 to n clusters.

Theorem 5.2 The unit processing cost, Y_n, for $n \geq 1$ has a local minimum value for sufficiently small selectivity values.

Proof. Equation 26 can be written in a recursive form as follows,

$$\begin{aligned} Y_1 &= \frac{C_s}{K_1} + \frac{K_1}{K_0} C_s P + C_d K_1 K_0 P^2, \text{when } n = 0; \\ Y_{n+1} &= \frac{C_s}{K_{n+1}} + K_{n+1} P(Y_n), \text{when } n \geq 1. \end{aligned} \tag{27}$$

When the derivative of the above equation is set to zero, we have a solution that will lead to the second derivative being zero. We proceed with the proof by induction on n.

(Basis): When $n = 0$, by Theorem 5.1, we have proved that the Y_1 has a local minimum for sufficiently small P[1]. In other words, we have the minimum of Y_1 at $(K_0, K_1) = \left(\sqrt{\frac{C_s}{C_d}}, \left(\frac{C_s}{4C_dP^3}\right)^{\frac{1}{4}}\right)$.

(Induction): We assume that Y_k in Equation 26 has a local minimum at $\mathbf{K}' = (K'_0, \ldots, K'_k)$ for $k \geq 1$. We need to show that Y_{k+1} also has a local minimum at $\mathbf{K}'' = (K''_0, \ldots, K''_{k+1})$. First, there is a solution, $\mathbf{K}$, for the system of equations, $\frac{\partial Y_{k+1}}{\partial K_i} = 0$, where $(k+1) \geq i \geq 1$. Second, the corresponding Hessian of Y_{k+1} is positive-definite. The use of the Hessian function is a standard technique used in calculus to verify if the turning point of a function having multiple variables is a minimum (cf. [14]).

From Equation 27, we express the first-order derivative of Y_{k+1} for $k \geq i \geq 1$ as follows,

$$\frac{\partial Y_{k+1}}{\partial K_i} = K_{k+1}P\left(\frac{\partial Y_k}{\partial K_i}\right). \tag{28}$$

By the inductive assumption, it follows that $\frac{\partial Y_k}{\partial K_i} = 0$ at $\mathbf{K}' = (K'_0, \ldots, K'_k)$ for $k \geq i \geq 1$. By Equation 28, it then follows that $\frac{\partial Y_{k+1}}{\partial K_i} = 0$ for $k \geq i \geq 1$. We now show that $\frac{\partial Y_{k+1}}{\partial K_i} = 0$ when $i = k+1$.

From Equation 27 again, we have the first-order derivative for $i = k+1$ as follows,

$$\frac{\partial Y_{k+1}}{\partial K_{k+1}} = -\frac{C_s}{{K_{k+1}}^2} + P(Y_k), \tag{29}$$

which has the solution of $K_{k+1} = \sqrt{\frac{C_s}{PY_k(\mathbf{K}')}}$ when $\frac{\partial Y_{k+1}}{\partial K_{k+1}} = 0$.

It remains for us to show that the $k+1$ dimensional point $\mathbf{K} = (K'_0, \ldots, K'_k, K_{k+1})$ is the minimum of Y_{k+1}. Equivalently, we show that the Hessian of Y_{k+1} at $\mathbf{K}$ is *positive-definite*.

We first define the Hessian function, H, of Y_{k+1} by (cf. see page 252 in [14])

$$HY_{k+1}(\mathbf{K})(\mathbf{h}) = \frac{1}{2}\sum_{i,j=1}^{k+1}\frac{\partial^2 Y_{k+1}}{\partial K_i \partial K_j}(\mathbf{K})h_i h_j, \tag{30}$$

where $\mathbf{h} = \langle h_1, \ldots, h_{k+1}\rangle$.

By expanding Equation 30, we have the following expression,

$$\begin{aligned} HY_{k+1}(\mathbf{K})(\mathbf{h}) = \frac{1}{2}\Big(&\sum_{i,j=1}^{k}\frac{\partial^2 Y_{k+1}}{\partial K_i \partial K_j}(\mathbf{K})h_i h_j + \sum_{i=1}^{k}\frac{\partial^2 Y_{k+1}}{\partial K_i \partial K_{k+1}}(\mathbf{K})h_i h_{k+1} + \\ &\sum_{j=1}^{k}\frac{\partial^2 Y_{k+1}}{\partial K_{k+1} \partial K_j}(\mathbf{K})h_{k+1} h_j + \frac{\partial^2 Y_{k+1}}{\partial^2 K_{k+1}}h_{k+1}h_{k+1}\Big). \end{aligned} \tag{31}$$

[1]When $n = 0$ in Equation 27, Y_1 is simply the case of CSS as already shown in Equation 21.

