A Hilbert Space Compression Framework for Parallel Relational OLAP

David Cueva

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April, 2007

Your file Votre référence
ISBN: 978-0-494-28950-1
Our file Notre référence
ISBN: 978-0-494-28950-1

ABSTRACT

A Hilbert Space Compression Framework for Parallel Relational OLAP

David Cueva

The Data Cube is the central abstraction behind the power of On-Line Analytical Processing (OLAP) systems. It enables knowledge workers to analyze vast amounts of enterprise data, in order to make timely and informed decisions. Nevertheless, this potential is accompanied by massive amounts of storage requirements. This fact challenges even the most powerful parallel systems to efficiently pack and manage the data, allowing the delivery of fast responses to user queries.

An ample number of methods concentrate on generic data compression, but relatively few have been proposed which fulfill the requirements of distributed OLAP environments. This thesis investigates these opportunities by presenting a compression framework specially tailored for parallel OLAP. New algorithms for data and index compression are proposed. The data encoding method is based on the Hilbert space-filling cuve. It eliminates the inherent redundancies in OLAP data and preserves compression granularity at the block level. The index encoding technique is based on the concept of packed R-trees, and when combined with our native query engine provides fast and I/O efficient access to the specific blocks satisfying a user request. Additionally, unlimited scalability is achieved by a set of supporting techniques permitting the framework to handle arbitrarily large data sets.

We have performed a broad evaluation of our framework, including comparison benchmarks with several influential published methods. Our compression algorithms deliver state-of-the-art ratios with averages above 80% for data and 95% for indexes. Finally, the number of blocks accessed by the query engine is typically reduced by a factor of 10.

ACKNOWLEDGEMENTS

Thanks to Magaly and Sonia for their understanding and support.

Thanks to Dr. Todd Eavis for his guidance.

Thanks to Dr. Sabine Bergler, Ing. Francisco Villavicencio, Ing. Gustavo Samaniego and Ing. Carlos Montenegro for their helping hand.

And a big thanks to Concordia for making this research possible.

To my family

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

We live in an era of rapid evolution for corporations around the world. Decisions affecting the future of companies must be taken every day to keep pace with the exigencies of the market. Decision Support Systems (DSS) are meant to provide the tools to access and analyze the appropriate information in order to make choices that are effective and timely. However, accomplishing this task is not easy. To provide users with meaningful information, these systems must deal with the transmission, storage and processing of massive amounts of data collected both from across the enterprise and from external sources.

This thesis introduces a compression framework, applied in the context of a distributed On-Line Analytical Processing (OLAP) environment. Because of the unique characteristics of data and indexes in this domain, special strategies are required to design an effective and integrated compression architecture. Consequently, the starting point is to analyze the particular aspects of OLAP data. The most important of these features is *multi-dimensionality.* This concept presents tuples as a set of points scattered in a d-dimensional space, where the dimensions can be collapsed by means of the aggregation of attributes.

Gray et al. [1996] presented the data cube operator, an extension of the Structured Query Language (SQL) to facilitate the generation of the multi-dimensional structures. Nevertheless, the size of the generated data cubes can be exceptionally large, and additionally, the size of the corresponding indexes can reach up to 10% of the cube size. Fortunately, redundancies and clustering across the space can be exploited by specially tailored compression algorithms.

Our framework takes advantage of these characteristics by converting multidimensional tuples into ordinals on a Hilbert space-filling curve, recording only the distances between two ordinals along the curve. These differences are then packed into disk blocks, forming the basis of a compressed R-tree index structure. Finally, a special-purpose query engine provides the user with an efficient and transparent mechanism to obtain the desired data. The framework itself constitutes a module of the Sidera OLAP system, which is a pioneer in the investigation of distributed Relational OLAP (ROLAP) technologies.

1.1 Overview of the Research

A brief description of the key elements of the thesis is presented in the remainder of this section. We note that, despite their independent descriptions, each component is actually integrated to form a single compression architecture, providing the user with the means to query compressed data using effective indexing techniques.

1.1.1 Compressing the Data Cube

We design, implement and test an algorithm for compressing summarized data cube views. The generation of the data cube is performed using the Sidera system developed by Eavis [2003]. The approach is based on a ROLAP paradigm, where each view is represented by a relational table of d dimensions. To efficiently generate the data cube, Sidera works in a distributed environment, where each processor computes one or more views according to the data cube *lattice* descibed in Harinarayan et al. [1996]. The lattice is partitioned among the nodes to balance the processing load. The multidimensional tuples are then encoded into ordinals of a Hilbert space-filling curve [Hilbert, 1891] and transmitted across the nodes where the compression and index generation takes place.

The compression algorithm starts by grouping the received Hilbert ordinals into blocks of data. Each compressed block is made to correspond to a fully packed disk page. Additionally, every block includes all the necessary meta-data to be decompressed without accessing any other blocks. These two very important features allow queries to examine and extract only small parts of the file at a time, therefore minimizing Input/Output (I/O) and processing time.

Each block is composed of a meta-data header followed by the ordinals in Hilbert sequence. Only the differences between an ordinal and the preceeding one are stored. These differences are packed into variable-size arrays of bits which are written irrespectful of byte boundaries. From a geometrical perspective, the method is in fact storing only the minimum distance between two positions along the Hilbert curve. The redundant information, shared by the two points, is given by the distance from the first ordinal to the origin, which is being eliminated. The method is based on Tuple Differential Coding (TDC) introduced by Ng and Ravishankar [1997].

Experimental evaluation demonstrates that our method achieves high compression rates of up to 85% when tested on views configured with 1 to 15 dimensions, cardinalities from 10 to 1 million and a row count as high as 100 million. Also, a set of standard tests proposed by the authors of TDC was performed. In sum, the results show that our method is comparable or surpasses the performance of the best techniques currently in the domain, namely TDC and Dwarf [Sismanis et al., 2002].

1.1.2 Building Compressed Indexes

We design, implement and test an algorithm for building compressed multi-dimensional indexes on top of compressed views. The algorithm follows the packed R-tree method introduced by Roussopoulos and Leifker [1985]. This method is called *sort-pack* because it starts by sorting the data points in a desirable order, which in our case is the already applied Hilbert order, followed by a bulk load of the R-tree, that fills each node to capacity. Recall that the leaf level of the R-tree is already packed by the data-compression algorithm into data blocks.

Each index block is composed of a meta-data header followed by a sequence of compressed hyper-rectangles. A hyper-rectangle is a space boundary defined by two d-dimensional vertexes, which establish the low and high values of the boundary. The R-tree structure starts bottom-up with the compressed data points, then a level of hyper-rectangles enclosing these points, and continues up to the root with boundaries enclosing lower-level hyper-rectangles. This structure is compressed by storing the differences between the vertexes of the contained hyper-rectangles and a *pivot vertex*. The pivot is defined as the low vertex of the container-parent hyper-rectangle.

Experimental evaluation shows that our method for index compression delivers an outstanding rate of up to 99.39%, with an average of 97.24%. Queries are resolved by following the path described by the R-tree hyper-rectangles top-down from the root to the leaf nodes and decompressing only the required blocks to find the results. Few index compression algorithms for OLAP environments have been presented in the literature. Perhaps the most significant result is the method described by Goldstein et al. [1998], which achieves lower compression rates and requires a user-defined initial estimation of the number of hyper-rectangles.

1.1.3 Querying compressed data

We implement and test an extension of the Sidera query engine to support compressed indexes and views, resolving point and range queries. The former type of query matches against a single record, while the latter matches against a range of values in each of the d dimensions. When a query is passed to the engine, the root node of the compressed R-tree index is read and decompressed. The children hyper-rectangles are matched against the query. The ones that fall inside the query are pushed into a queue for further processing. When a leaf node is reached, the corresponding disk block is read. Then the Hilbert ordinals are calculated by using the stored differentials, and subsequently transformed back to multidimensional tuples by applying a conversion function. Finally, those tuples that match the query are returned to the user.

The tests performed on the query engine show a dramatic 90% reduction on the number of blocks retrieved from disk, when compared to the non-compressed case.

1.1.4 Optimizations

The data and index compression algorithms significantly reduce space requirements, making efficient use of disk I/O operations and CPU processing by decompressing the required disk blocks only. However, several other practical issues had to be addressed in order to handle arbitrarily large data sets and to keep CPU intensive operations to a minimum.

- *Buffering:* While modern data warehouses climb to Terabyte (TB) sizes due to cheap data storage [Auerbach, 2006], main server memory remains a more restricted resource. Therefore, OLAP products must support buffering schemas to effectively use the available system memory. Our implementation of the compression algorithms incorporates buffering in all stages of the process, including the Hilbert ordinal generation, external memory sorting, distribution to the parallel nodes and R-tree creation.

- *Secondary memory sorting:* The Sidera parallel load balancing subsystem and our compression module rely on an initial sort of the data in Hilbert order. Because of the massive amount of the aforementioned data, an external sort algorithm is required. For this task, we used a well known *p-way sort*, which is a combination of Quicksort [Hoare, 1961] and Mergesort [Knuth, 1998]. Thanks to buffering and p-way sorting, the upper size limit of the input file is only bound by the intrinsic file system.

- *Hilbert ordinal generation:* The conversion from multidimensional records to Hilbert ordinal and vice versa are expensive operations in terms of CPU processing. Consequently an optimized version of the Hilbert conversion functions following Lawder [2000] has been implemented. In addition, the indexes are precomputed so no conversions are performed during the external sorting stage.

1.2 Thesis Organization

The organization of the thesis is as follows:

- Chapter 2 presents background information on DSS and OLAP. We also introduce the Data Cube, the basic multidimensional model, along with the indexing techniques specially designed to efficiently access its data. The chapter also includes sections on space-filling curves and compression algorithms, central components of our research.

- Chapter 3 discusses our new compression framework, which constitutes the main contribution of the thesis. The components of the framework are the data and index compression algorithms, and the compression-ready query engine. Additionally, we present the buffering sub-system, the fast hilbert external memory sort, and the incorporation of a multi-precision arithmetic library. These techniques are used to provide unlimited scalability, in order to handle high dimensions and arbitrarily large data sets. Finally we describe how the framework works in a distributed environment.

- Chapter 4 presents the environment where our framework is evaluated, and analyzes experimental results on various data sets. It also performs comparisons of the framework with other published relevant methods.

- Chapter 5 offers the conclusions and possible future work.

Chapter 2

Background

2.1 Introduction

The world has experienced an important transformation in the last few decades. McLuhan [1962] referred to our planet as a "global village", where distances are no longer a limitation for communications. This reflects our expectations for faster and easier access to an ever-growing set of data, regardless of its geographical location. Every person in contact with the overwhelming amount of information available on the Internet has experimented this phenomenon. Many organizations around the globe are facing a similar challenge with their data. In this domain, data retrieval and management becomes critical when important decisions are derived from the extracted information.

The present chapter introduces the concepts behind the management of vast amounts of decision-enabling data. We start with an overview of the field in section 2.2, including a glance to Decision Support Systems, their evolution and taxonomy. Then we deepen into one of the branches of the area, presenting in section 2.3 the origins and architechture of OLAP. Immediately, in section 2.4, perhaps the single most important abstraction for the current research is introduced: the Multidimensional Data Cube. The cube is then explored by showing its perspectives and components, along with a model to represent data in this environment, departing from the traditional Entity Relationship approach.

Once the basis for multi-dimensional analysis has been defined, more research-specific subjects are discussed. The first of these subjects is reviewed in section 2.5. Here we present the most prominent indexing methods to facilitate efficient searches on multi-dimensional data. Then, in section 2.6 we introduce the Space Filling Curves, a central concept for our research, with special attention on Hilbert curves. Finally, in section 2.7 we explain several compression methods, starting with traditional approaches and then focusing on the special needs for databases and OLAP.

2.2 Decision Support Systems

Decision Support Systems (DSS) are computer-based tools allowing a person or group to make informed choices that will drive the present and the future of an organization. Although generally associated with managerial decisions in large commerical enterprises, the applicability of these systems is very wide, ranging from retail vendors choosing brands to be sold next season, to doctors deciding which drugs are best for a certain treatment, to environmentalists pursuing the fastest ways to contain an oil spill.

The evolution of DSS started in the late 60's. At that time Morton and Stephens [1968] first described a DSS, and the concept rapidly gained interest from industry and academia. Guidelines for the development of such systems were formalized by various authors in the 70's and 80's, including Davis [1974], Alter [1980], and Sprague and Carlson [1982]. However, DSS flourished and matured in the 90's with the publication of influential work on data-warehouses by Inmon [2005] (1st ed. 1990) and the star-chema by Kimball [1995], and with the advent of big commercial players into the market, including IBM, Hyperion, Microsoft, Business Objects, and others. DSS presently supports an ample range of applications in many diverse fields.

2.2.1 DSS taxonomy

Although many different criteria can be used to classify DSS, we have chosen a categorization discussed by Power [2002] which separates DSS into five distinct groups according to the dominant functionality they provide. In practice, characteristics from these groups may be combined to build a flexible and robust system.

- *Data-driven DSS* analyze internal and external structured company data, gathered over a period of time. Examples of these systems are Executive Information Systems (EIS) and Business Intelligence (BI).

- *Model-driven DSS* access and manipulate simulation models with data and parameters provided by the users, to assist in the analysis of different scenarios. These systems are usually not data intensive.

- *Knowledge-driven DSS* are based on the structured representation of knowledge. Two clear examples of this category are: expert systems, which use rules and facts to infer new information; and description logics, a formalism oriented to konwledge access and reasoning.

- *Document-driven DSS* use unstructured documents as the input. The term *documents* here refers not only to textual data, e.g. manuals, legislation, emails, web pages, etc.; but also to multimedia sources, e.g. images, sound and video. Some technologies applied in this field include Natural Language Processing (NLP) and pattern recognition.

- *Communication-driven DSS* facilitate shared tasks of decision-makers working as a group. Such technologies include: virtual meetings, scheduling, white boards, etc.

2.2.2 Data-driven DSS

Our research focuses on the management and processing of large amounts of data, therefore we will concentrate on data-driven DSS. This particular category relies upon the extraction, management and analysis of enterprise data to provide useful information to the users empowering them to make appropriate and timely decisions. Data-driven DSS is by itself a very broad field, thus Eavis [2003] classifies it in three categories according to the type of user interaction and analysis complexity:

- *Information Processing:* Basic querying and reporting funcionality, with limited visualization modules. Queries are designed by IT specialists, extracting information directly from the Database Management Systems (DBMS).

- *On-Line Analytical Processing:* Provides complex analysis of data and support for ad-hoc queries. Data is collected from various sources over a period of time and structured as specialized multidimensional cubes. This category constitutes the groundwork for our research and is therefore further developed in section 2.3.

- *Data Mining:* These systems take one step further in automatization by using knowledge-based techniques to discover hidden patterns, relationships and trends in the data with minimal user interaction. Large amounts of structured data are usually required to obtain meaningful results, therefore data-mining tools are considered a hybrid data-driven and knowledge-driven DSS.

2.3 On-Line Analytical Processing

On-Line Analytical Processing (OLAP) is a category of DSS with focus on providing the user with powerful analysis and reporting tools built on top of specialized data structures. Data in these structures are arranged in a hierarchy of categories called *dimensions*, thus Thomsen [2002] defines OLAP as a set of multidimensional information systems.

The term OLAP was coined for the first time by Codd et al. [1993]. Codd's paper presented twelve rules tailored to evaluate any OLAP system. Although the article intended to become a set of standard guidelines, it was widely criticized for being sponsored by Arbor Software, a proprietary vendor of OLAP technology. An alternative and simpler definition is provided by the [OlapReport], and it is known as the FASMI definition. The acronym stands for Fast Analysis of Shared Multidimensional Information (FASMI), concisely describing five intuitive requirements, which are more general than Codd's rules. Therefore, an OLAP system should provide fast analysis tools in a concurrent environment, and is also required to be based on a multidimensional model, scalable to large data sets.

2.3.1 OLAP origins

Although the term OLAP was not used until 1993, its foundations had seen light almost half a century ago. Based on the timeline depicted by Pendse [2006], the origins of multidimensional analysis can be tracked back to the work of Iverson [1962]. Iverson presented A Programming Language (APL), a mathematically defined language which, although syntactically complex, provided clean definitions and transformations of multidimensional variables. The use of arrays presented in APL was adopted by a company called Management Decision Systems to build the first multidimensional application [Vlamis, 2002]. This product, called Express, was created in 1970 and eventually became part of the Oracle suite.

After Express, several other commercial systems added their own innovations to the developing OLAP technologies. Some notable contributions were made in the early 80's by Comshare's System W and Metaphor's Data Interpretation System (DIS). The former was the first to use a hypercube approach, while the latter, now part of IBM, was a pioneer in using the emerging client-server paradigm and in proposing multidimensional analysis on top of a relational database.

Starting in the late 80's and 90's spreadsheets grew increasingly popular and became the front-end of choice for OLAP tools. In more recent years, the developments have geared towards more user-friendly interfaces with powerful spreadsheet add-ins and intuitive graphical tools, while at the same time strengtening specialized OLAP back-ends. A more detailed architectural view of modern OLAP systems is presented in the next section.

2.3.2 OLAP architecture

Modern OLAP systems have evolved towards a clearly defined three-tier architecture [Han and Kamber, 2001], as illustrated in figure 2.1. Raw data is extracted from internal company sources e.g. operational On-Line Transaction Processing (OLTP) databases, legacy systems, etc., and possibly from external sources e.g. web services. Next, data from these potentially heterogeneous sources is unified by cleaning any inconsistencies and fitting it into a new data model, to finally be loaded into a repository. This repository is usually a Database Management Systems (DBMS) and constitutes the physical underlying data layer of the OLAP system, known as the *Data Warehouse (DW)*. Smaller repositories oriented to the different departments of a company are called *data marts*. Altogether, this set of data preparation steps is known as the Extraction Transformation and Load (ETL) process.

Once the Data Warehouse has been loaded, the OLAP back-end servers further transform the data by building specialized structures to be able to answer user queries more efficiently. These structures are known as *Data Cubes* and are discussed with greater detail in section 2.4. Finally, the front-end layer provides a user-interface that will allow the decision-making person or group to interact with the OLAP server, obtaining useful analytical reports usually in grid or graphical format.

It is important to make a clear distinction between OLAP systems and their transactional counterpart OLTP. The first systems deliver reports obtained from data collected over a period of time, to be used as analytical support for decision-making.

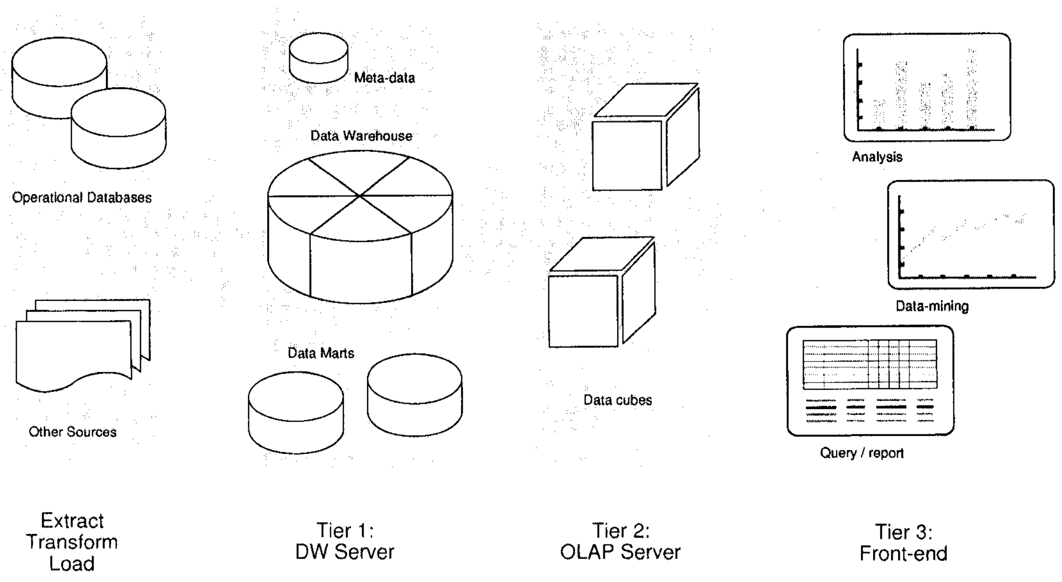


Figure 2.1: OLAP three-tier architecture

The second, on the other hand, manage day-to-day operational data, and are unable to provide historical analysis because they only deal with the present snapshot of the data. Nevertheless, as was mentioned in the previous architecture description, OLTP systems play an important role by providing the raw data that constantly feeds the Data Warehouse. As descibed by the "father of the data warehouse" William Inmon [2005], "Data warehouse data is nothing more than a sophisticated series of snapshots, each taken at one moment in time".

2.4 The Data Cube

The OLAP architecture presented in the previous section clearly depicts the components involved in providing the user with efficient decision-making support. Processing starts by extracting data from various sources and integrating it in a Data Warehouse. However, before OLAP front-end tools produce meaningful reports, special structures tailored to multidimensional analysis must be created. The current section presents these structures in detail, including definitions, functionality and modeling.

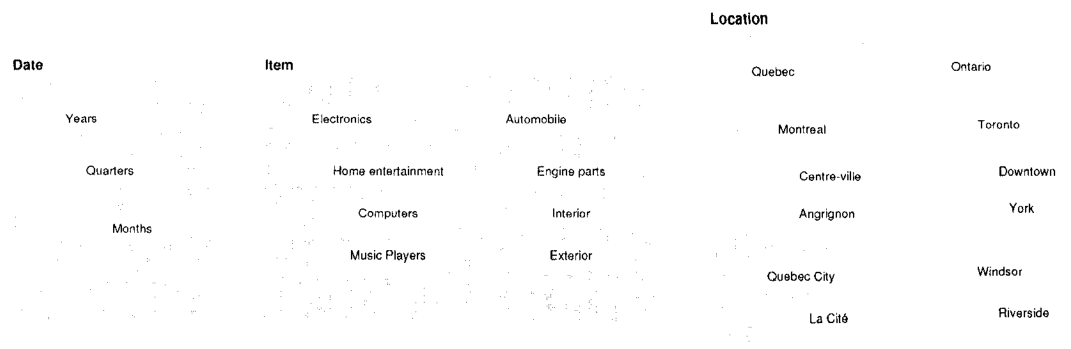


Figure 2.2: Dimension hierarchies example

2.4.1 Basic concepts

The single most important definition in OLAP is that of a *dimension.* A dimension is a category of entities relevant to an organization. For example, a retail store might record the sales for the following dimensions: date, item, location, etc. This will allow the store to know the monthly sales for specific lines of items in a given branch. Dimensions can have sub-dimensions organized in a hierarchy. For instance the hierarchy of *date* can be defined as: years, quarters and months, while the the *item* dimension can be subdivided into: electronics, automobile, furniture, etc.; and even further down as: home entertainment, computers, portable music players, etc. (see figure 2.2). In the literature, dimensions are often called *feature attributes* or simply *attributes.*

In figure 2.3 we can observe a cube of data formed by labeling the coordinate axes with the elements of each of the dimensions. By extrapolating this concept into *d* dimensions, we obtain a hypercube, where every orthogonal axis corresponds to a different dimension. This generalized structure is known as a *d-dimensional Data Cube*, the central piece of multidimensional analysis. Depending on the hierarchy level used to visualize the cube, different cube perspectives are obtained as described in section 2.4.2.

