Performance Comparison of Property Map and Bitmap Indexing

Ashima Gupta
Division of Pediatric Informatics
Cincinnati Children's Hospital Medical Center, Cincinnati, OH 45221
ashima.gupta@chmcc.org

Karen C. Davis
ECECS Dept.
University of Cincinnati
Cincinnati, OH 45221-0030
karen.davis@uc.edu

Jennifer Grommon-Litton
Matrix Systems, Inc.
Dayton, OH 45459
jennifer.litton@matrixsys.com

ABSTRACT

A data warehouse is a collection of data from different sources that supports analytical querying. A Bitmap Index (BI) allows fast access to individual attribute values that are needed to answer a query by representing the values of an attribute for all tuples separately, as bit strings. A Property Map (PMap) is a multidimensional indexing technique that pre-computes attribute expressions, called properties, for each tuple and stores the results as bit strings [DD97, LD02]. This paper compares the performance of the PMap and the Range-Encoded Bit-Sliced Index (REBSI) [CI98] using cost models to simulate their storage and query processing costs for different kinds of queries over a benchmark schema. We identify parameters that affect performance of these indexes and determine situations in which either technique gives significant improvement over the other. We also explore ways to improve PMap design to enhance performance.

Categories and Subject Descriptors: H.2.2 [**Information Systems**]: Database Management – *physical design, access methods.*

General Terms: Design, Performance

Keywords: data warehouse, bitmap index, performance study.

1. INDEXING DATA WAREHOUSES

Bitmap Indexes (BIs) are suitable for data warehousing environments as they consume only a fraction of the size of the indexed data and provide dramatic performance gains. Boolean operations such as AND, OR and NOT are extremely fast for bitmap vectors, also called bitmaps or bit-vectors [OQ97]. In a relational data warehouse, bitmaps indicate whether an attribute in a tuple is equal to, greater than, or less than (depending upon the type of BI) a specific value or not. The length of a bit-vector is equal to the cardinality of the indexed table. The position of a bit in a bit-vector denotes the position of a tuple in the table.

The Property Map (PMap) is a multidimensional indexing technique that pre-computes attribute expressions, called properties, for each instance in a relation and stores the results as bit strings [DD97, LD02]. Multiple predicates over one relation may be pre-computed and stored in a single PMap. PMaps are able to jointly encode attributes, unlike any other bitmap indexing technique. Properties are defined based on knowledge of an application, such as a known set of queries, and created using heuristic design algorithms [LD02] that are not discussed here due to space limitations. The value of each property is computed as a bit string for each instance of a relation, and these are concatenated to form a *pstring*. A PMap consists of the pstrings calculated for the tuples of a relation and a B+ Tree index to aid in pstring searches. To illustrate a PMap, consider an example relation *Employee* with attributes *age, workyears,* and *gender* with a PMap having the following properties:

p1: range on *age*	p2: enumerated on *workyears*	p3: Boolean on "*gender* = Female"
[20, 30) = 00	1 = 000, 2 = 001	F = 1
[30, 40) = 01	3 = 010, 4 = 011	M = 0
[40, 50) = 10	5 = 100	
[50, 60) = 11		

Table 1 shows sample tuples in *Employee* and corresponding pstring values that comprise the PMap in the final column.

Table 1. Example Database and Property Map

(age, workyears, gender)	p1	p2	p3	pstring
(39, 2, F)	01	001	1	010011
(24, 5, F)	00	100	1	001001
(52, 4, M)	11	011	0	110110

A single BI indexes an attribute; bit-vectors store bits representing the values of that attribute for all the tuples. A single PMap indexes a table; a pstring stores bits representing all the indexed attributes for a tuple, with respect to the properties over those attributes. A PMap is a specialized indexing method with a unique multi-attribute approach for fast processing of frequently used queries. With a BI, data items that satisfy each predicate in the query are located quickly, and then an intersection (or another operation) on the tuple identifiers yields the answer to the query. With a PMap, the answer to a query is found by examining the pstring associated with each tuple only once and no additional set operations over intermediate results are required, though in-memory filtering may be needed. For the purpose of this study, we assume a write-once-read-many DSS environment consisting of selection queries emphasizing range predicates.

In order to compare a suitable bitmap index with the PMap, we investigate the features of bitmap indexing techniques in the literature and classify them into three categories. The techniques in the *Simple* category, such as SBI, store a 0 or a 1 for each tuple and attribute value for the indexed attribute, although KEBI and RBBI use simple bitmap representations for clusters of values. The *Encoded* techniques (total-order preserving, range-based encoding, hierarchy encoding, and groupset indexing) use a binary encoded bitmap index, along with a mapping table and retrieval functions. *Bit-Sliced* techniques are based on the idea of decomposing values and encoding the components separately (described in Section 3.2.1).

We choose a suitable bitmap index for comparison with the PMap based on the following three criteria:

1. published algorithms for design and creation,
2. reasonable performance characteristics evidenced by published performance or comparison studies, and

DOLAP '02, November 8, 2000, McLean, VA, USA.

3. demonstrated capability to effectively process both range and equality selection queries.