From Equations 28 and 29, we express the Y_{k+1} derivative in terms of the Y_k derivative as follows,

$$HY_{k+1}(\mathbf{K})(\mathbf{h}) = \frac{1}{2}(K_{k+1}P\sum_{i,j=1}^{k}\frac{\partial^2 Y_k}{\partial K_i \partial K_j}(\mathbf{K})h_i h_j + P\sum_{i=1}^{k}\frac{\partial Y_k}{\partial K_i}(\mathbf{K})h_i h_{k+1} + P\sum_{j=1}^{k}\frac{\partial Y_k}{\partial K_j}(\mathbf{K})h_{k+1}h_j + \frac{2C_s}{K_{k+1}^3}). \quad (32)$$

By the inductive assumption, we have $\frac{\partial Y_k}{\partial K_i}(\mathbf{K}) = 0$ and therefore the second and third summation terms are zero. Furthermore, it follows by Equation 30 the first summation term is equal to $K_{k+1}P(HY_k(\mathbf{K})(\mathbf{h}))$, which is positive-definite by the inductive assumption, since P and K_{k+1} are positive numbers. The fourth term, $\frac{C_s}{K_{k+1}^3}$, is simply a fraction of positive parameters. Therefore, we conclude that $HY_{k+1}(\mathbf{K})(\mathbf{h})$ is positive-definite. $\square$

From Equation 27 we now deduce the expression of the minimum cost, $Y_{min}(n)$, as follows:

$$\begin{aligned} Y_{min}(n+1) &= \frac{C_s}{K_{n+1}} + K_{n+1}P(Y_{min}(n)) \\ &= C_s\sqrt{\frac{PY_{min}(n)}{C_s}} + PY_{min}(n)\sqrt{\frac{C_s}{PY_{min}(n)}} \\ &= 2\sqrt{PC_sY_{min}(n)}. \end{aligned} \quad (33)$$

Now, we let $\alpha = 2\sqrt{PC_s}$ and $\beta = \sqrt{Y_{min}(1)} = \sqrt{\frac{C_s}{K_1} + \frac{K_1}{K_0}C_sP + C_dK_1K_0P^2}$. From Equation 27, we expand the expression of the minimum cost for an n-layer MSS, $Y_{min}(n)$, and obtain the expression as follows:

$$\begin{aligned} Y_{min}(n+1) &= \alpha \cdot \alpha^{\frac{1}{2}}\alpha^{\frac{1}{4}}\cdots\alpha^{\frac{1}{2(n-1)}}\sqrt{Y_{min}(1)} \\ &= \beta\alpha^{2(1-\frac{1}{2^n})}. \end{aligned} \quad (34)$$

We remark that from Equation 34, as $n \to \infty$, $Y_{min} \to \beta\alpha^2$. In other words, Y_{min} is a decreasing function and its range is between $\beta\alpha$ and $\beta\alpha^2$.

6 Verification of the PPB Engine

In this section we examine the essential features of the PPB engine, which are developed based on the techniques introduced in Section 2 and analyzed in Section 3. In particular, we study the relationship among the query response time, the block size, the selectivity, and the selection and aggregate queries. The different selectivities in the experiments are obtained by varying the interval range of the queries. The experiments are run on a notebook computer that has the following configuration: PIII machine 600MHz, 192 MB RAM main

memory, 20 GB hard disk (Ultra DMA/66, 4200 rpm, 512KB cache, 12 ms seek time), and Windows2000 Professional SP2 platform. We assume that the unit processing cost is proportional to the response time in our system and that the block size used here, which is measured in *records per block*, is proportional to the block size we used in Section 3, which is defined as data items per block. Formally, we have *Response time (s)* $= a_1 \cdot Y$ and *Block Size (records per block)* $= a_2 \cdot K$, where a_1 and a_2 are the proportional constants determined by our system configuration.