While dimensions embody the scaffolding of the Data Cube, *measure attributes*

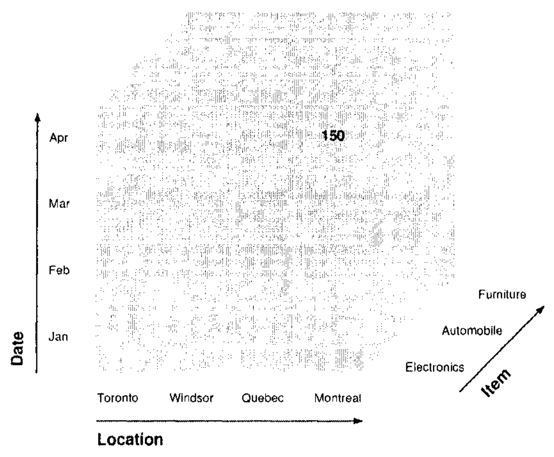


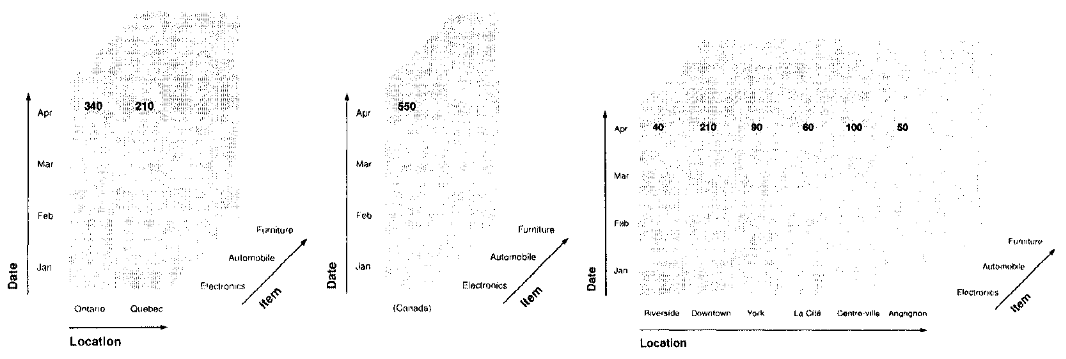
Figure 2.3: Data cube example

constitute the values filling each one of its cells. Following our example, the measure attribute *sales* has the value of 150,000 CAD for electronics sold in the Montreal store during the month of April. Note how the set of dimensions represents the coordinates in the cartesian space for each measure. Examples of measure values are: sales, inventory, revenue, Return of Investment (ROI), etc. One or more measure attributes can be recorded in each cell, however for simplicity only a single one is considered in most research papers.

A Data Cube is a finite structure in all dimensions. This means that every dimension must have a finite number of elements called its *cardinality*, and represented as $c = |d|$. In our example, the cardinality of the "date" dimension is $|date| = 12$, when expressed in months. The set of all the values for a dimension is known as its *domain*.

2.4.2 Perspectives

The Data Cube and its inherent dimension hierarchies allow users to visualize data from different perspectives. Each one of these perspectives is defined by the following OLAP functions:



(a) Collapse (b) Reduction (c) Drill down

Figure 2.4: Roll-up and Drill-down perspectives

- *Roll-up:* The roll-up perspective performs *aggregation* of the measure values by either collapsing a dimension hierarchy or by reducing dimensions. Collapsing a dimension implies climbing up the hierarchy. For example, in figure 2.4a we obtained provincial sales by adding up the sales for every corresponding city. Likewise, reducing a dimension means entirely removing it from the Data Cube, and can be interpreted as a full collapse to the highest level of the dimension hierarchy. In the example depicted on figure 2.4b, we have reduced "location", obtaining a cube with the total countrywide sales ordered by item and date.

  Although many aggregation operators are possible e.g. summation, min, max, standard deviation, etc.; here, as in the remainder of the thesis, the *summation* operator is used.

- *Drill-down:* The drill-down perspective is the exact opposite of the roll-up. It either steps down the dimension hierarchy or introduces a new dimension. This results in an increased level of detail, also called *granularity.* In figure 2.4c, we have performed a drill down on the location dimension, so specific district store sales are shown.

- *Slice:* As its name implies, this perspective takes a slice of the cube, similar to a physical slice of bread. A single value is selected from a given dimension.

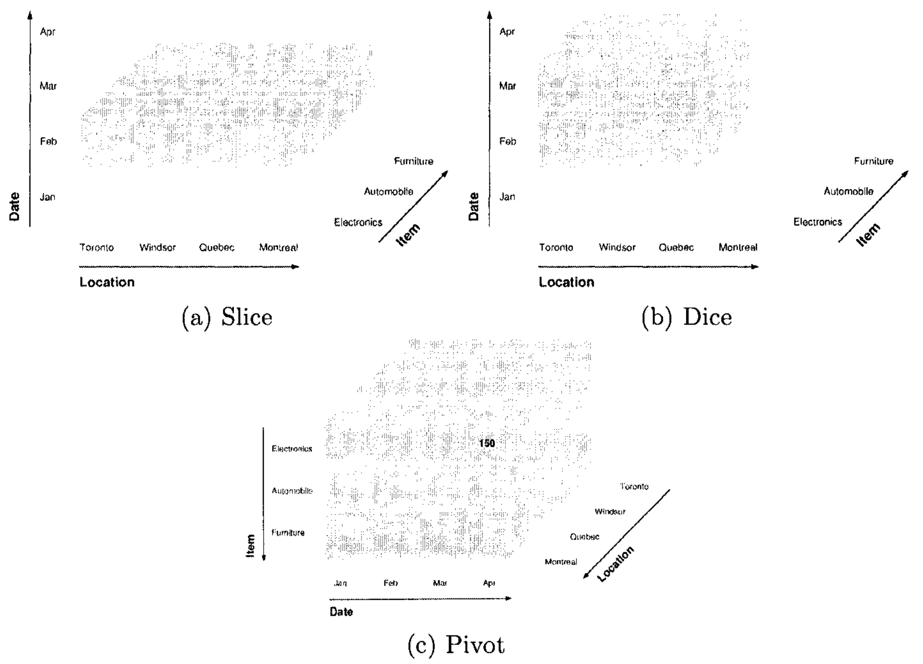


Figure 2.5: Slice, dice and pivot perspectives

Figure 2.5a presents a slice with *date= "february"*. No aggregation is performed.

- *Dice:* The dice perspective defines a sub-cube by applying a selection on more than one dimension. It is in fact, a generalization of the slice perspective. Figure 2.5b shows a cube with the dice selection: *(product = "electronics" or "automobile") and (location ≠ "Montreal") and (date = "february" or "march")*

- *Pivot:* The pivot perspective provides an alternative visualization of the data by rotating the cube along one or more of the axes. In figure 2.5c the original cube has been rotated clockwise around the Z and X axis.

2.4.3 The Data Cube Lattice

Data architects select a group of d dimensions to be loaded by ETL into the Data Warehouse. The dimensions are then used as the cartesian axes to form a Data Cube. However, it is possible to select only a sub-set of dimensions to obtain different levels of *aggregation* for the measure values. From our example, if we form a cube with

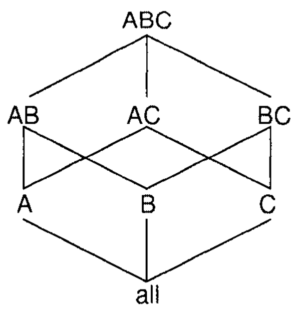


Figure 2.6: A Data Cube Lattice

the "date" and "item" dimensions only, each cell will represent the countrywide sales aggregate value for a specific month and item (figure 2.4b).

A sub-cube formed by aggregating zero or more dimensions is called a *cuboid* or *view.* A cuboid is equivalent to a roll-up perspective where zero or more dimensions have been collapsed, and its measure values aggregated, showing different levels of granularity.

For a cube with d dimensions, there exist 2^d cuboids, which can be organized in a *lattice* framework to model dependencies among views [Harinarayan et al., 1996]. Figure 2.6 shows a lattice for a sample 3-dimensional cube, where every cuboid corresponds to a node. Connectors indicate that a child node can be calculated from its parent by collapsing one dimension and aggregating the corresponding measure values.

The root node of the lattice is called the *base cuboid* and presents the finest granularity because no dimensions have been collapsed. This node has the property of being able to generate all the other cuboids. The "all" node at the bottom of the lattice is a single aggregated value for the whole cube. Finally, the set of all cuboids is known as the *full Data Cube.* This aggregation procedure is rather cumbersome to implement in plain SQL, thus a new operator was proposed by Gray et al. [1996], known as the *CUBE operator.*

2.4.4 OLAP taxonomy

The Data Cube is a conceptual model used to represent multidimensional data. Various research groups have approached the implementation of these definitions in different manners, giving birth to distinct OLAP server technologies. Query and processing performance, as well as storage requirements are affected by the choice of one of these OLAP flavors.

- *Multidimensional OLAP (MOLAP):* The Data Cube is mapped directly into a multidimensional array. This approach is straightforward and can provide fast access to the data by using simple array indexes. On the other hand, special array-based engines are required to manage the data. These engines often suffer from scalability problems in *sparse spaces* [Morfonios and Ioannidis, 2006], which is the case when many attribute combinations do not have an associated measure value. These "empty" cells are very common in most real-life applications.

- *ROLAP:* One table of a relational or extended-relational DBMS is used to store each cuboid. These systems profit from all the existent leverage of mature relational DBMS technology, including standard query resolution mechanisms. Empty cells producing sparse spaces do not need to be represented at all. To minimize the lack of a direct array index, additional structures are built on top of the data. This approach constitutes the groundwork for this thesis.

- *Hybrid OLAP (HOLAP):* This approach combines Multidimensional OLAP (MOLAP) and ROLAP technologies. Generally dense regions are stored in multidimensional arrays, while the rest use a relational DBMS [Kaser and Lemire, 2003].

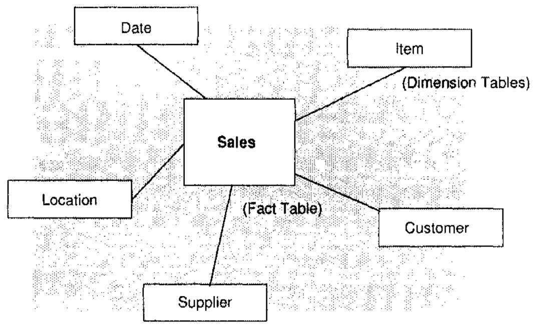


Figure 2.7: A Star Schema

2.4.5 The Star Schema

ROLAP solutions have the advantage of using solid and powerful Relational DBMS (RDBMS) to manage extremely large amounts of cube data. However, the conventional Entity-Relationship model published by Chen [1976] to conceptualize the database layout is not appropriate for the OLAP environment. Entity Relationship (ER) models are tailored to OLTP systems where efficient concurrent data access is provided for large numbers of transactions involving a few records. The OLAP world is very different, with few, less predictable and usually non-concurrent transactions accessing a large number of records. Furthermore, as we have already described, OLAP data is gathered over a period of time, rather than holding a single current snapshot as in OLTP.

To reflect all these differences, Data Warehouse (DW) architects use a *Dimensional Modeling (DM)* approach. Instead of creating a complex ER model for all of the organization's processes, DM identifies relevant business processes, and creates individual models for each of them [Kimball and Ross, 2002]. These logical-level models were proposed by Informix [1995], and are known as *Star Schemas* because of the distribution of the tables in the resulting diagram, resembling a star, as shown in figure 2.7.

The star schema is composed of a central fact-table and satellite dimension tables. The *dimension tables* include descriptive information for a given dimension.

The primary keys from all dimension tables are combined to form the *fact table.* The rows of this table are a combination of attribute values, each representing a given dimension, and one or more corresponding measure values. This simple de-normalized design minimizes the joins in multi-dimensional requests, thus improving query performance.

The fact table corresponds to the base cuboid in the Data Cube lattice presented in section 2.4.3. All the other cuboids are generated from this one by performing aggregations on one or more dimensions. A row in the base cuboid represents a single business operation performed at a given point in time. The historical collection of every operation implies that the base and aggregated cuboids can reach enormous sizes. One of the goals of our research is to reduce the storage footprint of the Data Cube by removing the redundancies inherent to the multidimensional cube model.

2.5 Indexing

DBMS had their origins in flat-file processing systems and have evolved into efficient managers of extremely large amounts of data. From those early days to the modern Data Warehouse systems, handling large data sets has always been accompanied by various indexing techniques [Cueva, 2002], designed to provide reasonable response times to user queries by avoiding exhaustive searches of the data.

An *Index* is a particular ordering or structure built with the purpose of reducing the volume of data that must be analized to answer a query. There exist two types of indexes: primary and secondary. Primary indexes contain the actual data records, and are generally simple orderings of the data. Secondary indexes are additional structures including only references to the data [Salzberg, 1996]. In the present thesis the term *index* will refer to a secondary structure, unless indicated otherwise.

Various index structures and associated techniques have been proposed, and different ones are used for each flavor of database system. *B-tree* indexes or one of its variants are widely applied in OLTP and often constitute the foundation for more sophisticated techniques. In our area of interest, OLAP systems, along with spatial databases, e.g. Geographic Information System (GIS), *Bitmap* or *R-tree* indexes are used, therefore we will further discuss them in the present section. Other particular database systems require specialized structures; for example: colour and spatial information are combined in indexes for graphics retrieval in Image Databases [Bertino et al., 1997].

2.5.1 B-trees

A *B-tree* is a structure proposed by Bayer and McCreight [1972], as an alternative to AVL trees, the de-facto standard index at the time, defined by Adelson-Velskii and Landis [1962]. On top of providing logarithmic-time insertions and deletions as with AVL trees, B-trees guarantee a uniform minimal height and can be efficiently managed in secondary storage, loading only required parts of the index in main memory at a time, and improving query response times substantially [Gupta et al., 1997].

In a typical implementation, a node is composed of a list $(p_0, x_1, p_1, ...x_l, p_l)$. The list contains l data keys x_i in sequential ascending order, and $l+1$ pointers p_i. Data keys uniquely identify an element of the index, and are generally either the actual data, or complemented with an extra pointer to the data. In the triplet (p_{i-1}, x_i, p_i) the pointer p_{i-1} leads to the child node with the elements smaller than x_i, while p_i leads to the child node with the elements greater than x_i. Searching a record on a B-tree is a procedure similar to searching a binary tree. At each node the algorithm chooses a pointer by simple arithmetic comparisons with the keys, and follows the path until the leaves are reached.

In a B-tree structure, each path from the root to any leaf has the same length h, $(h \geq 0)$, which is also called the *height* of the tree. A range $[a, b]$ is defined, such that any node has a number of children n within this range: $a \leq n \leq b$. Figure 2.8 shows a B-tree where $[a, b] = [2, 3]$. Insertion and deletion operations are carried out

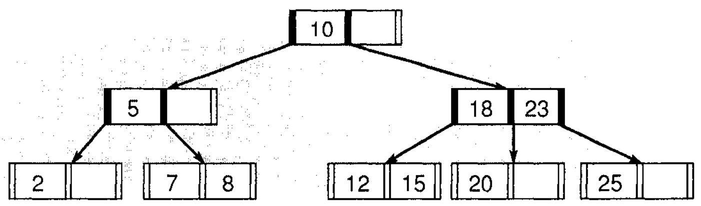


Figure 2.8: B-tree

at the leaf level and propagated in a single pass to the root. If the root is to be split, then a new root is created and the height of the tree is increased. B-trees are well suited for database environments beacuse of its flexibility to define a custom number of children per node. This characteristic allows to conveniently store each node in a multiple of a physical secondary-memory block by adjusting the values in the range. Values are usually modified by changing a sole factor k, where $k \in \mathbb{N}$, $a = k + 1$ and $b = 2k + 1$.

A variant called *B+ tree* is the most commonly used index structure in databases [Salzberg, 1996]. Enjoying the full advantages of a regular B-tree, a B+ tree stores all data keys in the leaf level, regardless if they already exist in intermediate nodes. Additionally, pointers are placed between leafs creating a complete linked list to facilitate the execution of range queries.

2.5.2 R-trees

In section 2.4 we presented the Data Cube, the central piece of the OLAP technology. This model is based on a multidimensional paradigm where the relevant facts in the problem domain are translated into axes of a d-dimensional space. The inherent spatial nature of the Data Cube has lead researchers to investigate access solutions that reflect this particular characteristic of OLAP data. An efficient spatial access method must adapt relevant indexing techniques to support phenomena such as clustering, spatial range queries, and massive amounts of data.

The ideas behind the simple design of the B-tree have served as the groundwork for spatial indexes. A B-tree builds a hierarchical division of a 1-dimensional space;

therefore the next logical step is to build hierarchical structures that divide the d-dimensional space. Some of the most prominent among these structures, described by van Oosterom [1999], are: the *KD-tree* [Bentley, 1975], which uses a data point to divide the space in two, similarly to the triplet described in the previous section; the *BSP tree* [Fuchs et al., 1980], well suited for 3D graphics; the *Quadtree* [Samet, 1984] and the *Grid File* [Nievergelt et al., 1984]. However, the most widely accepted spatial index method, namely the *R-tree*, uses the opposite approach, grouping data points first in order to form the space divisions. Greene [1989] and Smith and Gao [1990] have demonstrated the R-tree performance superiority when compared to the KDB-tree and the Grid File respectively. Moreover, Quadtrees and KD-trees do not easily map to pages in secondary storage, limiting their application in real database environments.

The R-tree was created by Guttman [1984], and shares similarities with a B+ tree. Leaf nodes are a composed of a set of pairs of the form (I, p), where p is a pointer to a group of d-dimensional records, and I is a d-dimensional rectangle, also called a *hyper-rectangle.* The hyper-rectangle is defined as $I = (I_0, I_1, ...I_{d-1})$, a set of intervals of the form $[a, b]$ in each of the d dimensions. Geometrically, I is defined as the smallest bounding box in d-dimensions to enclose the records pointed by p. Non-leaf nodes are also a set of pairs of the form (I, p), with the difference that p points to a child node, and I is the bounding box of all the hyper-rectangles defined in the aforementioned child.

Figure 2.9 shows an R-tree over a two-dimensional space, with data points represented as numbered dots. Hyper rectangles, marked by capital letters, group data points producing higher levels of the tree. Some data points (e.g. "2") lie in an overlapping region; nevertheless they belong to a single hyper-rectangle.

In the implementation, a node is made to correspond to a disk page, therefore the size of the page will define the maximum number of entries M to be allowed in any

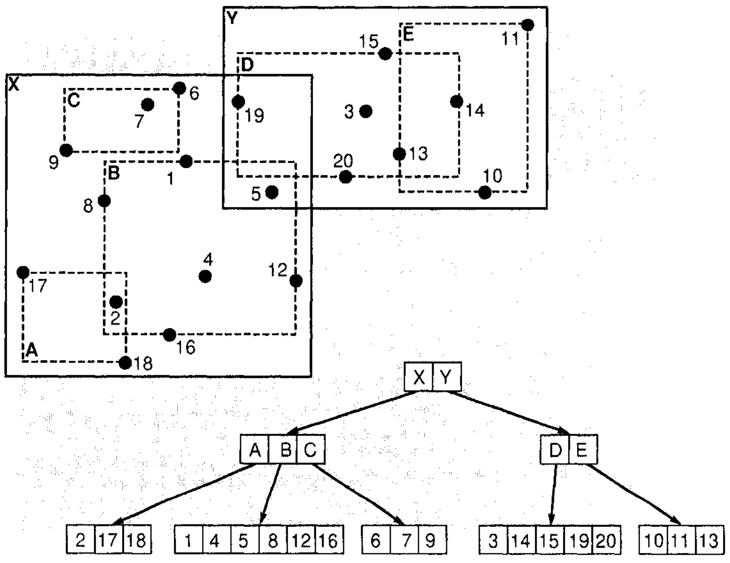


Figure 2.9: R-tree

node. The minimum number of elements is defined as $m \leq M/2$. This open-interval definition allows m to be fine-tuned for performance improvements. As in the B-tree, in an R-tree, all leaves appear at the same level. The height of the R-tree is at most $|log_m N| - 1$, with a branching factor of m, and N total index nodes.

Searching the R-tree is similar to the same operation in a B-tree; however because hyper-rectangles may overlap, it might require the traversal of more than one sub-tree. Inserting records is also analogous to a B-tree, starting from the leaves, the record is included in a node, modifying the corresponding hyper-rectangle. If the node overflows, it is split, and splits propagate up to the root. Although a B-tree approach is feasible for deletion, the authors of the R-tree have chosen to use a different approach. If a record deletion produces a node with less than m entries, it is eliminated, and the orphaned entries are re-inserted in sibling nodes.

Roussopoulos and Leifker [1985] described a "packed" version of the R-tree, which minimizes both hyper-rectangle coverage and overlap. In essence, packing a tree means saturating each node with as many children as will fit into a disk block. This basic technique is used in our index compression method presented in section 3.5.

RID	Item	bitmaps		
		E	A	F
1	Auto.	0	1	0
2	Elec.	1	0	0
3	Elec.	1	0	0
4	Furn.	0	0	1
5	Auto.	0	1	0

(a) Item Bitmaps

RID	Loc	bitmaps	
		ON	QC
1	ON	1	0
2	ON	1	0
3	QC	0	1
4	ON	1	0
5	QC	0	1

(b) Location Bitmaps

Table 2.1: Bitmap indexes

Other variations of the R-tree include: the *R+ tree* [Faloutsos et al., 1987], which minimizes overlap with a tradeoff of an increased number of nodes and references; the *R* tree* [Beckmann et al., 1990], which attempts to re-insert a percentage of an overflowed node, instead of splitting it; and the *Sphere tree* [van Oosterom and Claassen, 1990], which uses spheres instead of bounding boxes.

Spatial indexes, including the R-tree will benefit from ordering techniques to better cluster the data points to be inserted in the trees. One of these techniques is space filling curves, presented in section 2.6

2.5.3 Bitmap indexes

Bitmap indexes [O'Neil, 1989] are a technique used to provide fast access to relational tables, including those stored in a Data Warehouse, or to complement other spatial indexing techniques at the leaf level. It is based on the assumption that binary operations (AND, OR, XOR, NOT) are very fast to execute by most available computer platforms. For a dimension with c different values, the method generates c bitmaps. These bitmaps are a sequence of n bits, where n is the number of records in the table. A bit in a bitmap is set to 1 if the record holds a specific value for the attribute.

For example, tables 2.1a and 2.1b show the bitmaps for the Item (I) and Location (L) dimensions. The item can belong to the Electronics, Automobile or Furniture type, while the location can be either Ontario or Quebec. Therefore, three bitmaps

RID	Auto		ON		Res.
1	1		1		1
2	0		1		0
3	0	AND	0	=	0
4	0		1		0
5	1		0		0

Table 2.2: Bitmap query resolution

are created for I and two for L. The table includes five records. Record 4 for instance, corresponds to a sale of a furniture item in Ontario. Thus, row 4 in table 2.1a includes a 1 under the Furniture column, and zeroes for the other two columns. Similarly, for the same row in table 2.1b we observe a 1 in the Ontario column, and 0 under Quebec.