Table 2. Feature Analysis of Bitmap Indexes

Acronym	Reference	Category	1.	2.	3.
SBI	[OQ97]	Simple	Y		Y
KEBI	[K00]	Simple	Y	Y	
RBBI	[WY96]	Simple	Y		
EBI	[WB98]	Encoded			Y
BSI	[CI98]	Bit-Sliced	Y	Y	Y

Table 2 presents bitmap indexing techniques in the literature with respect to these three criteria. "Y" indicates that the technique satisfies the criteria and a blank indicates that it does not. BSI techniques satisfy all three criteria and are designed to process high cardinality queries more efficiently than SBI. We choose range encoding (REBSI) over equality encoding for our comparison study since a performance study indicates that REBSI is more efficient for both range and equality selections [CI98].

We simulate storage and query processing costs for PMaps and REBSIs, and obtain performance estimates to compare their storage and retrieval requirements. By varying parameter values in simulations and observing trends in the results, we observe the effects of these parameters on the performance of both the techniques and identify situations in which either kind of index is better suited. By analyzing the resulting data from different perspectives, we identify key factors that affect the individual and relative performance of PMaps. We suggest improvements to PMap design that enhance its performance and also suggest scenarios for its application.

2. PROPERTY MAP

To illustrate PMap terminology and approaches to query processing, we use a benchmark database [G93]; we use the same database in our performance study. The benchmark database has a single table called BENCH that contains 1 million rows of 200 bytes each (224 bytes with overhead) and 13 uniformly distributed, indexed attributes. Each attribute has integer values ranging from 1 to its cardinality, which is represented in the attribute name, e.g., the attribute *K4* has 4 values: 1, 2, 3 and 4, and *K100k* has 100,000: values 1, 2, …, 100,000. The benchmark offers a wide range of queries and selectivity values. We categorize these queries to form 6 different test sets, based on criteria such as attribute cardinality and selectivity, to study their impact. Figure 1 gives the query numbers and selection conditions for one of the six test sets, *Low Cardinality Attribute Query Set* (LCAQS), where at least 50% of a query's attributes have cardinality less than or equal to 100. Table 3 gives a PMap for the LCAQS created based on heuristic design algorithms [LD02]. We use this query set to illustrate terminology and introduce another one to illustrate the methodology of our performance studies in Section 4.

A *property* is a many-to-one mapping of the values of an attribute or expression to a bit string *b*. In essence, a property is used to perform expression evaluation over a tuple, returning true/false or a numeric value, which is converted into a fixed length bit string *b* representing that property value. Three types of properties, Boolean, range, and enumerated, are described here.

A *Boolean* property describes the truth value of a predicate and hence needs only one bit. This is set to 1 in the pstring if the tuple evaluates to TRUE for that property, otherwise it is set to 0. For example, for the Boolean property *p2* on predicate "K100 < 41," all tuples that have *K100* value less than 41 have the bit corresponding to this property set to 1, as in the case of tuples 1 and 3 in Table 3(b). For all other tuples, it is set to 0.

An *enumerated* property consists of discrete values of an attribute. For example, the enumerated property *p4* on *K25* in Table 3(a) describes values {1, 2, …, 24, 25}. These values are represented by bit strings. An enumerated property occupies $\lceil \log_2 Y \rceil$ bits, if *Y* discrete values are represented. Boolean and enumerated properties exactly cover the predicates on which they are defined and no unnecessary records are retrieved during query processing.

A *range* property defines range intervals on a predicate's values. The intervals do not have to have identical lengths; range interval boundaries are determined by the constant values of the predicates, where the first interval begins with the domain lower limit and the last interval ends with domain upper limit + 1. For example in Table 3, range property *p1* has intervals "[0, 2), [2, 3), [3, 4), [4, 101)" on the attribute *K100* covering the predicates "K100 = 2," and "K100 = 3" in the query set. A range property occupies $\lceil \log_2 Y \rceil$ bits, where *Y* is the number of intervals.

Q1	(i) K2 = 2	Q2A	(i) K2 = 2 AND K4 = 3
	(ii) K4 = 2		(ii) K2 = 2 AND K5 = 3
	(iii) K5 = 2		(iii) K2 = 2 AND K10 = 3
	(iv) K10 = 2		(iv) K2 = 2 AND K25 = 3
	(v) K25 = 2		(v) K2 = 2 AND K100 = 3
	(vi) K100 = 2		
Q4A	(i) K4 = 3 AND (K25 = 11 OR K25 = 19) AND K5 = 3		
	(ii) (K25 = 11 OR K25 = 19) AND K4 = 3 AND K100 < 41		

Figure 1. Selection Conditions from the LCAQS

A *property string* or *pstring* is a concatenation of the property bit string values from a set of properties calculated for a single tuple. A *property map* (PMap) is the set of pstrings corresponding to the tuples in a database.

