Accelerating Queries on Very Large Datasets*

Ekow Otoo and Kesheng Wu
Lawrence Berkeley National Laboratory

February 1, 2009

Abstract

In this chapter, we explore ways to answer queries on large multi-dimensional data efficiently. Given a large dataset, a user often wants to access only a relatively small number of the records. Such a selection process is typically performed through an SQL query in a database management system (DBMS). In general, the most effective technique to accelerate the query answering process is indexing. For this reason, our primary emphasis is to review indexing techniques for large datasets. Since much of scientific data is not under the management of DBMS systems, our review includes many indexing techniques outside of DBMS systems as well. Among the known indexing methods, bitmap indexes are particularly well suited for answering such queries on large scientific data. Therefore, more details are given on the state of the art of bitmap indexing techniques. This chapter also briefly touches on some emerging data analysis systems that don't yet make use of indexes. We present some evidence that these systems could also benefit from the use of indexes.

1 Introduction

One of the primary goals of a data management system (DBMS) is to retrieve the records under its control upon user request. In the SQL language, such retrievals are typically formed as queries. Answering these queries efficiently is a key design objective of a data management system. To achieve this goal, one needs to consider many issues including data organization, available methods for accessing the data, user interface, effective query execution planning, and the overall system design. Other chapters in this book have touched on many of these issues. In this chapter, we primarily focus on the aspects that have the most direct influence on the efficiency of query processing, which primarily include three of them: the data organization, access methods and query execution planning [67, 25, 47, 79, 99, 39].

Usually a query can be answered in different ways, for example, the tables and columns involved may be retrieved in different orders or through different access methods [44, 81, 23]. However, these choices are built on top of a set of good access methods and data organizations. Therefore, we choose to concentrate more on the issues of data organizations and access methods. Furthermore, many common types of queries on scientific data do not require complex execution plans as we explain in the next section. Thus optimizing the execution plan is less important than the other two issues. Furthermore, much of the scientific data is not under the control of a DBMS system, but are under the control of some stand-alone systems or emerging scientific data management systems. A discussion on the core data access methods and data organizations may influence the design and implementation of such systems.

On the issue of data organization, a fundamental principle of the database research is the separation of logical data organization from the physical data organization. Since most data management systems are based on software that does not have direct control of the physical organization on secondary storage systems, we primarily concentrate on logical organization in this chapter. One common metaphor of logical

*This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

data organization is the relational data model, consisting of tables with rows and columns. Some times, a row is also called a tuple, a data record, or a data object; while a column is also known as an attribute of a record, or a variable in a dataset.

There are two basic strategies of partitioning a table: the row-oriented organization that places all columns of a row together, and the column-oriented organization that places all rows of a column together. The row-oriented organization is also called the horizontal data organization, while the column-oriented organization is also known as the vertical data organization. There are many variations based on these two basic organizations, for example, a large table is often horizontally split into partitions, where each partition is then further organized horizontally or vertically. Since the organization of a partition typically has more impact on query processing, our discussion will center around how the partitions are organized. The data organization of a system is typically fixed, therefore to discuss data organization we can not avoid touching on different systems even though they have been discussed elsewhere already, most notably, Chapter 11 has extensive information about systems with vertical data organizations.

This chapter primarily focuses on access methods and mostly on indexing techniques to speed up data accesses in query processing. Because these methods can be implemented in software and have great potential of improving query performance, there have been extensive research activities on this subject. To motivate our discussion, we review key characteristics of scientific data and queries in the next section. In Section 3, we present a taxonomy of index methods. In the following two sections, we review some well-known index methods with Section 4 on single column indexing and Section 5 on multi-dimensional indexing. Given that scientific data are often high-dimensional data, we present a type of index that have been demonstrated to work well with this type of data. This type of index is the bitmap index; we devote Section 6 to discussing the recent advances on the bitmap index. In Section 7 we revisit the data organization issue by examining a number of emerging data processing systems with unusual data organizations. All these systems do not yet use any indexing methods. We present a small test to demonstrate that even such systems could benefit from an efficient indexing method.

2 Characteristics of Scientific Data

Scientific databases are massive datasets accumulated through scientific experiments, observations, and computations. New and improved instrumentations now provide not only better data precision but also capture data at a much faster rate, resulting in large volumes of data. Ever increasing computing power is leading to ever larger and more realistic computation simulations, which also produce large volumes of data. Analysis of these massive datasets by domain scientists often involve finding some specific data items that have some characteristics of particular interest. Unlike the traditional information management system (IMS), such as management of bank records in the 70's and 80's, where the database consisted of a few megabytes of records that have a small number of attributes, scientific databases typically consist of terabytes of data (or billions of records), that have hundreds of attributes. Scientific databases are generally organized as datasets. Often these datasets are not under the management of traditional DBMS system, but merely appears as a collection of files under certain directory structure or following certain naming convention. Usually, the files follow a format or schema agreed among the domain scientists.

An example of such a scientific dataset with hundreds of attributes is the data from the High Energy Physics (HEP) STAR experiments [87], that maintains billions of data items (referred to as *events*), on over hundred attributes. Most of the data files are in a format called ROOT [18, 70]. To search for a subset of the billions of events that satisfy some conditions based on a small number of attributes, requires special data handling techniques beyond traditional database systems. We address specifically some of the techniques for efficiently searching through massively large scientific datasets in this Chapter.

The need for efficient search and subset extraction from very large datasets is motivated by the requirements of numerous applications in both scientific domains and statistical analysis. Here are some such application domains:

- high energy physics and nuclear data generations from experiments and simulations,

- remotely-sensed or in-situ observations in the earth and space sciences, e.g., data observations used in climate models,
- seismic sounding of the earth for petroleum geophysics (or similar signal processing endeavors in acoustics/oceanography),
- radio astronomy, nuclear magnetic resonance, synthetic aperture radar, etc.,
- large-scale supercomputer-based models in computational fluid dynamics (e.g., aerospace, meteorology, geophysics, astrophysics), quantum physics and chemistry, etc.
- medical (tomographic) imaging (e.g., CAT, PET, MRI),
- computational chemistry,
- bioinformatic, bioengineering and genetic sequence mapping,
- intelligence gathering, fraud detection and security monitoring,
- geographic mapping and cartography
- census, financial and other "statistical" data.

Some of these applications are discussed in [95, 92, 36]. Compared with the traditional databases managed by commercial database management systems (DBMS), one immediate distinguishing property of scientific datasets is that there is almost never any simultaneous read and write access to the same set of data records. Most scientific datasets are *Read-Only* or *Append-Only*. Therefore, there is a potential to significantly relax the ACID [1] properties observed by a typical DBMS system. This may give rise to different types of data access methods and different ways of organizing them as well.

Consider a typical database in astrophysics. The archived data include observational parameters such as the detector, the type of the observation, coordinates, astronomical object, exposure time, etc. Besides the use of data mining techniques to identify features, users need to perform queries based on physical parameters such as magnitude of brightness, redshift, spectral indexes, morphological type of galaxies, and photometric properties, etc., to easily discover the object types contained in the archive. The search usually can be expressed as constraints on some of these properties, and the objects satisfying the conditions are retrieved and sent downstream to other processing steps such as statistics gathering and visualization.

The datasets from most scientific domains (with the possible exception of bioinformatics and genome data), can be mostly characterized as time varying arrays. Each element of the array often corresponds to some attribute of the points or cells in 2- or 3-dimensional space. Examples of such attributes are temperature, pressure, wind velocity, moisture, cloud cover and so on in a climate model. Datasets encountered in scientific data management can be characterized along three principle dimensions:

Size: This the number of data records maintained in the database. Scientific datasets are typically very large and grow over time to be terabytes or petabytes. This translates to millions or billions of data records. The data may span hundreds to thousands of disk storage units and are often are archived on robotic tapes.

Dimensionality: The number of searchable attributes of the datasets may be quite large. Often, a data record can have a large number of attributes and scientists may want to conduct searches based on dozens or hundreds of attributes. For example, a record of a High-Energy collision in the STAR experiment [87] is about 5 MB in size, and the physicists involved in the experiment have decided to make 200 or so high-level attributes searchable [101].

[1] Atomicity, Consistency, Isolation and Durability

Time: This concerns the rate at which the data content evolves over time. Often, scientific datasets are constrained to be *Append-Only* as opposed to frequent random insertions and deletions as is typically encountered in commercial IMS databases.

The use of traditional database management systems such as ORACLE, Sybase and Objectivity have not had much success in scientific data management. These have had only limited applications. For example a traditional relational database management system, e.g., MySQL, is used to manage the metadata, while the principal datasets are managed by domain specific data management systems such as ROOT [18, 70]. It has been argued by Gray et al. [35], that managing the metadata with a non-procedural data manipulation language combined with *data indexing* is essential when analyzing scientific datasets.

Index schemes that efficiently process queries on scientific datasets are only effective if they are built within the framework of the underlying physical data organization understood by the computational processing model. One example of a highly successful index method is the bitmap index method [4, 102, 103, 48, 21], that is elaborated upon in some detail in Section 6. To understand why traditional database management systems and their accompanying index methods such as B-tree, hashing, R-Trees, etc., have been less effective in managing scientific datasets, we examine some of the characteristics of these applications.

Data Organizational Framework: Many of the existing scientific datasets are stored in custom-formatted files and may come with their own analysis systems. ROOT is a very successful example of such a system [18, 70]. Much of astrophysics data are stored in FITS format [41] and many other scientific datasets are stored in NetCDF format [61] and HDF format [42]. Most of these formats including FITS, NetCDF and HDF, are designed to store arrays, which can be thought of as a vertical data organization. However, ROOT organizes data as objects and is essentially row-oriented.

High Performance Computing (HPC): Data analysis and computational science applications, e.g., Climate Modeling, have application codes that run on high performance computing environments that involve hundreds or thousands of processors. Often these parallel application codes utilize a library of data structures for hierarchical structured grids where the grid points are associated with a list of attribute values. Examples of such applications include finite element, finite difference and adaptive mesh refinement method (AMR). To efficiently output the data from the application programs, the data records are often organized in the same way as they are computed. The analysis programs have to reorganize them into a coherent logical view, which add some unique challenges for data access methods.

Data Intensive I/O: Often highly parallel computations in HPC also perform data intensive data inputs and outputs. A natural approach to meet the I/O throughput requirements in HPC is the use of parallel I/O and parallel file systems. To meet the I/O bandwidth requirements in HPC, the parallel counterparts of data formats such as NetCDF and HDF/HDF5 are applied to provide consistent partitioning of the dataset into chunks that are then striped over disks of a parallel file system. While such partitioning is efficient during computations that produce the data, the same partitioning is usually inefficient for later data analysis. Reorganization of the data or an index structure is required to improve the efficiency of the data analysis operations.

None-Transactional ACID Properties: Most scientific applications do not access data for analysis while concurrently updated the same data records. The new data records are usually added to the data in large chunks. This allows the data management system to treat access control in a much more optimistic manner than it is possible with traditional DBMS systems. This feature will be particularly important as data management systems evolve to take advantage of multi-core architectures and clusters of such multi-core computers, where concurrent accesses to data is a necessity.

3 A Taxonomy of Index Methods

An access method defines a data organization, the data structures and the algorithms for accessing individual data items that satisfy some query criteria. For example, given N records, each with k attributes, one very simple *access method* is that of a sequential scan. The records are stored in N consecutive locations and for any query, the entire set of records is examined one after the other. For each record, the query condition is evaluated and if the condition is satisfied, the record is reported as a hit of the query. The data organization for such a sequential scan is called the *heap*. A general strategy to accelerate this process is to augment the heap with an *index scheme*.

An *index scheme* is the data structure and its associated algorithms that improve the data accesses such as insertions, deletions, retrievals, and query processing. The usage and preference of an index scheme for accessing a dataset is highly dependent on a number of factors including:

Dataset Size: Whether the data can be contained entirely in memory or not. Since our focus is on massively large scientific datasets, we will assume the latter with some consideration for main memory indexes when necessary.

Data Organization: The datasets may be organized into fixed size *data blocks* (also referred to as *data chunks* or *buckets* at times). A data block is typically defined as a multiple of the physical page size of disk storage. A data organization may be defined to allow for future insertions and deletions without impacting the speed of accessing data by the index scheme. On the other hand the data may be organized and constrained to be *read-only*, *append-only* or both. Another influencing data organization factor is whether the records are of fixed length or variable length. Of particular interest in scientific datasets are those datasets that are mapped into very large k-dimensional arrays. To partition the array into manageable units for transferring between memory and disk storage, the array is partitioned into fixed size sub-arrays called *chunks*. Examples of such data organization methods are *NetCDF* [61], *HDF5* [42] and FITS [41].

Index Type: A subset of the attributes of a record that can uniquely identify a record of the dataset is referred to as the *primary key*. An index constructed using the *primary key* attributes is called the *primary index*, otherwise it is a *secondary index*. An index may also be classified as either *clustered* or *non-clustered* according to whether records whose primary keys are closely similar to each other are also stored in close proximity to each other. A metric of similarity of two keys is defined by the collating sequence order of the index keys.

Class of Queries: The adoption of a particular index scheme is also highly dependent on the types of queries that the dataset is subsequently subjected to. Let the records of a dataset have k attributes $A = \{A_1, A_2, \ldots A_k\}$, such that a record $r_i = \langle a_1, a_2, \ldots a_k \rangle$, where $a_i \in A_i$. Typical classes of queries include:

Exact-Match: Given k values $\langle v_1, v_2, \ldots v_k \rangle$ an exact-match query asks for the retrieval of a record $r_i = \langle a_1, a_2, \ldots a_k \rangle$ such that. $a_i = v_i, 1 \leq i \leq k$.

Partial-Match: Let $A' \subseteq A$ with $a'_j \in A'_j$ and $a_j \in A_j$ respectively, for $j \leq |A'| = k' \leq k$. Given values $\{v_1, v_2, \ldots v_{k'}\}$ for the k' attributes of A', a partial-match query asks for the retrieval of all records whose attribute values match the corresponding specified value, i.e, $a_j = v_j$, for $a_j \in A'_j, 1 \leq j \leq k'$. The exact-match query is a special class of a partial-match query.

Orthogonal-Range: For categorical attributes we assume that an ordering of the attribute values is induced by the collating sequence order of the values and for numeric attribute values the ordering is induced by their respective values. Then given k closed intervals of values $\langle [l_1, h_1], [l_2, h_2], \ldots [l_k, h_k] \rangle$ an orthogonal-range query asks for the retrieval of all records $r_i = \langle a_1, a_2, \ldots a_k \rangle$ such that $l_i \leq a_i \leq h_i, 1 \leq i \leq k$.