If we need to retrieve the sales for automobile items in Ontario, we simply perform a binary AND between the two corresponding bitmaps, as shown in table 2.2. The rows, whose bit is 1 after the operation are returned. In this case, only the first row satisfies the query.

Although sometimes not considered true indexing structures [Chaudhuri and Dayal, 1997], bitmaps can be advantageous compared to their tree counterparts [Han and Kamber, 2001], specially in low cardinality domains, because index comparison, intersection and union are reduced to bit arithmetic. However, for higher cardinalities or dimension counts, the number of bitmaps can be very large, and some compression techniques are required to reduce storage footprint. A second limitation is that bitmaps do not enforce any ordering on disk, so range queries can be costly due to the lack of clustering (see section 3.3.4).

2.6 Space Filling Curves

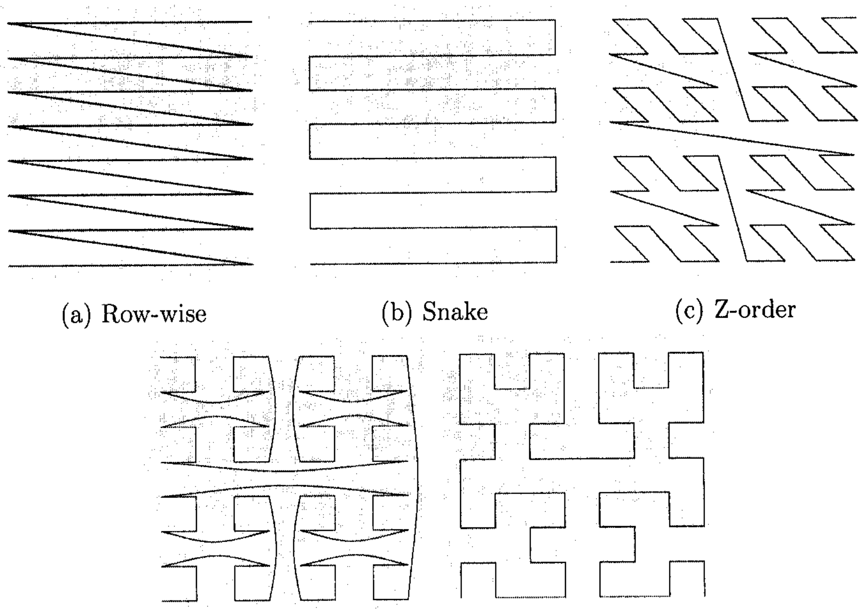
Spatial indexing techniques allow to efficiently search multidimensional data to answer user queries. However, while data is modelled in the d-dimensional space by using a Data Cube abstraction, in the physical world storage is provided in most cases by a strictly uni-dimensional disk or tape. Therefore, a transformation from multidimensional into linear space is required before loading the index. This transformation is accomplished by means of a *total order* of the data points.

A total order is defined as a relation R (e.g. $\leq$) on a set S, which $\forall a, b, c \in S$ complies with the properties of: Reflexivity ($a \leq a$); Anti-simmetry ($(a \leq b) \wedge (b \leq a) \rightarrow a = b$), Transitivity ($(a \leq b) \wedge (b \leq c) \rightarrow a \leq c$) and Trichotomy ($(a \leq b) \veebar (b \leq a)$) [Séroul, 2000].

Different total orders can be applied to the spatial data points; however an appropriate order should preserve as much as possible the spatial proximity of the points, after they have been transformed to 1-dimensional space. This *clustering* property is specially important for range queries. A range query defines a region in space and retrieves all the points included in such region. Thus, to answer these types of queries, the most efficient distribution of the index leaves is obviously the one that best keeps points that were close in space in the same or adjacent leaves. This task is well suited for space-filling curves.

First described by Peano [1890] as a 2-dimensional function, a *space-filling curve* can be generalized to d-dimensions as a bijective non-differentiable function from the unit interval $U_1 = [0, 1]$ into the unit hyper-cube $U_n = [0, 1]^d$ [Sagan, 1994]. A function $f : X \rightarrow Y$ is said to be bijective if $\forall x \in X, \exists! y \in Y : f(x) = y$. In other words, a space filling curve is a continuous curve that visits every point in space once and only once.

Some of the most well-known space-filling curves are shown in figure 2.10. The simplest curve visits each row consecutively, hence the name *row-wise*. A slight modification reverses the scan direction for alternate rows, obtaining the *snake curve*. Although extensively used in traditional TV cathode ray tubes, and many graphical hardware and software applications these curves are not well adapted to databases due to the long jumps which are translated into poor clustering capabilities.



(a) Row-wise (b) Snake (c) Z-order

(d) Gray (e) Hilbert

Figure 2.10: Space-filling curves

Some more suitable curves are: the *Z-order curve* [Orenstein, 1983], defined by interleaving the binary representation of point coordinates; the *Gray curve* [Faloutsos, 1986], drawn similarly to the Z-order curve, but first encoding the coordinates as Gray codes [Gray, 1953]; and the *Hilbert curve*, to be further described in section 2.6.1.

Various studies have been presented analyzing the properties of space-filling curves. Fortunately the choice is greatly facilitated by both empirical [Jagadish, 1990, Abel and Mark, 1990] and analytical results [Moon et al., 2001] which consistently point to the Hilbert curve as having the best clustering properties, outperforming its counterparts.

2.6.1 The Hilbert curve

The *Hilbert curve* is a space-filling curve defined by Hilbert [1891]. In contrast with the purely analytical approach by Peano, Hilbert described his curve geometrically as a mapping from U_1 onto U_2. If we consider an inifinite granularity in U_2, the curve

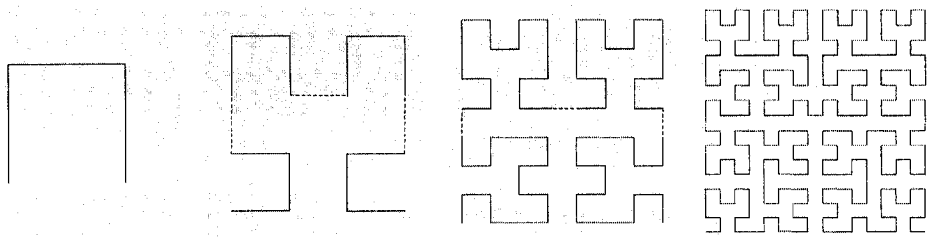

(a) $\mathcal{H}_1^2$ (b) $\mathcal{H}_2^2$ (c) $\mathcal{H}_3^2$ (d) $\mathcal{H}_4^2$

Figure 2.11: Hilbert 2-dimension curves

will give the rather counter-intuitive result of reducing $\mathbb{R}^2$ into $\mathbb{R}^1$. Nevertheless, all practical definitions for space filling curves use a discrete d-dimensional Euclidean space with finite granularity. Therefore, we adopt the notation proposed by Moon et al. [2001], where $\mathcal{H}_k^d$ denotes the k^{th}*-order*, d-dimensional Hilbert curve, a discrete non-differentiable bijection of the form: $\mathcal{H}_k^d : \left[0, 2^{kd} - 1\right] \rightarrow \left[0, 2^k - 1\right]^d$.

Although the Hilbert curve has traditionally been circumscribed to a $2^k \times 2^k$ space, Butz [1969] presented an analytical extension of the curve to d-dimensions. Moreover, Jagadish [1990] proposed a geometrical extension to d-dimensions, along with two techniques to produce Hilbert curves in non-square spaces by either replication of the curve or by defining non-square primitives with special rotation properties.

Figure 2.11 shows the steps for drawing a 2-dimensional Hilbert curve with orders 1 to 4, i.e. $\mathcal{H}_1^2$ to $\mathcal{H}_4^2$. We start with a basic shape from figure 2.11a; then we augment the subdivisions of the grid by a factor of 2, and at the same time we shrink the basic shape to half its size. The four shapes are placed on the grid, with SW rotated 90° clockwise, SE rotated 90° counter-clockwise. The four pieces are connected with line segments to obtain the order 2 curve in figure 2.11b. This curve is then used as the basic shape for order 3, and so on.

A $\mathcal{H}_k^2$ Hilbert curve can also be described as a *Lindenmeyer System*, where string-rewriting is used to generate fractals. The encoding is composed of an initial string "L", rewriting rules "L" $\rightarrow$ "+RF-LFL-FR+", "R" $\rightarrow$ "-LF+RFR+FL-" and angle of 90°. [Peitgen and Saupe, 1988]. This relatively simple encoded definition is often used by graphical programs to draw the Hilbert curve. An "F" indicates drawing a line segment one unit in the current direction, and each "+" or "-" indicates a turn with the given angle clockwise or counter-clockwise respectively. Starting with "L", the rewriting rules can be iteratively applied to obtain different-order fractal curves. When the desired order has been reached, the rewriting symbols are eliminated to obtain the final string. For example, stopping at the first iteration we obtain "F-F-F", which describes a $\mathcal{H}_1^2$ curve (fig 2.11a). Going one rewrite further: "+-LF+RFR+FL-F-+RF-LFL-FR+F+RF-LFL-FR+-F-LF+RFR+FL-+" $\rightarrow$ "F+F+F-FF-F-F+F+F-F-FF-F+F+F", which generates $\mathcal{H}_2^2$ (fig 2.11b), etc.

2.7 Compression

The size of a full Data Cube obtained by applying the CUBE operator can be hundreds or even thousands of times the size of the original fact table [Eavis, 2003]. However, unique characteristics of OLAP data play in favor of special-purpose compression methods, created to reduce the footprint of both cuboids and their respective indexes. In the current section we present the basics of data compression, along with the most influential work on statistical and dictionary techniques, followed by a description of the requirements and current research on database-oriented compression.

2.7.1 Definitions

Data compression is "the process of *encoding* a body of data D into a smaller body of data $\delta(D)$." The opposite process is called decompression, and is defined as *decoding* $\delta(D)$ "back to D or some acceptable approximation of D". [Storer, 1988]

Note that the decompression definition states the possibility of the decoded data D' not being exactly the same as D, but an acceptable approximation of it. When this technique is used, it is known as *lossy compression*, which is very common in multimedia applications, e.g. jpeg, mp3, etc., where some data non-noticeable to the human perception is trimmed from the original version. In database environments, lossy compression is found only in few special situations. An example is the "squashing" method proposed by DuMouchel et al. [1999] to scale down a set for data mining. However, in the vast majority of cases, no data can be lost during the process, and therefore we will focus only on *lossless compression.*

2.7.2 Statistical Techniques

One of the simplest compression techniques is Run Length Encoding (RLE). The central idea behind RLE is to replace n consecutive occurrences of a data item d by a pair nd. For example, the string *aaaabbccc* can be encoded as $4a2b3c$. Some considerations should be taken in account; for instance, if the input data consists also of numbers, an escape character must be introduced for the decompressor to be able to tell input characters from the repetition factors.

The naïve approach used in RLE is the basis for the *entropy coding* methods. These methods seek to replace symbols according to their frequency of occurrence. A variable length *codeword* is used for each symbol, such that more frequently used symbols are represented by smaller codewords. The codewords themselves must be unique, and should be *prefix-free*, which means that no codeword should be the prefix of another codeword. This characteristic is known as the *prefix property*, and is highly desirable in any environments where the decoder sequentially reads an encoded stream. For example, the set of codewords (10, 110) is prefix-free because its elements are uniquely identifiable in any sequence e.g. : 10|110|110, 110|10|110, etc.

All methods using the probability of occurrence to replace symbols from an input data set are called *statistical compression methods.* Some of the methods that form the groundwork for more advanced techniques are:

- *Morse Code:* Developed by Morse [1840], it encodes the letters of the English alphabet into sequences of dots and dashes, originally intended to be transmitted as electric impulses over a telegraph. To speedup transmission over the wire, the code assigns the shortest codes for the most frequently used letters, e.g. e='·', t='–', i='··', and longer codes to the least frequently used, e.g. q='––·–'. The code is still in use today for military and emergency purposes.

- *Shannon-Fano Coding:* Discovered independently in the late 1940's by both Shannon [1948] and Fano [1949], it orders the items from most to least probable. The set is then divided in two subsets with nearly-equal total probability. The first set is assigned the bit '0', while the second the bit '1'. This procedure is recursively repeated until unary sets are reached, resulting on unique variable-size binary codes assigned to each symbol.

- *Huffman Coding:* This method was presented by Huffman [1952], as a class assignment for Professor Claude Shannon. The data items are sorted in descending order according to their frequencies, and the two items with the minimum frequencies are grouped, assigning bits '0' and '1' to them. Next the list is re-sorted, with the newly-formed group representing a single combined item. The steps are repeated until only one group is present. Both Shannon-Fano and Huffman methods build a binary tree, where the codes are easily read from the root of the resulting tree to the nodes. However, the Huffman method is preferred most of the times for producing better codes [Salomon, 2004]. Additionally, Huffman codes have been demonstrated to be optimal when the probabilities of the source items are negative powers of 2. [Pu, 2006]

- *Golomb Code:* The Huffman and Shannon-Fano methods are applicable only for a finite list of input symbols, because the coding trees must be computed and stored. Therefore Golomb [1966] created a technique which defines an infinite set of variable-length codes using the distribution law of probabilities. Golomb defined his codes in terms of RLE, where no matter how long the run-length is, there will be an optimal code to represent it.

- *Arithmetic coding:* Mostly the work of Rissanen [1976], Arithmetic Coding tries to overcome the limitation of Huffman coding requiring to assign an integer number of bits for the length of a code [Bodden et al., 2002]. Instead of replacing each input symbol with a code, it replaces the whole input stream with a floating point number $a \in [0, 1)$. To encode and decode this number, each symbol is assigned a sub-range $s \subseteq [0, 1)$, according to its probability. Each time a symbol is encoded, the range is recursively restricted to that of the given symbol [Howard and Vitter, 1992]. Arithmetic coding has been the most successful alternative to Huffman, and achieves better results with reduced symbol sets.

2.7.3 Dictionary Techniques

The efficiency of the statistical techniques greatly depends on how close the probability distribution is to that of the input set. When such statistics are not easily available another approach must be used. *Dictionary-based compression methods* use a pattern respository, called a *dictionary*, to store strings previously encountered in the input stream. These methods do not require a statistical model, obtain a performance similar to their statistical counterparts, and are also well suited for general purpose compression, working on text, images, audio, etc.

Most of the dictionary-based techniques are extensions of the work of Lempel and Ziv in the late 70s, and honour their contribution by using the prefix "LZ" in their names. The basic algorithms are as follows:

- *LZW:* LZW was proposed by Welch [1984], as an improvement to LZ78, and became very popular aided in part by its simplicity when compared to the original LZ methods. The technique starts by initializing the dictionary with pairs (a, i), where a represents each symbol in the input alphabet, and i a corresponding index. One variable containing the current longest pattern l is initialized to the empty string. Next, it reads the input sequentially one character c at a time and appends it to the pattern l. If this newly concatenated string $l + c$ is in the dictionary, then nothing is written to the output, a known pattern has been found, so l is updated as $l = l + c$. On the other hand, if $l + c$ is not in the dictionary, it is a new pattern, therefore the index for the current pattern l is written to the output, $l + c$ is included in the dictionary, and l is updated as $l = c$. LZW has been widely used, some examples being the UNIX compress/uncompress commands, the GIF image format, and the V42bis modem transmission protocol.

- *LZ77:* The original compression algorithm, presented by Ziv and Lempel [1977]. It uses a buffer called a "sliding window" to search for the longest string pattern. The window is divided in two, a *history (H)* and a *lookahead (L)* buffer. Each time the window is advanced one character, then the longest substring found in H that is a prefix of L is stored in the dictionary as triplet, including a relative offset from the right side of H and a length.

- *LZ78:* A major problem in LZ77 is that it cannot recognize patterns that have been shifted out of the sliding window. LZ78 [Ziv and Lempel, 1978] adresses this issue by keeping a permanent dictionary with all the patterns seen so far. A very well known method, based on LZ77 and LZ78, is *Deflate.* Created by Phillip Katz [Deutsch, 1996] and widely used in the "Zip" and "Gzip" families of compression products, it features the LZSS extension [Storer and Szymanski, 1982], combined with Huffman codes.

2.7.4 Database compression

In section 2.4.3 we described the full Data Cube as the set of all cuboids conforming a lattice structure. Each one of the cuboids is obtained by aggregating zero or more dimensions from its parents. The purpose of this operation is to provide different pre-computed levels of granularity, readily available for OLAP queries. However, the sizes of the resulting full cubes can be daunting even for modern parallel processing and storage systems, easily reaching hundreds to thousands of times the size of the original fact table. Moreover, index structures built on top of the cuboids can grow to be as large as 10% of the total stored data.

Clearly, a compression method is required in order to reduce the physical space footprint. However, the traditional techniques decribed in the last section cannot be directly applied to a database environment. First, the size of a cuboid invalidates the possibility of a full compression or decompression when a query is executed. Instead, the concept of *compression granularity* must be introduced. This concept implies that compression is confined to a certain database structural level, namely: attribute, tuple, page or file. Examples of the application of the granularity definition while using traditional techniques can be found in the work of Ray et al. [1995] and Westmann et al. [2000]. The former investigate the application of statistical and dictionary techniques to databases, and propose a method combining attribute frequency distributions and arithmetic coding. The latter use attribute-level compression, applying numeric, string or dictionary compression, depending on the type of field to be processed.

Second, statistical and dictionary methods were defined for general input data; however, data stored in OLAP cuboids present unique characteristics that must be considered when designing a compression method. One such feature is the redundancy produced when a group of records has identical values for certain attributes. Therefore, only the differences between these values should be stored. This is the main idea behind TDC, a method proposed by Ng and Ravishankar [1997]. This method sorts the records in lexicographical order, and stores only differences between consecutive tuples, compressing one disk page at a time. A similar technique was later presented by Goldstein et al. [1998]. Here, the authors define a *frame of reference*, where high and low interval points are defined for each dimension, and tuples are then represented as differences with these points. Both algorithms constitute the groundwork for the present thesis; therefore they will be described in detail in sections 3.2.2 and 3.4.1 respectively.

Although not a compression method in the strict sense, a *partial cube* technique can also be used to reduce the size of the computed lattice. This method calculates a reduced number of cuboids, based on a user-defined aggregate condition, as in Iceberg Cube [Beyer and Ramakrishnan, 1999], while queries falling in non-materialized cuboids can be answered by *surrogate views* as in Eavis [2003].

Sismanis et al. [2002] presented Dwarf, a compression method for OLAP that abandons the relational model for a specialized structure which functions as both data repository and index, using pointer indirection for navigation. Compression is achieved by identifying and eliminating *prefix* and *suffix* redundancies. In the lattice of figure 2.6 the cuboids AB and ABC present prefix redundancy, while ABC and BC show suffix redundancy. This method is used in comparison tests in section 4.3.4.

Finally, Lakshmanan et al. [2003] proposed the QC-trees method. This technique is a data structure that allows the storage and search of quotient cubes. Quotient cubes [Lakshmanan et al., 2002] are summary structures created by partitioning the set of cube cells into classes. The Dwarf and QC-trees are simlar in that they always compress full data cubes. However, in the lower levels of the lattice almost no aggregation is performed. Therefore, it is likely that a partial cube with a number of un-compressed materialized bottom cuboids would outperform both Dwarf and QC-trees at query time.

2.8 Conclusions

Decision Support Systems play an important role in any organization expecting to profit from current technologies in order to make informed decisions. We have categorized these systems, and indicated that the focus of our reasearch falls on the data-driven DSS, and specifically on OLAP. These types of systems rely on enterprise data collected from various sources over a period of time to provide useful decision-enabling information. For this purpose, OLAP systems build specialized structures, which permit the execution of ad-hoc queries.

We have presented the Data Cube, the most prominent OLAP abstraction. This structure is built from multi-dimensional data, where each orthogonal axis corresponds to a given data dimension. The distinct cube perspectives: slice, dice, pivot, roll-up and drill-down were also introduced. Then, we showed how the cube is built by following a lattice construction, and how the multi-dimensional data should be modelled by using a Star Schema.

Because Data Cubes can reach extremely large sizes, various techniques must be applied to allow for efficient access of the data. The first presented method consists of building indexes on top of the exisiting data. These indexes are additional structures designed to minimize the volume of data that must be analyzed to answer a query. Among these structures we have mentioned the B-trees, extensively used in OLTP, as well as R-trees, probably the most effective indexing method for OLAP.

The second technique, namely compression, aims to reduce the footprint of Data Cubes and to improve the access to the data. We explored the traditional exisiting methods and highlighted the reasons making their direct application infeasible in a database context. As explained, database compression requires a definition of granularity to limit the amount of processed data. Additionally and most importantly, general-purpose methods do not profit from regularities found in the database. Therefore, specially tailored methods will be presented in chapter 3.

Finally, in order to build the indexes and to apply a compression method, a total order of the data is required. Thus, we have examined the Space Filling Curves, with special attention to the Hilbert curve because of its superior clustering properties.

Chapter 3

Compression Framework

3.1 Introduction

Modern OLAP systems are effective tools designed to support the decision-making processes of an institution. These systems are based on a multi-dimensional paradigm, with the data cube as their central model. In the ROLAP environment, the well-proven relational concepts are applied to handle the large amounts of historical data obtained from different enterprise sources. However the generation of the data cube usually leads to a considerable size explosion of the already large historical input set. These massive amounts of data pose several challenges for OLAP systems, such as the storage requirements for the cuboids and the ability to efficiently answer ad-hoc user queries.

In the present chapter we address these issues by proposing a framework for data compression, indexing and querying, specially tailored for OLAP. In section 3.2 we introduce basic definitons for database encoding, along with one of the most influential works in the field, namely the TDC method. Then, we propose our own compression algorithm for multi-dimensional data in section 3.3. This technique, based on the Hilbert space-filling curve, overcomes the limitations found in TDC, offering a clean integration with our indexing schema.

Section 3.5 explains our algorithm for the creation of compressed R-tree index structures. Our approach is influenced by the ideas behind the GRS method, discussed in section 3.4. We identify and exploit the strengths of this method, while at the same time analyzing and overcoming some of its disadvantages. Next, the strategy for efficient query resolution, using the created indexing structures, is presented in section 3.6. The final sections 3.7 and 3.8 address the important subjects of scalability and paralellization. Our framework provides the facilities to handle data sets with arbitrarily large row and dimension counts. At the same time it takes advantage of the processing power in a parallel environment to build the compressed structures and to answer user queries, with almost optimal load balancing across the nodes.

3.2 OLAP Data Compression

Traditional methods presented in section 2.7 are not directly applicable to database compression. These methods are designed for single block processing, where data is considered as a finite stream of bytes to be encoded as an undivided entity. In an OLAP environment, the size of the stored data makes this single block approach infeasible, and therefore the concept of compression granularity had to be introduced. The database is then considered as a hierarchy of structural elements: attribute, page, tuple and file. Even when using this concept, the application of traditional techniques on each structural component ignores how OLAP data is composed and how to take advantage of this special disposition.

For these reasons we have developed a compression method specially designed for OLAP data, as part of a complete framework that also includes indexing and querying capabilities. Before presenting our compression alternative we must define certain basic concepts and analyze one of the most influential works in database compression, which constitutes the groundwork of our own research.