A *property mask* or *pmask* is a bit string with the same length as a pstring. It is used to mask out irrelevant properties and is obtained by setting bits corresponding to the properties covering the query to 1 and setting all other bits to 0. For example, for Q2A(i) "K2 = 2 AND K4 = 3," the *pmask* is 000100000110000, masking out properties *p1*, *p2*, *p4*, *p6* and *p7*. A *property filter* or *pfilter* is a bit string with the same length as a pstring and is used to determine the pstrings that satisfy the query criteria. It is obtained by setting all bits corresponding to properties covering the query to the value desired and setting all other bits to 0. For example, Q2A(i) "K2 = 2 AND K4 = 3" has the *pfilter* 000100000100000, since *p3* is equal to 1 for "K2 = 2" and *p5* is equal to 10 for "K4 = 3."

A *filter formula*, of the form *pstring* AND *pmask* = *pfilter*, is used to filter out pstrings that do not satisfy the query. For example, Q2A(i) "K2 = 2 AND K4 = 3," has the *filter formula* "*pstring* AND 000100000110000 = 000100000100000."

Property ID	Attribute/ Expression	Number of Bits	Property Type	Value Set of the Property
p1	K100	2	range	[0, 2)=00, [2, 3)=01, [3, 4)=10, [4, 101)=11
p2	"K100 < 41"	1	Boolean	0, 1
p3	"K2 = 2"	1	Boolean	0, 1
p4	K25	5	enumerated	1=00001, 2=00010,…,11=01011, …, 25=11001
p5	K4	2	range	[0, 2)=00, [2, 3)=01, [3, 4)=10, [4, 5)=11
p6	K5	2	range	[0, 2)=00, [2, 3)=01, [3, 4)=10, [4, 6)=11
p7	K10	2	range	[0, 2)=00, [2, 3)=01, [3, 4)=10, [4, 11)=11

(a) PMap Metadata

Tuple	(K2, K4, K5, K10, K25, K100)	p1	p2	p3	p4	p5	p6	p7	pstring
1	(2, 3, 2, 6, 19, 40)	11	1	1	10011	10	01	11	1111100111100111
2	(1, 3, 3, 2, 11, 78)	11	0	0	01011	10	10	01	1100010111101001
3	(2, 2, 3, 3, 1, 1)	00	1	1	00001	01	10	10	0011000010111010

(b) Example Property Strings (pstrings)

Table 3. A PMap for the Low Cardinality Attribute Query Set

Each *pmask* has a set of one or more *pfilters* associated with it. There is one *filter formula* for each *pmask-pfilter* pair. To account for multiple *pfilters* (as in case of required attribute values that fall in multiple range intervals) and expedite the search, pfilter low ($pfilter_l$) and pfilter high ($pfilter_h$) are used. The string $pfilter_l$ is the *pfilter* with the lowest ordinal value of all the *pfilters.* If any of the *filter formulae* use less than (<) or less than-equal to (<=) operators, $pfilter_l$ is the pstring with all bits set to 0. The string $pfilter_h$ is obtained by setting all irrelevant bits to 1 in each *pfilter* and selecting the one with the highest ordinal value. If any of the *filter formula* uses greater than (>) or greater than or equal to (>=) operator, $pfilter_h$ is the pstring with all bits set to 1. The AND operation in the *filter formula* does not have to be performed for all *pstrings* but only for those limited by $pfilter_l$ and $pfilter_h$.

To illustrate query processing with a PMap in more depth, consider the query Q4A(i) from Figure 1:

SELECT * FROM BENCH

WHERE K4 = 3 AND (K25 = 11 OR K25 = 19) AND K5 = 3.

We propose two strategies for processing this query using the PMap given in Table 3.

The query processor determines the *pmask* equal to 000011111111100, which is obtained by setting the bits corresponding to the properties covering the query attributes *K25, K4* and *K5* to 1, and setting property bits for attributes *K100, K2,* and *K10* to 0. The *K4* value equal to "3" has property value 10, the *K25* value equal to "11" has property value 01011, the *K25* value equal to "19" has property value 10011, and the *K5* value equal to "3" has property value 10.

Strategy A splits the high level query into two or more separate queries by rewriting them in disjunctive normal form and processing each disjunct as a separate query, taking the union of the result set as the final answer. For example, the selection condition Q4A(i) can be rewritten as conditions "(K4 = 3 AND K25 = 11 AND K5 = 3)" and "(K4 = 3 AND K25 = 19 AND K5 = 3)." The query "K4 = 3 AND K25 = 11 AND K5 = 3" has $pfilter_l$ equal to 000001011101000 and $pfilter_h$ equal to 111101011101011. The query "K4 = 3 AND K25 = 19 AND K5 = 3" has $pfilter_l$ 000010011101000 and $pfilter_h$ equal to 111110011101011. These subqueries are processed individually as described below and the results for both constitute the answer to the query.

Strategy B processes a high level query as a single query using multiple *pfilters*. Since there are two predicates on *K25*, we have two *pfilters*: $pfilter_1$ = 000001011101000 in the case of "K25 = 11," and $pfilter_2$ = 000010011101000 in the case of "K25 = 19." The corresponding $pfilter_h$*s* are 111101011101011 and 111110011101011, respectively. Essentially, all $pfilter_l$*s* and all $pfilter_h$*s* from *Strategy A* are considered as candidates. The $pfilter_l$ with the lowest ordinal value (000001011101000) is chosen as $pfilter_l$ and the $pfilter_h$ with the highest ordinal value (111110011101011) is selected as $pfilter_h$ for the complete query.