Partial-Orthogonal-Range: Let $A' \subseteq A$ with $a'_j \in A'_j$ and $a_j \in A_j$ respectively, for $j \leq |A'| = k' \leq k$. Under the same assumptions of attribute values as in Orthogonal-Range query, and given values $\langle [l_1, h_1], [l_2, h_2], \ldots [l_{k'}, h_{k'}] \rangle$ for the k' attributes of A', a partial-orthogonal-range query asks for the retrieval of all records whose attribute values lie the respective specified intervals, i.e, $l_j \leq a_j \leq h_j$, for $a_j \in A'_j, 1 \leq j \leq k'$. The orthogonal-range query is a special case of a partial-orthogonal-match query. Since partial-orthogonal-range query subsumes the preceding classes we will use the measure of the efficiency of processing partial-orthogonal-range queries and the measure of the efficiency of processing partial-match queries as the comparative measures of the efficiencies of the various indexing schemes to be addressed. If the columns involved in a partial-orthogonal range query vary from one query to the next, such queries are also known as *ad hoc range queries*. In the above definition, the condition on each column is of the form $l_j \leq a_j \leq h_j$. Since it specifies two boundaries of the query range, we say it is a two-sided range. If the query range only involves one boundary, e.g., $l_j \leq a_j$, $a_j \leq h_j$, $l_j < a_j$, and $a_j > h_j$, it is a one-sided range.

Other Query Processing Considerations: There are numerous query processing conditions that influence the design and choice of an index scheme besides the preceding ones. For example, the orthogonal range query is very straight forward to specify by a set of closed intervals. The range coverage may be circular, spherical or some arbitrary polygonal shape. There is a considerable body of literature that covers these query classes [79].

Attribute Types: The data type of the attribute keys, plays a significant role in the selection of an index scheme used in an access method. For example when the data type is a long alphabetic string or bit-strings, the selection of index scheme may be different from that when the data type of the index is an integer value.

Index Size Constraint: The general use of an index structure is to support fast access of data items from stored datasets. One desirably feature of an index is that its size be relatively small compared to the base data. An index may contain relatively small data structures that can fit entirely in memory during a running session of the application that uses the index. The memory resident information is then used to derive the relative positions or block addresses of the desired records. Examples of these indexing schemes include inverted indexes [47], bitmap indexing [21, 103], and direct access index (or *hashing* [47, 79]). However, when the index structure is sufficiently large, this will require that it be stored on a disk storage system and then page-in the relevant buckets of the index into memory on demand in a manner similar to the use B-Tree index and its variants [24]

Developing an efficient access method in scientific applications is one of the first steps in implementing an efficient system for a data intensive application. The key of which is to select an efficient indexing scheme for accessing the data of the application. The question that immediately arises then is what measures constitute a good metric for evaluating an efficient index scheme. Let an index scheme G be built to access N data items. For a query Q, the most important metrics are the query response time and the memory requirement, how long and how much memory does it take to answer the query. Often, the query response time is dominated by I/O operations and the number of bytes (or disk sectors) read and written is some times used as a proxy of the query response time.

Additional metrics for measuring an indexing scheme include: *recall* denoted by ρ, *precision* denoted by π and *storage utilization* denoted by $\mu(G)$. Let $r(Q)$ denote number of correct results retrieved for a query and let $R(Q)$ denote the actual number of results returned by a query in using G. Let $c(Q)$ denote the actual number of correct results desired where $c(Q) \leq r(Q) \leq R(Q)$. The *precision* is defined as the ratio of desired correct result to the number of correct results, i.e., $\pi = c(Q)/r(Q)$, and the *recall* is the ratio of the number of correct results to the number of retrieved results, i.e., $\rho = r(Q)/R(Q)$. Let $S(G)$ denote the actual storage used by an index scheme for N data items where an average index record size is b. The storage utilization is defined as $\mu(G) = bN/S(G)$.

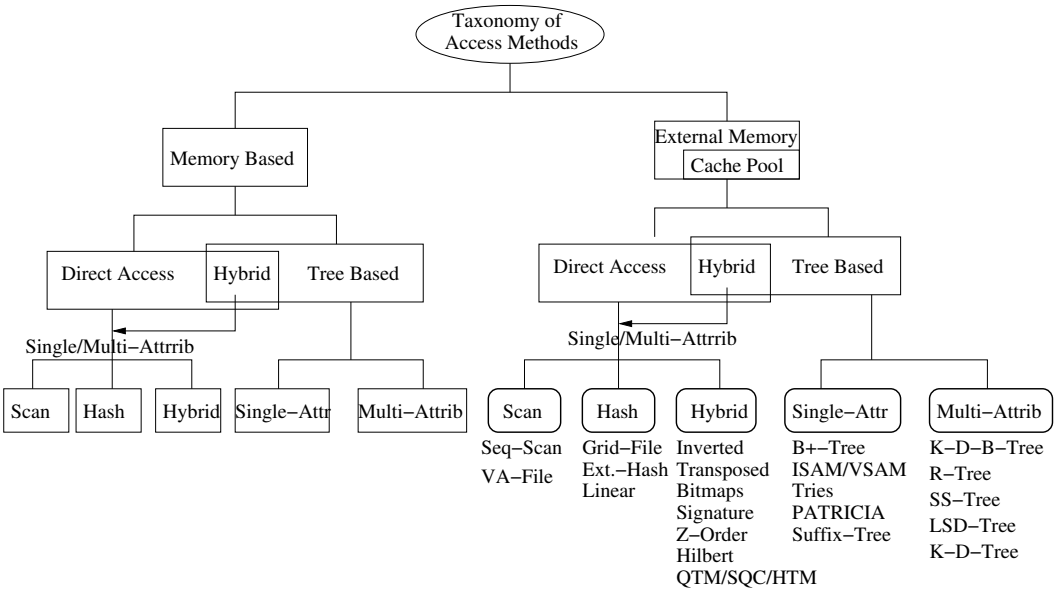


Figure 1: A Taxonomy of Indexed Access Methods

Another metric, sometimes considered, is the time to update the index. This is sometimes considered as two independent metrics; the *insertion* time and the *deletion* time. In scientific data management, deletions are hardly of concern.

There is a large body of literature on index structures. It is one of the highly researched subjects in Computer Science. Extensive coverage of the subject is given by [25, 47, 79, 99, 39]. Together, these books cover most of the indexing structures used in datasets and databases of various application domains. Index schemes may be classified into various classes. Figure 1 shows a taxonomy used to guide our review of known indexing methods.

The first level of classification distinguishes these schemes according to whether they are designed for memory resident datasets or disk resident datasets. Each class is then partitioned into three subclasses of whether accessing the data is by *direct access* method, *tree-structured* index or some combination of tree-structured index and a direct access method which we term as a *hybrid index*. Each of these subclasses can be subsequently grouped according to whether the search key is a *single attribute* or a combination of *multiple attributes* of the data item. Such combined *multi-attribute* index schemes are also referred to as *multi-dimensional* index schemes.

Since the classification of hybrid indexes is fuzzy, we will discuss them under direct access methods in this chapter. Further, to restrict the classification space to a relatively small size, we will not distinguish between single-attribute and multi-attribute indexing when we discuss both the direct-access and the hybrid index methods. There is a further level of classification that distinguishes index schemes into those that are more suitable for static data and those that more suitable of dynamic data. Most large scale scientific datasets are append-only and as such are not typically subjected to concurrent reads and writes in a manner that requires models of transactional accessing. As such, we do not consider index methods under further classification of dynamic versus static methods.

In this chapter, we focus primarily on disk-based datasets that are stored on external memory and are paged in and out of memory during processing using *memory buffers* or *cache pools*. Consequently, we address access methods, denoted as the highlighted ovals of the leaf nodes of our classification hierarchy shown in Figure 1 where we also show some examples of the indexing methods that have been used in practice. In the next sections we describe methods of accessing massively large datasets by: i) simple scans, ii) hashing, iii) single attribute indexes, iv) multi-attribute indexes, and v) hybrid schemes that are comprised of one or more combinations of hashing and scanning, hashing and tree indexes or a combination of hashing, tree index and scanning. Note also that methods that generate a single key by combining the

Rec No	Y	X	Label
1	y0	x6	A
2	y2	x6	B
3	y4	x3	C
4	y5	x3	D
5	y2	x1	E
6	y4	x1	F
7	y1	x7	G
8	y2	x5	H
9	y1	x5	I
10	y6	x4	J
11	y7	x2	K
12	y6	x7	L
13	y5	x2	M
14	y3	x0	N
15	y4	x5	N

X-Vals	Rec No
x6	1
x6	2
x3	3
x3	4
x1	5
x1	6
x7	7
x5	8
x5	9
x4	10
x2	11
x7	12
x2	13
x0	14
x5	15

X-Index List	
x0 → {14} x1 → {5, 6}	B_0
x2 → {11, 13} x3 → {3, 4}	B_1
x4 → {10} x5 → {8, 9, 15}	B_2
x6 → {1, 2} x7 → {7, 12}	B_3

Figure 2: An Example of a Single-Attribute Index.

multiple attribute values of the data, either by concatenation or bit interleaving, and then using the generated key in a single attribute index scheme, are considered under the hybrid methods.

4 Single Attribute Index Schemes

4.1 Sequential Scan Access

A natural approach to searching unordered datasets without an index is by a simple sequential scan. This gives an $O(N)$ search time for datasets of N items. Although seemingly naive and simplistic, it has the advantage that insertions are fast since new data items are simply appended at the end of the existing file. If order is maintained on either a single or combined attributes, searches can be performed in $O(\log N)$ time, using a binary search. Of course, this requires a pre-processing cost of sorting the items. As new records are added, one needs to ensure that the sorted order is preserved.

For very large high dimensional datasets, where searches are allowed on any combination of attributes, it has been shown that such simple sequential scans of processing queries can be just as effective, if not more efficient, as building and maintaining multi-dimensional indexes [97, 14, 82], particularly when nearest neighbor queries are predominant. With parallel computing and parallel file systems, a sequential scan can achieve near linear speed up with a balanced data partition. We revisit the case of sequential scans for high-dimensional datasets in Subsection 5.4.

4.2 Tree-Structured Indexing

Most applications require accessing data both sequentially and randomly. The tree structured indexing methods facilitate both of them. These indexing schemes are sometimes referred to as multi-level indexing schemes. A file organization scheme, representative of a tree structured index is the *indexed sequential access method* (ISAM). The index keys in the table shown in Figure 2 are first grouped into first level nodes, or blocks, of size b records per node. The idea is illustrated in Figure 3 that represents the ISAM organization of the blocked indexes shown in Figure 2. In the Figure 3, $b = 2$. The lowest index keys (alternatively the largest index key), of a node, each paired with its node address, are formed into records that are organized as the next higher (or second) level index of the first level nodes. The process is recursively repeated until a single node (i.e., the root node) of size b can contain all the index records that point to the lower level nodes.

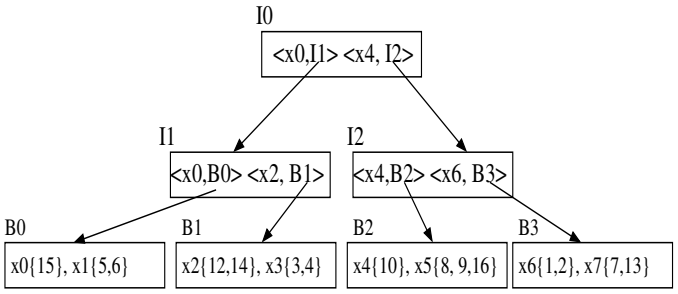


Figure 3: Illustrative Tree-Structured Index by ISAM organization. The data blocks are labeled I0, ..., I2, B0, ..., B3.

Searching for a record begins at the root node and proceeds to the lowest level node, i.e., the leaf node, where a pointer to the record can be found. Within each node, a sequential or binary search is used to determine the next lower level node to access. The search time for a random record is $O(\log_b(N/b)$ given a file of N data items. Note that, we are concerned with indexing very large datasets so that in practice $b >> 2$. The early commercial design and implementation of the indexed sequential access method was by IBM. The acronym ISAM actually implies this design approach. In this scheme the ISAM index was mapped directly to the layout of data on a disk storage where the root level index searches gives the proper cylinder number of the record. The first track of each cylinder gives the track number and this corresponds to the second level of indexing for searches. At the lowest level, this corresponds to the locations of records within a track. A sequential scan at this level is used to locate the records. The ISAM index method is a *static* index method. Subsequent insertions require the records to be managed as overflow records that are periodically merged by reorganizing the entire ISAM index.

The *indexed sequential organization* illustrates the structural characteristic of tree-structured index schemes. To circumvent the static limitations of the ISAM, the B-Tree indexing scheme was developed. Detailed coverage of the B-Tree and its variants such as B^+-Tree, B^*-Tree Prefix-B-Tree are given in [9, 25, 24, 47]. The B-Tree is a dynamic height-balanced index scheme that grows and shrinks by recursively splitting and merging nodes from the lowest level of the index tree up to the root node. The VSAM file organization [55], is a B^+-Tree that is mapped directly to the layout of data on disk storage.

Tree-structured index schemes typically maintain the fixed-sized keys, such as integers or fixed length character strings, in the index nodes. When the keys are variable length strings, then rather than storing the entire keys, only sufficient strings of leading characters that form separator keys, are stored. This is the basic idea of the prefix-B-Tree. An alternative index method for long alphabetic strings is the use of a *trie* [39, 47, 79, 99]. Suppose the keys are formed from a domain of alphabet set Σ with cardinality $|\Sigma|$. Each node of the trie-index at level i is comprised of all occurrences of the distinct i^{th} characters of keys with the same $i-1$ prefix string. A node in a trie index structure has size of at most $|\Sigma|$ entries where each entry is pair of a character and a pointer to the lower level node.

A special kind of trie, called the suffix tree [39, 47, 43, 37], can be used to index all suffixes in a text in order to carry out fast full or partial text searches. A basic trie implementation has the disadvantage of having single path which is not space efficient. A more efficient space efficient implementation of a trie index is the *PATRICIA* index [58, 47]. PATRICIA stands for Practical Algorithm to Retrieve Information coded in Alphanumeric. Tries are the fundamental index schemes for string oriented databases, e.g., very large text database used in information retrieval problems, genome sequence database, bio-informatics, etc. Tries can be perceived as generic index methods with variants such as *Suffix-Trees*, *String B-Trees* and *Burst Tries* [39, 43, 37, 47, 31, 99].