3.2.1 Definitions

Differential encoding [Gottlieb et al., 1975] is a transformation applied to a set of data, in which only the changes in succesive values are represented, instead of taking a fixed reference point. Formally, the transformation is defined as:

$$D : X \rightarrow Y$$

$$\forall x \in X, y \in Y; \quad y_i = \begin{cases} x_0, & \text{if } i = 0 \\ x_i - x_{i-1}, & \text{if } i > 0 \end{cases}$$

where X is the *ordered* set of original values $X = \{x_0, x_1, ...x_{i-1}, x_i, ...x_n\}$, and Y is the resulting transformed set. Note that D is not an injective function because the mapping does not produce unique elements of Y. However, the transformation effect can be reversed in practice by storing each element of Y as a distinct value in the same order as the original elements.

Another important concept is *bit compaction*, a well-known basic compression technique where each of the values of a set Y' is represented as a binary number of b bits. If the range of Y' is r, the value b is calculated as $b = \lceil log_2 r \rceil$.

As an example of the two previous concepts, consider the original ordered set $X = \{55, 58, 60, 66\}$. If each element is stored in a standard 32-bit integer record, the whole set X would occupy 128 bits. Applying the differential encoding transformation we obtain $D(X) = \{55, 58 - 55, 60 - 58, 66 - 60\}; \quad Y = \{55, 3, 2, 6\}$. Note that differential encoding by itself does not provide any compression, as the resulting set would still be stored in 128 bits. However, by defining $Y' = \{y \in Y : y \neq y_0\}; \quad Y' = \{3, 2, 6\}$ we obtain a subset of Y with an integer range of 7 values ($[0, 6]$). By bit compaction we can represent each element of Y' as a 3-bit binary value. With the combination of these two simple techniques, the final representation for our example is $Y = \{55_{10}, 011_2, 010_2, 110_2\}$, occupying only 41 bits, a figure representing 67% of compression with respect to the original size. Here as in the rest of the thesis, the underscores x_{10} and x_2 denote decimal and binary numbers respectively.

3.2.2 Tuple Differential Coding

Proposed by Ng and Ravishankar [1997], TDC is a database compression technique based on the differential encoding concept. The main idea behind the method is to store the differences between consecutive tuples instead of the tuples themselves.

The basic notation presented in the original work is as follows. A relation is defined as a set of d attributes $\mathcal{R} =< A_1, A_2, ...A_d >$, indicating a d-dimensional space composed of tuples from the cartesian product $A_1 \times A_2 ... \times A_d$. Each attribute has $|A_i|$ different values, corresponding to its cardinality. A tuple is represented as $t =< a_i, a_2, ...a_d > \in \mathcal{R}$.

The compression process is performed in four stages:

1. *Tuple reordering:* Each d-dimensional tuple is first converted into a unique integer, by means of the function $\varphi : R \longrightarrow \mathcal{N}_R$, defined as:

$$\varphi < a_1, a_2, ...a_d >= \sum_{i=1}^{d}(a_i \prod_{j=i+1}^{n} |A_j|) \tag{3.1}$$

   The integer obtained by applying $\varphi(t)$ in equation 3.1 is an ordinal position of t within the $\mathcal{R}$ space. Sorting the resulting set of integers in ascending order we obtain $\mathcal{N}_R = \{0, 1, ...||R|| - 1\}$, where $||R|| = \prod_{j=1}^{n} |A_j|$. This is equivalent to a standard *lexicographical order* with respect to the attribute sequence.

2. *Attribute ranking:* Different attribute orders yield different values for the conversion function φ. Therefore, the authors recommend sorting the attribute fields in increasing order according to their cardinality as a heuristic to achieve good compression results. This step should be done before performing the tuple conversion and reordering.

3. *Block partitioning:* As a method built for database environments, TDC uses the concept of granularity to limit the compression and decompression process

to a single disk page. When a tuple is requested, only the block where that tuple resides is transferred to main memory and decompressed.

4. *Block encoding:* Differential encoding is applied to each block. The first tuple is used as a reference, while the rest are replaced by the differences with respect to the preceeding tuple. As noted before, differential encoding does not provide compression *per se*, consequently the tuples are stored using bit compaction on each attribute field, combined with RLE to eliminate leading zeroes.

Decompression is performed using the inverse of the conversion function previously defined in equation 3.1. This inverse is $\varphi(e)^{-1} =< a'_1, a'_2, ..., a'_d >$, where each a'_i is calculated independently with the following equations:

$$a'_i = \left\lfloor \frac{a^r_{i-1}}{\prod_{j=i+1}^{d} |A_j|} \right\rfloor \tag{3.2}$$

$$a^r_i = a^r_{i-1} - a'_i \prod_{j=i+1}^{d} |A_j| \tag{3.3}$$

where $i = 1, 2..., d-1$, $a^r_0 = e$ and $a'_d = a^r_{d-1}$. Equation 3.2 finds the integer quotient a' corresponding to attribute i, while equation 3.3 obtains the remainder, used to calculate the next attribute value.

For example, consider the tuples $t_0 =< 0, 0, 2, 34, 12 >$ and $t_1 =< 0, 1, 0, 7, 52 >$, where cardinalities have been ranked in ascending order: 4,4,4,64,64. Assume both tuples will be stored in a single disk block. The first one is the reference point, thus it is stored unchanged. For the second tuple we calculate the difference with the first one by following the conversion: $\delta = \varphi(t_1) - \varphi(t_0) = 16884 - 10380$; $\delta = 6504$. This difference is equivalent to the tuple $\delta =< 0, 0, 1, 37, 40 >$ obtained with equations 3.2 and 3.3. The leading zeroes are eliminated with RLE: $\delta =< 2, 1, 37, 40 >$, and the rest of the attribute values are encoded with bit compaction as $01_2\,100101_2\,101000$.

Although TDC obtains high compression rates varying from 75% to 85%, it suffers from several problems. Probably the most important deficiency is the lack of an integrated indexing solution to execute queries over the compressed data. This is definitely not a trivial issue because a compressed data set is of no use if it cannot be efficiently searched.

Al least two other drawbacks are evident after a deeper analysis of TDC. The first is related to the lexicographical order applied to the tuples prior to compression. This total order is equivalent to a row-wise scan, providing poor clustering capabilities, as described in section 2.6. The second is the size of the resulting ordinal numbers. With a modest cardinality of 100, in 5 dimensions, the resulting ordinal will overflow standard 32-bit registers, and even 64-bit registers cannot deal with higher dimension data sets. Details on the manipulation of arbitrarily long numbers and its related overhead are not provided in the original paper.

3.3 Hilbert Differential Coding

Our algorithm, Hilbert Differential Coding (HDC) is the first building block of the OLAP compression framework. It provides state of the art compression ratios above 80%, comparable to TDC, while at the same time addressing TDC's deficiencies. Furthermore, the method is cleanly integrated with the high performance multi-dimensional indexes, presented in section 3.5.

3.3.1 HDC algorithm

We assume an initial cuboid C modelled within a space of d-dimensions. Each dimension corresponds to a feature attribute A_i, whose cardinality is represented as $|A_i|$ and defines the number of different values on the i-th dimensional axis. The cuboid stores a total of n tuples arbitrarily scattered through the space. Each tuple is a point defined by a set of d coordinates, corresponding to its attribute values, plus a single aggregated measure (see section 2.4.1). This thesis, as with most research papers, focuses on the compression of the attribute values only. The output of the algorithm will be a compressed cuboid C'.

A Hilbert space-filling curve [Hilbert, 1891] is used to represent the multidimensional tuples. As previously described in section 2.6.1, a Hilbert curve visits every point of the d-dimensional space once and only once. Therefore, following the curve order, we obtain a set of strictly increasing values $H = \{h_0, h_1, ..., h_{s^d}\}$, $h_i \in \mathbb{N}$, (s =space side length), belonging to each space position. We will refer to these values as *Hilbert ordinals.* Every tuple is in fact a point in the space whose coordinates determine a unique Hilbert ordinal. A simple 2-dimensional representation is shown in figure 3.1a.

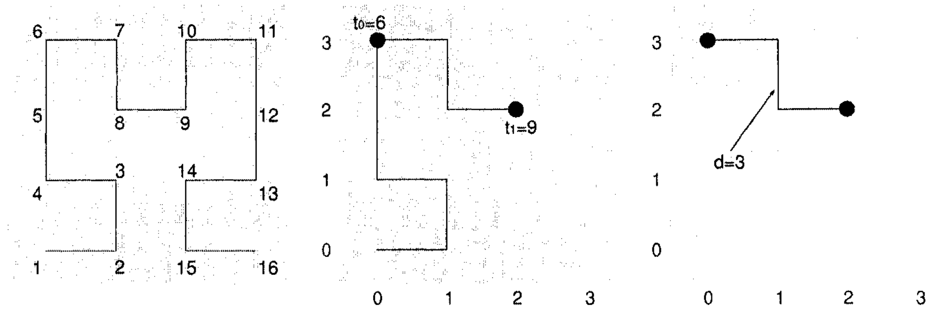
Algorithm 1 presents the complete HDC data point compression method. It starts by converting all tuples into their corresponding Hilbert ordinals. Recall that a Hilbert Curve is a bijective mapping, and therefore invertible. Thus, a record encoded into a Hilbert ordinal can be uniquely unencoded again into its multidimensional tuple form. Once all the tuples have been encoded, they are sorted according to their Hilbert ordinal values. The relevance of the order of these two initial steps is further discussed on section 3.7.2.

Using the concept of compression granularity at the disk page level, the algorithm proceeds by filling one disk block at a time in the following manner. Each Hilbert ordinal h_i is read from the sorted cuboid. If the page P is currently empty, h_i is used as the initial reference value. Otherwise, we calculate the difference δ between the current Hilbert ordinal and its immediate predecessor h_{i-1}. We continue to calculate differences until the block has been filled to capacity.

To determine when the space in the block has been exhausted, we use a counter p, which stores the maximum number of bits required for any already-seen difference. Each time a difference δ is calculated, the number of bits b it requires in bit-compacted form is calculated. If this value is greater than the current maximum, p is updated and a new page size estimation P_{est} is determined based on both the value of p and

Algorithm 1 HDC Data compression

```
Input: An aggregated and arbitrarily sorted cuboid $C$, with $n$ tuples consisting of
    $d$ feature attributes and one measure. A temporary buffer $P$. A required disk
    block size $B$ in bits.
Output: A fully compressed cuboid $C'$ with page level granularity.
 1: for all $n$ tuples of $C$ do
 2:   convert each tuple to its Hilbert ordinal value
 3: end for
 4: re-sort the $n$ ordinals of $C$
 5: for each tuple $i$, with $i \leq n$ do
 6:   read next Hilbert ordinal $h_i$
 7:   while estimated page size $P_{est} < B$ do
 8:     if $P$ is currently empty then
 9:       record $h_i$ as the uncompressed reference value
10:       set the max $\delta$ bit count $p = 0$
11:     else
12:       calculate differential $\delta = (h_i) - h_{i-1}$
13:       compute bit count $b$ required to represent $\delta$
14:       if $b > p$ then
15:         $p = b$
16:       end if
17:     end if
18:   end while
19:   write page $P$ to disk, appending the measures as required
20: end for
```



(a) $\mathcal{H}_2^2$ ordinals (b) Redundancy (c) Differential

Figure 3.1: Hilbert Differential Coding

the number of processed differences. When this estimation surpasses the block size in bits B, the bit-compacted differences are written to disk, appending the uncompressed measure aggregation values.

HDC effectively addresses the redundancy inherent in multidimensional cuboids by eliminating the distances shared by the Hilbert ordinals along the curve. Figure 3.1b geometrically exemplifies this concept with a simple $\mathcal{H}_2^2$ curve. Two tuples are considered, namely $t_0 =< 0, 3 >$ and $t_1 =< 2, 2 >$. Their corresponding Hilbert ordinals are $h_0 = 6$ and $h_1 = 9$ respectively. Figure 3.1c illustrates the differential $\delta = h_1 - h_0$, which will be used to represent t_1. Note how the redundant distance from the origin to h_0 has been eliminated. In terms of storage, the original t_1 should be stored in two 32-bit registers, while the encoded differential version requires only two bits when bit-compacted: $\delta = 3_{10} = 11_2$.

3.3.2 Decompression

In order to decompress an HDC block, we first obtain the Hilbert ordinals h from the stored differentials δ by applying the simple formula $h_i = h_{i-1} + \delta i$. Recall that a reference Hilbert ordinal h_0 was stored uncompressed in the block B. The first differential δ_1 is added to h_0 to obtain ordinal h_1, the second differential δ_2 is added to h_1 to obtain h_2 and so on. Finally a transformation function is applied to every h

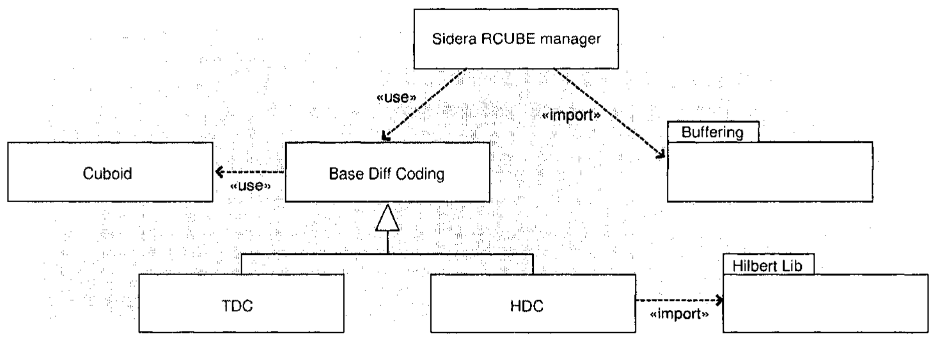


Figure 3.2: HDC UML Class Diagram

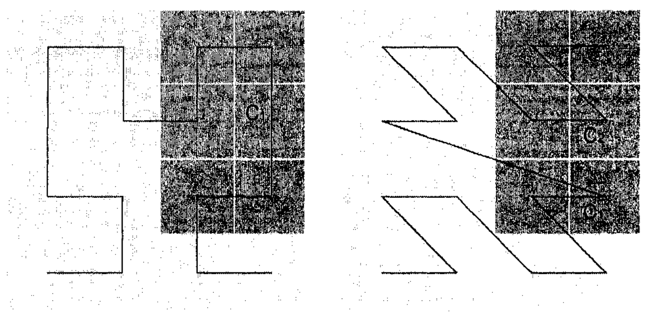
to convert them to multi-dimensional tuple form.

Although obtaining Hilbert ordinals from the diferentials is quite straightforward, obtaining the d-dimensional tuple representation from the ordinals is a complex and computationally intensive operation. The HDC module originally incorporated the open source library by Moore [2005]. Unfortunately, these functions proved to be unacceptably slow in early stages of the development. Therefore, we implemented an optimized version of the Hilbert transformation routines.

Our Hilbert conversion library follows the algorithm proposed by Butz [1971]. This approach applies several succesive bitwise-operations to a byte-divided Hilbert ordinal in order to obtain each of the d-dimensional coordinates and vice versa. The algorithm is well suited for modern processors where bit-wise operations are extremely efficient. Furthermore, we have incorporated the optimizations to the algorithm proposed by Lawder [2000], including the use of Gray Codes when computing Butz's σ factor. Aditionally, we have implemented various engineering improvements, notably in memory management and fast bit-reading routines. As a result, we have gained an enhancement of more than one order of magnitude for the Hilbert transformations.

3.3.3 Class Hierarchy

The package of classes participating in the HDC method is shown in the UML diagram [OMG, 2005, Fowler, 2003] in figure 3.2. The base class uses a cuboid abstraction



(a) Hilbert: 1 cluster (b) Z-order: 2 clusters

Figure 3.3: Clustering comparison

to provide generic differential coding without bit-compaction. The HDC class encapsulates the calculation of the Hilbert ordinal differentials, using our optimized Hilbert routines, while TDC acts as a comparison benchmark. The required previous conversion and sorting is performed by an external manager, which also handles the interaction with the buffering sub-system. Some utility classes are omitted for clarity.

3.3.4 Clustering

By using a Hilbert curve we overcome the clustering problem found in TDC. As mentioned in section 2.6, the Hilbert curve presents the best clustering properties among its space-filling relatives. A *cluster* is defined as a group of points inside a query, consecutively connected by the curve. Figure 3.3, based on Moon et al. [2001], shows an example of a spatial query, represented as a shadowed area, where the Z-curve forms two clusters, while the Hilbert curve forms only one. Jagadish [1990] and Rong and Faloutsos [1991] report a minimum average of clusters on 2×2 queries for three space-filling curves. The results show the Hilbert curve with the best minimum average of 2, compared with 2.625 for the Z-curve and 2.5 for the Gray curve.

Superior Hilbert clustering also translates into a more balanced load among processors when the framework is used in the Sidera distributed OLAP environment, as will be explained in section 3.8 .

3.3.5 High dimensionality

In our approach, we have targeted the scalability to higher dimensions, identified earlier as a deficiency in the TDC method. As explained in section 3.2.2, the lexicographical order upon which TDC is based requires a previous transformation of every tuple into an ordinal following equation 3.1. The maximum length in bits l_{max} of the resulting integer value can be calculated from the product of the attribute cardinalities: $l_{max} = \left\lceil log_2(\prod_{j=1}^{n} |A_j|) \right\rceil$. A standard 32-bit register will overflow with a rather small 5-dimensional cuboid with uniform cardinalities of 100, where $l_{max} = \lceil log_2(100^5) \rceil = 34$ bits. In the same manner, by just doubling the number of dimensions, the resulting ordinals will overflow a 64-bit register.

In our framework, we have used the GNU Multiple Precision (GMP) library [GMP] to overcome bit-size limitations in the calculated Hilbert ordinals. GMP is an open-source project, providing functions for arbitrary precision arithmetic. It is widely used on academic research and commercial products, due to its consistent interface, flexibility and efficiency. The designers of the library assert that the precision of calculations handled by GMP is only limited by the available memory of the underlying system.

GMP is part of the GNU's Not Unix (GNU) project, and is therefore free to run and distribute, but most importantly free to study, change and improve [GNU]. The Lesser General Public License (LGPL) license allows even commercial programs to use the library. GMP is organized in modules, targeting integer, rational and floating-point numbers. Also included are facilities for random number generation, formatted output and a convenience C++ wrapper. Although most functions are high-level Application Programming Interface (API) calls, there exists the possibility of calling low-level routines that work "behind the scenes", purposedly exposed for time-critical applications such as our framework.

Although GMP is probably the most efficient multi-precision arithmetic library, our experimental evaluations have demonstrated an average time overhead factor of 4, as described in section 4.3.5. Therefore, we included several optimizations in our code. The first one, mentioned in the previous paragraph, was the use of low-level functions whenever possible, for instance for bit-twiddling tasks (e.g. mpn_rshift()). Second we centralized the creation and release of GMP related variables, and promoted the reuse of GMP allocated memory throughout the code. Finally, for places where time is absolutely critical, we reduced GMP usage to almost that of a long-number placeholder, working directly with the GMP internals, as is the case with the Hilbert transformations during the query resolution process.

3.4 OLAP Index Compression

Well designed indexes play a very important role in any DBMS by providing improved query resolution response times. Nevertheless, index complexity is greatly increased in OLAP because the multidimensional nature of the cube data requires the definition of special structures. The most relevant of these were presented in section 2.5.

The index compression module in our framework is based on the work of Goldstein et al. [1998], and addresses some of its drawbacks. We profit from the fact that the cuboid data was sorted in Hilbert order before applying the HDC compression. This allows our framework to incorporate a technique developed by Kamel and Faloutsos [1993], called *packed R-trees.* As a result, we obtain a robust multi-dimensional index with impressively high compression ratios of up to 98%.

3.4.1 GRS Compression

The database compression algorithm proposed by Goldstein et al. [1998], herafter referred as the Goldstein Ramakrishnan and Shaft (GRS) method, is a technique based on the principle of differential encoding. Although founded on the same concept as

TDC, the algorithm does not require a conversion of the tuples to a single lexicographical ordinal. Instead, differences are calculated between each one of the attributes and their respective pre-computed reference values in a column-wise fashion.

The input set is a relation with n attributes $\mathcal{R} =< A_1, A_2, ...A_n >$. Each attribute is defined as a set of k different values $A_i = \{a_1, a_2, ..., a_k\}$, where $k = |A_i|$ is the cardinality of the i-th attribute. The range of these values can be represented by a finite interval $r_i = [min(A_i), max(A_i)]$.

The compression process is divided in two constituent stages:

1. *Page level:* A *page* is a sub-set of the input relation $\mathcal{R}$, defining the unit of compression. The basic observation of the authors of GRS is that for a given page P, the range of values that appear in a particular attribute field i is smaller than the complete range for the aforementioned attribute: $|r_i^P| \leq |r_i|$. Therefore, we can use differential encoding to store the values in page P for attribute i as a difference with the minimum value of the i-th range.

   Extending these definitions to all attributes in the data set, we obtain value ranges for each of the dimensions. In a spatial representation, the minimum and maximum values of these ranges form the two opposing corners of a d-dimensional rectangle. This d-rectangle contains all points in the page and is called a *frame of reference.* In other words, the frame F indicates the range of possible values in each dimension, for all the tuples included in the page P.

   For example, consider the page $P = \{(29, 190), (53, 179), (37, 192)\}$, a subset of a 2-dimensional cuboid. The frame of reference is then defined by the two points $F = \{(29, 179), (53, 192)\}$. The ranges for each attribute include $|r_1^P| = 25$ and $|r_2^P| = 14$ values; therefore the points in F can be represented with bit compaction by using only 5 and 4 bits respectively. Each point is stored as the difference between itself and the minimum corner of the frame. Thus the final representation for the page is $P = \{(00000_2, 1011_b), (11000_2, 0000_2), (01000_2, 1101_2)\}$.

We have reduced the space used by the three points from 192 bits, using standard 32-bit registers, to only 27 bits. Note however, that in order to decompress the points, the frame of reference must be stored in the page in uncoded format. According to the authors, this is "well worth" the overhead.

2. *File level:* The second part of the method simply defines the number of tuples that should be included in a page. For this task, GRS uses a greedy approach in such a manner that one page will contain as many tuples as can be fit in a single disk block. Nevertheless, the compression granularity of the algorithm is not limited to the page level because a single tuple, or even a single attribute can be decompressed independently from the others in the same block. This is possible considering that the values are dependent only on the frame of reference, and not on the previous tuple, as with TDC. Although this is an advantage for environments requiring field-level granularity, it sacrifices a certain amount of compression, due to the minimum value being further away than consecutive ones.

For our research, the most important feature of GRS is the possibility of compressing R-tree structures. In order to compress an R-tree index the two d-dimensional points delimiting a node bounding box are considered one single 2d-dimensional point. The set of current-level hyper-rectangles is then processed with page and file level compression. The frame of reference can be either 2d-dimensional or just d-dimensional. In the latter case, a logical division must be conserved in the 2d-dimensional tuples to be treated as two points when interacting with the frame.