The pstrings in a PMap are stored as a B+ Tree for efficient searching. To answer the query, only those pstrings whose value is between $pfilter_l$ and $pfilter_h$ need to be searched. The tuples for which the *filter formula* "*pstring* AND *pmask = pfilter,*" where *pfilter* is either $pfilter_1$ or $pfilter_2$ in case of *Strategy B*, evaluates to TRUE are in the result set. For tuple 2 in Table 3(b), the *pstring* (110001011101001) ANDed with the *pmask* (000011111111100) evaluates to 000001011101000, which is equal to $pfilter_l$ for the first disjunct in the case of *Strategy A* and $pfilter_1$ in the case of *Strategy B*. Since no other tuple in this example satisfies the filter formula, tuple 2 is the query answer. We discuss these two strategies as the number of index pages retrieved using each is different and one may be better than the other for certain queries.

The record pointers related to each pstring in the result set are used to retrieve the blocks that contain the tuples corresponding to the pstring. The PMap cost model presented in the next section is based upon a B+ Tree storage structure and the pstring search algorithm described elsewhere [LD02].

3. STORAGE/RETRIEVAL COST MODELS

The cost models for PMap and REBSI are independent of query processing strategy. In our performance studies described in Section 4, we choose either *Strategy A* or *Strategy B*, whichever retrieves fewer pages for the PMap. Storage cost is measured in terms of the number of disk blocks required to store the index, and retrieval performance cost is measured in terms of the number of index blocks retrieved to answer a query, since the number of data blocks is the same for both techniques.

3.1 PMap Cost Model

We assume the PMap index to be packed for a write-once-read-many data warehousing environment.

3.1.1 Storage

We compute the total number of blocks occupied by a PMap using the following parameters: size of a pstring, pointer, and integer in bytes (S_{STR}, S_{PTR} and S_{INT}) and formulae: (1) The blocking factor of B+ tree index nodes, $bfr = \lfloor (S_B - S_{PTR}) / (S_{STR} + S_{PTR}) \rfloor$, (2) the utilization of a pstring, $pu = \prod p_i / 2^w$, where w is the word length, (3) the number of leaf nodes in the PMap index, $N_{LF} = \lceil pu * 2^w / bfr \rceil$, (4) The height of the PMap index, $h = \lceil \log_{(bfr+1)} (N_{LF}) \rceil + 1$, (5) the number of internal and leaf nodes, $N_I = N_{LF} + \lceil N_{LF} /(bfr+1) \rceil + \lceil N_{LF} /(bfr+1)^2 \rceil + \ldots + \lceil N_{LF} /(bfr+1)^{h-1} \rceil \approx \lceil N_{LF} * [(1 / (bfr+1)^h - 1) / (1 / (bfr+1) - 1)] \rceil$, (6) the number of records on average represented by a pstring, $Arec = t / (pu * 2^w$, (7) the blocking factor of the record pointer nodes, $bfr' = \lfloor (S_B - S_{PTR} - S_{INT}) / (S_{STR} + S_{INT} + Arec * S_{PTR}) \rfloor$, (8) the total number of record pointer nodes, $N_R = \lceil pu * 2^w / bfr' \rceil$, and (9) the total number of blocks in the index, $N = N_I + N_R$.

3.1.2 Retrieval

To find $pfilter_l$ requires reading one page for each level of the B+ tree and so is the same as the height of the tree. Thus, the number of index pages needed to find $pfilter_l$ is $M1 = h = \lceil \log_{(bfr+1)} (N_{LF}) \rceil + 1$. The number of leaf level nodes between $pfilter_l$ and $pfilter_h$ is $M2 = \lceil (pfilter_h - pfilter_l + 1) * pu / bfr \rceil - 1$.

For the number of record pointer nodes read, a minimum and maximum number of nodes are calculated. The maximum number of nodes is read if all the pstrings between $pfilter_l$ and $pfilter_h$ satisfy the query criteria and therefore, all the corresponding record pointers are retrieved. Thus, the maximum number of record pointer nodes read is $MR_{max} = \lceil (pfilter_h - pfilter_l + 1) * pu / bfr' \rceil$. The minimum number of nodes is read if all the pstrings (and corresponding record pointers) that satisfy the query criteria are stored sequentially between $pfilter_l$ and $pfilter_h$. Thus, the minimum number of record pointer nodes read is $MR_{min} = \lceil (S_q * 2^w * pu) / bfr' \rceil$, where S_q is the query selectivity. The actual number of record pointer nodes accessed (*MR*) depends on the percentage of pstrings that match the search criteria, where $MR_{max} \geq MR \geq MR_{min}$. The total number of index pages read is between *PMin* and *PMax*, where $PMin = M1 + M2 + MR_{min}$ and $PMax = M1 + M2 + MR_{max}$.