4.3 Hashing Schemes

Hashing methods are mechanisms for accessing records by the address of a record. The address is computed using a key of the record; usually the *primary key* or some unique combination of attribute values of the record. The method involves a set of n disk blocks also termed *buckets* with index values numbered from 0 to $n-1$. A bucket is a unit of storage containing one or more records. For each record $\langle k_i, r_i \rangle$, whose key is k_i and entire record is r_i, a hash function $H()$ is used to compute the bucket address I where the record is stored, i.e., $I = H(k_i)$. The result I of computing the hash function, forms the index into the array of buckets or data blocks that hold the records. A hashing scheme can either be *static* or *dynamic*.

In *static* hashing, the number of buckets is pre-allocated. Records with different search-key values may map to the same bucket, in which case we say that a *collision* occurs. To locate a record in a bucket with multiple keys the entire bucket has to be searched sequentially. Occasionally a bucket may have more records that hash into it than it can contain. When this occurs it is handled by invoking some overflow handling methods. Typical overflow handling methods include, separate chaining, rehashing, coalesced chaining, etc. To minimize the occurrence of overflow, a hash function has to be chosen to distribute the keys uniformly over the allocated buckets. Hashing methods on single attributes are discussed extensively in the literature [25, 47, 79].

An alternative to static hashing is *dynamic* hashing. Dynamic hashing uses a dynamically changing function that allows the addressed space, i.e., the number of allocated buckets, to grow and shrink with insertions and deletions of the records respectively. It embeds the handling of overflow records dynamically into its primary address space, to avoid explicit management of overflow buckets. Various techniques for dynamic hashing have been proposed. Notable among these are: dynamic hashing [51], linear hashing [54], extendible hashing [30, 68].

5 Multi-Dimensional Index Schemes

Multi-Dimensional indexing arises from the need for efficient query processing in numerous application domains where objects are generally characterized by *feature vectors*. They arise naturally from representation and querying of geometric objects such as points, lines, and polyhedra in computational geometry, graphics, multimedia and spatial databases. Other applications that have seen a considerable research in multi-dimensional data structures include data mining of scientific databases and geographic information systems. Objects characterized by k-dimensional feature vectors, are typically represented either by their spatial extents in the appropriate metric space or mapped as points in $k-$dimensional metric space that defines an appropriate metric measure of relationship between any two points [33, 15, 74, 79]. Depending on the types of queries on the objects, different data structures that recognize the spatial extents of the objects can be utilized. A more general approach however is still the mapping of these objects as points and then partitioning either the points or the embedding space. To illustrate these general approaches of representing objects that can be characterized by feature vectors, we consider the representation of line segments as in Figure 4, that are subjected to stabbing line and intersection queries.

Two well known memory resident data structures for representing such line segments for efficient processing of the stabbing line queries are the *interval tree* and *segment tree* [74, 79]. They also have corresponding disk based counterparts [96]. Since each line segment is characterized by a vector $\langle lx, rx \rangle$ of its left and right x-values, these can be mapped as points in 2-dimensional space as shown in Figure 5. The stabbing line query $Q1 = 4.5$ translates to a range query of finding all line segments whose $lx \leq 4.5$ and $rx \geq 4.5$ as depicted in Figure 5. Similarly, a query that asks for all lines that overlap line $F = \langle 2, 6 \rangle$ also translates to a range query of finding all lines whose $lx \leq 6$ and $rx \geq 2$. The shaded region of Figure 6 is the area where the points of the response to query lie. In the illustrative example given, all lines are mapped onto points that lie above the 45-degree line. It is possible to choose a mapping that distributes the points uniformly in the embedding space. The basic idea is easily extended to objects of arbitrary shapes and dimensions such as rectangles, circles, spheres, etc. For example, a collection of rectangles may be repre-

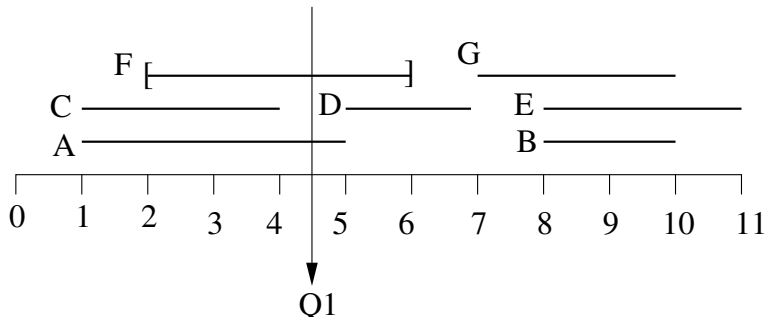


Figure 4: A collection of X-Axis parallel line segments

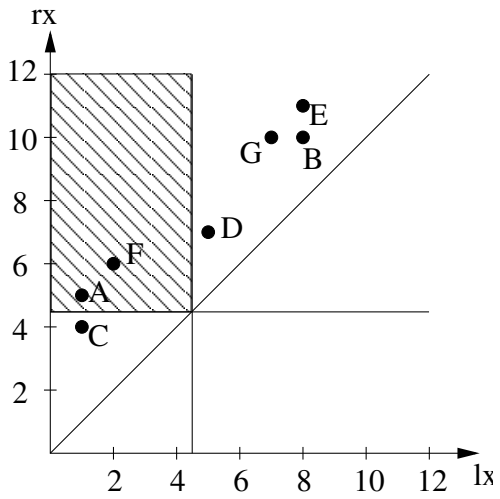


Figure 5: Shaded region representing a stabbing query.

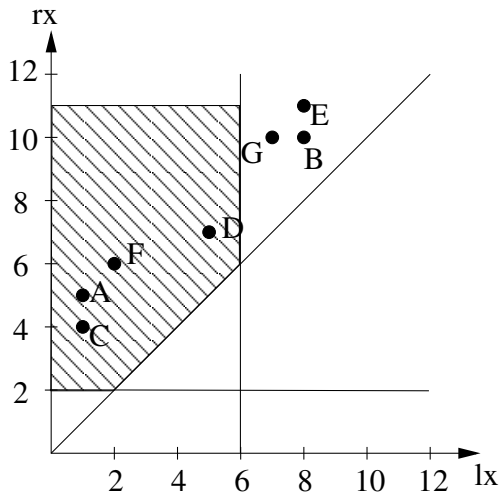


Figure 6: Shaded region representing an intersection query.

sented either explicitly as rectangles with the appropriate index structure, such as R-trees [94] layered over it to processing queries on them, or mapped as points in a four dimensional data-space. In the subsequent discussion, we focus mostly on datasets that are $k-$dimensional and are perceived as points in $k-$dimensional space.

A sample of a 2-dimensional dataset is given in the table shown in Figure 7. The first three columns give the Y-values, the X-values and the labels respectively. The mapping of the dataset as points in a two dimensional space is shown in Figure 8, and is based on the Y- and X-values. The indexing mechanism used to access the data is highly dependent on the physical organization of the datasets. In high dimensional very large scientific datasets, the datasets are typically vertically partitioned and stored by columns. Such physical storage of datasets is amenable to efficient access by bitmaps and compressed bitmap indexes [4, 21, 48, 38, 102, 103]. Another popular physical representation of datasets, particularly when the datasets are perceived as points in a k-dimensional data space, is by tessellating the space into rectangular regions and then representing each region as a *chunk* of data, i.e., all the points in a region are clustered in the same *chunk*. A chunk typically corresponds to the physical page size of disk storage but may span more than one page. The different methods of tessellating and indexing the corresponding *chunks* give rise to the numerous multi-dimensional indexing methods described in the literature. For example, the tessellation of the data space may be:

- overlapping or non-overlapping,
- flat or hierarchical, and

Y	X	Label	Linear Quad-Code	Linear Binary Gray-Code
y0	x6	A	110	010001 ⇒ 30
y2	x6	B	130	011011 ⇒ 18
y4	x3	C	211	101100 ⇒ 55
y5	x3	D	213	101110 ⇒ 52
y2	x1	E	021	001011 ⇒ 13
y4	x1	F	201	101001 ⇒ 49
y1	x7	G	113	010010 ⇒ 28
y2	x5	H	121	011111 ⇒ 21
y1	x5	I	103	010111 ⇒ 26
y6	x4	J	320	110110 ⇒ 36
y7	x2	K	232	100101 ⇒ 57
y6	x7	L	331	110010 ⇒ 35
y5	x2	M	212	101111 ⇒ 53
y3	x0	N	023	001000 ⇒ 15
y4	x5	N	301	111101 ⇒ 41

Figure 7: A Sample Multi-attribute data

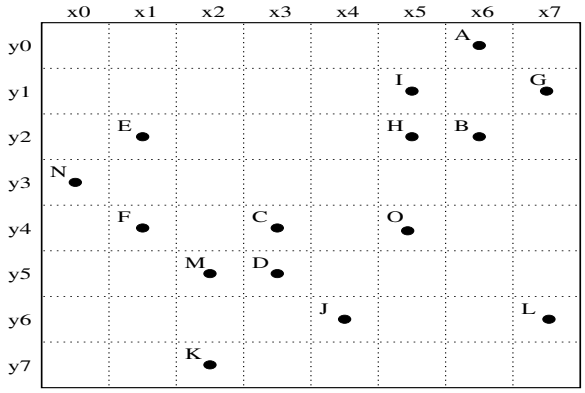


Figure 8: Multi-attribute data in 2D Space

- based on equalizing the points in each region as much as possible or based on generating equal regions of spatial extents.

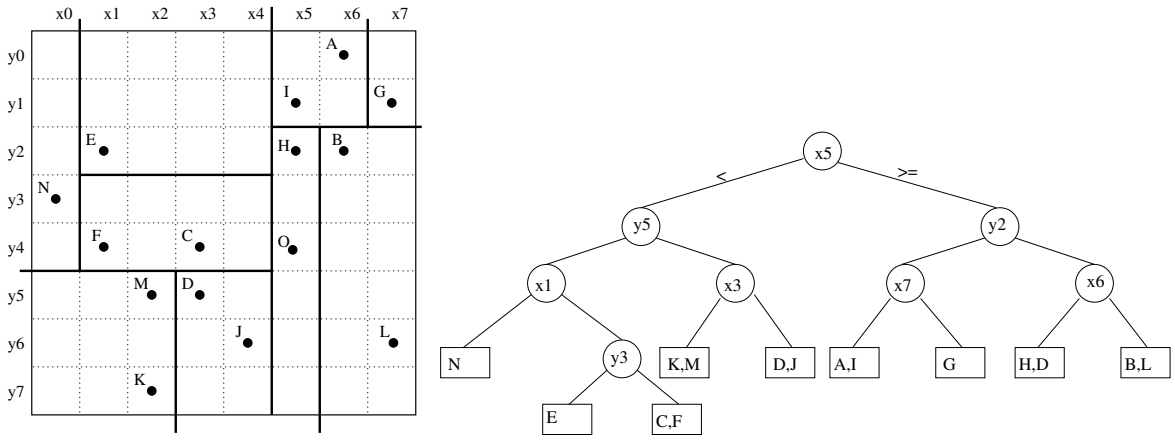
These chunking methods give rise to typical formats for *array files*. Examples of *array files* include the file formats known as NetCDF (network Common Data Format) [61], HDF4/HDF5 (Hierarchical Data Format) [42]. We briefly explore some of the widely used indexing methods that are based on tessellation of the data-space.

5.1 Multi-Dimensional Tree-Structured Indexing Methods

From the general approach of designing multi-dimensional index schemes, we examine now some special cases that lead to the class of tree-structured multi-dimensional indexing.

5.1.1 K-D-Tree and LSD-Tree

The *K-D-Tree* [11, 33, 25] was designed as a memory resident data structure for efficient processing of range-, partial-match and partial-range queries. Given the k-dimensional data-space of a collection of points, the K-D-Tree is derived by tessellating the space with (k-1)-dimensional hyper-planes that are parallel to all but the axis being split. The split plane occurs at the position that splits the number of points as evenly as possible within the subspace being split. The choice of which axis to split is done cyclically. The subspace



(a) K-D-Tree Induced Region Partitioning

(b) Region K-D-Tree (Also LSD-Tree)

Figure 9: K-D-Tree Indexing of 2-D dataset

tessellation proceeds recursively until the number of points in a region forms the desired chunk size of the data-space. Figure 9a illustrates the idea for a 2D-space tessellation that is represented explicitly as a tree structure in Figure 9b, for a chunk size of 2. Such a K-D-Tree is referred to as a region K-D-Tree as opposed to a point K-D-Tree. Rather than maintaining the leaf nodes, i.e., the data chunks, in memory these can be maintained on disks while the internal nodes of comparator values remain in memory. This variant of the K-D-Tree is referred to as the LSD-Tree (or Local Split Decision Tree), and is more suitable of large disk resident datasets. The idea is that when insertion causes a data chunk to exceed its capacity, a decision can be made to split the data chunk and to insert the split value as an internal node with pointers to the newly created leaves of data chunks. The memory resident tree nodes can always be maintained persistent by off-loading onto disk after a session and reloading before a session.

The complexity of building the a K-D-Tree of N points takes $O(N/B \log_2(N/B))$, for a chunk size of B. Insertion and deletion of a new point into a K-D-Tree takes $O(\log(N/B))$ time. One key result of the K-D-Tree is that partial-match and partial-range queries that a query involving s of k dimensions take $O((N/B)^{1-s/k} + r)$ time to answer, where r is the number of the reported points, and k the dimension of the K-D-Tree.