6.1 PPB Compression Performance

We now study the compression performance of the PPB engine by using the four XML datasets of $Weblog$ [24], $DBLP$ [22], *TPC-H* [26] and $XMark$ [27], which are commonly used in XML research (see the experiments in [3, 10, 21]). Some characteristics of these data sources are shown in Table 4, where E_num and A_num refer to the number of elements and attributes in the document, respectively. We briefly introduce each dataset as below.

1. *Weblog* is constructed from the Apache webserver log [24]. The original documents are not in XML format.

2. *DBLP* is a collection of the XML documents freely available in the DBLP archive [22]. The documents are already in XML format.

3. *TPC-H* is an XML representation of the TPC-H benchmark database, which is available from the Transaction Processing Performance Council [26].

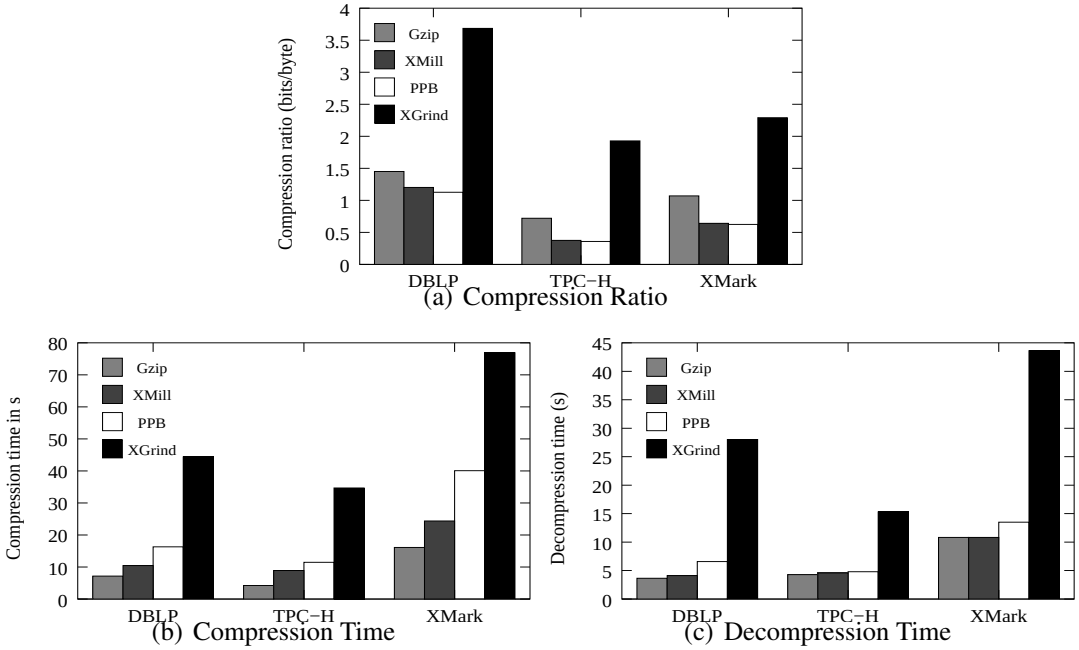
4. *XMark* is an XML document that models an auction website. It is generated by the generation tool provided in [27].

Dataset	File Size	Depth	E_num	A_num
Weblog	32.72MB	3	641037	0
DBLP	40.90MB	6	1107711	118028
TPC-H	32.30MB	3	1022976	0
XMark	103.64MB	11	2873293	621490

Table 4: XML Datasets for Studying PPB Compression Performance

Figure 12(a) shows the compression ratio (expressed in the number of bits per byte) that is achieved by the four compressors. Notably, both XMill and PPB consistently achieve better compression ratios than Gzip achieves. Figures 12(b) and 12(c) show the compression and decompression times (expressed in ms), which indicate that Gzip outperforms other compressors. The time overhead can be explained by the fact that both XMill and PPB introduce

a pre-compression phase for restructuring XML documents to help in the main compression process. XMill adopts by default an *approximation match* on a *Reversed DataGuide* for determining which container a data value belongs to. This *group by enclosing tag* heuristic runs faster than the grouping method used in the PPB engine. Thus, XMill runs slightly faster than PPB, since the compression buffer window size in XMill is solely optimized for better compression [10]. One observation from Figure 12(c) is that, in general, XGrind requires longer compression and decompression times than other three decompressors, which agrees with the findings reported in [21].