3.2 Range-Encoded Bit-Sliced Index

Chan and Ioannidis present various ways to create REBSIs. Due to the fact that access time and not the available disk space is usually the important factor in a data warehouse (though unlimited space allocation is unrealistic to assume), we consider the Range-Encoded Bit-Sliced Index that is time optimal under a given space constraint [CI98] for our comparison study. In this section, we define REBSI and present a storage and retrieval cost model for it.

3.2.1 REBSI Definition

A Bit-Sliced Index is defined by two factors, attribute value decomposition and encoding scheme [CI98]. The first factor, *attribute value decomposition*, defines the arithmetic that represents the values of an attribute. It is the decomposition of an attribute's values in digits according to a chosen base. For example, 124 can be decomposed into <1, 2, 4> according to base <10, 10, 10>. Consider an attribute with cardinality C, and a sequence of *n-1* numbers $<b_{n-1}, b_{n-2}, \ldots, b_1>$. Let $b_n = \lceil C / (\Pi_{i=1,\ n-1} b_i) \rceil$. Then each choice of *n* and sequence $<b_n, b_{n-1}, \ldots, b_1>$, called the base of the index, gives a different representation of attribute values, according to the formulae given below, and therefore a different index. The index consists of *n* components, i.e., one component per base digit. Each component is a collection of bitmaps limited by the value of the base for that component. If $b_{n-1} = b_{n-2} = \ldots = b_1$ are all equal to *b*, then the base is called uniform. Consider an attribute value *v*, number of components *n* and base $<b_n, b_{n-1}, \ldots, b_1>$. Then *v* can be decomposed into a sequence of *n* digits $<v_n, v_{n-1}, \ldots, v_1>$ as follows:

$$\begin{aligned} v &= V_1 \\ &= V_2 b_1 + v_1 \\ &= V_3 (b_2 b_1) + v_2 b_1 + v_1 \\ &= V_4 (b_3 b_2 b_1) + v_3 (b_2 b_1) + v_2 b_1 + v_1 \\ &= \ldots \end{aligned}$$

where $v_i = V_i \bmod b_i$, $V_i = \lfloor (V_{i-1} / (b_{i-1}) \rfloor$, $1 < i <= n$, $V_1 = v$, and each digit v_i is in the range $0 <= v_i < b_i$. An index is well-defined if $b_i >= 2$, $1 <= i <= n$.

The second factor that defines a BSI is the *encoding scheme*. Consider the i^{th} component of an index with a base value b_i. The two schemes to encode the corresponding values v_i ($0 <= v_i <= b_i - 1$) in bits are equality encoding and range encoding. In *equality encoding*, there are b_i bits, one for each possible value. The representation of value v_i has all bits set to 0, except for the bit corresponding to v_i, which is set to 1.

In *range encoding,* there are b_i bits (one for each possible value) set so as to satisfy an inequality condition between the value represented by them and the decomposed attribute value of the corresponding record for that component. The representation of value v_i has the v_i rightmost (or leftmost) bits set to 0 and the remaining bits (starting from the one corresponding to v_i, and to the left (or right)) are set to 1. Since the bitmap B_i^{bi-1} has all bits set to 1, it does not need to be stored, so a Range-Encoded Bit-Sliced Index (REBSI) component consists of $(b_i - 1)$ bitmaps.

3.2.2 REBSI Cost Model

In an encoded bit-sliced index that is time optimal under a given space constraint, a seed index is first created that satisfies the space constraint measured in terms of the number of bitmaps stored (*FindSmallestN*); improvements to the time efficiency of the seed index are done by adjusting its base numbers (*RefineIndex*) [CI98]. We determine the base of a REBSI for each attribute, the number of bitmaps to store it, and the expected number of bitmap scans answer a selection predicate on this attribute using these algorithms.

To compare the bitmap structure with the PMap structure, we define the space constraint for the REBSI to be equal to the PMap space multiplied by a scaling factor *sf*. The REBSI corresponding to a PMap consists of individual REBSIs constructed for each attribute in the query set. The space constraint *M* given for all dimensions together is $M = \lceil (N / z) * sf \rceil$, where the number of blocks (z) to store 1 bitmap vector is $z = \lceil t / (8 * S_B) \rceil$, where t is the number of tuples and S_B is the block size. The maximum number of bitmaps *M* is split into separate M_j for each dimension by weighting *M* with the attribute's cardinality c_j using the following formula $M_j = \lfloor M (\log c_j / \Sigma_{j=1}^{d} \log c_j \rfloor$, where d is the number of dimensions, so that the space for each attribute is proportional to its cardinality.

Each M_j and the corresponding attribute's cardinality c_j is input to the algorithm *FindSmallestN* to obtain the least number of components *n* such that *Space(I)* is less than or equal to *M,* and the base values constituting the *n*-component index *I* for that dimension. The space measurement for a range encoding on an

attribute *j* in terms of the number of bitmaps is given by Space(I) = $\Sigma^{n}_{i=1}$ (b_i - 1). The metric *Time(I)* for a range encoding on an attribute *j* is in terms of the expected number of bitmap scans for a selection query evaluation and is given by Time(I) = 2 * (n - $\Sigma^{n}_{i=1}$ $1/b_i$ + 1/3 * ($1/b_1$ - 1)).