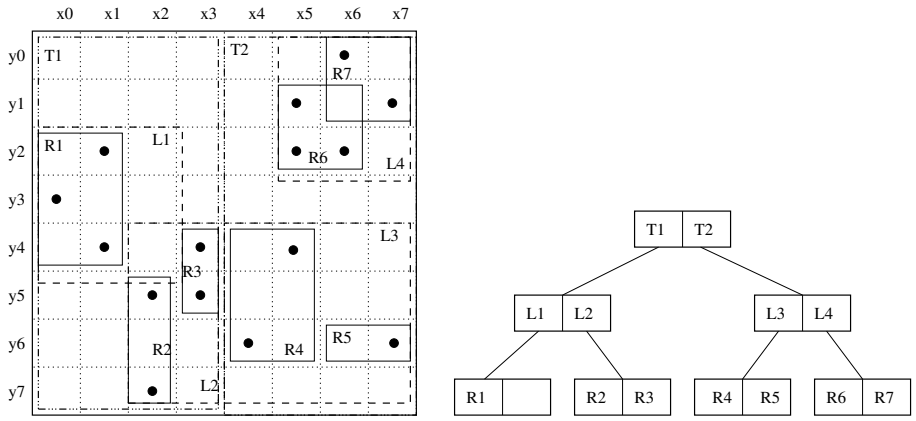
5.1.2 R-Tree and its Variants

The R-Tree, first proposed by Guttman [40], is an adaptation of the B^+-Tree to efficiently index objects contained in k-dimensional bounding boxes. The R-Tree is a height balanced tree consisting of internal and leaf nodes. The entries in the leaf nodes are pairs of values of the form $\langle RID, R\rangle$, where RID is the row identifier and R is a vector of values that defines the minimum rectilinear bounding box (MBR) enclosing the object. Note that the object may simply be a point. An internal node is a collection of entries of the form $\langle ptr, R\rangle$, where ptr is a pointer to a lower level node of the R-Tree, and R is a minimum bounding box that encloses all MBRs of the lower level node. Figure 10a, shows a collection of points grouped into rectangular boxes of at most 3 points per box and indexed by an R-Tree.

Like the B^+-Tree, an R-Tree specifies the maximum number of entries B, that can be contained in a node and satisfies the following properties:

i. A leaf node contains between $B/2$ and B entries unless it is the root node.

ii. For an entry $\langle RID, R\rangle$ in a leaf node, R is the minimum rectilinear bounding box of the object RID.



(a) Rectilinear Bounded Regions

(b) R-Tree Representation

Figure 10: R-Tree Indexing of Bounded Rectilinear 2-D dataset

iii. An internal node contains between $B/2$ and B entries unless it is the root node.

iv. For an entry $\langle ptr, R\rangle$ in an internal node, R is the minimum rectilinear bounding box that encloses the MBRs in the node pointed to by ptr.

v. A root node can contain at least two children unless it is a leaf node.

vi. All leaf nodes appear at the same level.

The R-Tree representation of the rectangular regions of Figure 10a is shown in Figure 10b. Since the introduction of the R-Tree, various variants have been introduced. The R-Tree portal [94] gives implementation codes and papers of the different variants. It has had numerous applications in spatial databases, GIS, VLSI design and applications that depend on nearest neighbor searching in low multi-dimensional space. The R-Tree and its variants use rectilinear bounding boxes. The use of other geometric shapes as bounding boxes, such as circles/spheres have led to the development of similar index schemes such as the SR-Tree [46],

5.2 Multi-Dimensional Direct Access Methods

The conceptualized optimal multi-dimensional access method is one that can be correctly defined as a *dynamic order preserving multi-dimensional extendible hashing method*. The idea being that the location where a record is stored is derived by a simple computation; the utilized data space of the dataset will grow and shrink with insertions and deletions of data items and accessing records in consecutive key order should be just as efficient as a simple sequential scan of the data items given the first key value. Such a storage scheme is impossible to realize. However numerous close approximations to it have been realized. Notable among these is the *Grid-File* [62]. Other related multi-dimensional storage schemes are the optimal partial-match retrieval method [2], the multi-dimensional extendible hashing [69], and the BANG-file [32]. Other similar methods are also presented in [33, 79].

Consider the mapping of the dataset of Figure 7, as points in a two dimensional space as shown in Figure 8. The mapping is based on the Y- and X-values. A 2-dimensional Grid-File partitions the space rectilinearly into a first level structure of a grid-directory. The Y-values define a Y-axis that is split into I_Y segments. Similarly the X-values define an X-axis that is split into I_X segments. The grid-directory is comprised of an array of $I_Y \times I_X$ elements. An element of the grid-directory array stores the bucket address of the data buckets where the data item is actually stored. Given a key value $\langle y, x\rangle$, the y-value is mapped

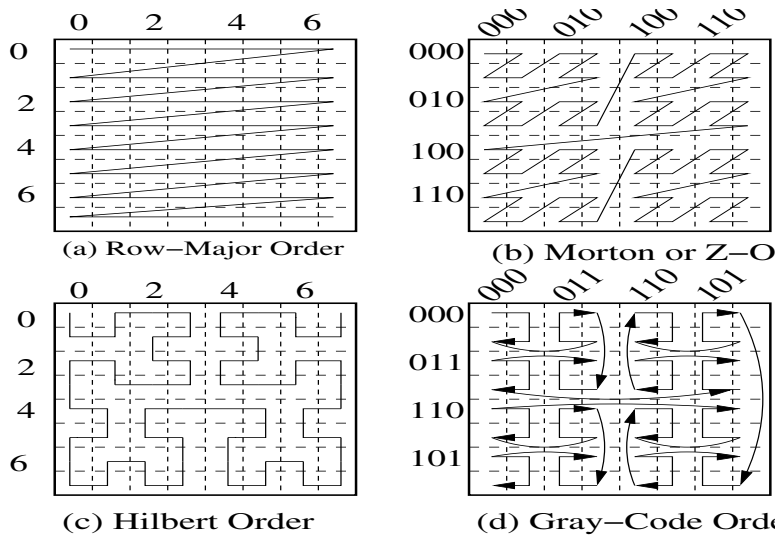


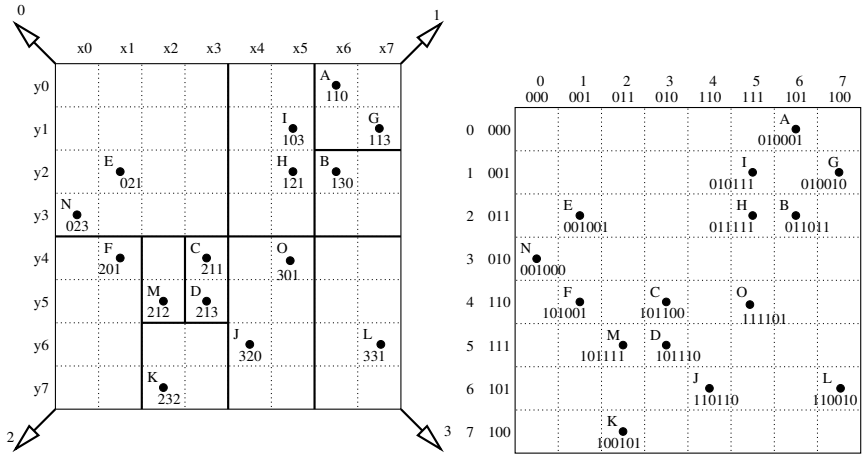
Figure 11: Examples of 2-dimensional Space Curves

onto a y-coordinate value i_y and the x-value is mapped onto an x-coordinate value i_x. A look up of the grid-directory array entry $\langle i_y, i_x \rangle$ gives the bucket address where the record can be found or inserted. Since grid-directory and data buckets are disk resident, accessing an element using the grid director requires at most 2 disk accesses. The basic idea, illustrated with a 2-dimensional key space, can easily be generalized to arbitrary number of dimensions. For a dataset of n-buckets, a partial-match retrieval that specifies s out k-dimensional values can be performed in $O(n^{1-s/k} + r)$ time where r is the number of records in the response set.

5.3 Hybrid Indexing Methods

The term *hybrid index* refers to index structures that are composed of two or more access structures such as hashing (or direct access method), tree-based indexes, and simple sequential scans. In the preceding section on *Grid-File*, we saw that in the 2-dimensional grid-directory mapping, the key value of the form $\langle y, x \rangle$ is first translated into a $\langle i_y, i_x \rangle$ coordinate index that is used as an index of an array entry. The use of the $\langle i_y, i_x \rangle$ coordinate index is actually a mapping onto a 1-dimensional array which in a 2-dimensional array, happens to be either a row-major or a column-major addressing method. In general, given a k-dimensional index $K = \langle i_0, i_1 \ldots i_{k-1} \rangle$ one can generate a 1-dimensional mapping denoted as I_K, of the k-dimensional key and then use this in any of the tree-based index schemes described in the preceding sections. There are different methods by which 1-dimensional mapping can be formed. These are formally referred to as *space filling curves* [78]. Figure 11 gives some typical space filling curves generated from a 2-dimensional index and mapped onto 1-dimensional codes.

The most popular of the space filling curves that have been used in numerous application is the Morton or Z-order mapping. It is generated simply by k-cyclic interlacing of the bits of the binary representation of the k-dimensional index of the data-space. Its simplicity and the fact that it is consistent with the *linear quad-tree* encoding of the space, makes it one of the most widely encoding methods in applications [34, 52, 71, 79]. Alternative methods to the Z-order encoding are the Hilbert order encoding [57] and the Gray-Code encoding [79]. In the table shown in Figure 7, column 4 shows the linear quad-code (or *Z-order* code), generated from a 8×8 grid partitioning of the data-space. Note that linear quad-code is a base k string of digits formed by taking k-bits of the Z-order codes. The Gray-Code encodings of the points are shown in column 5. The Figures 12a and 12b depict the spatial representations and code labels of points in a 2-D data-space for the Z-order and Gray-Code encoding respectively.



(a) Morton-Order (or Z-Order) Labeling (b) Gray-Code Order Labeling

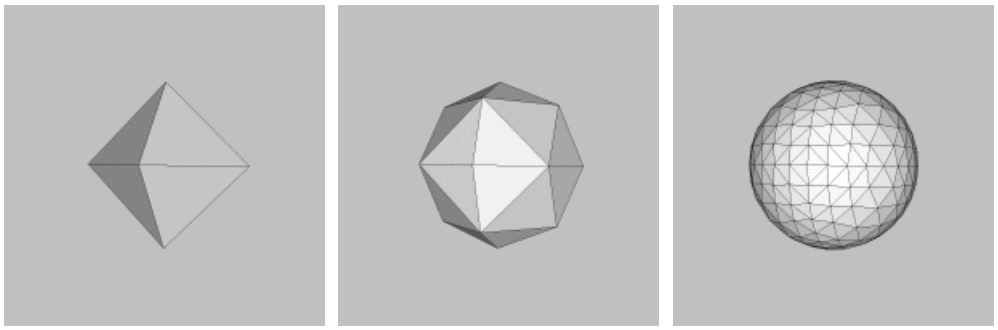
Figure 12: Hybrid Indexing by Space-Filling Curves of 2-D dataset

5.3.1 Quaternary/Hierarchical Triangular Mesh

While most database applications deal with planar and hyper-rectilinear regions, some large scientific applications deal with datasets that lie on spherical surfaces. Examples of these are datasets from climate models, GIS and Astronomy. The approach to indexing regions on spherical surfaces is similar to the method of linear quad-code or Morton-sequence order encoding of planar regions. The basic idea is to approximate the sphere by a base *platonic solid* such as a tetrahedron, hexahedron (or cube), octahedron, icosahedron, etc. If we consider say the octahedron, a spherical surface is approximated at level 0 by 8 planar triangular surfaces. By bisecting the mid-points of each edge and pushing the mid-pints along a ray from the center of the sphere that passes through the mid-point to the surface, 4×8 triangular surfaces are generated at level 1. Using such recursive edge bisection procedure, the solid formed from the patches of triangular planes approximates closer and closer to the sphere. The process is depicted in Figure 13. Two base platonic solids that have been frequently used in such approximation schemes of the sphere are the octahedron and the icosahedron shown in Figures 14a and 14b respectively. Consider the use of an inscribed octahedron as the base solid. Indexing a point on a spherical surface, at any level of the tessellation, amounts then to indexing its projection on the triangular patch that the point lies at that level. If we now pair the triangular upper and lower triangular patches to form four quadrants, then the higher level tessellation of each quadrant is similar to the higher level tessellation of planar regions that can now be encoded using any of the space filling curve encoding scheme. In the Figure 14a we illustrate such a possible encoding with the Z-order encoding. Variations of the technique just described, according to whether the base inscribing platonic solid is either an octahedron, cube or an icosahedron, and the manner of labeling the triangular regions form the basis of the various techniques known as the Quaternary Triangular Mesh (QTM), Hierarchical Triangular Mesh, Semi-Quadcode (SQC), etc., [7, 8, 53, 79, 93].

5.4 Simple Scanned Access of Multi-Dimensional Datasets

Most of the multi-dimensional access methods described in preceding sections are only suitable for low dimensional datasets of the order of $k \leq 8$. For datasets with higher dimensionality the *Curse of Dimensionality* sets in. The term curse of dimensionality was coined by Bellman to describe the problem caused by the exponential increase in volume associated with adding dimensions to a metric space [10]. This has two main implications on an indexing method. As the number of dimensions of data increases, the index size increases as well. Such indexes are usually only effective for exact-match queries and range queries

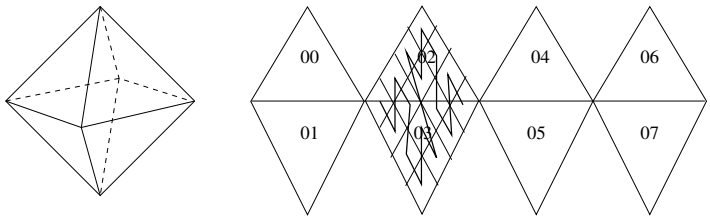


(a) Inscribed Base Octahedron

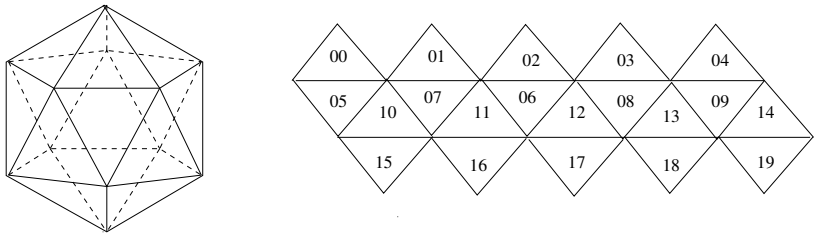
(b) Level 1 Hierarchical Tessellation

(c) Level 3 Hierarchical Tessellation

Figure 13: Spherical Indexing with HTM (Courtesy of [93]).



(a) Spherical Indexing by Base Octahedron



(b) Spherical Indexing by Base Icosahedron

Figure 14: Spherical Indexing with Quaternary/Hierarchical Triangular Mesh

where all indexed dimensions are used, if only a few of the indexed dimensions are used, the effectiveness of the index deteriorates dramatically.