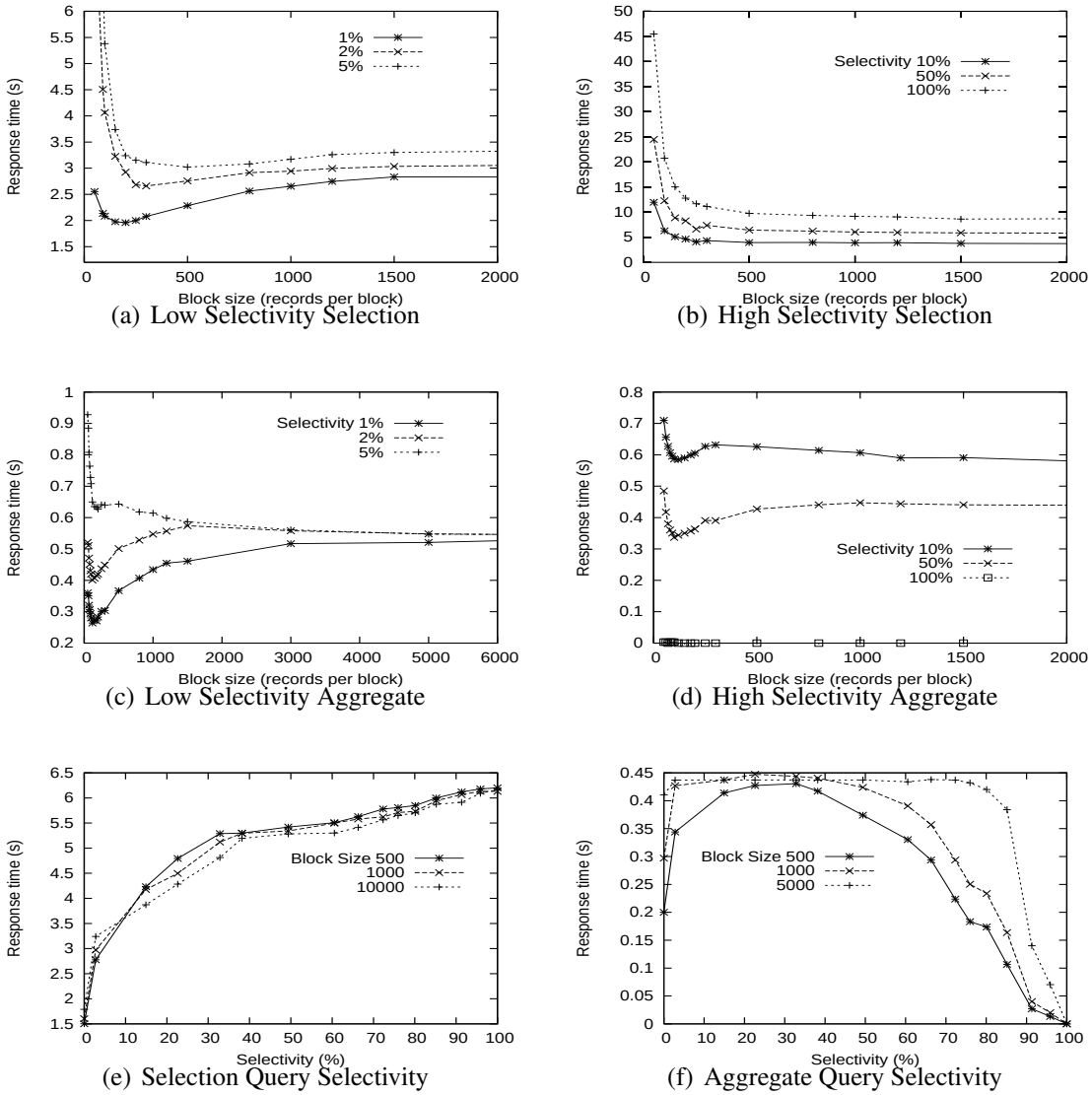
(a) Compression Ratio
(b) Compression Time
(c) Decompression Time