4. PERFORMANCE COMPARISON

We conduct a performance study to address the following questions:

- What are the comparative storage and retrieval cost of REBSI and PMaps in different scenarios?
- How is performance affected by parameters such as blocksize, database size, selectivity of queries, cardinality of attributes, kind of queries, and property ordering?
- Can PMap design and performance be improved using this knowledge?
- Under what conditions is it better to use either index?

We create six query sets over BENCH [G93] based on different criteria: high cardinality attributes, very high cardinality attributes, low cardinality attributes, low selectivity, high selectivity, and mixed queries. In our simulations we vary the following parameters for each query set: number of tuples (50,000 and 1,000,000), block size (2048, 4096, and 8192 bytes), word size (16 and 32 bits), and scaling factor (minimum per corresponding PMap up to 10). For each query set, we create PMaps and REBSIs and measure their performance for the same queries that were used to create them. In this study, PMap ranges are designed to exactly cover the query predicates and hence no excess tuples are retrieved. Performance is measured in terms of the number of index pages retrieved. We consider 43 different queries and perform more than 145 simulations [G02].

4.1 Methodology

We illustrate our methodology using one of our smaller test sets, the *High Cardinality Attribute Query Set* (HCAQS). General results over all the query sets are discussed in Section 5. The PMap for the HCAQS is comprised of the following four properties:

1. Range *K100k*, 2 bits: [0, 2), [2, 3), [3, 4), and [4, 100000],
2. Range *K10k*, 2 bits: [0, 2), [2, 3), [3, 4), and [4, 10000],
3. Range *K1k*, 2 bits: [0, 2), [2, 3), [3, 4), and [4, 1000], and
4. Boolean "K2 = 2," 1 bit.

Each query and its corresponding *pmask*, $pfilter_l$, and $pfilter_h$ are shown in Table 4.

A separate REBSI is created for each attribute referenced in the query set, that is, for *K2*, *K1k*, *K10k* and *K100k*. Using cost model formulae, we evaluate the number of bitmaps read and thus, the number of index pages retrieved for REBSIs with different space allocations. In this case, we create REBSIs with 5 times and 10 times the space occupied by the PMap, apart from the minimum *sf*, which is 3 in the case of 1,000,000 tuples. Table 5 shows the average number of bitmap scans for each attribute for both database sizes, with *min_sf* and 8k blocksize, and also the number of blocks to store one bitmap, *z*, in each case.

4.2 Observations

Figure 2 shows the performance comparison graph for the two techniques for the HCAQS for a database size of 1,000,000 tuples. This figure summarizes the results of 4 simulations. The *x*-axis shows the queries and the *y*-axis shows the index pages retrieved for each query; the queries are ordered in decreasing order of the difference between the average number of index pages retrieved by the PMap (*PAvg*) and the number of pages retrieved by the REBSI with the smallest scaling factor (*min_sf*), i.e., occupying the minimal required space. We show the minimum (*PMin*), maximum (*PMax*) and average (*PAvg*) number of index pages retrieved by a particular PMap, and the pages retrieved by the corresponding REBSIs with three scaling factor values, including the minimum. We analyze the performance graph to observe overall behavior for the HCAQS and to find traits to explain these trends.

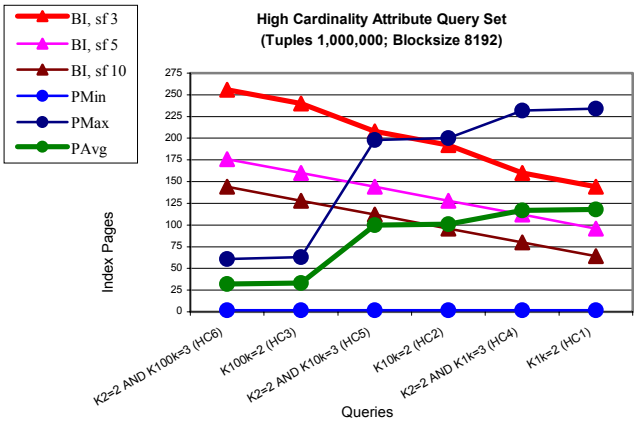

Figure 2. REBSI and PMap Performance for HCAQS

As expected, the number of index pages retrieved for each query decreases proportionally with increasing blocksize for both techniques. The varying blocksizes do not have any other impact on performance and therefore we only present the largest (8k) blocksize graph. The parameter word length (*ws*) varies with two values, 16 and 32, to create PMaps with different pstring lengths (which should be the same as or smaller than word length for efficiency). For the queries in our study, this parameter did not have any impact, as PMaps created in both cases are the same, and usually have pstring lengths less than 16. Due to the trade-off between space and time, the scaling factor (*sf*) has a predictable impact on the performance of a REBSI. REBSI performance improves proportionally to increased space allocation.