Many index methods have been proposed to address the curse of dimensionality including the well-known X-Tree [13] and pyramid tree [12]. These methods typically address the index size issue, but fail to address the performance issue on partial range queries. For a k-dimensional data structure partitioned into n buckets, the complexity of processing a partial range query involving s out of k dimensions is $O(n^{1-s/k}+r)$ for both X-Tree and pyramid tree. If $s = d$, which is the case for an exact match query or a full range query, $O(1)$ buckets are accessed. If k is large and $s = 1$, nearly all the pages are accessed. This prompted the question of whether a simple sequential scan is satisfactory for high dimensional data sets [14, 82]. In the case of similarity searches in high-dimensional space, one effective method is the use of sequential scan but with the attribute values mapped onto fixed length strings of about $64\tilde{}128$ bits. The method is termed the VA-File approach [97]. Other more efficient methods are the bitmap indexes described in the next section.

RID	**A**	bitmap index			
		=0	=1	=2	=3
1	0	1	0	0	0
2	1	0	1	0	0
3	2	0	0	1	0
4	2	0	0	1	0
5	3	0	0	0	1
6	3	0	0	0	1
7	1	0	1	0	0
8	3	0	0	0	1
		b_1	b_2	b_3	b_4

Figure 15: An illustration of the basic bitmap index for a column **A** that can only take on four distinct values from 0 to 3. Note RID is the short-hand for "row identifiers".

6 Bitmap Index

In this section, we separately review the current work on bitmap indexes because it is a more effective in accelerating query processing on large scientific datasets than other techniques reviewed earlier. The bitmap indexes are generally efficient for answering queries. In fact, certain compressed bitmap indexes are known to have the theoretically optimal computational complexity [103]. They are relatively compact compared to common implementations of B-Trees, and they scale well for high-dimensional data and multi-dimensional queries. Because they do not require the data records to be in any particular order, they can easily take on data with any organization to improve the overall data processing task beyond the querying step.

In this section, we explain the key concept of bitmap index, and review three categories of techniques for improving bitmap indexes: compression, encoding and binning. We end this section with two case studies on using a particular implementation of bitmap indexes called *FastBit*[2].

6.1 The basic bitmap index

Figure 15 shows a logical view of the basic bitmap index. Conceptually, this index contains the same information as a B-tree [24, 65]. The key difference is that a B-tree would store a list of Row IDentifiers (RIDs) for each distinct value of column **A**, whereas a bitmap index represents the same information as sequences of bits, which we call *bitmaps*. In this basic bitmap index, each bitmap corresponds to a particular value of the column. A bit with value 1 indicates that a particular row has the value represented by the bitmap. What is required here is a durable mapping from RIDs to positions in the bitmaps [63].

The mapping used by the first commercial implementation of bitmap index [65] is as follows. Let m denote the maximum number of rows that can fit on a page, assign m bits for each page in all bitmap. The first record in a page is represented by the first bit assigned for the page, the second record by the second bit, and so on. If a page contains less than m records, then the unused bitmaps in the bitmap are left as 0. An additional existence bitmap may be used to indicate whether a bit position in a bitmap is used or not. Such existence bitmap may also be used to indicate whether a particular record has been deleted without recreating the whole bitmap index. This mapping mechanism is robust to changes and can be applied to all bitmap indexes of a table.

In most scientific applications, data records are stored in densely packed arrays [35, 49, 59], therefore, a more straightforward mapping between the RIDs and positions in bitmaps can be used. Furthermore, most scientific data contain only fix-sized data values, such as integers and floating-point values, and are stored in multi-dimensional arrays. In these cases, the array index is a durable mapping between bit positions and data records. Usually such RIDs are not stored explicitly.

[2]FastBit software is available from `https://codeforge.lbl.gov/projects/fastbit/`.

The bitmap indexes are particular useful for query-intensive applications, such as data warehousing and OLAP [66, 112]. One of the key reasons is that queries can be answered with bitwise logical operations on the bitmaps. In the example shown in Figure 15, a query "$\mathbf{A} < 2$" can be answered by performing bitwise OR on b_1 and b_2 ($b_1 \mid b_2$). Since most computer hardware support such bitwise logical operations efficiently, the queries can be answered efficiently in general. Another key reason is that answers from different bitmap indexes can be easily combined. This is because the answers from each bitmap index is a bitmap and combining the different answers simply requires additional bitwise logical operations. Because combining answers from different indexes efficiently is such an important consideration, a number of DBMS that do not support bitmap indexes, such as PostgreSQL and MS SQL server, even convert intermediate solutions to bitmaps to combine them more effectively.

Because results from different indexes can be efficiently combined, a bitmap index is built for one column only, and composite bitmap indexes for multiple columns are rarely used. This simplifies the decisions on what indexes to build because one does not need to consider composite indexes. This also simplifies the query optimization because there are less indexes to consider.

The biggest weakness of the basic bitmap index is that its size grows linearly with the number of distinct values of the column being indexed. Next we review three sets of strategies to control the index sizes and improve the query response time, namely, compression, encoding and binning.

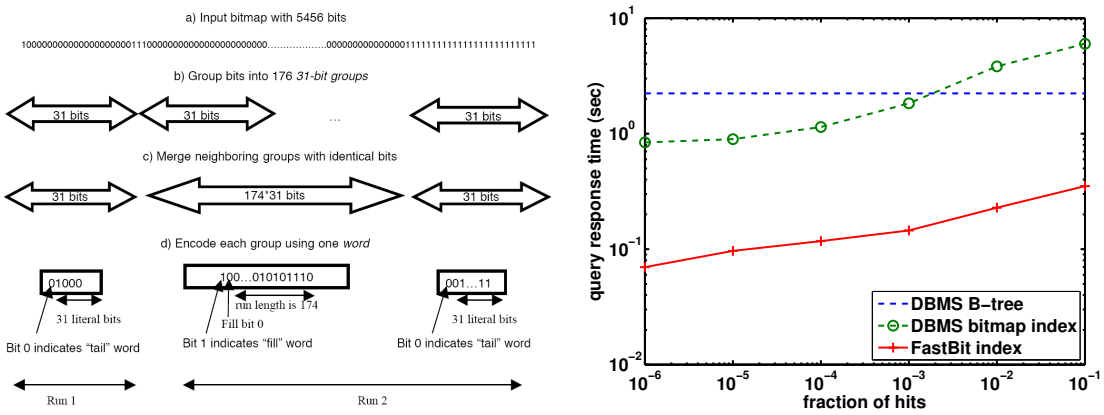
6.2 Compression

Each individual bitmap in a bitmap index can be compressed with a data compression method [60]. Any lossless compression may be used. However, the specialized bitmap compression methods typically offer faster bitwise logical operations and consequently faster query response time [3, 45]. The most widely used bitmap compression method is Byte-aligned Bitmap Code (BBC) [4, 5]. More recently, another bitmap compression method called Word-Aligned Hybrid (WAH) code was shown to perform bitwise logical operations more than 10 times faster than BBC [107, 109].

WAH gains its speed partly from its simplicity. For long sequences of 0s or 1s, it uses run-length encoding to represent them and for relatively short sequences of mixed 0s and 1s, it represents the bits literally. Hence, it is a hybrid of two methods. Another key feature that enables it to achieve performance is that the compressed data are word-aligned. More specifically, WAH compressed data contains two types of words; *literal* words and *fill* words.

A *literal* word contains one bit to indicate its type and uses the remaining bits to store the bitmap literally. On a 32-bit word system, it may use the most significant bit to indicate the type of the word, and use the remaining 31-bit to store the bitmap. A *fill* word similarly needs 1 bit to indicate its type. It uses another bit to indicate whether the bits are all 0s or all 1s, and the remaining bits is used to store the number of bits in a bitmap it represents. The number of bits represented by a WAH *fill* word is always a multiple of the number of bits stored in a *literal* word. Therefore, the length of a *fill* is stored as this multiple instead of the actual number of bits. For example, a *fill* of 62-bits will be recorded as being of length 2 because it is 2 times the number of bits that can be stored in a *literal* word (31). This explicitly enforces the word-alignment requirement and allows one to easily figure out how many *literal* words a *fill* word corresponds to during a bitwise logical operation. Another important property is that it allows one to store any *fill* in a single *fill* word as long as the number of bits in a bitmap can be stored in a word. This is an important property that simplifies the theoretical analysis of WAH compression [103]. An illustration of WAH compression on a 32-bit machine is shown in Figure 16(a).

Figure 16(b) shows some examples to illustrate the effects of compression on overall query response time. In this case the commercial DBMS implementation of compressed bitmap index (marked as "DBMS bitmap index") uses BBC compression, while "FastBit index" uses WAH compression. The query response time reported are average time values over thousands of ad hoc range queries that produce the same number of hits. Over the whole range of different number of hits, the WAH compressed indexes answer queries about 14 times faster than the commercial bitmap indexes.



(a) an illustration of WAH

(b) time (s) to answer queries

Figure 16: The effects of compression on query response time. The faster WAH compression used in FastBit reduces the query response time by an order of magnitude. (Illustration adapted from [89])

In addition to being efficient in timing measurements, WAH compressed basic bitmap index is also theoretically optimal. In the worst case, the query response time is a linear function of the number of hits according to our analysis in [102, 103]. A few of the best B-tree variants have the same theoretical optimality as the WAH compressed bitmap index [24]. However, bitmap indexes are much faster in answering queries that return more than a handful of hits as illustrated in Figure 16. Since the basic bitmap index contains the same information as a typical B-tree, it is possible to switch between bitmaps and RID lists to always use the more compact representation as suggested in the literature [65]. This is an alternative form of compression that was found to perform quite well in a comparison with WAH compressed indexes [63].

The bitmap compression methods are designed to reduce the index sizes and they are quite effective at this. Discussions on how each compression method controls the index size are prominent in many research articles. Since there is plenty of information on the index sizes, we have chosen to concentrate on the query response time. Interested readers can obtain more information on discussions on the index sizes in [109, 45].

6.3 Bitmap encoding

Bitmap encoding techniques can be thought of as ways of manipulating the bitmaps produced by the basic bitmap index to either reduce the number of bitmaps in an index or reduce the number of bitmaps needed to answer a query. For example, to answer a range query of the form "$\mathbf{A} < 3$" in the example given in Figure 15, one needs to OR the three bitmaps b_1, b_2 and b_3. If most of the queries involve only one-sided range conditions as in this example, then it is possible to store C bitmaps that correspond to $\mathbf{A} \leq a_i$ for each of the C distinct values of $\mathbf{A}$. We call C the *column cardinality* of $\mathbf{A}$. Such a bitmap index would have the same number of bitmaps as the basic bitmap index, but can answer all one-sided range queries by reading one bitmap. This is the *range encoding* proposed by Chan and Ioannidis [21]. The same authors also proposed another encoding method called the *interval encoding* that uses about half as many bitmaps as the basic bitmap index, but answers any range queries with only two bitmaps [22]. The encoding used in the basic bitmap index is commonly referred to as the *equality encoding*. Altogether, there are three basic bitmap encoding methods: equality, range, and interval encodings.

The three basic bitmap encodings can be composed together to form two types of composite encoding schemes: multi-component encoding [21, 22] and multi-level encoding [86, 108]. The central idea of multi-component encoding is to break the key value corresponding to a bitmap into multiple components in the same way an integer number is broken into multiple digits in a decimal representation. In general, each "digit" may use a different basis size. For an integer attribute with values between 0 and 62, we can use two

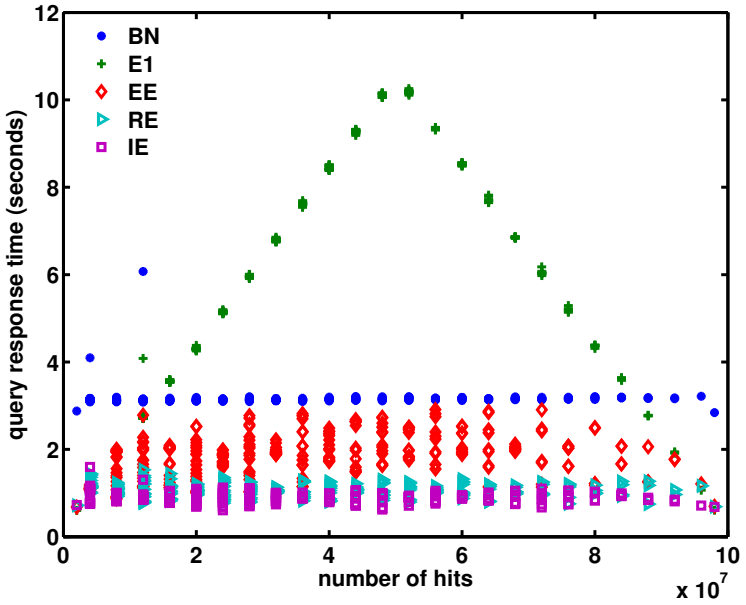


Figure 17: The query response time of five different bitmap encoding methods with WAH compression (BN: binary encoding, E1: the basic one-component equality encoding, EE: two-level equality-equality encoding, RE: two-level range-equality encoding, IE: two-level interval-equality encoding).

components of basis sizes 7 and 9, and index each component separately. If the equality encoding is used for both components, then we use 7 bitmaps for one component and 9 bitmaps for the other. Altogether we use 16 bitmaps instead of 63 had the equality encoding been used directly on the key values. It is easy to see that using more components can reduce the number of bitmaps needed, which may reduce the index size. To carry this to the extreme, we can make all base sizes 2 and use the maximum number of components. This particular multi-component encoding can be optimized to be the binary encoding [100], which is also known as the bit-sliced index [64, 66]. This encoding produces the minimum number of bitmaps and the corresponding index size is the smallest without compression. A number of authors have explored different strategies of using this type of encoding [20, 112].

To answer a query using a multi-component index, all components are typically needed, therefore, the average query response time may increase with the number of components. It is unclear how many components would offer the best performance. A theoretical analysis concluded that two components offer the best space-time trade-off [21]. However, practitioners have stayed away from two-component encodings; existing commercial implementations either uses one-component equality encoding (the basic bitmap index) or the binary encoding. This discrepancy between the theoretical analysis and the current best practice is because the analysis has failed to account for compression which is a necessary part of a practical bitmap index implementation.

A multi-level index is composed of a hierarchy of nested bins on a column. Since a level in such an index is a complete index on its own, a query may be answered with one or a combination of different levels of a multi-level index. Therefore, this type of composite index offers a different type of trade-off than the multi-component index [86, 108]. We will give more detailed information about the multi-level indexes in the next subsection after we explain the basic concept of binning.