Figure 12: Compression Performance of the PPB Engine

6.2 Selection and Aggregate Queries

We study the selection query, Q_3, and the aggregate query, Q_5, given in Section 4.1 running on 89MB XMLized Weblog data [24]. Figure 13(a) shows the query response time for evaluating *low selectivity* selection queries under different block sizes. As shown in this figure, the PPB engine performs as expected and enjoys the benefits of using PPB data streams and block partitioning. The optimal point depends on the selectivity of the query. Even as the block size further increases, the response time for evaluating queries that have different selectivities approaches the same value (recall Figure 4(a) and Theorem 3.1). Figure 13(b) shows the response time for evaluating *high selectivity* selection queries. We see that the response time for the queries is similar to our analysis given in Figure 4(b). In essence, the decompression overhead increases when finer partitioning is used. When the block size increases, the response time decreases due to the fact that the decompression overhead decreases.

Figures 13(c) and 13(d) show the query response time for evaluating aggregate queries



(a) Low Selectivity Selection (b) High Selectivity Selection

(c) Low Selectivity Aggregate (d) High Selectivity Aggregate

(e) Selection Query Selectivity (f) Aggregate Query Selectivity

Figure 13: Query Processing in the PPB Engine

under different block sizes, which are what we expected in Figures 7(a) and 7(b). As we see in Figure 13(c), for aggregate queries with low selectivity, PPB can take advantage of the data stream partitioning in processing the queries, which is similar to the case of selection queries. It should be noted that the query response time has an optimal point as expected. For an aggregate query to have a high selectivity value (say at the extreme near 100%), the corresponding query response time is almost zero. The underlying reason is that when the selectivity of an imposed query is extremely large, the probability that l_β overlaps the predicate interval, l_p, of the imposed query is near unity. In this case, the PPB engine uses only the BSS indexes to compute the aggregate value without decompressing many blocks. It is straightforward to see that the optimized block size for the response time of different queries roughly falls within a short range of 250 to 500 records per block. Thus, we can further take advantage of this finding for the PPB engine configuration, when block size

estimation should be done under insufficient or no query information.

6.3 Selectivity and Block Size

Figure 13(e) shows the query response time achieved by the PPB engine for a set of selection queries having different selectivity values and block sizes. As the selectivity of the imposed queries increases, the query response time increases. This is due to the fact that more XML fragments are expected to be selected and returned in the output as the selectivity of an imposed query increases. Figure 13(f) shows the query response time achieved by the PPB engine for aggregate queries with different selectivities and block sizes. When the selectivity of the imposed query is low, the query response times are extremely small. As the selectivity values of the imposed queries increase, the query response time increases rapidly to its maximum value and decreases gradually again as the selectivity further increases. Compared to Figures 8(a) and 8(b), the trends of the response time in both cases are similar, except that the decreasing rate in the region of high selectivity aggregate queries seems to depend more heavily on the block size than expected.