We use a scatterplot to analyze relative performance. Each (*x*, *y*) point on the graph is a query, where the *x*-value represents the number of pages retrieved by the REBSI with the minimum

Table 4. Queries, *pmasks* and *pfilters* for the High Cardinality Attribute Query Set

ID	Queries	K100k		K10k		K1k		K2	K100k		K10k		K1k		K2	K100k		K10k		K1k		K2
HC1	K1k = 2	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	1	1	1	0	1	1
HC2	K10k = 2	0	0	1	1	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	1	1
HC3	K100k = 2	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1
HC4	K2 = 2 AND K1k = 3																					

HC5	K2 = 2 AND K10k = 3	0	0	1	1	0	0	1	0	0	1	0	0	0	1	1	1	1	0	1	1	1
HC6	K2 = 2 AND K100k = 3	1	1	0	0	0	0	1	1	0	0	0	0	0	1	1	0	1	1	1	1	1

Table 5. Number of Bitmap Scans for Attributes in HCAQS

Attribute	1,000,000 tuples (min_sf=3, z=16)	50,000 tuples (min_sf=4, z=1)
K2	1	1
K1k	9	8
K10k	12	10
K100k	15	13

scaling factor, and the *y*-value represents the average number of index pages retrieved by the PMap. We group these points into one of three groups: Group A: PMap performance is relatively better, Group B: REBSI performance is relatively better, and Group C: performance of both techniques is similar. For the 1,000,000 tuple database size shown in Figure 3, all queries in the HCAQS are in Group A; for the 50,000 tuple size (not shown) only two queries are in Group A and the others are in Group C. From the scatter plot, we see that the savings for the PMap for queries with *K100k* (HC3 and HC6) are much greater than queries with *K10k* (HC2 and HC5), which again are greater than queries with *K1k* (HC1 and HC4).

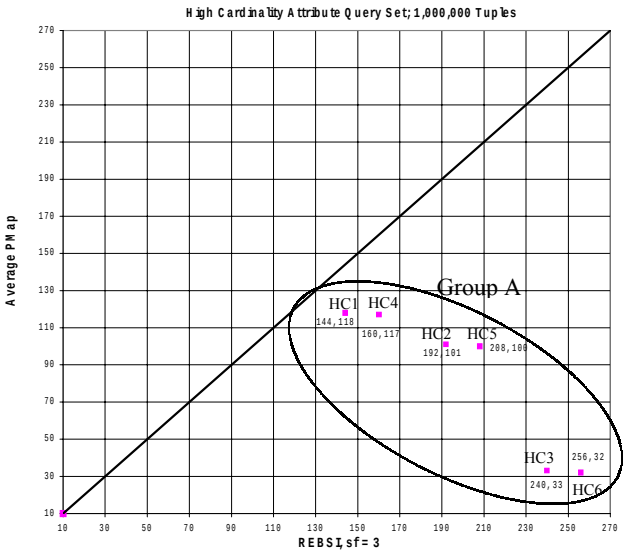

Figure 3. Scatter Plot of the HCAQS; 1,000,000 Tuples

4.3 Analysis

We discuss observations using HCAQS over three categories: impact of storage allocation, property ordering, and multi-attribute queries. We explore possible reasons to explain these observations.

1. *Impact of Storage Allocation*: In the case of the database size of 1,000,000 tuples with a blocksize of 8k, the least space that the REBSI requires to be able to create bitmaps for all the attributes is 3 times the space required by the PMap (*min_sf* = 3). For the *min_sf*, the average PMap performance is better than REBSI for all the queries in this set. For the REBSI occupying 5 times the space of PMap, the performance of the PMap is better in the first four queries, comparable in one case and worse in the last case. For the REBSI occupying 10 times the space of the PMap, the average performance of the PMap is better in the first three queries, comparable in one case, and worse in the last two queries. Thus, the REBSI performance improves as the scaling factor becomes larger. The more space allotted for the bitmaps, the better they perform.

2. *Impact of Property Ordering*: The PMap performance deteriorates for lower cardinality attributes in the HCAQS. However, this is due to the position of the properties covering these attributes in the pstring, rather than the attribute cardinality.

The further a property is from the beginning of the pstring, the lower the $pfilter_l$ value. This increases the number pstrings that have to be checked for satisfying the filter formula and thus, the number of index blocks to be retrieved. The savings for the PMap are much higher for queries with attribute *K100k* (HC3 and HC6) compared to the other queries. That is because the property covering *K100k* is the first property in the pstring and that significantly reduces the difference between $pfilter_h$ and $pfilter_l$. This is significant in other query sets as well.

3. *Multi-attribute Queries*: For each high cardinality attribute *KN*, the PMap retrieves slightly fewer pages for queries of the form *K2* AND *KN* than ones with only *KN*. Multiple conditions increase the number of bits that are set in the $pfilter_l$, which reduces the difference between $pfilter_h$ and $pfilter_l$, resulting in fewer pages retrieved. Thus, the PMap performs better than the REBSI for multi-attribute queries for the HCAQS.

5. CONCLUSIONS

We conclude the following based on our simulation results over all query sets and parameter variations.

1. The storage cost of REBSIs is higher than that of the PMap because separate REBSIs have to be created for each of the attributes accessed in the frequently used queries.

2. The position of a property covering a predicate in the pstring significantly affects the number of pages retrieved for a query accessing that predicate.