The compression efficiency in GRS greatly depends on the set of tuples stored in a page. The authors propose as part of the method, an algorithm called GBPack to be applied during the loading process. In GBPack, the points are sorted in ascending order in the first dimension. Then this dimension is divided in p partitions, and each partition is sorted in alternating ascending and descending order in the second dimension. This process continues for all dimensions, obtaining a "join-the-dots" linearization of the data. The pages are then packed by following the linearization. The resulting curve is similar to a d-dimensional row-wise scan, therefore point clustering is quite likely to be compromised, particularly as we increase the dimension count.

A serious drawback of GRS is precisely the GBPack linearization, which fully determines the final degree of compression. The problem lies in the definition of the number of partitions to subdivide each dimension. The authors use $p = \sqrt[d]{P}$, but this implies an *a priori* knowledge of the number of resulting compressed pages P, which is in turn not known until after the set has been compressed. This issue is weakly acknowledged in the original paper, suggesting an estimation of P by using the bounding box of the whole set and assuming the data is uniformly distributed. This assumption is also not true in most real life environments where a certain level of skew is present.

3.5 Hilbert R-Tree Coding

The second core element of our compression framework is the Hilbert R-tree Coding (HRC) indexing method. It combines the best multidimensional indexes available, with extremely high compression ratios of up to 98% for the resulting structures. Moreover, it is seamlessly integrated on top of the HDC-processed data and addresses the shortcomings found on GRS.

3.5.1 HRC Algorithm

The input set for HRC is the compressed cuboid C' obtained by applying HDC compression to a d-dimensional raw-data cuboid. The C' cuboid is composed of b disk blocks. Recall that a block includes a header reference tuple and a variable number of bit-compacted Hilbert differentials. These blocks constitute the leaf nodes of the R-tree, and form the basis for the bottom-up construction of the index.

Every node N of the R-tree is represented by a d-dimensional hyper-rectangle as explained in section 2.5.2. A hyper-rectangle is fully defined by its two opposing d-dimensional vertexes. The lower vertex V_{min} includes the set of minimum values for every dimension from all the points enclosed in the node. We will refer to this vertex as the *pivot* point. In a simliar manner, V_{max} is a point whose coordinates are the maximum dimension values from all the enclosed points.

Algorithm 2 describes our index generation and compression method. We start by calculating the hyper-rectangle vertexes for every node N^i in the leaf level. Each of the computed node boundaries is stored uncompressed as a pair $N_0^i = (V_{min}\ ,\ V_{max})$ in a temporary initial working index level L.

The next step is to create compressed nodes in level $L + 1$. A node N_L from the lower level is sequentially included in an inmediately higher-level node N_{L+1} until the latter reaches the disk block size. Including N_L into N_{L+1} implies the following tasks. The new vertexes of N_{L+1} are calculated by comparing the current V_{min} and V_{max} with their respective points in N_L. A change in the vertexes also triggers a modification in the number of bits required to represent the range of values of each dimension. These values are recalculated, and then used to compute the estimated page size P_{est} by a simple product with the current number of nodes in N_{L+1}.

If the last inclusion of a node N_L in level $L + 1$ produced a page size greater than the physical block B it is rolled back, and the page is written to disk in the following manner. For every node included in $L + 1$, a differential δ is calculated between each of the attributes of its vertexes, and the corresponding coordinates of the current pivot G. Next, the differentials are written to disk applying bit compaction. The two vertexes of the node are stored sequentially and kept as two logically separated entities.

We continue creating nodes in level $L+1$ until all nodes in L have been processed. Then level $L+1$ becomes the current level, and the process keeps going until reaching

Algorithm 2 HRC Index compression

```
Input: A cuboid $C'$ containing $b$ blocks compressed with the HDC method. Temporary buffers $P$ and $T$. A required disk block size $B$.
Output: A fully compressed r-tree index $I$ with one node per page.
1: for all $b$ blocks of $C'$ do
2:   calculate $V_{min}$ and $V_{max}$ vertexes of $b$,
     $V = \{A_1, A_2, ..., A_k\}$
3:   write node $N_0^b = (V_{min}\ , V_{max})$ to $L$ uncompressed
4: end for
5: repeat
6:   retrieve number of nodes $nN$ from level $L$
7:   for each $i$, with $i \leq nN$ do
8:     read one node $N_L = (V_{min}, V_{max})$ from level $L$
9:     if page $P$ is empty then
10:      create node $N_{L+1}$ in level $L + 1$
11:      initialize $N_{L+1} = N_L$
12:    else
13:      include vertexes of $N_L$ in $N_{L+1}$
14:      recalculate pivot of $N_{L+1}$: $G = min_{L+1}$
15:      recalculate bits to represent $V_{max}$ in $N_{L+1}$
16:      if estimated page size $P_{est} > B$ then
17:        rollback last inclusion
18:        write $N_{L+1}$ to page $P$ uncompressed
19:        for all $N_L$ nodes of level $L$ do
20:          for all $k$ attributes of each vertex $V_L$ in $N_L$ do
21:            calculate differential $\delta$ as $V_L(k) - G(k)$
22:            write $\delta$ to $P$ using Bit Compaction
23:          end for
24:        end for
25:      end if
26:    end if
27:  end for
28:  write level $L$ to a temporary storage $T$
29: until number of nodes $nN == 1$
30: move all $L$ from $T$ to $I$, top to bottom
```

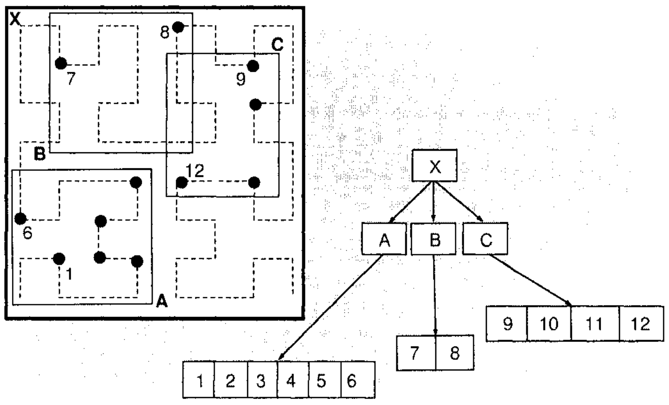


Figure 3.4: R-tree over Hilbert data

a single node, the root of the R-tree. Finally, all levels are written from top to bottom to the resulting index I, in order to facilitate search (see section 3.6.1).

Consistent with the HDC data compression algorithm, the unit of compression is mantained at the page level, therefore one disk block can be compressed and decompressed at a time. Another similarity is the variable number of points included in each block. In both cases, the number of points is dynamically determined by the algorithms and directly depends on the geometrical distance of the points in the d-dimensional space. The smaller the differences with the previous tuple in HDC and with the pivot in HRC, the more points that will fit in a single block. Figure 3.4 shows an R-tree built on top of a simple Hilbert-ordered 2-dimensional data set.

Our approach completely eliminates the linearization problem found in GRS. Not only is an initial estimation of compressed pages not required, but clustering is mantained and enforced by the underlying Hilbert ordering. Loading data into the R-tree nodes is effortless because the point sequence is pre-sorted by the HDC algorithm. It suffices to take the next available point until a disk block is filled to capacity. The same holds true for higher levels of the index.

We note that packing an R-tree by following a Hilbert order was proposed by Kamel and Faloutsos [1993]. This article shows the superiority of the Hilbert curve over *lowx* [Roussopoulos and Leifker, 1985] and z-order [Orenstein, 1983] to pack and produce smaller R-trees with higher fanout.

An added advantage over GRS is the fact that the equivalent of the frame of reference is not stored in the hyper-rectangle it bounds. Instead, it is compressed in the level inmediately above. Searching the index always requires traversing the nodes from top to bottom, therefore the pivot stored in level $L+1$ will necessarily be decompressed before its corresponding hyper-rectangle in level L (see section 3.6.2). The block however, requires a header indicating the number of bits used in each vertex dimension. These values are also stored with bit compaction.

3.5.2 Decompression

Decompressing an HRC block is a straightforward process. First, the query engine traverses the R-tree, retrieving the hyper-rectangle corresponding to the node $N = (V_{min}, V_{max})$. Then, it loads the disk block that is delimited by the aforementioned hyper-rectangle. This block contains the children of N in hyper-rectangle form. The i-th child is defined by the formula $N^i = (G + \delta_{min}, G + \delta_{max})$. Here, the differentials coresponding to the two opposing vertexes of N^i are separately added to the pivot of the enclosing hyper-rectangle $G = V_{min}$. A similar procedure is applied for the children of N^i. A detailed explanation of the query engine traversal of the index is given in section 3.6.2.

3.5.3 Class Hierarchy

Figure 3.5 shows the UML class diagram for the HRC method. The Sidera server provides a manager class, which acts as the process flow controller. It loads the raw data using elements from the Buffering package and uses the External Sort class to arrange the tuples in Hilbert order. The bulk of the index creation is delegated to the Compressed R-tree class. Hyper-rectangles are represented by a class with the same name, while their compression is handled by a separate class, of which they are a constitutive part. In order to interact with the leaf node level, some classes from

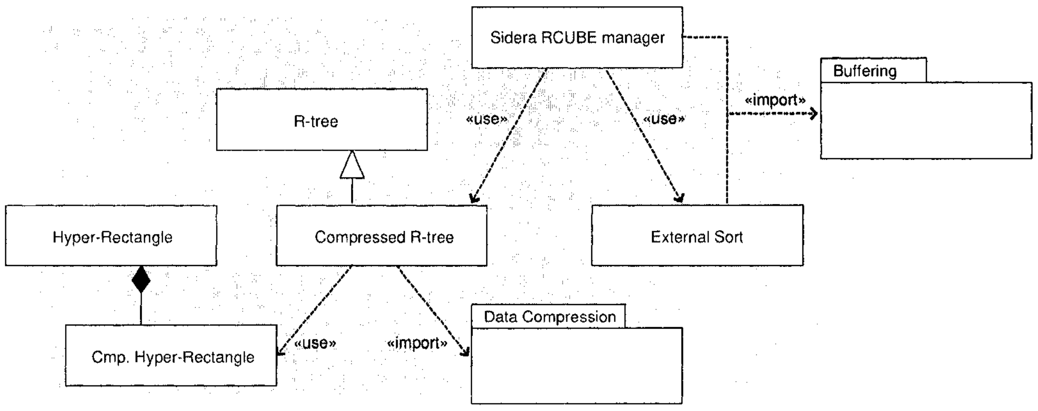


Figure 3.5: HRC UML Class Diagram

the Data Compression package are imported.

3.6 Query Engine

The third core component of the framework is the Query Engine. This module accepts a user query and performs the required search on the compressed data, retrieving and presenting the requested results. The algorithm is based on the orginal Sidera Engine, and uses the services from the data and index sub-systems for the decompression process. The Query Engine benefits from increased fanout and reduced I/O provided by the compressed R-tree and data, and also from fast Hilbert transformations, engineered specifically to improve the performance of query resolution.

3.6.1 Search strategy

For choosing an effective search strategy for query resolution we must first consider the behavior of the underlying storage given the disposition of the compressed data. A hard disk drive will perform more efficiently when long sequential reads are executed, as opposed to small random accesses [Lo and Ravishankar, 1996], because of the time involved in moving the read-head back and forth in the latter case. The R-tree packing algorithm from section 3.5 effectively adapts to this condition by writing the index nodes sequentially from "left-to-right" into levels, and then storing the levels

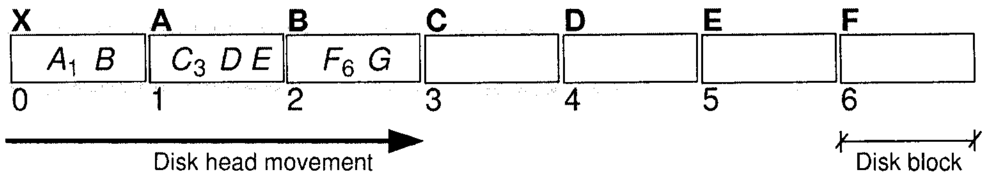


Figure 3.6: Disk track of a packed R-tree

themselves in a top-to-bottom fashion.

A standard Depth-First search (DFS) traversal is clearly not the best option. By searching up and down the R-tree, it will produce undesired random seeks leading to an increased disk head movement. On the other hand, the Linear Breadth-First search (BFS) approach proposed by Eavis [2003] is well suited for this environment. It examines the nodes in a BFS manner, but visiting only the ones that would be explored in a DFS search. This is possible thanks to an offset o included in each of the tree nodes, which indicates the position of its first child node. The offset allows to selectively identify the blocks corresponding to children nodes which satisfy the user query, and will be the only ones to be analyzed in a subsequent iteration of the algorithm. This condition, along with the packed R-tree disposition of nodes and levels, maximizes disk sequential reads and guarantees that the disk head travels in one logical direction only.

Figure 3.6 illustrates the concept enunciated in the previous paragraph. We observe a single disk track divided in numbered physical disk blocks represented by bold letters, where X is the root of the r-tree. Each block stores the logical representation of its children hyper-rectangles, shown in italics. The offset of the first child is shown as an underscore. For example, in block A, the first child is C, found at offset 3. The position of the siblings of C is then easily calculated as: $E_{off} = 3 + 2 = 5$. See figure 3.7 for the hierarchical representation.

3.6.2 Query Resolution Algorithm

Algorthm 3 shows the query procesing method over compressed data, based on the Linear BFS technique. The input is a user query q, consisting of ranges in the cuboid dimensions. Note that although q is defined in terms of range queries, a point query is also possible by limiting all intervals to a single value. Also required are the cuboid C' and index I, compressed with HDC and HRC respectively.

Although logically equivalent, for the R-tree index we will make a distinction between node and block, in order to better understand the algorithm. Here, a *node* or *hyper-rectangle* refers only to the pair of two opposing vertexes $N = (V_{min}\ , V_{max})$, while a *block* refers to the physical portion of data, equal in size to a disk page.

The following procedure is applied to retrieve the results for the query q. The first block of I is read. This block corresponds to the root node N of the packed R-tree. A task is then created as a pair $t = (N, o)$, where o is the block offset in I of the first child of N. The created task t is pushed on the queue T. Every task in the queue corresponds to a child node to be analyzed. Only children who satisfy the user query are included in the queue.

Each task t is processed in a First-In First-Out (FIFO) manner, until the queue T is empty. If the node N, part of task t, is an index node (i.e. not a leaf node), a seek operation is performed in I to retrieve its corresponding block B at offset o. Recall that the block includes the compressed hyper-rectangle representations of all the children of N. The block B is then uncompressed as explained in section 3.5.2. Next, all the children of N are matched against the query q. If q intersects the hyper-rectangle of child N^i, then a new task is pushed at the end of the queue T. This task includes N^i and the corresponding block offset, calculated by adding the offset stored in B plus the position i of this child among its siblings.

In case the node N is a leaf node, the offset o points to a block in the compressed data file C'. This block is uncompressed, as indicated in section 3.3.2, and all the

Algorithm 3 Query Resolution

Input: A user query q. A compressed cuboid C' and its corresponding compressed R-tree index I. A FIFO task queue T.

Output: A set of tuples R in uncompressed form, which satisfy the query ranges.

```
1: read first node $N$ from $I$ (root node)
2: create task $t = (N, o)$ with node and offset
3: push task $t$ on queue $T$
4: while $T$ is not empty do
5:    pop next task $t = (N, o)$ from $T$
6:    if $N$ is not a leaf node then
7:       seek offset $o$ in $I$ and load into $B$
8:       uncompress $B$ as explained in section 3.5.2
9:       for all children $N^i$ in $B$ do
10:         if $N^i$ satisfies the query $q$ then
11:            push task $t = (N^i, o^i)$ on $T$
12:         end if
13:      end for
14:   else
15:      seek offset $o$ in $C'$ and load into $B$
16:      uncompress $B$
17:      for all points $p$ in $B$ do
18:         if $p$ is inside query $q$ then
19:            write $p$ to result set $R$
20:         end if
21:      end for
22:   end if
23: end while
24: return result set $R$
```

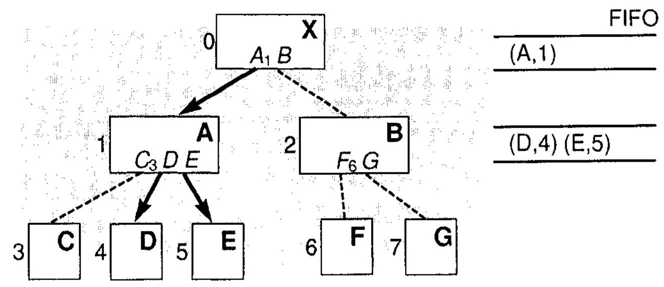


Figure 3.7: Linear BFS Query resolution

points p are compared against the query. Those points falling inside the query are written to the result set R.

When all tasks in T have been processed, the results are returned to be presented to the user, or to possibly undergo further OLAP processing.

Figure 3.7 shows an example of the query search. The process starts by decompressing the root block X. From its children, A and B, only A satisfies the query, thus it is pushed on the queue. The offset 1 allows the retrieval and subsequent decompression of block A, which yields hyper-rectangles C,D and E. From this set, D and E match the query, therefore, two tasks are pushed on the queue. Notice the offsets 4 and 5, calculated from from the offset $C_{off} = 3$. This process continues until reaching the leaf nodes, which are in turn decompressed, and their included tuples returned. Note that by using the offsets in the queue, the linear BFS visits only the matching nodes, marked with bold arrows.

3.6.3 Class Hierarchy

The UML diagram in figure 3.8 depicts the classes involved in the resolution of a query over compressed indexes and data. The package is an extension of the Sidera Query Engine. The main class derives from the original R-tree handler. To interact with the compressed tree nodes it imports a class from the indexing package, while HDC compressed data is managed by an embedded block decoding class. Other significant

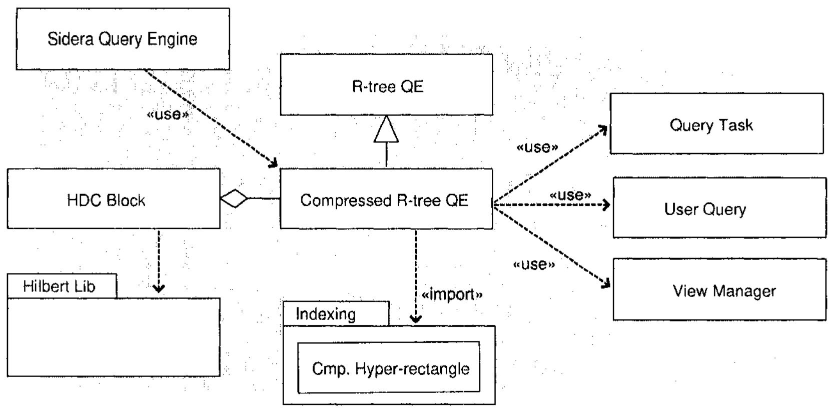


Figure 3.8: Query Engine UML Class Diagram

classes are the Query Task to be pushed on the queue, an abstraction of the User Query, and the View Manager to handle attribute order conversions to match the requested format.

3.7 Scalability

One of the objectives of our framework is to allow the processing of arbitrarily large data sets. This is made possible by a *streaming compression* model, where buffering objects are used througout the process to insure that the available memory in the system is never exhausted. To better understand the approach, we present an activity diagram in figure 3.9, which shows the sequence of actions or stages during the streaming compression. These stages are based on a pipeline producer/consumer model, were each stage produces a set of intermediate results that are consumed in the next step. In the diagram, only a few representative products are shown.

In addition to guaranteeing a controlled memory management, the framework must also also be able to efficiently sort the large sets containing the Hilbert ordinals. Both concepts, buffering and external sorting, are presented below in greater detail.

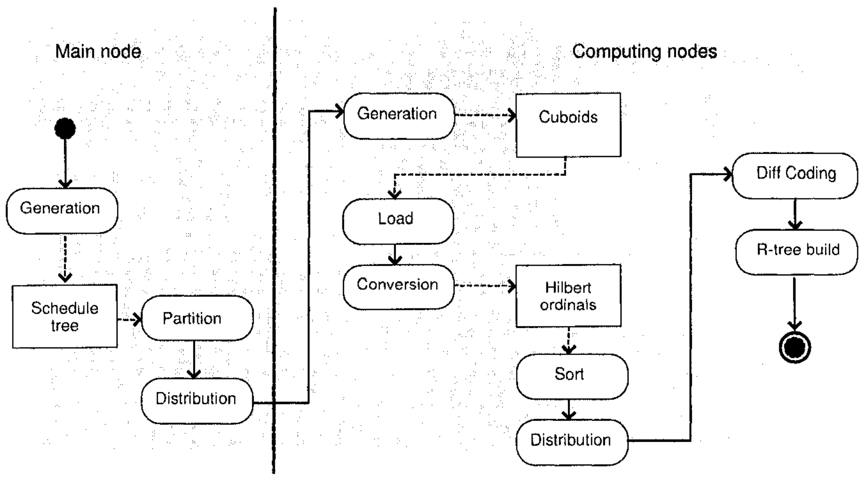


Figure 3.9: Streaming Compression UML Activity Diagram

3.7.1 Buffering

The UML class diagram for the buffering sub-system is depicted in figure 3.10. Note that an abstract class and a template have been used to maximize extensibility and reuse. The relations between the classes and the pipeline stages are described below.

- *Raw data buffer:* Used after the cuboid generation to load the raw multi-dimensional data, and to convert it into Hilbert ordinals.
- *Merge buffer:* Active during the merge phase of the secondary memory sort. It manages a set of non-contiguous data sections from each already quick-sorted partition, and allows merging the data into a single set. Although similar in some aspects to the Hilbert buffer, it stands outside the class hierarchy because it is the only multiple-buffer manager.
- *Hilbert buffer:* This is the single most important buffering class, used in practically all parts of the pipeline dealing with Hilbert ordinals; including: conversion, secondary memory sorting, compression and decompression.
- *Aligned Hilbert buffer:* A modified version of the Hilbert buffer class, used during the data distribution stage among computing nodes. It aligns Hilbert

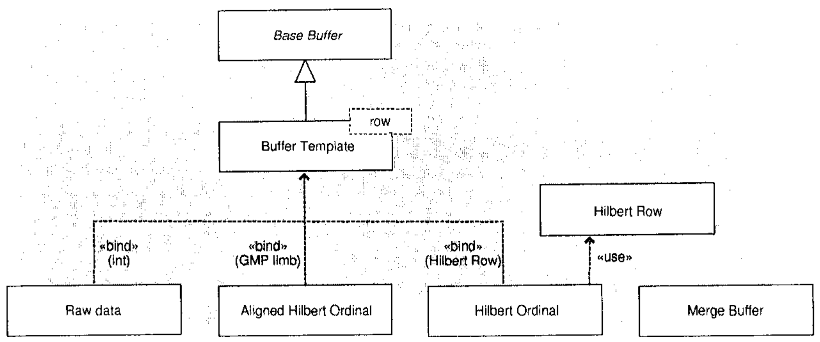


Figure 3.10: Buffering UML Class Diagram

ordinals into fixed size slots in order to efficiently transmit them using the Message Passing Interface (MPI) specification [MPI-Forum, 1994]. MPI is the de-facto standard for process communication in parallel applications.