Because of the simplicity of WAH compression, it is possible to thoroughly analyze the performance of WAH compressed indexes [110]. This analysis confirms the merit of the basic equality encoding and the binary encoding. Among the multi-level encodings, the new analysis reveals that two levels are best for a variety of parameter choices. More specifically, it identifies three two-level encoded indexes that have the same theoretical optimality as the WAH compressed basic index, but can answer queries faster on average. Figure 17 shows some timing measurements to support the analysis. In this case, we see that two-level

encodings (equality-equality encoding EE, range-equality encoding RE, and interval-equality encoding IE) can be as much as ten times faster than the basic bitmap index (marked E1). On the average, the two-level encoded indexes are about 3 to 5 times faster than both the basic bitmap index and the binary encoded index (BN) [110].

6.4 Binning

Scientific data often contains floating-point values with extremely high column cardinality. For example, the temperature and pressure in a combustion simulation can take on a large range of possible values and each value rarely repeats in a dataset. The basic bitmap index will generate many millions of bitmaps in a typical dataset. Such indexes are typically large and slow to work with, even with the best compression and bitmap encoding. We observed that such precise indexing is often unnecessary since the applications don't usually demand full precision. For example, a typical query involving pressure is of the form "pressure $> 2 \times 10^7$ Pascal." In this case, the constant in the query expression has only one significant digit. Often, such constants have no more than a few significant digits. One may take advantage of this observation and significantly reduce the number of bitmaps used in a bitmap index.

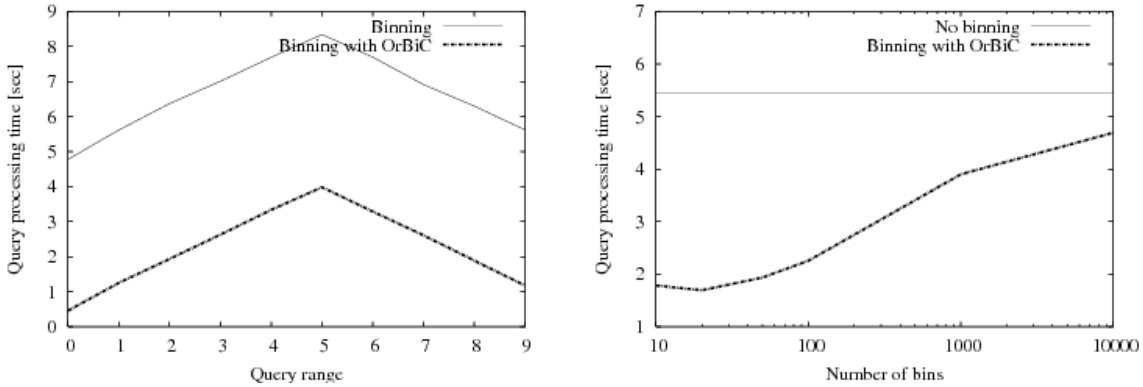
In general, the technique of grouping many values together is called *binning* [48, 83, 88, 105]. The values placed in a bin are not necessarily consecutive values [48]. However, the most common forms of binning always place values from a consecutive range into a bin. For example, if the valid pressure values are in the range between 0 and 10^9, we may place values between 0 and 1 in the first bin, values between 1 and 10 in the second bin, values between 10 and 100 in the third bin, and so on. This particular form of binning is commonly known as logarithmic binning. To produce a binned index that will answer all range conditions using one-digit query boundaries, we can place all values that round to the same one-digit number into a bin[3].

A simple way to divide all pressure values between 0 and 10^9 into 100 bins would be to place all values between $i \times 10^7$ and $(i+1) \times 10^7$ in bin i. We call them *equal-width bins*. Since each equal-width bin may contain a different number of records, the corresponding bitmaps will have varying sizes and the amount of work associated with them will be different too. One way to reduce this variance is to make sure that each bin has the same number of records. We call such bins *equal-weight bins*. To produce such equal-weight bins, we need to first find the number of occurrences for each value. Computing such detailed histogram may take a long time and a large amount of memory, because there may be a large number of distinct values. We can sample the data to produce an approximate histogram, or produce a set of fine-grain equal-width bins and coalesce the fine bins into the desired number of nearly equal-weight bins.

The multi-level bitmap indexes are composed of multiple bitmap indexes; each one corresponding to a different granularity of binning. To make it easier to reuse information from different levels of binning, we ensure that the bin boundaries from coarser levels are a subset of those used for the next finer level of bins. This generates a hierarchy of bins. To minimize the average query processing cost, the multi-level bitmap indexes mentioned in the previous subsection always uses equal-weight bins for the coarse levels. These indexes all use two levels of bins with the fine level having one bitmap for each distinct value. We consider such indexes to be *precise* indexes because they can answer any queries with the bitmaps, without accessing the base data.

Even though, binning can reduce the number of bitmaps and improve the query response time in many cases. However, for some queries, we have to go back to the base data in order to answer the queries accurately. For example, if we have 100 equal-width bins for pressure between 0 and 10^9, then the query condition "pressure $> 2.5 \times 10^7$" can be resolved with the index only. We know bins 0 and 1 contain records that satisfy the query condition, and bins 3 and onwards contain records that do not satisfy the condition, but we are not sure which records in bin 2 satisfy the condition. We need to examine the actual values of all records in bin 2 to decide. In this case, we say that bin 2 is the *boundary bin* of the query

[3] A caveat: we actually split all values that round to a low precision number $\bar{x}$ into two bins, one for those round up to $\bar{x}$ and one for those that round down to $\bar{x}$.



(a) individual query (b) average time

Figure 18: Time needed to process range queries.

and call the records in bin 2 *candidates* of the query. The process of examining the raw data to resolve the query accurately is called *candidate checking*. When a candidate check is needed, it often dominates the total query response time. There are number of different approaches to minimize the impact of candidate checks. One approach is to reorder the expression being evaluated to minimize the overall cost of candidate checks [88]. Another approach is to place the bin boundaries to minimize the cost of evaluating a fixed set of queries [48, 77, 76, 75].

Both approaches mentioned above do not actually reduce the cost of a candidate checking operation. More recently, a new approach was proposed to do just that [111]. It does so by providing a clustered copy named Order-preserving Bin-based Clustering (OrBiC) of the base data. Since the values of all records in a bin are organized contiguously, the time needed for a candidate checking operation is minimized. In tests, this approach was shown to significantly outperform the unbinned indexes. Figure 18 shows some performance numbers to illustrate the key advantages of the new approach. In Figure 18(a), we see that the binned index with OrBiC outperforms the one without OrBiC for all query condition tested. In Figure 18(b), we see how the advantage of OrBiC varies with the number of bins. Clearly, we see that the advantage of OrBiC is significantly affected by the number of bins used. The analysis provided by the authors can predict the optimal number of bins for simple types of data [111], but additional work is needed to determine the number of bins for more realistic data.

6.5 Implementations

The first commercial implementation of a bitmap index was in Model 204 from early 1980s [65], and it is still available as a commercial product from Computer Corporation of American. A number of popular DBMS products have since implemented variants of bitmap index. For example, ORACLE has a BBC compressed bitmap index, IBM DB2 has the Encoded Vector Index, IBM Informix products have two versions of bitmap indexes (one for low cardinality data and one for high cardinality data), and Sybase IQ data warehousing products have two versions of bitmap indexes as well. These bitmap index implementations are either based on the basic bitmap index or the bit-sliced index, which are two best choices among all multi-component bitmap indexes [110].

There are a number of research prototypes with numerous bitmap indexes [63, 106]. In particular, FastBit is freely available for anyone to use and extend. We next briefly describe some of the key features of the FastBit software.

FastBit is distributed as C++ source code and can be easily integrated into a data processing system. On its own, it behaves as a minimalistic data warehousing system with column-oriented data organization. Its strongest feature is a comprehensive set of bitmap indexing functions that include innovative techniques in

all three categories discussed above. For compression, FastBit offers WAH as well as the option to uncompress some bitmaps. For encoding, FastBit implements all four theoretically optimal compressed bitmap indexes in addition to a slew of bitmap encodings proposed in the research literature. For binning, it offers the unique low-precision binning as well as a large set of common binning options such as equal-width, equal-weight and log-scale binning. Because of the extensive indexing options available, it is a good tool for conducting research in indexing. In 2007, two PhD theses involving FastBit software were successfully completed, which demonstrate the usefulness of FastBit as a research tool [72, 85]. FastBit has also been successfully used in a drug screening software TrixX-BMI, and was shown to speed up virtual screening by 12 times on average in one case and hundreds of times in another [80]. The chapter on visualization describes another application of using FastBit for network traffic analysis. Later in Section 7.3 we will briefly describe another application of using FastBit in analysis of High-Energy Physics data.

7 Data Organization and Parallelization

In this section, we briefly review a number of data management systems to discuss the different aspects of data organizations and their impact on query performance. Since many of the systems are parallel systems, we also touch on the issue of parallelization. Most of the systems reviewed here don't have extensive indexing support. We also present a small test comparing one of these systems again FastBit to demonstrate that indexing could improve the query performance. Finally, we discuss the Grid Collector as an example of a smart iterator that combines indexing methods with parallel data processing to significantly speed up large scale data analysis.

7.1 Data processing systems

To access data efficiently, the underlying data must be organized in a suitable manner, since the speed of query processing depends on the data organization. In most cases, the data organization of a data processing system is inextricably linked to the system design. Therefore we can not easily separate the data organization issue from the systems that support them. Next, we review a few example systems to see how their data organizations affects the query processing speed. Since most of the preceding discussion applies to the traditional DBMS systems, we will not discuss them any further.

Column-based systems The column-based systems are extensively discussed in Chapter 11. Here, we will only mention some names and give a brief argument on their effectiveness.

There are a number of commercial database systems that organize their data in column-oriented fashion, for example, Sybase IQ, Vertica, and Kx Systems [98]. Among them, Kx Systems can be regarded as an array database because it treats an array as a first-class citizen like an integer number. There are a number of research systems that uses vertical data organization as well, for example, C-Store [91, 90], MonetDB [17, 16], and FastBit. One common feature of all these systems is that they logically organize values of a column together. This offers a number of advantages. For example, a typical query only involves a small number of columns, the column-oriented data organization allows the system to only access the columns involved, which minimizes the I/O time. In addition, since the values in a column are of the same type, it is easier to determine the location of each value and avoid accessing irrelevant rows. The values in a column are more likely to be the same than values from different columns as in row-oriented data organization, which makes it more effective to apply compression on data [1].

Special-purpose data analysis systems Most of the scientific data formats such as FITS, NetCDF and HDF5 come with their own data access and analysis libraries, and can be considered as special-purpose data analysis systems. By far the most developed of such systems is ROOT [18, 70, 19]. ROOT is a data management system developed by physicists originally for High-Energy Physics data. It currently manages many petabytes of data around the world, more than many of the well-known commercial DBMS products.

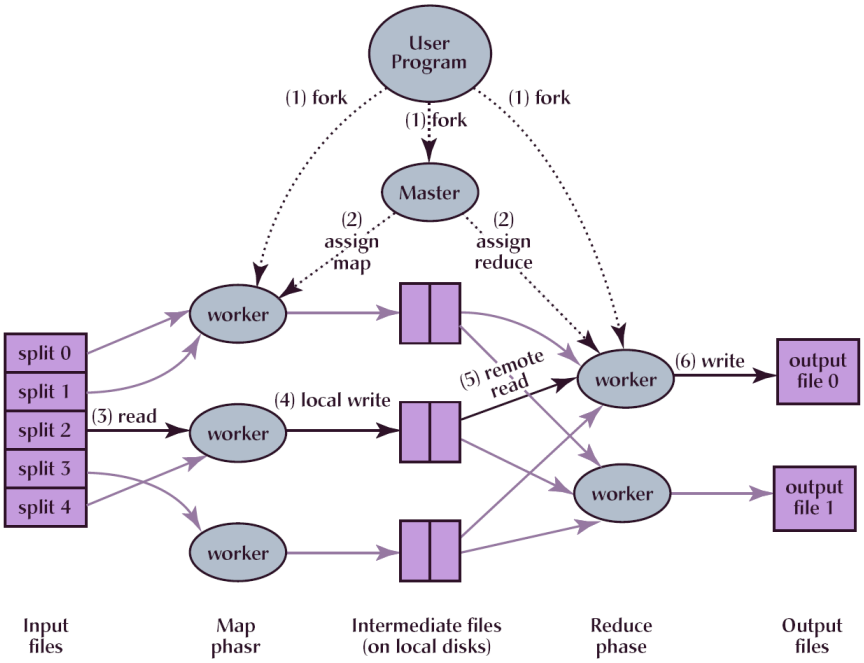


Figure 19: An overview of the MapReduce execution process (adapted from [28]).

ROOT uses an object-oriented metaphor for its data, a unit of data is called an object or an event (of High-Energy collision), which corresponds to a row in a relational table. The records are grouped into files and the primary access method to records in a file is to iterate through them with an iterator. Once an event is available to the user, all of its attributes are available. This is essentially the row-oriented data access. In recent versions of ROOT, it is possible to split some attributes of an event to store them separately. This provides a means to allow for column-oriented data access.

ROOT provides an extensive set of data analysis framework, which makes analyses of High-Energy Physics data convenient and interactive. Its interpreted C++ environment also offers the possibility of infinitely complex analysis that some users desire. Since each ROOT file can be processed independently, the ROOT system also offer huge potential for parallel processing on a cluster of commodity computers. This is a nice feature that enabled the cash strapped physicists to effectively process petabytes of data before anyone else could. The ROOT system is now being extensively used by many scientific applications, and has even gained some fans in the commercial world. More information about ROOT can be found at http://root.cern.ch/.

MapReduce parallel analysis system The MapReduce parallel data analysis model has gained considerable attention recently [27, 28, 50]. Under this model, a user only needs to write a `map` function and a `reduce` function in order to make use of a large cluster of computers. This ease of use is particularly attractive because many other parallel data analysis systems require much more programming effort. This approach has been demonstrated to be effective in a number of commercial settings.

There are a number of different implementations of MapReduce system following the same design principle. In particular, there is a open-source implementation from the Apache Hadoop project which is available for anyone to use. To use this system, one needs to place the data on a parallel file system supported by the MapReduce run-time system. The run-time system manages the distribution of the work onto different processors, selecting the appropriate data files for each processor, and passing the data records from the file to the `map` and `reduce` functions. The run-time system also manages the coordination among the parallel tasks, collects the final results, and recovers any errors. An illustration of these steps is shown in Figure 19 [28].