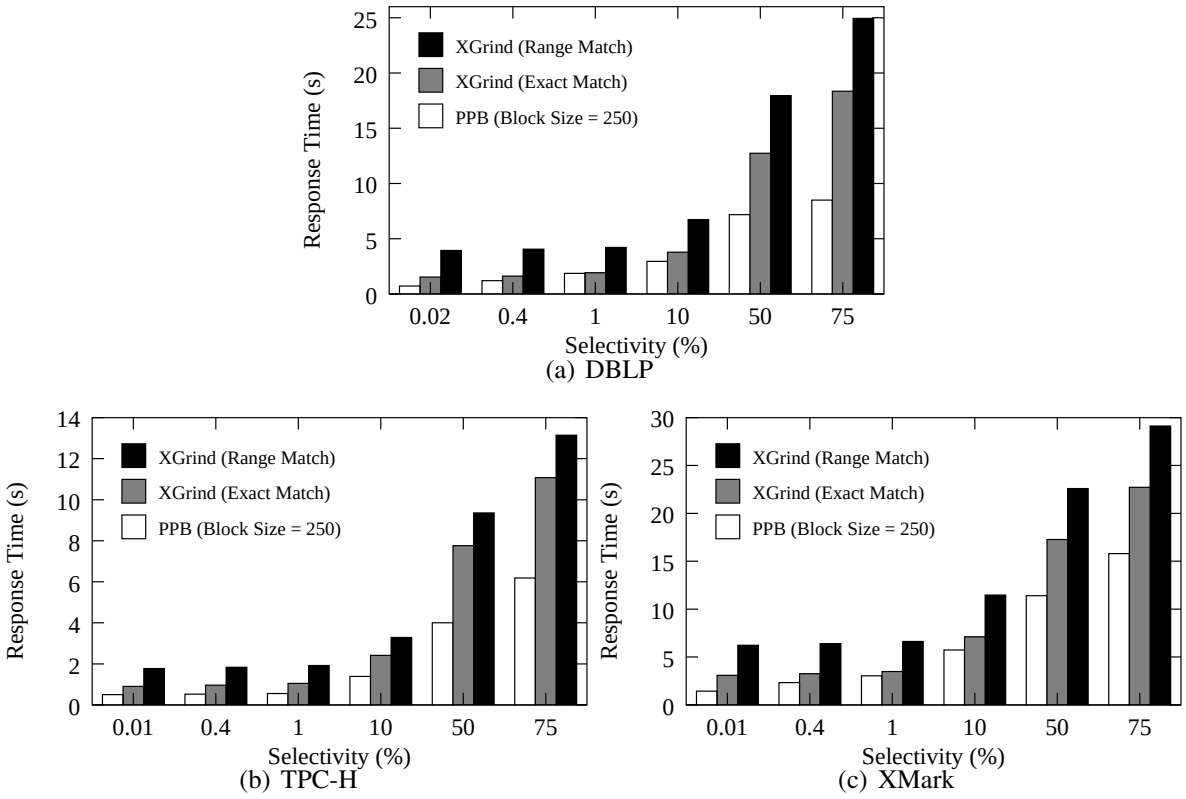


Figure 14: Processing Selection Queries over Different Datasets in PPB and XGrind

6.4 Comparing PPB with XGrind

We now compare the performance of XGrind with that of the PPB engine by running the following queries.

(Q_7): /dblp/article[1998 $>$ year $>$ 1950].

(Q_8): /table/T[10010 $>$ L_ORDERKEY $>$ 10000].

(Q_9): /open_auctions/open_auction[100 $>$ current $>$ 20].

As XGrind only supports processing of *exact match* and *partial match* selection queries, we use these selection queries in the tests. Figures 14(a), 14(b) and 14(c) show the query response times of XGrind and PPB for processing queries Q_7 to Q_9 on the respective datasets. We varied the interval ranges of `year`, `L_ORDERKEY` and `current` to obtain different selectivities for the experiment. Clearly, the query response time obtained by the PPB engine is consistently shorter than that obtained by XGrind. The savings in response time by PPB is attributed to the use of partial decompression of data streams and to the BSS indexing adopted in the PPB engine.

7 Conclusions

In this paper, we developed and analyzed the PPB data grouping strategy in order to support querying over compressed XML data. We demonstrated that the technique is effective from both analytical and empirical points of view. We introduced the BSS indexing scheme for compressed blocks in a data stream. We then presented a critical analysis of partial decompression performance based on the PPB and BSS indexing techniques. Our analysis presents query processing cost expressions to answer selection and aggregate queries. We showed that the optimal cost exists in low selectivity selection queries and an asymptotic cost exists in high selectivity values in Theorem 3.1. We discussed the extension of BSS into CSS and found that optimal sizes of clusters and blocks exist in the CSS indexing scheme. The optimal block size and cluster size values are independent of the data size, N, but not of the selectivity value, P. The CSS indexing scheme can be further generalized into the MSS indexing scheme. We established the generalized result in Theorem 5.2 that there exists a local minimum of the unit processing cost for all MSS indexing schemes if the selectivity and scanning costs are sufficiently low. We demonstrated that the relationship between processing time, selectivity and block size is consistent with our analysis in Sections 6.2 and 6.3. We also showed that the compression performance is comparative to XMill and the querying performance of PPB is superior to XGrind in the the scope of our study.

Overall, our analysis paves the way to optimize query processing of compressed XML data within a path-partitioned based framework. We need to go in three directions in our future work. First, we need to verify empirically how the compression performance is impacted by the MSS indexing scheme, though we proved from the theoretical point of view

that the processing cost can be reduced. Second, we need to analyze further the cost of using the PPB technique to handle a wider scope of XML semantics. For example, the join (for the *For clause*), semi-join (for the *Where clause*) and outer-join (for the *Return clause*) in standard XQuery expressions [29]. Third, it is indeed challenging and practical to consider XML query processing over compressed XML documents in distributed applications, such as the study of distributed XML query processing and result dissemination in a co-operative framework [8]. However, when the query evaluation is carried out over a network of clients, the work is more complex and challenging. We need to devise some succinct data structures and query rewriting efficient techniques in order to organize multi-XML queries from network clients and support query evaluation of compressed documents.