The buffering objects are specialized to handle the particular data types and stage requirements; nevertheless they share a straightforward common buffering technique. First, a user-configurable amount of memory is allocated and initialized. Then, s bytes are read from the current working data file, where s is the alloted buffer size. When the current executing module requests data, it is immediately returned if found inside the current buffer; otherwise, the loaded data is either written back to disk or discarded, and the next set of s bytes is read from the data file. More complex page management techniques are not required because the buffered stages process the data sequentially; however, the method still allows for basic random access.

3.7.2 External Hilbert Sort

The buffering sub-system guarantees a controlled use of the available system memory. However, an additional condition is required for the pipeline to be able to process arbitrarily large data sets. The input data must be sorted in Hilbert order before the differential encoding and bit compaction can be applied. However, the size of the data set negates the use of a straightforward in-memory sort and favors an external memory algorithm.

The external memory module employs a standard *P-way* external sort [Knuth, 1998]. This algorithm first divides the data in P partitions, where $P = \lceil n/m \rceil$; n is the total size of the input data set, and m is the available memory. Each partition P is sorted by using a standard quicksort and written to external memory. Once all the input file has been processed, the partitions must be merged in a single ordered set. For this task, B buffers are allocated, where $B = \lceil m/(P+1) \rceil$. One buffer is loaded from each one of the P partitions. A standard merge sort is then performed, the results are written to the remaining buffer and subsequently to the final sorted disk file. Both quicksort and mergesort are $O(n\,log(n))$ bounded algorithms in the average case; therefore, the external sort algorithm can also be bound in average by $O(n\,log(n))$.

On average, we expect to perform $O(log(n))$ comparisons for each of the n Hilbert ordinals obtained from the conversion of multi-dimensional tuples. The point at which the conversion should be performed is open to two possible options, compared in section 4.3.5. The first one would convert the multi-dimensional tuples into Hilbert ordinals for each comparison of the sort. This naïve option is highly inefficient because it redundantly repeats the required conversions. Therefore, our approach in the streaming pipeline is to convert all multi-dimensional tuples to Hilbert ordinals first, and only then perform the external sort, using the supporting buffering objects, as shown in figure 3.9.

The streaming compression including the buffering sub-system and the external memory sort allows our framework to process arbitrarily large data sets, with sizes only limited by the underlying file system. At the same time, it minimizes the I/O, while remaining transparent to the calling routines.

3.8 Parallelization

One of the main reasons for choosing a distributed system is to sub-divide the workload to efficiently balance the use of the available processing power [Ricardo, 2004]. In our framework, load balancing is particularly important for computationally intensive tasks, such as the data compression stage, and for time-critical tasks, such as the query resolution process. The framework is part of the OLAP Sidera server, which is specially designed for working in a distributed environment. This project is the work of Eavis [2003], and is based on the Relational OLAP paradigm.

3.8.1 Distributed compression

Two of the main features of the Sidera server are: the efficient generation of full and partial data cubes, and their associated multi-dimensional indexing. In order to compute a data cube, the master node calculates the cheapest way of computing each of the 2^d views and builds a schedule tree, which is then divided in p equally weighted sub-trees. These sub-plans are sent to each of the p parallel processing nodes, where the cuboids C_p indicated in the sub-tree schedule are computed.

In this parallel environment, the HDC compression algorithm works in two separate stages. The first stage starts once the C_p cuboids have been materialized in each computing node. It includes only the calculation of the Hilbert ordinal for each tuple, and their subsequent sorting. This computing intensive task is well balanced among the p nodes thanks to the equally-weighted partitions of the original schedule tree.

With all tuples in Hilbert order, the Sidera server starts a data distribution task. Every cuboid is striped across all available nodes, using a round robin schema. Each tuple, in Hilbert ordinal format, is sent to the next node. A single processor P_i receives the set of tuples $\{t_i, t_{i+p}, t_{i+2p}...\}$, until all tuples in the current cuboid have been distributed. Because Hilbert ordinals may span beyond standard register bit sizes, a special buffering schema was developed, which is presented in section 3.7.1.

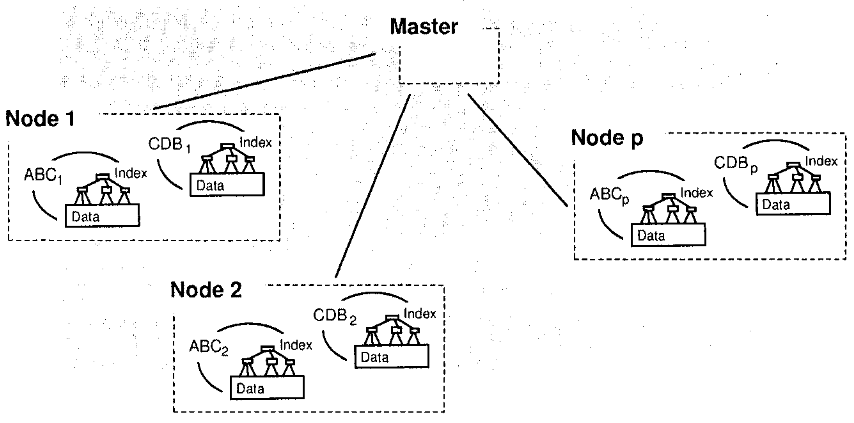


Figure 3.11: Distributed compressed cube

After the distribution, each P_i node holds a striped partition from every materialized cuboid. Here, the second stage of the HDC algorithm takes place. These *partial cuboids* are already sets of ordered Hilbert values, therefore algorithm 1 continues by calculating the differentials and writing them to fully-packed disk blocks in bit-compacted representation. Once the data in the partial cuboids is processed, each computing node builds the corresponding compressed R-tree index, applying algorithm 2.

Figure 3.11 shows a representation of the distributed data cube after building the indexes. Two cuboids, ABC and CDB, are striped across the p processing nodes. Each node computes a compressed R-tree over a distinct set of HDC compressed data. The front end node acts only as a distributor of schedule-tree partitions and user queries. All process-intensive tasks are balanced among the computing nodes.

3.8.2 Distributed Query Resolution

The Hilbert Space-Filling Curve provides superior clustering properties, which are exploited by our compression framework. The same properties play an important role in the round robin method for striping data across processing nodes. Eavis [2003] demonstrated that this striping pattern over a Hilbert curve, increases the likelihood of having a homogeneous distribution of points falling inside the query hyper-rectangle, over all the processing nodes.

Answering queries in a parallel environment is handled using a *front-controller* paradigm, implemented in the query resolution module of Sidera. Initially, the query q is broadcast form the controller main node to all p processors. Because the cuboid C might have an attribute order different from that of the query, q is transformed in q', an equivalent query following the actual physical order of the cuboid C. Then, q' is passed to the query engine in the current node, returning the result set R_p. The attributes of R_p are sorted to have the order of the original query q. A parallel sample sort of R_p is performed across the processors to produce the order specified by the user. Finally all R_p sets are returned to the main node, and unified. Further details lie outside the scope of this thesis, and can be found in the original work by Eavis.

3.9 Conclusions

The compression framework presented in this chapter offers an efficient platform for handling vast amounts of OLAP data. We have introduced a new method for compressing multi-dimensional data using the space-filling Hilbert curve. When combined with differential encoding and bit-compaction, this method yields state-of-the art compression ratios above 80%. Moreover, it preserves the access granularity at block level, hence optimizing I/O operations. We have contrasted our method with TDC, one of the most influential works in database compression, highlighting how our approach addresses the limitations found in TDC by providing integrated indexing, unlimited scalability and superior clustering.

Our indexing solution, built on top of the compressed data, obtains extremely high compression ratios of up to 98%. In order to generate the index we have combined the ideas behind packed R-tree structures, with differential encoding, obtaining a very low footprint and efficient index. Our approach overcomes some of the problems encountered in GRS, an important previously published method. Specifically, it eliminates the need for an initial block estimation, and provides a solid linearization by means of the underlying Hilbert curve. We have also presented our query engine, which has beeen specially designed to work with compressed data and indexes, and can be used in a distributed environment.

Two other very important notions introduced in this chapter are scalability and paralellization. The first one allows the framework to accept a data set with no limitations other than the underlying file system, for the row, cardinalities and dimension counts. This is achieved by a combination of the buffering sub-system, the external p-way sort algorithm, and a judicious use of the GMP library to handle multi-precision arithmetic. The second concept, parallelization, uses the existing Sidera facilities to provide distributed compression and index generation, and permits balancing the computing load among nodes to process user queries.

Chapter 4

Evaluation

4.1 Introduction

In the previous chapter we introduced the components of our compression framework for OLAP. Specifically, we presented HDC, our Hilbert-based data encoding method, which is seamlessly integrated with HRC, our algorithm to generate compressed R-tree indexes. We have also explained the distributed query engine, and the scalability and parallelization advantages included in our approach. The next natural step is to obtain quantitative references for the performance of the methods and comparative measurements with other related algorithms.

In the present chapter, we evaluate the performance of several elements of our framework. We start by defining the test environment, in section 4.2, specifying the precautions adopted to guarantee a fair comparison with other methods. This includes the software and hardware platforms and the methodologies used to generate the data and to perform the tests. Next, in section 4.3.1 we show the compression results for the HDC method when tested against several data configurations. Then, several comparisons are carried out to contrast our results with those obtained by TDC, GRS and the Naïve method. In sections 4.3.2 and 4.3.3 we focus on index compression and query resolution experiments, complemented with an additional comparison against an influential but non-relational method called Dwarf, in section 4.3.4.

In the later sections, starting with 4.3.5 we provide indicators on the optimizations performed to the framework regarding the External Hilbert sort, the GMP library and the query response time. Additionally, section 4.3.6 presents the load imbalance percentage when a query is executed in a distributed environmet. Finally, in section 4.3.7 we offer statistics on the processed Hilbert differences, the deep constitutive elements of the compressed data blocks. We expect these closing statistics to be useful for future developments.

4.2 Test Environment

The first step in the evaluation process of our framework is the defintion of the *test environment.* The environment includes the software and hardware characteristics that support the correct development and testing of all the implemented components. It also indicates the particular features of the data sets used during the experiments and the procedures applied to run such tests. A clear description of each of these elements is provided in order to facilitate the verification and repeatability of the presented results.

A uniform and well-defined environment is of great importance because it establishes a single common platform to be used during the comparison rounds. The platform does not favor any particular method and therefore constitutes a neutral ground for a fair benchmarking effort. In addition, the methods used as comparison benchmarks have been developed to comply as close as possible with the decriptions in the original works.

4.2.1 Software

The implementation of the compression framework, methods used for comparison, and supporting tools were written in the C++ programming lanuguage [Stroustrup, 1997]. We chose C++ for various reasons. First, to easily integrate our code with the existing Sidera modules. Second, to obtain all the benefits of a widely used, high-level and flexible language without compromising performance. Third, to be able to use the object oriented concepts to build a robust and modularized solution, where classes can be extended and reused. Finally, to incorporate some widely-available C++ libraries, used to support specific tasks, as indicated in the following paragraphs.

We lightly use the C++ Standard Template Library (STL), originally proposed by Stepanov and Lee [1995] and developed at SGI. STL is an extension of the C++ Standard Library, providing a set of routines, classes and iterators, which implement basic data structures and algorithms of Computer Science. The library is fully parametrized by using templates, hence the name, meaning that the components adapt to any required data type. In our implementation, we profit from the container classes, specially the *deque*, the *vector* and the *map*.

The Sidera server is designed to function in a distributed environment. The communication between the nodes of this parallel application is achieved by using the Message Passing Interface (MPI) de-facto industry standard [MPI-Forum, 1994]. Specfically, the Local Area Multicomputer (LAM) implementation of MPI was chosen because it is free, comprehensive and open source [LAM]. Additionally, the Sidera modules require the graph classes from the Library of Effcient Data Structures and Algorithms [LEDA] packages in order to manipulate the cube lattice structure. Finally, our framework uses the GMP library to handle arbitrarily large Hilbert ordinals as explained in section 3.3.5.

For the development of the framework we use simple but powerful tools. These include the excellent VI IMproved (VIM) editor [VIM], Doxygen [van Heesch] for generating the standard documentation and the *make* utility [Feldman, 1979] for execution of complex tasks. This last tool allows the execution of compilations, code synchronization, document generation and experimental runs, with a single command.

The following is a list of the primary software elements of the environment:

- Linux Fedora Core 5, kernel version 2.6.17-1.2157

- GNU Compiler Collection (GCC), C++ compiler, version 4.1.1
- C++ Standard Library and STL, version 3.4 (included in GCC)
- LAM/ MPI, version 7.1.2
- Library of Effcient Data Structures and Algorithms (LEDA), version 5.1
- VIM editor, version 7.0.42
- GNU make, version 3.8
- Doxygen, version 1.4.6

4.2.2 Hardware

The development and testing of the framework is performed in a single workstation with the following configuration:

- Dell Precision WorkStation 380
- Motherboard Dell 0CJ774
- Intel Pentium 4 Processor @ 3.2 GHz
- L1 Cache 16 KB, L2 Cache 2048 KB
- DDR Memory 1 GB @ 667 MHz
- Western Digital 160 GB SATA Hard Drive, model WD1600JS.
  Linux partition is 80 GB and swap partition is 2 GB.
- Broadcom BCM5751 NetXtreme Gigabit Ethernet Controller

Although all modules of the Sidera server are designed for a parallel configuration, we did not have the resources to test our framework in such an environment. However, the compression ratios will not vary from the ones presented in this thesis, because cuboid partitions and their corresponding indexes are processed independently in each node. Moreover, compression and query processing times will be improved due to the increased computing power and the balanced load distribution, as explained in section 4.3.6. The installation of a new 16-node cluster is currently in its final stage to allow such tests.

4.2.3 Methodologies

Engineering

Our compression framework was engineered to be a robust high-quality sub-system of the Sidera Server. The concepts behind the Agile Software Development Manifesto [Fowler and Highsmith, 2001] were applied throughout the life-cycle of our project. These concepts facilitate the production of software in a fast and adaptive way, being ready to respond to changes and placing people interaction over tools and excessive documentation.

Agile divides the development in interation cycles, where each iteration is a miniature project of its own. At the end of a cycle an incremental update is realeased, and the priorities are re-evaluated. Our iterations were influenced by the Agile Unified Process (AUP) [Ambler and Jeffries, 2002], which identifies a set of different stages to be performed. Specifically, the modelling stage produces light analysis and design documentation, as shown in previous UML diagrams. Implementation and testing is done in parallel, with code being heavily annotated for clarity and automated document generation. Management was performed by regular meetings and result evaluations.

Data generation

In order to have an effective evaluation of our framework, we performed several tests over data with a broad range of characteristics. These data features follow those likely to be encountered in practical settings. A very flexible tool to obtain such data is the *data generator* implemented by Eavis [2003]. This utility produces "raw-data" files containing records composed entirely of integer values. This is typical of an OLAP application, where all attributes would be previously mapped to integers. Most importantly, all features of the produced data are easily configured, including the number of rows, dimensions, cardinality of each dimension and skew. The skew parameter is used to generate non-uniformly distributed data, using the Zipf power-law [Zipf, 1935].

An experimental *data set* is defined as a group of generated files. Standard values are specified for: row count, dimension count, cardinalities and skew. One file in the data set is created with the standard values, while the rest are generated by varying one parameter at a time within a given range.

Although our data sets are synthetically created, they model the features of a real OLAP data set. Of course there is no absolute rule to define a "real" set, but there exists a consensus over the general characteristics it should have. This means high row counts, from millions of records up to billions [Auerbach, 2006], less than 15 dimensions [Kimball and Ross, 2002], a set of mixed cardinalities with few high values greater than 10^4, and a non-uniform distribution of points [Eavis, 2003]. The data sets used in the experiments highlight various aspects of the methods being tested, while remaining close to the desirable "real" features, given the limitations on resources.

Experimentation

When running compression experiments and comparisons, the following process is applied. The raw-data files corresponding to the required tables are copied to a given folder. Then, the component being tested is run with this data as input. It analyzes and processes one file at a time and produces a compressed output file. Relevant measurements are taken at this point and recorded, most commonly internal functioning statistics, and the compression ratio. A test is performed once for each cuboid. All input files are processed sequentially in an automatized batch task, using the *make* utility.

A small in-house tool was implemented to calculate compression ratios. The tool simply compares the input and output files and expresses the ratio as: $\%c = 100(1 - O/I)$, where O is the size of the resulting compressed file and I is the size of the input file. Therefore, the numbers being shown represent the effective size reduction, expressed as a percentage of the original file. With this method, higher numbers mean better compression. For example, if data set A is compressed 80% into file B, it means that B is 20% the size of A. Note that measure attributes, defined in section 2.4.1, are excluded from the calculation. This is done in order to produce more intuitive results, because only the processed parts of the data are compared. Measure values are not compressed by any of the methods presented in this thesis.

For query experiments, we must consider that the resolution time of a single query is usually very short, generally in the sub-second range. Comparisons at this scale are therefore hard to interpret. Thus, query tests consist of a batch of 1000 random queries, as opposed to a single one. For this purpose we use a tool developed by Eavis [2003], which we refer to as the *query generator*. This utility produces the query batch after randomly selecting the view to search, the order of its attributes and the range for each attribute.

4.3 Results

In the current section we present the results of the evaluation of our framework. The effect of the variation of the data features on the data and index compression is analyzed. We also perform comparisons with other methods and provide query resolution observations.

4.3.1 Data Compression

We introduce the evaluation by presenting the compression ratios obtained with our HDC method. The standard values for the data set are: 10 dimensions, 1 million rows, cardinality of 100 for each dimension and no skew. This set is used in several other experiments, thus we will refer to it as the *standard set A*. One data parameter is varied at a time while the others remain constant.

The first general observation when analyzing the charts in figure 4.1 is an average ratio above 80%. This is an outstanding number for HDC when compared with other methods, as will be demonstrated in subsequent sections.

Figure 4.1a shows the variation of the compression ratio when the number of rows in the cuboid is raised from 100,000 to 20 million. We observe an increment in the compression performance from 81.85% to 84.64%. By keeping the number of dimensions and cardinalities at fixed values, the size of the cuboid space remains constant. At the same time the number of data points inside this space is increased, driving the points closer together. Our HDC method profits from this distance reduction between points by storing a higher number of smaller Hilbert differences in each block.

We will use the concept of point *density* to describe the amount of points inside a delimited region of space. In the last paragraph for example, we described a density increase. Note that this closely resembles the chemical density law for ideal gases, where $d = m/V$. Here, the density d increases if the volume V is kept constant while incrementing the mass m.

The opposite takes place in figure 4.1b. Point density is lowered because the space grows in size with the increase in dimension count from 2 to 15, while the number of points remains fixed. Therefore, the distance between points along the Hilbert curve is incremented. In this scenario, the HDC algorithm requires more bits to store the differences in each block, producing a decrease in compression. Nevertheless, our method still produces very high compression ratios in the range 97.94% to 81.35%.

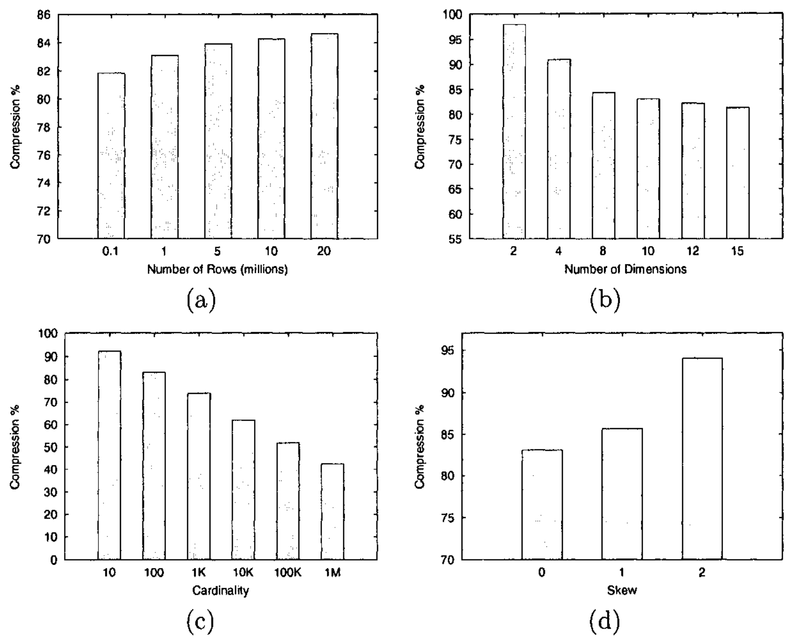


Figure 4.1: HDC Data Compression

A similar but more pronounced behaviour is shown in figure 4.1c. Here we vary the cardinality of each dimension from 10 to 1 million. Recall that the Hilbert curve visits every point in space, therefore the length of the curve, i.e. the last visited point, is given by $h = s^d$, where s is the space side length, and d is the number of dimensions (see section 3.3). In this graph, the left hand side is more representative, however it is interesting to see the behaviour of the method with high cardinalities. For example, with a drastically unrealistic uniform cardinality of 1 million, $h = (10^6)^{10} = 10^{60}$, a number requiring 200 bits to be represented. Compression ratios drop accordingly to 42.5%. Fortunately, as we have metioned in section 4.2.3 real settings usually have cardinality values no greater than $10,000$ for most dimensions.

Finally, we tested HDC with three skew values. A value of 0 is a random distribution of points in the space, while 2 means high amounts of clustering, with points close to each other in some regions. Naturally, higher clustering means smaller distances along the Hilbert curve, and thus imply better compression ratios of up to 94%.

HDC vs TDC

The first comparison round for HDC is naturally against its closest sibling, the TDC method introduced in section 3.2.2. For this experiment we have implemented a TDC compression module adhering to the original specification by Ng and Ravishankar [1997]. To make it even more interesting, we will use a data set following the guidelines proposed by the aforementioned authors. In this set, the number of dimensions is fixed at 8, while the number of rows assumes the values 10^4, 10^5 and 10^6. The tests are defined by the characteristics of the cardinalities and skew, as shown in table 4.1. Cardinality has a *low variance* when the difference between domain sizes is less than 10% of the average, and it a *high variance* when the difference is more than 100%.

In all four tests, both methods obtain ratios in excess of 80%, as shown in figure 4.2. Although TDC obtains slightly higher measures, the difference is never greater than 3%. This variation is due to the fact that the Hilbert axes must be rounded to the next "power of two", sacrificing a small degree of compression.

Both methods respond positively to an increase in the row count, with HDC reducing the existing difference and surpassing TDC in the 1 million row experiments in figures 4.2a and 4.2c. This is attributed to the superior clustering properites of the Hilbert curve, which are more evident in dense spaces.