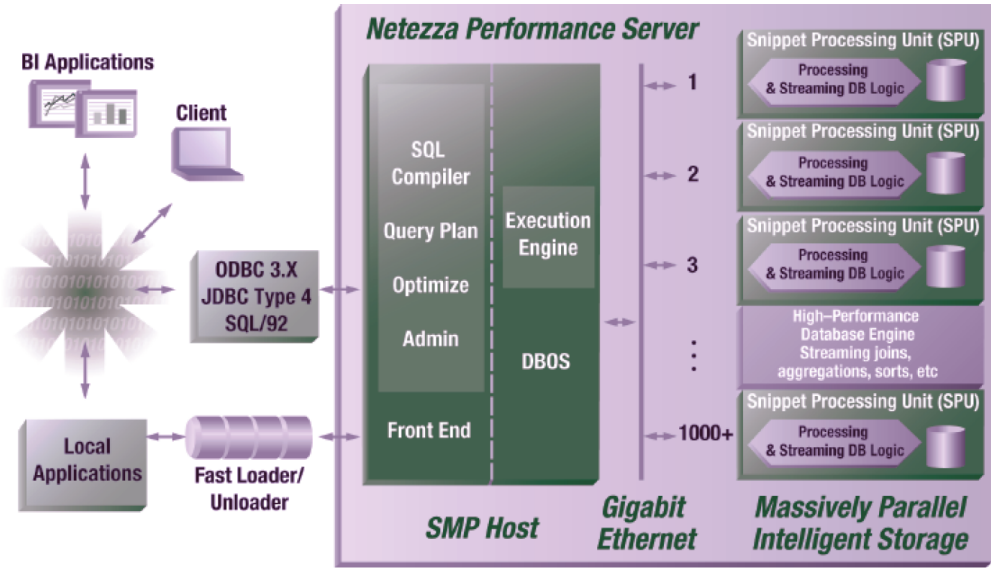


Figure 20: An overview of Netezza architecture (adapted from [26]).

The MapReduce system treats all data records as key/value pairs. The primary mechanism offer in this model is an iterator (identified by a key). Recall that the ROOT system also provides a similar iterator for data access. Another similarity is that both ROOT and MapReduce can operate on large distributed data. The key difference between ROOT and MapReduce is that the existing MapReduce systems rely on underlying parallel file systems for managing and distributing the data, while the ROOT system uses a set of daemons to deliver the files to the parallel jobs. In a MapReduce system, the content of the data is opaque to the run-time system and the user has to explicitly extract the necessary information for processing. In the ROOT system, an event has a known definition and accessing to attributes of an event therefore requires less work.

The data access mechanism provided by a MapReduce system can be considered as row-oriented because all values associated with a key are read into memory when the iterator points to the key/value pair. On structured data, for a typical query that requires only a small number of attributes, a MapReduce system is unlikely to deliver poorer performance than a parallel column-oriented system such as MonetDB, C-Store, or Vertica. The MapReduce systems is proven effective for unstructured data.

Custom data processing hardware The speed of accessing secondary storage in the past few decades practically remain unchanged compared with the increases in the speed of main memory and CPU. For this reason, the primary bottleneck for efficient data processing is often the disk. There has been a number of commercial efforts to build data processing systems using custom hardware to more efficiently answer queries. Here we very briefly discuss two such systems: Netezza [26] and Teradata [29, 6].

Netezza attempts to improve query processing speed by have smart disk controllers that can filter data records as they are read off the physical media. An high-level illustration of the system is shown in Figure 20 [26].

In a Netezza server, there is a front-end system that accepts the usual SQL commands, so the user can continue to use the existing SQL code developed for other DBMS systems. Inside the server, a SQL query is processed on a number of different Snippet Processing Units (SPUs), where each SPU has its own disk and processing logic. The results from different SPUs are gathered by the front-end host and presented to the user. In general, the idea of off-loading some data processing to the disk controllers to make an *active storage* system could benefit many different applications [73, 56].

The most unique feature of Teradata warehousing system is the BYNET interconnect that connects the main data access modules called AMPs (Access Module Processors). The designed of BYNET allows bandwidth among the AMPs to scale linearly with the number of AMPs (up to 4096 processors). It also is

fault tolerant and performs automatic load balancing. The early versions of AMPs are similar to Netezza's "smart disk controllers," however, the current version of AMPs are software entities that utilizes commodity disk systems.

To the user, both Netezza and Teradata behave as a typical DBMS system, which is a convenience feature for the user. On disks, both systems appear to be followed the traditional DBMS systems, i.e., storing their data in the row-oriented organization. Potentially, using the column-oriented organization may improve their performances. Teradata has hash and B-Tree indexes, while Netezza does not use any index method.

7.2 Indexes still useful

Many of the specialized data management systems mentioned above do not employ any index method. When the analysis task calls for all or a large fraction, say one-tenth, of records in a dataset, then having an index may not accelerate the overall data processing. However, there are plenty of cases where indexes can dramatically speed up the query processing. For example, for interactive exploration, one might selected a few million records from a dataset with billions of records. Going through the whole dataset to find a few million is likely to take longer than using an effective index. Another example is in query estimation. Often, before users commit to evaluate subsets of data records, they may want to know the size of the result and the possible processing time. This task is usually better accomplished with indexes. Additionally, indexes may provide a faster way to gather some statistics. To illustrate the usefulness of an index in accomplishing such tasks, we next give a specific example of answering count queries from the Set Query Benchmark.

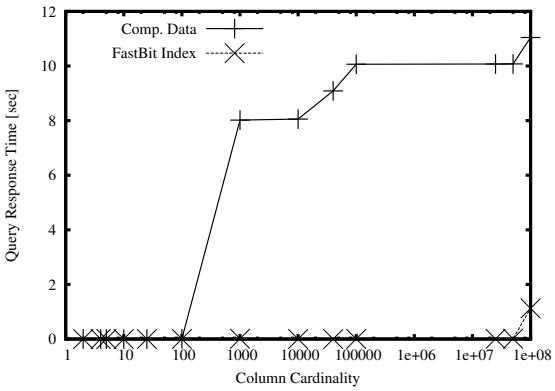
The Set Query Benchmark is a benchmark designed to evaluate the performance of OLAP type applications [67, 63]. The test data contains 12 columns of uniform random numbers plus a sequence number that serves as row identifiers. Our test uses queries of the form "Select count(*) From Bench Where ...," which we call count queries [63]. We use a test dataset with 100 million rows instead of the original specification of 1 million rows. In the test dataset, we also adjust the column cardinality of the two random columns to be 25 million and 50 million instead of 250,000 and 500,000 as the benchmark specification indicated. We note that such uniform random values with extremely high column cardinalities are the worst type of data for the compressed bitmap indexes. The table in Figure 21 contains the names of these count queries, Q1, Q2A, Q2B, and so on. Each of these queries have a number of different instances that involves different columns or different query conditions. Figure 21(a) shows the total time to complete all instances of a query and Figure 21(b) shows the query response time for each instance of Q1.

The query response times are gathered from two systems; one with a compressed bitmap index and the other with compressed columns but without any index. The bitmap index is from FastBit and the indexless system is a commercial product that reorders and compresses the base data. This indexless system is advertised as the fastest data processing system. During our testing, we consulted with the vendor to obtain the best ordering and compression options available at the time. The two systems are run on the same computer with a 2.8GHz Intel Pentium CPU and a small hardware RAID that is capable of supporting about 60MB/s throughput.

The time results in Figure 21 clearly indicate that indexes are useful for these set of queries. Overall, one can answer all the queries about 11 times faster using bitmap indexes than using the compressed base data. Figure 21(b) shows some performance details that helps to explain the observed differences. The horizontal axis in Figure 21(b) is the column cardinality. The best reordering strategy that minimizes the overall query response time is to order the lowest cardinality column first, then the next lowest cardinality column, and so on. More specifically, the column with cardinality 2 (the lowest cardinality) is complete sorted; when the lowest cardinality column values are the same, the rows are ordered according to the column with cardinality 4; when the first two columns have the same value, the rows are ordered according to the next lowest cardinality column. This process continues on all columns including the column of sequence numbers. After this reordering, the first few columns are very nearly sorted and can be compressed very well. Their compressed sizes are much smaller than the original data and Q1 queries (Select count(*) From Bench Where :col = 2, where :col is the name of a column in the test dataset) can be answered very

	FastBit index	Comp. Data
Q1	1.157	66.436
Q2A	0.540	28.478
Q2B	0.541	28.367
Q3A0	1.004	76.431
Q3B0	4.995	128.688
Q4A0	9.799	5.235
Q4B0	12.784	0.015
total	30.819	333.65

(a) Total query response time



(b) Time to answer Q1 queries

Figure 21: Time needed to answer count queries from the Set Query Benchmark with 100 million records.

quickly. In these cases, the time required by two test systems are about the same. However, when higher cardinality columns are involved in the queries, the indexless system requires much more time. Overall, the total time required to answer all 13 instances of Q1 is about 1.2 seconds with the FastBit indexes, but about 66 seconds with the compressed base data. The indexless system works well on queries Q4A0 and Q4B0 because these queries only involve low cardinality columns where the compressed columns are very small.

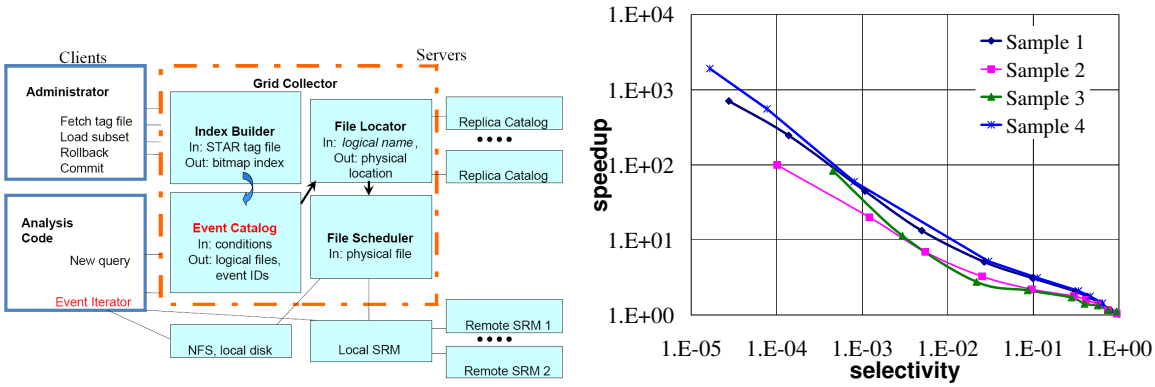
Note that the bitmap indexes were built on the data that were reorder in the same way that the indexless system did. Had we not reorder the data and built the bitmap indexes on the data in the original order, the differences would be smaller because the reordering also helps reduce the sizes of bitmap indexes. Nevertheless, even without reordering the data, the overall performance of compressed bitmap index is about 3 times better than the indexless system. In short, the indexes are very useful in some applications; indexless approaches are unlikely to completely replace systems with indexes.

7.3 Using Indexes to Make Smart Iterators

The previous set of tests demonstrate that one can use indexes to count the number of results of queries. Next, we present another example where indexes can be used to make "smart iterators" to speed up data analysis. More specifically, we present a case where FastBit is used to implement an efficient Event Iterator for a distributed data analysis of a large collection of High-Energy Physics data from STAR.

The Event Iterator was implemented for a plug-in to the STAR analysis framework called Grid Collector [101, 104]. STAR is a high-energy physics experiment that collects billions of events on collisions of high-energy nuclei. The records about collisions are organized into files with a few hundred events each. Most analysis tasks in STAR go through a relatively small subset of the events (from a few thousand of events to a few million events out of billions) to gather statistics about various attributes of the collisions. The basic method for analyzing the volumes of data collected by the STAR experiment is to specify a set of files and then iterate through each collision events in the files. Typically, a user program filters the events based on a set of high-level summary attributes called tags and compute statistics on a subset of desired events. The Grid Collector allows the user to specify the selection criteria as conditions on the tags and directly deliver the selected events to the analysis programs. Effectively, the Grid Collector replaces the existing simple iterator with a smart iterator that understands the conditions on the tags and extracts events satisfying the specified conditions for analysis. This remove the need for users to manage the files explicitly, reduce the amount of data read from the disks and speed up the overall analysis process as shown in Figure 22.

In Figure 22(a), we show a block diagram of the key components of the Grid Collector. The main Grid Collector component can be viewed as an Event Catalog that contains all tags of every collision events collected by the STAR experiment. This component runs as a server and is also responsible for extracting the



(a) Schematics of Grid Collector

(b) Speedup using Grid Collector

Figure 22: Using Grid Collector can significantly speed up analysis of STAR data [101, 104].

tag values, building bitmap indexes, resolving conditions on tags to identify relevant data files, locating these files, and coordinating with various storage resource managers [84] (see Chapter 3 for details) to retrieve the files when necessary. All of these tasks together delivers the selected events to the analysis code.

In Figure 22(b), we show the observed speedup values versus the selectivity of the analysis task. In this case, the selectivity is defined as the fraction of events in the data files that are selected for an analysis task. The Grid Collector speeds up analysis tasks primarily by reducing the amount of disk pages accessed. Because the selected events typically are randomly scattered in the data files, the data files are compressed in blocks [18], and the analysis jobs often involve significant amount of computation, therefore, the speed up is not the inverse of selectivity. However, as the selectivity decreases, the average speedup value increases. When one out of 1000 events is selected, the speedup values are observed to be between 20 and 50. Even if one in ten events are used in an analysis job, the observed speedup is more than 2. STAR has hundreds of users at their various analysis facilities, improving these facilities overall throughput by a factor of 2 is a great benefit to the whole community.

As mentioned before, other parallel data management systems such Hadoop currently iterate through all data records as well. A smart iterator similar to that of Grid Collector could benefit such a system as well.

8 Summary and Future Trends

In this chapter, we discussed two basic issues for accelerating queries on large scientific datasets, namely indexing and data organization. Since the data organizations are typically tied to an individual data management system, we also briefly touched on a number of different systems with distinct data organization schemes.