References

[1] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient Query Evaluation over Compressed XML Data. *Proc. of EDBT* (2004).

[2] P. Buneman, M. Grohe and C. Koch. *Path Queries on Compressed XML*. Proc. of VLDB, (2003).

[3] J. Cheney. *Compressing XML with Multiplexed Hierarchical PPM Models*. Proc. of IEEE Data Compression Conf., pp. 163-172, (2000).

[4] J. Cheng and W. Ng. XQzip: Querying Compressed XML Using Structural Indexing. *Proc. of EDBT* (2004).

[5] T. M. Cover and J. A. Thomas. Elements of Information Theory. *WILEY-INTERSCIENCE, John Wiley & ons, Inc., New York*, (1991).

[6] A. Datta and H. Thomas. *Accessing Data in Block-Compressed Data Warehouses*. Workshop on Information Technologies and Systems (WITS), (1999).

[7] C. Faloutsos and S. Christodoulakis. *Design of a Signature File Method that Accounts for Non-Uniform Occurrence and Query Frequencies*. Proc. of VLDB, pp. 165-170, (1985).

[8] J. He, W. Ng, X. Wang and A. Zhou. *An Efficient Co-operative Framework for Multi-Query Processing over Compressed XML Data*. Proc. of DASFAA 2006, LNCS Vol. 3882, pp. 218-232, (2006).

[9] B. Iyer and D. Wilhite. *Data Compression Support in Databases*. Proc. of VLDB Conf., pp. 695-704, (1994).

[10] H. Liefke and D. Suciu. *XMill: An Efficient Compressor for XML Data*. Proc. of ACM SIGMOD, pp. 153-164, (2000).

[11] Z. Lin and C. Faloutsos. *Frame-Sliced Signature Files*. IEEE TKDE, 4(3), pp.281-289, (1992).

[12] W. Ng, W. Y. Lam, P. T. Wood and M. Levene. *XCQ: A Queriable XML Compression System.* An International Journal of Knowledge and Information Systems, (2005).

[13] W. Ng, W. Y. Lam and J. Cheng. *Comparative Analysis of XML Compression Technologies*. Journal of World Wide Web: Internet and Web Information Systems, **9**(1), pp. 5-33, (2005).

[14] J. E. Marsden and A.J. Tromba. *Vector Calculus* W.H. Freeman and Company, (1988).

[15] J. K. Min, M. J. Park and C. W. Chung. *XPRESS: A Queriable Compression for XML Data*. Proc. of ACM SIGMOD, (2003).

[16] W. Ng and C. Ravishankar. *Block-Oriented Compression Techniques for Large Statistical Databases* IEEE TKDE 9(2), pp. 314-328 (1997).

[17] M. Poess and D. Potapov *Data Compression in Oracle*. Proc. of VLDB Conf., (2003).

[18] L. Segoufin and V. Vianu. *Validating Streaming XML Documents*. Proc. of PODS, (2002).

[19] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, Vol. 27, pp. 398-403 (1948).

[20] N. Sundaresan and R. Moussa. *Algorithms and Programming Models for Efficient Representation of XML for Internet Applications*. WWW Conf., pp. 366-375, (2001).

[21] P. M. Tolani and J. R. Haritsa. *XGRIND: A Query-friendly XML Compressor*. IEEE ICDE Conf., pp. 225-234, (2002).

[22] DBLP, http://dblp.uni-trier.de/

[23] Gzip, http://www.gzip.org/

[24] Log Files - Apache, http://httpd.apache.org/docs/

[25] SAX, http://www.saxproject.org/

[26] TPC-H, http://www.tpc.org/tpch/default.asp

[27] XMark, http://monetdb.cwi.nl/xml/

[28] XPath 1.0, http://www.w3c.org/TR/xpath/

[29] XQuery 1.0, http://www.w3c.org/TR/2002/WD-xquery-20020816.

[30] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, Vol. IT-23, No. 3, pp. 337-343, May (1977).