We also observe that skew, present in figures 4.2a and 4.2b yields higher compression ratios, close to 90%. The differential nature of the methods profit from the localized clustering produced by the skew by storing more differences per block. On the other hand, cardinalities with high variance as shown in figure 4.2d decrease the ratio for HDC, because the method requires taking the value of the highest cardinality

Test	A	B	C	D
Skew	1	1	0	0
Cardinality variance	Small	Large	Small	Large

Table 4.1: TDC-Defined Tests

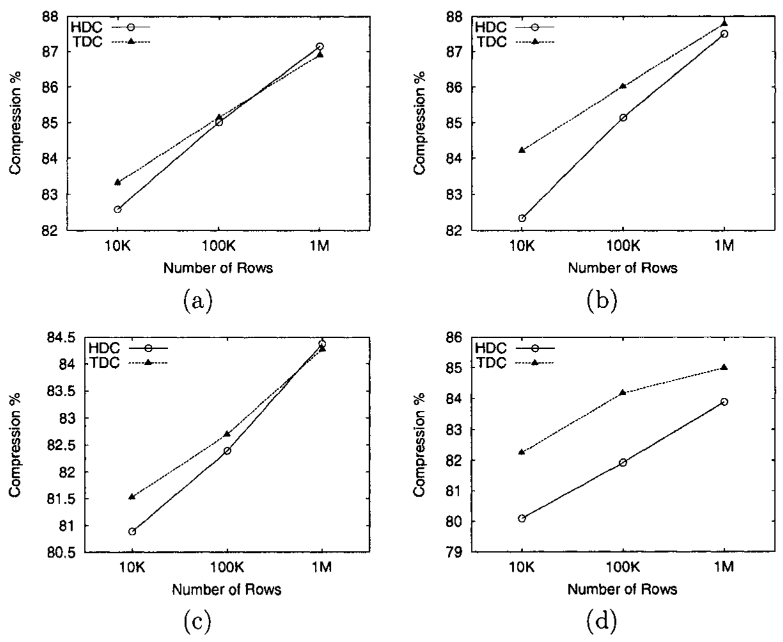


Figure 4.2: HDC vs TDC - Data compression

to calculate the Hilbert ordinals.

Finally, it is important to mention that both methods are based on Differential Encoding. However, for differentials to be effective, they require bit compaction to fully exploit their potential. Figure 4.3 shows a comparison between TDC and pure Differential Encoding with varying row-count on the standard data set A. While TDC surpasses the 80% ratio, differential encoding with no bit-compaction obtains 20%. This value results from the differences being smaller than the original $\varphi(t)$ ordinals (see section 3.2.2).

Four methods comparison

This experiment adds two more methods to the data compression comparison. The first one, GRS, is an implementation of the method proposed by Goldstein et al. [1998], discussed in section 3.4.1. The second one is the product of an observation by Eavis [2003], applicable to OLAP environments. This remark implies that by simply bit compacting each field of a cuboid we could obtain reasonable levels of compression.

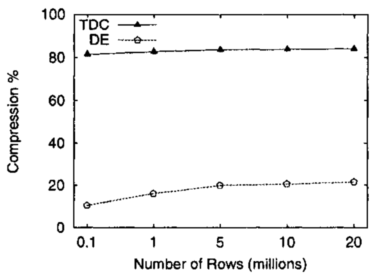


Figure 4.3: TDC vs Differential Encoding

We refer to this as the *Naïve* method.

The standard values for the data set are: 6 dimensions, 1 million rows, a skew value of 1 and a constant group of mixed cardinalities taken from the range [4, 1000]. We will refer to this set as the *standard set B.*

All the charts in figure 4.4 show HDC and TDC with the highest compression ratios, consistent with the observations in the previous experiment. GRS obtains lower values, a result associated with the requirement of estimating the number of compressed blocks beforehand, assumming a uniform distribution of points. On the other hand, the Naïve method obtains a repectable and constant 78.125%, as shown in figure 4.4a. The method is unaffected by an increase in dimensions because each extra attribute will have the same cardinality as the existing ones, and therefore will be compressed at exacly the same ratio. Moreover, note that in very high dimensions, the Naïve method will eventually catch up with the differential ones. This is easily explained because as the size of the space increases, distances between points also increase until they are too far apart for the differentials to have any impact on compression.

In figure 4.4b we examine the effect of setting a skew value of three, producing a huge amount of clustering. We have mentioned that non-uniformity is characteristic of real data sets, and also that the differential methods are well suited for this cases because differences are small in the dense regions of space. The chart clearly ratifies this fact, where HDC and TDC boost their ratios over 95%, and the advantage over

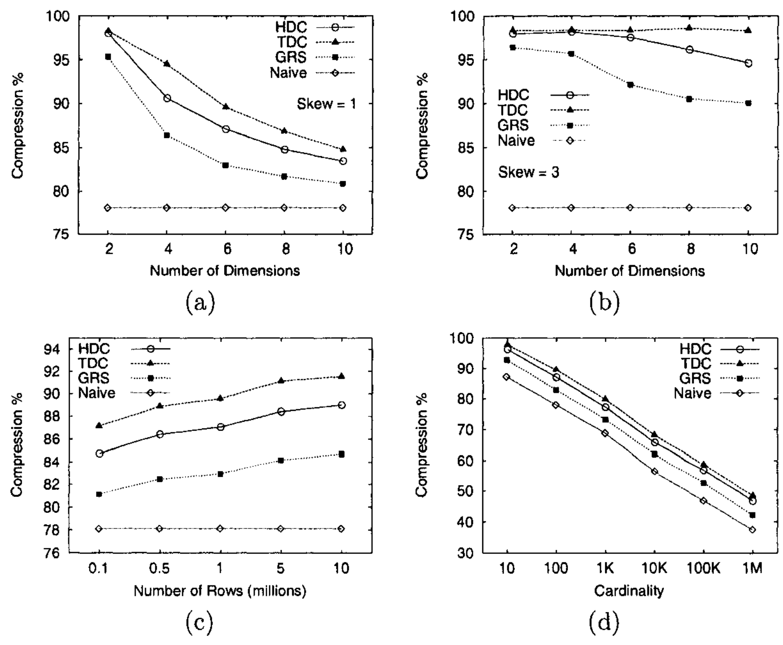


Figure 4.4: Four methods comparison

the Naïve method becomes very pronounced.

The figures 4.4c and 4.4d confirm the trends observed in previous tests. An increase in row count produces high-density spaces, which translate into better compression ratios, while an increase in cardinality lowers the density and ratios by increasing the space size. Note that although the definition of GRS stated in section 3.4.1 seems dissimilar to the other methods, it is intrinsically based on differential encoding and therefore follows the behaviour of its differential siblings. On the other hand, the Naïve method is affected only by cardinality variations because it depends exclusively on bit compaction.

In general, the observed trends are quite encouraging, as our data compression method obtains state-of-the-art ratios when tested against the desired features of a realistic data set. As described in section 4.2.3, these features include: high row counts, less than 15 dimensions, a moderate cardinality average, and non-uniform point distributions.

4.3.2 Index Compression

Previous experiments have shown excellent compression ratios for our HDC algorithm. The ratios are very close to those for TDC, but we overcome a major limitation of this method by providing an integrated compressed indexing method. As we have stated before, compressed data is not useful without effective means for access and search. The current experiment focuses on our HRC index creation and compression method. The data set used is the standard set B.

The numbers obtained exceeded our expectations, and the majority of results surpassed the 95% mark. Figure 4.5 shows the variation of the index compression ratio when varying the data set parameters. In figure 4.5a the record count increases from 100K to 10 million, also increasing the index compression ratio from 94.90% to 98.98%. The underlying data set density allows the index hyper-rectangles to enclose a higher number of points, therefore requiring less nodes for the R-tree.

In figure 4.5b, index compression slightly increases from 96.63% to 98.68% when the dimension count is raised. Recall that in HDC, differentials are calculated between each point and the pivot vertex, in a column-wise manner. If we increment the dimensions, these per-column differentials do not increase in size, because the cardinalities of the new attributes are in the same range as the existing ones. This behaviour can be contrasted with the Hilbert ordinals and differentials, whose size increases with a dimension increase. The small improvement in the compression ratio can be attributed to gains obtained from not storing the pivot vertexes in the current block, which becomes more evident when points have more attributes (see the last paragraph in section 3.5.1).

Figures 4.5c and 4.5d show the effect of cardinality and skew, similar to that observed with data compression. The main difference is the higher ratios over 90%, even for high cardinalities. Skew clustering produces extremely high compression measures of up to 99.29%, for a skew value of 2.

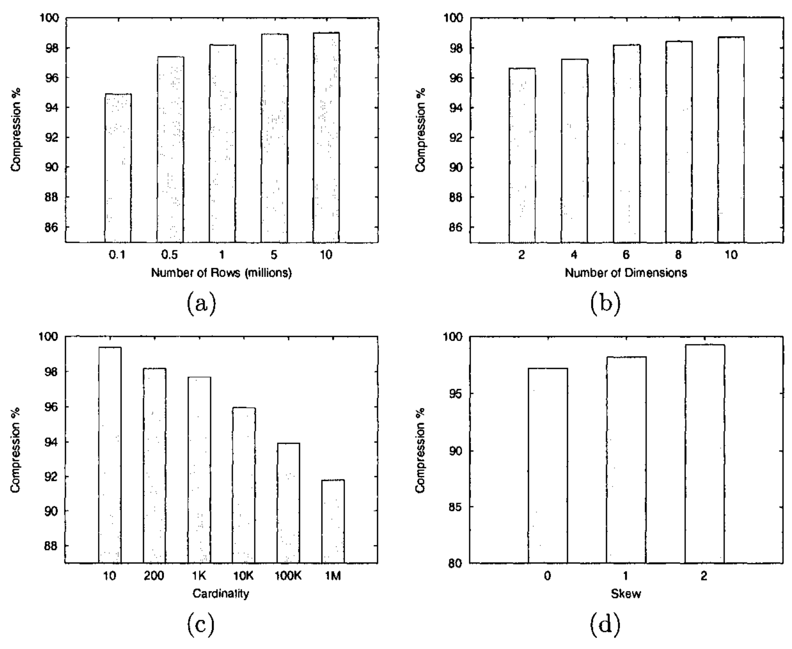


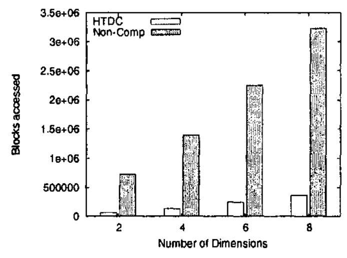
Figure 4.5: Index compression

In order to test the benefits of the scalability provided by the framework, we performed an additional test augmenting the row count to 100 million. The uncompressed cuboid occupied approximately 1.6 GB of disk space. After processing, the data set was reduced to just 58 MB corresponding to an impressive ratio of 96.36 %. The associated R-tree index occupied a tiny 553 KB.

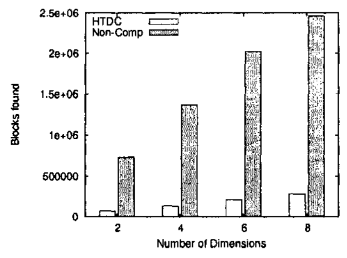
4.3.3 Query Resolution

After obtaining high index compression results of above 90% for all tests, we analyze the performance characteristics of those indexes. For this purpose, we experiment with standard set B, using batches of queries as explained in section 4.2.3.

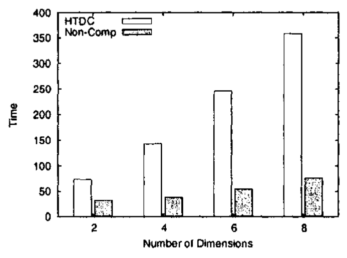
Figure 4.6a compares the number of blocks accessed when executing the batch of queries on the compressed indexes and data versus the same execution on uncompressed files. We observe that in order to resolve the query batch, the uncompressed version requires the retrieval of 10 times more blocks than the compressed one. This factor remains roughly constant when dimension count is increased. This alone is a



(a) Blocks accessed



(b) Blocks found



(c) I/O

Figure 4.6: Query Resolution

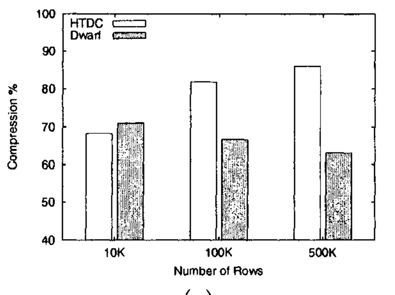
gain of one order of magnitude in terms of disk block accesses.

A slight variation, presented in figure 4.6b, counts the number of blocks which effectively lead to tuples that were included in the final result. Recall that the R-tree structure requires the traversal of several branches when the search includes an ovelapping area. The result is simliar to the previous one, where the compressed structures requires 10 times less blocks than their uncompressed counterparts.

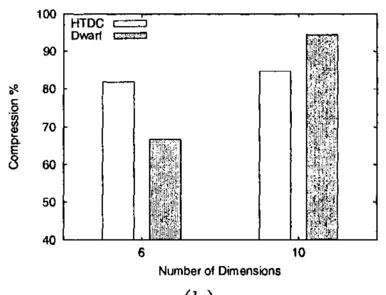
Finally, in figure 4.6c we compare the response times for the query resolution. Times are recorded from the moment the batch is handed to the query engine until the complete result set is gathered. The algorithm over compressed data and indexes requires additional processing time, in a factor of 2 for low dimensions and 4 for high dimensions. We have performed extensive optimizations in order to reduce these ratios, as explained in section 4.3.5. However, further fine-grained work is still possible in order to achieve a threshold not noticeable by end users. We believe this can be obtained with a factor of 2 for high dimensions.

4.3.4 Non-relational comparison

A different method for cube compaction was proposed by Sismanis et al. [2002]. This technique builds a speciallized structure known as a Dwarf which functions as both index and data repository. Compression is achieved by identifying prefix and suffix redundancies and eliminating them by coalescing their storage. Prefix redundancy refers to cuboids with common leading attributes e.g. (ab, abc) and is high



(a)



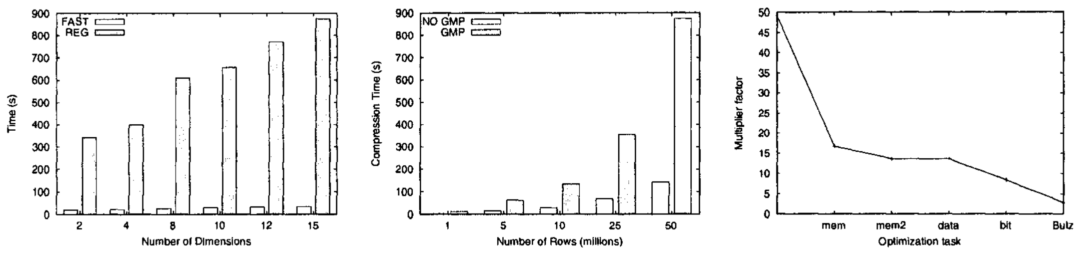
(b)

Figure 4.7: Comparison with Dwarf

on dense areas. Suffix redundancy exists in cuboids with common trailing attributes e.g. (abc, bc) and is high in sparse areas.

Note that Dwarf is not relational in nature, but we consider it as an influentlial yardstick in OLAP compression. Therefore, we have implemented the Dwarf algorithm closely following the description given in the original paper. For the comparative experiments, the data sets are full data cubes generated using the Sidera facilites. The cubes are based on fact tables with 6 dimensions, 100,000 rows, a mixed cardinality with an average value of 200, and skew 1. Note that by generating all cuboids in the data cube, the size of the resulting input set is roughly 20 times the size of the aforementioned original fact table.

Figure 4.7 shows the comparison between our integrated index and data compression methods and Dwarf. In figure 4.7a we observe that our framework clearly surpasess the performance of Dwarf for higher row counts. We can attribute this fact to the clustering properties of the Hilbert curve in dense spaces. Figure 4.7b shows our methods to consistently deliver high compression ratios for cubes with 6 and 10 dimensions. Dwarf scores much lower on the first test, but becomes particularly strong in high dimension environments. However, we note that the high dimensional views are typically the ones that are least useful for the users. This is the case since it is extremely difficult to directly interpret the results of, for example, an 8-dimensional query. So, in effect, the Dwarf cube does a very good job of representing views that



(a) Fast Hilbert Sort (b) GMP overhead (c) Query factor reduction

Figure 4.8: Optimizations

few people actually want.

The trend observed in figure 4.7a raises some questions about the scalability of Dwarf. Moreover, parallelization efforts and external memory implementations could prove specially difficult given the non-relational nature of the algorithm. Some other deficiencies have also been identified by the authors of the original paper, specifically on the resolution of range queries. The structure may span large areas of the disk causing excessive amounts of I/O. A variation of the algorithm was briefly proposed to alleviate this effect. This variation was not implemented because no query comparisons were performed with Dwarf.

4.3.5 Optimizations

In the present section we describe several important optimizations included during the development of the framework:

- *Fast Hilbert Sort:* An important addition to the Sidera Server implementation is the Fast Hilbert Sort, part of the External Sort discussed in section 3.7.2. This concept is applied during the initial ranking of the points in Hilbert order. It implies converting all multi-dimensional tuples to Hilbert ordinals before performing the sort. The Naïve approach performs a "just-in-time" conversion, for each comparison of the sort. This method is highly inefficient as shown in figure 4.8a because it duplicates transformations that have been already done in previous comparisons. This relatively simple improvement speeds up the sorting process by an factor of 23, on average. The test was carried out using the standard set A.

- *GMP overhead minimization:* The compression performed by HDC has been specially designed with scalability in mind. High dimension counts and cardinality values produce space sizes that easily overflow traditional 32-bit registers. Therefore, the GMP library was used to allow operations on arbitrarily large Hilbert ordinals. However, the overhead introduced by the library might not be negligible. To confirm this fact, a non-optimized version of TDC compression was run on a data set with 4 dimensions, cardinality of 100, and skew 0. Note the low values on dimension count and cardinalities to avoid overflowing the standard integer registers for the comparison.

  As expected, the module using GMP functions was 4 times slower on average than the module using standard integer operations, as shown in figure 4.8b. Consequently, several optimizations have been introduced, as explained in section 3.3.5 to minimize or avoid the library overhead.

- *Query response time:* In figure 4.8c we observe a chronology of the optimization of the reponse time when executing query batches. Initially a batch executed on compressed data took 50 times longer than a batch executed on non-compressed data. The two first optimization tasks focused on memory management, especially in the Hilbert transformation functions. By centralizing the allocation of memory and reusing existing buffers we improved the times by a factor of 15. Next, we changed the structure of the compressed files to avoid the use of bit-reading functions on uncompressed parts of the file (e.g. header, measures). Then, we optimized the extensively used bit-reading function to save as many

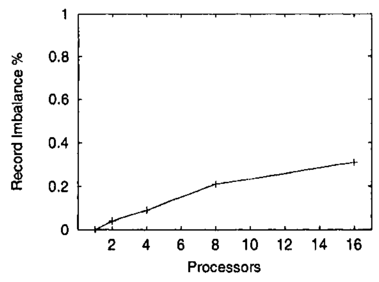


Figure 4.9: Imbalance with respect to number of rows retrieved in each node

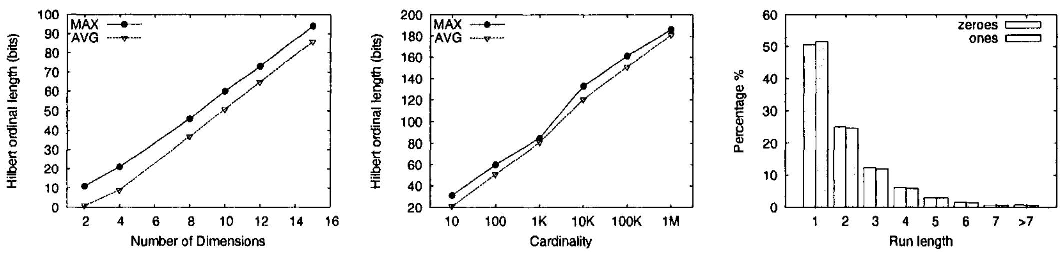
processor cycles as possible, reaching a factor of 8. Finally, the Hilbert transformations were completely rewritten as explained in section 3.3.2 attaining a factor of 2 for low dimensions and 4 for high-dimensions. Further optimizations are still possible, as suggested in section 5.2.

4.3.6 Parallel load balance

Our compression framework benefits from the distributed nature of the Sidera Server. As a constituent module of the server, it is able to perform all compression and query resolution in parallel computing nodes. As explained in section 3.8, the data is striped across the processing nodes following a round robin scheme. The clustering provided by the Hilbert curve produces an almost perfectly balanced environment, where the number of tuples retrieved on each node during a query operation differs by less than 0.3%. Figure 4.9, presented in Eavis [2003], shows the maximum balancing error as a percentage variation between the sizes of the resulting sets returned by each node. The standard set A is used with a batch of 100 queries.

4.3.7 Hilbert Statistics

During the development of the data compression module we collected various statistics from the processed Hilbert differentials. They may be useful for future development and improvement of the methods presented in this thesis. Therefore, in this section we briefly present some of the most relevant measurements, gathered from the processing of the standard set A.



(a) Diff length vs dim. (b) Diff length vs card. (c) Bit runs

Figure 4.10: Hilbert ordinals statistics

Figures 4.10a and 4.10b show the variation of the Hilbert differentials' maximum and average lengths when dimensions and cardinalities are increased. Dimension counts over 6 and fairly low cardinalities over 30 will overflow a standard integer record. A hybrid system using 64-bit registers and GMP could be studied, with the first type supporting up to 10 dimensions, and cardinalities of less than 100. On the other hand, differential lengths remain constant when increasing the row count, with a maximum of 59 and 49 bits on average. The difference between the maximum length and the average length is 9 bits on average, with a standard deviation of 4. The minimum ordinal size is always 1 bit.

Each Hilbert differential is composed of groups of consecutive bits with the same value. These groups are called *runs*, and are formed by one or more bits. The number of bits in a run is known as its length. We observe on figure 4.10c the various average run-lengths computed for the processed Hilbert differentials. Notice that the measures are very similar for 0s and 1s. Runs of 1 bit are predominant with more than 50%, while 2-bit runs account for 25%, and 3-bit runs for aroun 12.32%. The composition of run-lengths is an indicator for the possible application of additional compression techniques, in this case discarding RLE, but inviting to search for patterns in the combination of small runs.

4.4 Conclusions

In this chapter we have presented several evaluation tests to measure the performance of the components of our framework when applied to various data conditions. We have also carried out a number of benchmarks to compare our results with those obtained by several related methods. All these tests were performed in a single non-biased environment, a fact of of great importance because it is the basis to produce a fair comparison effort.

Our framework has been developed adhering to the ideas behind the Agile Manifesto, prioritizing people interaction and rapid evolution response, over specific tools. The C++ language was used, along with several open-source libraries to produce a robust object-oriented implementation with more than 10,000 lines of code, supported by UML diagrams and Doxygen documentation. The implementation of competing methods has closely followed the algorithms described in the original papers.

Initially, we defined the elements that form part of the standard test environment. All hardware and software components are widely available to ensure that the results are easy to reproduce and verify. Data generation is carried out by a very flexible tool, part of the Sidera system, which allows us to prepare cuboids with specific characteristics. This allows us to to analyze the impact of various feature changes on the studied methods. For compression experiments we report the compression ratio with respect to the original file. For query execution experiments we present the response times required to run batches of 1000 queries.

The results obtained are very positive, with an average ratio above 80% for the data compression and over 95% for R-tree index compression. The first one is attributed to the combination of differential encoding, bit-compaction and the Hilbert curve clustering. The second not only profits from these features, but also from a very compact representation where the reference pivot is not even enclosed in the block, but stored in higher levels of the tree, which are also compressed.