The bulk of this chapter discusses different types of index methods; most of which are better suited for secondary storage. Applications that use scientific data don't require simultaneous read and write accesses of the same dataset. This allows the data and indexes to be packed more tightly than in transactional applications. Furthermore, the indexes can be designed to focus more on query processing speed and with less on updating of individual records. In general, scientific data tend to have a large number of searchable attributes, and require indexes on every searchable attribute, whereas a database for a banking application may require only one index for the primary key.

After reviewing many of the well-known multi-dimensional indexes, we concluded that the bitmaps are the most appropriate indexing schemes for scientific data. We reviewed some recent advances in bitmap index research and discussed their uses in two examples. These examples use an open-source bitmap index

software called FastBit. The first example demonstrated the usefulness of indexes by measuring the time needed to answer a set of queries from the Set Query Benchmark. We saw that FastBit outperforms the best available indexless system by an order of magnitude. This demonstrates that there are situations where the use an index significantly improves performance of an application. The second example demonstrated the use FastBit indexes to implement a smart iterator for a distributed data analysis framework. Since an iterator is a convenient way to access a large datasets on parallel systems, the example demonstrated an effective way of using indexes for parallel data analysis.

As datasets grow in size, all data analyses are likely to be performed on parallel computers. Off-loading some data processing tasks to the disk controller (as the Netezza system does) or other custom hardware could be an effective strategy to improve the efficiency of query processing. However, advanced indexing techniques will continue to be an indispensable tool for analyzing massive datasets.

References

[1] D. Abadi, S. R. Madden, and M. C. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. ACM, 2006.

[2] A. V. Aho and J. D. Ullman. Optimal partial-match retrieval when fields are independently specified. *ACM Trans. Database Syst.*, 4(2):168–179, 1979.

[3] S. Amer-Yahia and T. Johnson. Optimizing Queries on Compressed Bitmaps. In *International Conference on Very Large Data Bases*, Cairo, Egypt, September 2000. Morgan Kaufmann.

[4] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference (DCC)*, March 28 1995.

[5] G. Antoshenkov and M. Ziauddin. Query Processing and Optimization in ORACLE RDB. *VLDB Journal*, 5:229–237, 1996.

[6] C. Ballinger and R. Fryer. Born to be parallel: Why parallel origins give teradata an enduring performance edge. *IEEE Data Eng. Bull.*, 20(2):3–12, 1997. An updatet version of this paper is available at http://www.teradata.com/library/pdf/eb3053.pdf.

[7] P. Barrett. Application of linear quadtree to astronomical database. *Astronomical Data Analysis Software and Systems IV*, 77:1–4, 1995.

[8] J. Bartholdi and P. Goldsman. Continuous indexing of hierarchical subdivisions of the globe. *Int. J. Geographical Information Science*, 15(6):489–522, 2001.

[9] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[10] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[11] J. L. Bentley. Mutidimensional binary search tree in database applications. *IEEE Trans. Soft. Eng.*, SE-5(4):333–337, 1979.

[12] S. Berchtold, C. Bohn, and H-P. Kriegel. The pyramid technique: Towards breaking the curse of dimensionality. In *SIGMOD*, pages 142–153, 1998.

[13] S. Berchtold, D. Keim, and H-P. Kriegel. The X-tree: An index structure for high -dimensional data. In *VLDB*, pages 28–39, 1996.

[14] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 1999.

[15] C. Böhm, S. Berchtold, and Hans-Peter Kriegel. Multidimensional index structures in relational databases. *Journal of Intelligent Info Syst.*, 15(1):322–331, 2000.

[16] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *The VLDB Journal*, pages 54–65, 1999.

[17] P. A. Boncz, W. Quak, and M. L. Kersten. Monet and its geographic extensions: A novel approach to high performance gis processing. In *EDBT'96*, volume 1057 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[18] R. Brun and F. Rademakers. ROOT : An object oriented data analysis framework. *Nuclear instruments & methods in physics research, Section A*, 289(1-2):81–86, 1997.

[19] J. J. Bunn and H. B Newman. Data intensive grids for high energy physics, 2003. Available at `http://pcbunn.cithep.caltech.edu/Grids/GridBook.htm`.

[20] A. Chadha, A. Gupta, P. Goel, V. Harinarayan, and B. R. Iyer. Encoded-vector indices for decision support and warehousing, 1998. US Patent 5,706,495.

[21] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2):355–366, 1998.

[22] C. Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD*, Philadelphia, PA, USA, June 1999. ACM Press.

[23] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS'98*, 1998.

[24] D. Comer. The ubiquitous B-tree. *ACM Comput. Surveys*, 11(2):121–137, 1979.

[25] C. Cormen, T. Stein, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 2nd edition, 2001.

[26] G. S. Davidson, K. W. Boyack, R. A. Zacharski, S. C. Helmreich, and J. R. Cowie. Data-centric computing with the netezza architecture. Technical Report SAND2006-3640, Sandia National Laboratories, 2006.

[27] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, 2004.

[28] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[29] D. J. DeWitt, M. Smith, and H. Boral. A single-user performance evaluation of the teradata database machine. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 244–276, London, UK, 1989. Springer-Verlag.

[30] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. on Database Syst.*, 4(3):315–344, 1979.

[31] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.

[32] M. Freeston. The bang file: a new kind of grid file. In *SIGMOD*, pages 260–269, 1987.

[33] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[34] I. Gargantini. An effective way to represent quad-trees. *Comm. ACM*, 25(12):905–910, 1982.

[35] J. Gray, D. T. Liu, N. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41, 2005.

[36] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41, 2005.

[37] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[38] A. Gupta, K. C. Davis, and J. Grommon-Litton. Performance comparison of property map and bitmap indexing. In *DOLAP'02*, pages 65–71. ACM, 2002.

[39] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, Mass., 1997.

[40] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM, 1984.

[41] R. J. Hanisch, A. Farris, E. W. Greisen, W. D. Pence, B. M. Schlesinger, P. J. Teuben, R. W. Thompson, and A. Warmork III. Definition of the flexible image transport system (FITS). *Astronomy and Astrophysics. Supp. Ser.*, 376:359–380, 2001.

[42] Hdf5 home page.

[43] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[44] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[45] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *VLDB*, Edinburgh, Scotland, September 1999. Morgan Kaufmann.

[46] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *SIGMOD'97*, pages 369–380. ACM, 1997.

[47] D. Knuth. *The art of computer programming: Sorting and searching*, volume 3. Addison-Wesley, Reading, Mass., 2e edition, 1973.

[48] N. Koudas. Space efficient bitmap indexing. In *CIKM'00*, pages 194–201. ACM, 2000.

[49] Z. Lacroix and T. Critchlow, editors. *Bioinformatics: Managing Scientific Data*. Morgan Kaufmann, 2003.

[50] R. Lämmel. Google's mapreduce programming model - revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

[51] P. Larson. Dynamic hashing. *BIT*, 18:184–201, 1978.

[52] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. In *BNCOD*, volume 1832 of *Lecture Notes in Computer Science*. Springer, 2000.

[53] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics*, 19(2):79–121, 2000.

[54] W. Litwin. Linear hashing: a new tool for table and file addressing. In *VLDB'80*, pages 212–223, 1980.

[55] D. Lovelace, R. Ayyar, A. Sala, and V. Sokal. VSAM demystified. Technical Report Redbook Series SG246105, IBM, 2001.

[56] X. Ma and A. L. N. Reddy. MVSS: An active storage architecture. *IEEE Transactions on Parallel and Distributed Systems*, PDS-14(10):993–1005, October 2003.

[57] B. Moon, H. V. Jagadish, and C. Faloutsos. Analysis of the clustering properties of the hilbert space-filing curve. *IEEE Trans. on Knowledge and Data Eng.*, 13(1), 2001.

[58] D. R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[59] Ron Musick and Terence Critchlow. Practical lessons in supporting large-scale computational science. *SIGMOD Rec.*, 28(4):49–57, 1999.

[60] Mark Nelson and Jean loup Gailly. *The Data Compression Book*. M&T Books, New York, NY, 2nd edition, 1995.

[61] NetCDF (network common data form) home page.

[62] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable symmetric multikey file structure. *ACM Trans. on Database Syst.*, 9(1):38–71, 1984.

[63] Elizabeth O'Neil, Patrick O'Neil, and Kesheng Wu. Bitmap index design choices and their performance implications. In *IDEAS 2007*, pages 72–84, 2007.

[64] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, Tucson, AZ, USA, May 1997. ACM Press.

[65] P. O'Neil. Model 204 architecture and performance. In *Second International Workshop in High Performance Transaction Systems*. Springer Verlag, 1987.

[66] P. O'Neil. Informix indexing support for data warehouses. *Database Programming and Design*, 10(2):38–43, February 1997.

[67] P. O'Neil and E. O'Neil. *Database: pronciples, programming, and performance*. Morgan Kaugmann, 2nd edition, 2000.

[68] E. J. Otoo. Linearizing the directory growth of extendible hashing. In *ICDE*. IEEE, 1988.

[69] E. J. Otoo. A mapping function for the directory of a multidimensional extendible hashing. In *VLDB*, pages 493–506, 1984.

[70] F. Rademaker and R. Brun. ROOT: An object-oriented data analysis framework. *Linux Journal*, July 1998. ROOT software is availe from http://root.cern.ch/.

[71] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a database system kernel. In *VLDB'2000*, 2000.

[72] F. R. Reiss. *Data Triage*. PhD thesis, UC Berkeley, Jun 2007.

[73] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *VLDB'98*, pages 62–73, New York, NY, August 1998. Morgan Kaufmann Publishers Inc.

[74] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases, With Applications to GIS*. Morgan Kaufmann, San Francisco, 2002.

[75] D. Rotem, K. Stockinger, and K. Wu. Minimizing I/O costs of multi-dimensional queries with bitmap indices. In *SSDBM, Vienna, Austria, July 2006*. IEEE Computer Society Press, 2005.

[76] D. Rotem, K. Stockinger, and K. Wu. Optimizing candidate check costs for bitmap indices. In *CIKM, Bremen, Germany, November 2005*. ACM Press, 2005.

[77] D. Rotem, K. Stockinger, and K. Wu. Optimizing I/O costs of multi-dimensional queries using bitmap indices. In *DEXA, Copenhagen, Denmark, August 2005*. Springer Verlag, 2005.

[78] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.

[79] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, San Francisco, CA, 2006.

[80] J. Schlosser and M. Rarey. TrixX-BMI: Fast virtual screening using compressed bitmap index technology for efficient prefiltering of compound libraries. ACS Fall 2007 Boston MA, 2007.

[81] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[82] U. Shaft and R. Ramakrishnan. Theory of nearest neighbors indexability. *ACM Trans. Database Syst.*, 31(3):814–838, 2006.

[83] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. Multidimensional indexing and query coordination for tertiary storage management. In *SSDBM*, pages 214–225, 1999.

[84] A. Sim and A. Shoshani. The storage resource manager interface specification, 2008. http://www.ogf.org/documents/GFD.129.pdf (also in: http://sdm.lbl.gov/srm-wg/doc/GFD.129-OGF-GSM-SRM-v2.2-080523.pdf)

[85] R. R. Sinha. *Indexing Scientific Data*. PhD thesis, UIUC, 2007.

[86] R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32(3):16, 2007.

[87] Star: Solenoidal tracker at rhic (star) experiment.

[88] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *DEXA*, Zaragoza, Spain, September 2004. Springer-Verlag.

[89] K. Stockinger and K. Wu. *Bitmap Indices for Data Warehouses*, chapter VII, pages 179–202. Idea Group, Inc., 2006.

[90] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One size fits all? part 2: Benchmarking studies. In *CIDR*, pages 173–184. www.crdrdb.org, 2007.

[91] S. Stonebraker, D. J. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *VLDB'05*, pages 553–564. VLDB Endowment, 2005.

[92] A. Szalay, P. Kunszt, A. Thakar, J. Gray, and D. Slutz. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In *SIGMOD*, Dallas, Texas, USA, May 2000. ACM Press.

[93] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner. Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. *SIGMOD Rec.*, 29(2):451–462, 2000.

[94] Yannis Theodoridis. The r-tree-portal, 2003.

[95] L. A. Treinish. Scientific data models for large-scale applications, 1995.

[96] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[97] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98*, pages 194–205. Morgan Kaufmann, 1998.

[98] Arthur Whitney. Abridged kdb+ database manual. http://kx.com/q/d/a/kdb+.htm, 2007.

[99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing, and Indexing documents and Images*. Van Nostrand Reinhold, International Thomson Publ. Co., New York, 1994.

[100] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proceedings of VLDB 85, Stockholm*, pages 448–457, 1985.

[101] K. Wu, J. Gu, J. Lauret, A. M. Poskanzer, A. Shoshani, A. Sim, and W.-M. Zhang. Grid collector: Facilitating efficient selective access from datagrids. In *International Supercomputer Conference, Heidelberg, Germany, June 21-24, 2005*, May 2005.

[102] K. Wu, E. J. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB'2004*, 2004.

[103] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.

[104] K. Wu, W.-M. Zhang, V. Perevoztchikov, J. Lauret, and A. Shoshani. The grid collector: Using an event catalog to speedup user analysis in distributed environment. In *Computing in High Energy and Nuclear Physics (CHEP) 2004*, Interlaken, Switzerland, September 2004.

[105] K.-L. Wu and P.S. Yu. Range-Based Bitmap Indexing for High-Cardinality Attributes with Skew. Technical report, IBM Watson Research Center, May 1996.

[106] Kesheng Wu. FastBit reference guide. Technical Report LBNL/PUB-3192, Lawrence Berkeley National Laboratory, Berkeley, CA, 2007.

[107] K. Wu, E. Otoo, and A. Shoshani. A performance comparison of bitmap indices. In *CIKM*. ACM Press, November 2001.

[108] K. Wu, E. J. Otoo, and A. Shoshani. Compressed bitmap indices for efficient query processing. Technical Report LBNL-47807, LBL, Berkeley, CA, 2001.

[109] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, pages 99–108, 2002.

[110] K. Wu, K. Stockinger, and A. Shoshani. Performance of multi-level and multi-component compressed bitmap indexes. Technical Report LBNL-60891, Lawrence Berkeley National Laboratory, Berkeley, CA, 2007.

[111] K. Wu, K. Stockinger, and A. Shoshani. Breaking curse of cardinality on bitmap indexes. In *SSDBM'08*, pages 348–365, 2008.

[112] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *International Conference on Data Engineering*, Orlando, Florida, USA, February 1998. IEEE Computer Society Press.