3. PMap performance is not significantly affected by variations in attribute cardinality or query selectivity.

4. REBSI retrieval cost increases as the number of attributes accessed in the query increases, even for a very small cardinality attribute such as *K2*. On the other hand, PMap retrieval cost remains the same or decreases with multi-attribute queries.

5. As the database size decreases, REBSI performance becomes better and the relative savings of PMap are reduced. This is intuitive since the number of tuples directly impacts bit-vector length and the number of blocks to read one bitmap decreases accordingly. Savings for PMaps over REBSI are higher for the larger database sizes, larger blocksizes, and high cardinality attributes.

We offer recommendations for scenarios that can benefit from using PMaps, followed by three enhancements for PMap processing and design.

5.1 Recommendations for PMap Usage

Based on our observations and analyses, we give the following general guidelines for the use of PMaps and applications where they may provide savings in query processing.

- PMap performance is not significantly affected by attribute cardinality; however the property position in the pstring is a significant factor. Since performance for most other indexing

techniques deteriorates for high cardinality attributes, we can achieve significant savings for these by creating PMaps on high and very high cardinality attributes. The property ordering should be in the decreasing order of savings desired for the attributes.

- PMaps are beneficial when space is limited for index creation, and performance savings increase as database size increases in our study.
- PMaps perform well for multi-attribute queries, even better than for single attribute queries, unlike REBSIs and most other indexing techniques. Hence, it is useful to create PMaps for frequently used multi-attribute queries.

5.2 Enhancements

Each range, Boolean or enumerated property is considered a separate property, even if two or more properties cover the same attribute. For example, consider attribute *K100* on which a Boolean property "K100 < 41" and range property "[0, 2), [2, 3), [3, 4) and [4, 101)" are defined. They are treated as separate properties *p1* and *p2*, respectively. In this case, the total number of property representations for *K100* = 2 * 4 = 8. If a query has the predicate "K100 < 41," then we do not consider *p2* at all; it is masked out in the *pmask* (100). Similarly, if the query has a range predicate, then the Boolean property *p1* is masked out in the *pmask* (011). We propose considering all the properties of one attribute as sub-properties of a single property. The value of one property limits the possibilities of the other property since they are on the same attribute. For example, if *K100* is in one of the first three intervals of the range property, then we know that "K100 < 41" is TRUE and hence, *p1* is set to 1 for range intervals 00, 01, 10. If *K100* value is in the last range interval (11), then *p1* could be either TRUE or FALSE. Thus, we have the following 5 possible property values: 100, 101, 110, 111, and 011. Combining the properties of an attribute to consider them as a single property reduces the total number of pstrings, which reduces the total storage space as well as the number of pages retrieved to answer a query.

The effects of property ordering have been observed for all the query sets and it is determined to be the most important criteria affecting PMap performance for different queries. In all cases, PMaps perform better for attributes that have properties in the beginning of the pstring. Performance deteriorates as the properties move to the right in the pstring. For example, in the case of the HCAQS, we show that the attribute cardinality has practically no impact on PMap retrieval in comparison to the property position.

6. FUTURE WORK

We propose various enhancements to the design of Pmaps and identify areas, such as the role of pstring utilization *(pu)* in the cost model, for further investigation. Future research could extend the PMap to evaluate aggregation and group by queries. A solution to the problem of finding a well-defined encoding discussed by Wu and Buchmann [WB98] can be used to make PMaps more efficient. A performance study of PMaps in a real environment with an actual database workload as well as with queries retrieving excess tuples is suggested. Application of PMaps for providing indexed access to genomic data could be explored.

7. REFERENCES

[CI98] C.Y. Chan, Y. Ioannidis, "Bitmap Index Design and Evaluation." *Proceedings of the ACM SIGMOD International Conference*, Seattle, Washington, June 1998, pp. 355-366.

[DD97] B. Dharmarajan, K.C. Davis "Property Maps: A New Secondary Access Mechanism." *Proceedings of the IRMA Conference,* Hershey, PA, May 1997.

[G93] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann Publishers, Inc., 1993.

[G02] A. Gupta, "Performance Comparison of Property Map Indexing and Bitmap Indexing for Data Warehousing," MS Thesis, ECECS Dept., University of Cincinnati, Cincinnati, OH, 45221-0030, 2002.

[K00] N. Koudas, "Space Efficient Bitmap Indexing." *Ninth International Conference on Information Knowledge Management*, McLean, Virginia, USA, Nov. 2000, pp. 194-201.

[LD02] J.G. Litton, K.C. Davis, "Heuristic Design Algorithms and Evaluation Methods for Property Maps." in preparation, 2002.

[OQ97] P. O'Neil, D. Quass, "Improved Query Performance with Variant Indexes." *Proceedings of the ACM SIGMOD Conference*, Tucson, Arizona, May 1997, pp. 38-49.

[WB98] M.C. Wu, A. Buchmann, "Encoded Bitmap Indexing for Data Warehouses." *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, USA, Feb. 1998, pp. 220-230.

[WY96] K.L. Wu, P.S. Yu, "Range-Based Bitmap Indexing for High Cardinality Attributes with Skew." *Twenty Second International Computer Software and Application Conference*, Vienna, Austria, Aug. 1998.