We have observed a uniform behaviour, where compression varies according to point density. Increments in row counts or skew produce dense spaces or zones, which translate into smaller differences along the Hilbert curve, and are therefore reflected by an increase in compression. On the other hand, increasing dimension counts or cardinalities functions in the opposite direction, expanding the space and thus producing a decrease in the compression ratio.

Comparisons have also been very encouraging, placing our results very close or even higher than state-of-the-art methods. For the benchmark against TDC, our method remains within 3% of their mark, while surpassing GRS in all cases. We have also included a comparison with a Naïve measure, which constitutes a bare per-field bit-compaction approach. Additionally, we compare our framework with Dwarf, a non relational system, discovering an interesting trend when the row-count is increased. This produces a degeneration of Dwarf's performance, with our results surpassing it by 20 percentage points in some cases.

In the final sections, we have presented indicators on the optimizations included in the framework, specifically the Fast Hilbert sort, GMP overhead minimization and query time reduction, obtaining excellent overall improvements. Next, measures of the load distribution in a parallel environment are shown. Here, the Sidera server obtains an almost perfect load balance with differences no greater than 0.3% for the number of rows retrieved in each node. Finally we include statistics for the bit-length and bit-runs found in Hilbert differentials, which could be useful for future developments.

Chapter 5

Conclusions

5.1 Summary

In this thesis we have presented a compression framework for OLAP. The framework is based on the Hilbert space-filling curve, which in combination with differential encoding and bit compaction, provides high levels of data cube and index size reduction and effective means for the execution of user queries. Specifically we have addressed the following topics:

- *Data compression:* Modern OLAP systems allow knowledge workers to obtain relevant information in order to support decision making processes. The underlying representation of decision-enabling data is the Cube abstraction. The generation of the Cube requires the summarization of historical enterprise data into a lattice of multi-dimensional cuboids. The resulting structure can reach extremely large sizes, presenting the aforementioned systems with the daunting tasks of providing efficient methods for the storage and access to the cube data.

  The first building block in our framework is the HDC method, proposing an algorithm to reduce the data cube storage requirements. The method departs from traditional compression algorithms by effectively exploiting the inherent redundancies found in OLAP data. Only the differences between consecutive tuples are stored, using a Hilbert space-filling curve to represent records as a totally ordered set of d-dimensional points. The Hilbert differentials are then bit-compacted into disk blocks, to match the unit of I/O with the compression granularity. This technique optimizes disk accesses and at the same time allows the independent decompression of selected disk blocks.

We prepared a test environment with strict measures to ensure a fair comparison. Our compression framework was developed using Agile engineering practices, with an object-oriented C++ implementation complemented by several open-source libraries. All methods used as benchmarks faithfully follow the guidelines published in the original papers. Several data configurations were prepared, and in all of them the results achieved by our framework were very competitive with those obtained by the compared state-of-the-art methods. On average, our method yields compression ratios in excess of 80%, very close to the TDC marks and noticeably higher than those from GRS. A constant observed trend is the increase of the compression efficiency with the augmentation of point space density, because the spatial distances between points along the Hilbert curve are shorter, and therefore the stored differentials are smaller.

- *Index compression:* An index is a particular structure built with the purpose of reducing the amount of data that must be analyzed to answer a query, consequently obtaining improved access times. In the OLAP context, R-trees have been proven to be one of the most effective indexing techniques. These structures are based on a hierarchy of d-dimensional hyper-rectangles, enclosing data points or other hyper-rectangles.

  The second major module of our framework is the HRC method, which presents an indexing algorithm based on *packed* R-trees. This special type of R-tree minimizes the total area covered by the leaf nodes and their overlap by filling to capacity one disk block per node of the tree. Compression is achieved by applying a schema where the local minima of the block is used to calculate per-dimension differentials, which are later stored using bit-compaction. The leaf nodes correspond to HDC-compressed data blocks, where the underlying total order is given by the Hilbert curve.

The evaluation runs for our indexes share the same methodologies previously explained. These tests yielded extremely high compression ratios in excess of 95% on average. Additionally, we performed a comparison against Dwarf, an influential non-relational method that uses a special pointer-based structure to represent both data and indexes. In this experiment, the results were quite positive, with our method surpassing Dwarf by more than 20 percentage points in data sets with high row-counts.

- *Query resolution:* The third core component of our framework is the Query Engine. It receives a user query, performs the search by traversing the index structure and retrieves the data satisfying the request. All compressed R-tree and data blocks are written in a sequential left-to-right and top-to-bottom fashion. This serves two purposes. First the query engine will perform long sequential reads instead of small random accesses, matching the preferred behaviour for hard drives. Second, it allows to perform a Linear BFS search, where only nodes relevant to the query are visited.

  Evaluation results for the query engine showed that the number of blocks accessed is reduced by a factor of 10, when compared to a search performed on non-compressed data. However, block decompression imposes an overhead on the response time. Extensive optimizations reduced this time from a factor of 50 to 2 for low dimensions and 4 for high-dimensions. Further optimizations are possible to lower this mark to a level not noticeable by end users, which we believe corresponds to a threshold of 2 for high-dimensions.

- *Scalability:* Our framework provides unlimited scalability, meaning that the row, cardinality and dimension count of the input set is only restricted by the underlying storage. This is made possible by a combination of the following techniques: first, a judicious use of the GMP libraries to handle arbitrary-precision arithmetic during the calculation of Hilbert ordinals. Second, a buffering subsystem conceived to transparently manage the memory usage. And third, a p-way external sort to produce the initial Hilbert order.

- *Parallelization:* Processing intensive and time critical tasks are excellent candidates for workload sub-division. Two clear examples of these cases are respectively: the initial ordering and compression, and query resolution tasks. Therefore, the framework was designed from the ground up to be executed in a distributed environment. Each computing node independently builds the compressed structures and participates on resolving a given portion of a user query. Thanks to the excellent clustering properties of the Hilbert curve, the Sidera Server attains an almost perfect load distribution with an imbalance below 0.3%.

5.2 Future Work

The investigation materialized in our compression framework opens a door for new research initiatives to extend the proposed methods or to apply them in different environments. In this section we identify several of these possibliities.

- *Measure compression:* In our work, as in all research papers, the compression focuses on the cuboid attributes. However, OLAP data contains one or more measure values that could also be part of a complementary encoding process. The publications in this area are practically non-existent, due to the difficulty of finding regularities that could be exploited.

- *Differential internals:* Although our method has reached very high data compression ratios, it might still be possible to apply additional compression techniques to the Hilbert differentials. In section 4.3.7 we have included some indicators of the internal bit-features of these differentials, which could be useful in this regard.

- *Query caching:* The Sidera Server includes a simple caching mechanism, that could be extended to obtain reduced query response times. A more thorough caching strategy will avoid I/O and decompression tasks for commonly requested data blocks. Also, if a batch of queries is known in advance, common regions for all queries could be identified, to insure that a given block is processed only once.

- *In-house GMP:* The GMP library has been used to manipulate arbitrary-length Hilbert ordinals. Although it is a widely-used and highly-optimized library, it imposes some overhead to all processes. Additionally, the number of functions and structures from this library that are used in the framework is minimal when compared to its full capabilities. Therefore, we believe that an in-house implementation with only the specific required features will cerainly improve the overall processing and response times.

- *Conversion optimization:* The method proposed by Butz [1971] and complemented by Lawder [2000] was implemented to reduce the time involved in the Hilbert transformations. Butz's algorithm divides the Hilbert ordinal into m columns (i.e. *words*) of d bits each, where m is the number of bits per axis and d is the dimension count. Then, it applies bit operations to every column to obtain the multi-dimensional tuple. Note that although we are using 32-bit registers for the columns, only d bits are representative. Because $d < 32$ for most cases, we consider that the technique can be improved to include more than one word per column, therefore reducing the number of bit operations executed during the conversion.

- *Non-squared spaces:* The Hilbert transformation algorithms in the literature are designed to work in spaces with an equal number of bits for all axes. Recall that this count is defined by the cardinality of each dimension, which most likely will not be uniform. Consequently, further compression gains will be achieved on mixed-cardinality data sets if the transformations are performed within a non-squared space. Jagadish [1990] provides an overview for possible approaches.

- *Incremental updates:* A Data Warehouse was defined by [Inmon, 2005] as being non-volatile, meaning not-changeable as opposed to an OLTP database. However, incremental updates would be a valuable addition to our framework. The objective should be to avoid the regeneration of the compressed cube and indexes when new historical data is available.

- *Distributed tests:* We are currently in the last stage of the configuration of a 16-node parallel cluster to be able to test our framework in a distributed hardware platform. No changes to the implementation should be required, as support for parallelization has been one of the design objectives.

- *Real data sets:* The Data Generator used in our research to produce the data sets is a very flexible tool. However, tests on real-wold data would be an interesting addition to the evaluation rounds. We suggest to consider a dataset by Hahn et al. [1996], used in the publications of Dwarf [Sismanis et al., 2002] and QC-trees [Lakshmanan et al., 2003]. This set contains cloud conditions observed at several land stations for September 1985.

5.3 Final Thoughts

Our objective has been to develop a practial solution to a problem pertaining to both commercial and academic OLAP systems. We have presented an integrated set of techniques for compressing the Data Cube and providing effective access to its data. All methods have been designed with efficiency and scalability in mind, including the benefits from parallel processing. Therefore, we believe that this thesis will be a valuable contribution to the current literature in the field.

Bibliography

David J. Abel and David M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990.

Georgii M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 143:263–266, 1962.

Steven Alter. *Decision Support Systems: Current Practice and Continuing Challenge.* Addison-Wesley, Reading, MA, USA, 1^{st} edition, 1980. ISBN 0201001934.

Scott W. Ambler and Ron Jeffries. *Agile modeling: Effective practices for extreme programming and the unified process.* John Wiley & Sons, Inc., New York, NY, USA, 2002. ISBN 0-471-20282-7.

Kathy Auerbach. 2005 TopTen Program Summary: Select Findings from the TopTen Program. White paper, WinterCorp, Waltham, MA, USA, May 2006.

Rudolf Bayer and Ed McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. ISSN 0001-5903.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-365-5.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. ISSN 0001-0782.

Elisa Bertino, Beng Chin Ooi, Ron Sacks-Davis, Kian-Lee Tan, Justin Zobel, and Boris Shidlovsky. *Indexing Techniques for Advanced Database Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 1997. ISBN 0792399854.

Kevin Beyer and Raghu Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 359–370, New York, NY, USA, June 1999. ACM Press.

Eric Bodden, Malte Clasen, and Joachim Kneis. Arithmetic Coding Revealed. In *Proseminar Datenkompression 2001.* RWTH Aachen University, 2002.

Arthur R. Butz. Convergence with Hilbert's Space Filling Curve. *Journal of Computer and System Sciences*, 3(2):128–146, May 1969.

Arthur R. Butz. Alternative Algorithm for Hilbert's Space-Filling Curve. *IEEE Transactions on Computers*, 20(4):424–426, April 1971. ISSN 0018-9340.

Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997. ISSN 0163-5808.

Peter Pin-Shan Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. ISSN 0362-5915.

E.F. Codd, S.B. Codd, and C.T. Salley. Providing OLAP to User-Analysts: An IT Mandate. White paper, E.F. Codd Associates / Arbor Software, 1993.

Comshare. Now part of Extensity Corporation. `http://www.extensity.com`.

David Cueva. Estudio comparativo de la gestión de recursos en DBMS mediante el uso de un benchmark. Engineering Thesis, Escuela Politécnica Nacional, Quito, Ecuador, 2002.

Gordon Davis. *Management Information Systems: Conceptual foundations, structure and development.* McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1974. ISBN 0070158274.

Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. *RFC-1951*, May 1996.

William DuMouchel, Chris Volinsky, Theodore Johnson, Corinna Cortes, and Daryl Pregibon. Squashing flat files flatter. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 6–15, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-143-7.

Todd Eavis. *Parallel Relational OLAP*. PhD thesis, Dalhousie University, Halifax, NS, Canada, June 2003.

Christos Faloutsos. Multiattribute hashing using Gray codes. In *SIGMOD '86: Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 227–238, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-191-1.

Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 426–439, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-236-5.

Robert Fano. The Transmission of Information. Technical report 149, Research Laboratory of Electronics, MIT, Cambridge, MA, USA, 1949.

Stuart I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3rd edition, September 2003. ISBN 0-321-19368-7.

Martin Fowler and Jim Highsmith. The Agile Manifesto. *Software Development*, 9 (8):28–32, August 2001.

Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM Press. ISBN 0-89791-021-4.

GMP. GNU Multiple Precision Library. `http://www.swox.com/gmp`.

GNU. GNU's not Unix. `http://www.gnu.org`.

Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing Relations and Indexes. In *ICDE '98: Proceedings of the 14th International Conference on Data Engineering*, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8289-2.

Solomon W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966. ISSN 0018-9448.

Doron Gottlieb, Steven A. Hagerth, Phillipe G. Lehot, and Henry S. Rabinowitz. A Classification of Compression Methods and their Usefulness for a Large Data Processing Center. In *Proceedings of the National Computer Conference*, volume 44, 1975.

Frank Gray. Pulse Code Communication. *U.S. Patent No. 2,632,058*, March 1953.

Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *International Conference on Data Engineering*, 00:152, 1996. ISSN 1063-6382.

Diane Greene. An Implementation and Performance Analysis of Spatial Data Access Methods. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 606–615, Washington, DC, USA, 1989. IEEE Computer Society. ISBN 0-8186-1915-5.

Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index Selection for OLAP. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 208–219, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7807-0.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on*

Management of data, pages 47–57, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-128-8.

Carole J. Hahn, Stephen G. Warren, and Julius London. Edited synoptic cloud reports from ships and land stations over the globe, 1982–1991. Technical report NDP-026B/4367, U.S. Department of Energy, Washington, DC, USA, February 1996.

Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-489-8.

Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 205–216, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-794-4.

David Hilbert. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, September 1891. ISSN 1432-1807.

C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961. ISSN 0001-0782.

Paul G. Howard and Jeffrey Scott Vitter. Practical Implementations of Arithmetic Coding. Technical report 92-18, Brown University, Providence, RI, USA, 1992.

David Huffman. A Method for the Construction of Minimum Redundancy Codes. In *Proceedings of the Institute of Radio Engineers (IRE)*, volume 40, pages 1098–1101, 1952.

IBM. International Business Machines. `http://www.ibm.com`.

Informix. Designing the Data Warehouse on Relational Databases. White paper, Stanford Technology Group Inc., 1995.

William H. Inmon. *Building the Data Warehouse.* Wiley Publishing, Inc., Indianapolis, IN, USA, 4^{th} edition, 2005. ISBN 0764599445.

Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, 1962. ISBN 0471430145.

H. V. Jagadish. Linear clustering of objects with multiple attributes. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 332–342, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-365-5.

Ibrahim Kamel and Christos Faloutsos. On packing R-trees. In *CIKM '93: Proceedings of the second international conference on Information and knowledge management*, pages 490–499, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-626-3.

Owen Kaser and Daniel Lemire. Attribute value reordering for efficient hybrid OLAP. In *DOLAP '03: Proceedings of the 6th ACM international workshop on Data warehousing and OLAP*, pages 1–8, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-727-3.

Ralph Kimball. The database market splits. *DBMS*, 8(10):12–ff., 1995. ISSN 1041-5173.

Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 2^{nd} edition, 2002. ISBN 0-471-20024-7.

Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 2^{nd} edition, April 1998. ISBN 0-201-89685-0.

Laks Lakshmanan, Jian Pei, and Jiawei Han. Quotient Cube: How to Summarize the Semantics of a Data Cube. In *VLDB '02: Proceedings of th 28th International Conference on Very Large Databases*, pages 778–789, August 2002.

Laks Lakshmanan, Jian Pei, and Yan Zhao. QC-trees: An efficient summary structure for semantic OLAP. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 64–75, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-634-X.

LAM. Local Area Multicomputer. `http://www.lam-mpi.org`.

Jonathan K. Lawder. Calculation of Mappings between One and n-dimensional Values Using the Hilbert Space-Filling Curve. Technical Report JL1/00, Birkbeck College, University of London, London, UK, 2000.

LEDA. Library of Effcient Data Structures and Algorithms. `http://www.algorithmic-solutions.com/enleda.htm`.

Ming-Ling Lo and Chinya V. Ravishankar. Towards Eliminating Random I/O in Hash Joins. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 422–429, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7240-4.

Marshall McLuhan. *Gutenberg Galaxy : The Making of Typographic Man.* University of Toronto Press, Toronto, ON, Canada, 1962. ISBN 0451616162.

Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001. ISSN 1041-4347.

Doug Moore. Fast Hilbert Curve Generation, Sorting, and Range Queries. `/http://www.caam.rice.edu/~dougm/twiddle/Hilbert/`, February 2005. Available at: `http://web.archive.org`.

Konstantinos Morfonios and Yannis Ioannidis. CURE for cubes: Cubing using a ROLAP engine. In *VLDB'2006: Proceedings of the 32nd international conference on Very Large Data Bases*, pages 379–390. VLDB Endowment, September 2006.

Samuel Morse. Telegraph Signs. *U.S. Patent No. 1,647*, June 1840.

Michael Scott Morton and J. A. Stephens. The impact of interactive visual display systems on the management planning process. In *International Federation for Information Processing (IFIP)*, volume 2, pages 1178–1184, Edinburgh, UK, August 1968.

MPI-Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, TN, USA, 1994.

Wee-Keong Ng and Chinya V. Ravishankar. Block-Oriented Compression Techniques for Large Statistical Databases. 9(2):314–328, 1997. ISSN 1041-4347.

J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984. ISSN 0362-5915.

OlapReport. The OLAP Report. `http://www.olapreport.com`.

OMG. Unified Modeling Language: Superstructure. Technical Report 05-07-04, Object Management Group, Needham, MA, USA, August 2005.

Patrick E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, 1989. Springer-Verlag. ISBN 3-540-51085-0.

Oracle. Oracle Corp. `http://www.oracle.com`.

Jack A. Orenstein. *Algorithms and data structures for the implementation of a relational database.* PhD thesis, McGill University, Montreal, QC, Canada, 1983.

Giuseppe Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36(1):157–160, March 1890. ISSN 1432-1807.

Heinz-Otto Peitgen and Dietmar Saupe. *The Science of Fractal Images.* Springer-Verlag, New York, NY, USA, 1988. ISBN 0-387-96608-0.

Nigel Pendse. The origins of today's OLAP products. *The OLAP Report*, April 2006. URL `http://www.olapreport.com`.

Daniel J. Power. *Decision Support Systems: Concepts and resources for managers.* Greenwood Publishers / Quorum Books, Westport, CT, USA, 1st edition, 2002. ISBN 1-56720-497-X.

Ida Mengyi Pu. *Fundamental Data Compression.* Butterworth-Heinemann – Elsevier, Oxford, UK, 1st edition, 2006. ISBN 0-7506-6310-3.

Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database Compression: A Performance Enhancement Tool. In *COMAD '95: Proceedings of the 7th International Conference on Management of Data*, Pune, India, December 1995.

Catherine Ricardo. *Databases Illuminated.* Jones and Bartlett Publishers, Inc., Sudbury, MA, USA, 2004. ISBN 0763733148.

Jorma Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976.

Yi Rong and Christos Faloutsos. Analysis of the clustering property of Peano curves. Technical Report CS-TR-2792, University of Maryland, College Park, MD, USA, 1991.

Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed R-trees. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, pages 17–31, New York, NY, USA, 1985. ACM Press. ISBN 0-89791-160-1.

Hans Sagan. *Space-Filling Curves.* Springer-Verlag, London, UK, 1^{st} edition, 1994. ISBN 0387942653.

David Salomon. *Data Compression: The complete reference.* Springer-Verlag, New York, NY, USA, 3^{rd} edition, 2004. ISBN 0-387-40697-2.

Betty Salzberg. Access methods. *ACM Computer Surveys*, 28(1):117–120, 1996. ISSN 0360-0300.

Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

Raymond Séroul. *Programming for Mathematicians.* Springer-Verlag, London, UK, 1^{st} edition, 2000. ISBN 354066422X.

SGI. Silicon Graphics, Inc. `http://www.sgi.com/tech/stl`.

Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, July 1948.

Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the PetaCube. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 464–475, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-497-5.

T.R. Smith and P. Gao. Experimental performance evaluations on spatial access methods. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, pages 991–1002, 1990.

Ralph H. Sprague and Eric D. Carlson. *Building Effective Decision Support Systems.* Prentice Hall Professional Technical Reference, Englewood Cliffs, NJ, USA, 1^{st} edition, 1982. ISBN 0130862150.

Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report 95-11, HP Laboratories, Palo Alto, CA, USA, November 1995.

James A. Storer. *Data Compression: methods and theory.* Computer Science Press, Inc., Rockville, MD, USA, 1988. ISBN 0-88175-161-8.

James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. ISSN 0004-5411.

Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Boston, MA, USA, 1997. ISBN 0201700735.

Erik Thomsen. *OLAP Solutions: Building Multidimensional Information Systems.* John Wiley & Sons, Inc., New York, NY, USA, 2^{nd} edition, 2002. ISBN 0-471-40030-0.

Dimitri van Heesch. Doxygen: Source code documentation generator tool. `http://www.doxygen.org`.

Peter van Oosterom. *Geographical Information Systems Principles, Technical Issues, Management Issues, and Applications*, volume 1, chapter Spatial Access Methods, pages 385–400. Wiley, 1999.

Peter van Oosterom and E. Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 1016–1029, 1990.

VIM. VI IMproved. `http://www.vim.org`.

Dan Vlamis. Effectively Using 9i OLAP in Business Intelligence Applications. In *OracleWorld 2002*. Oracle Corp., 2002.

Terrence A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, June 1984. ISSN 0018-9162.

Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3): 55–67, 2000. ISSN 0163-5808.

George Kingsley Zipf. *The Psycho-Biology of Language. An Introduction to Dynamic Philology*. Houghton-Mifflin, Boston, MA, USA, 1^{st} edition, 1935.

Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. ISSN 0018-9448.

Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

Glossary

API	*Application Programming Interface:* Source code interface provided by a program to allow service requests from other software components, 51
APL	*A Programming Language:* Array-based programming language invented by Kenneth Iverson in 1962, with the ability to define and manipulate multidimensional variables, 11
AUP	*Agile Unified Process:* Software develpment methodology, using Agile concepts with a simplified version of the Unified Process, 77
BFS	*Breadth-First search:* An algorithm for traversing a tree structure. It explores all sibling nodes in a level before continuing with the next level, 61, 64, 98
BI	*Business Intelligence:* Data-driven DSS, tailored to support business decisions, 9
DBMS	*Database Management Systems:* Software system whose objective is to facilitate the access and management to a set of structured data stored in a Database, 10, 12, 19, 21, 52
DFS	*Depth-First search:* An algorithm for traversing a tree structure. It explores from the root to the end of a given branch before taking the next branch, 61
DM	*Dimensional Modeling:* Modeling approach specially tailored to build Data Warehouses, 20
DSS	*Decision Support Systems:* A system that supports a person or group in the task of decision-making, 1, 8–10, 37
DW	*Data Warehouse:* A database structured to be accessed effectively by OLAP Systems, 12, 20

EIS	*Executive Information Systems:* Data-driven DSS, targeted to senior managers, 9
ER	*Entity Relationship:* A model used to design denormalized OLTP databases, 19, 20
ETL	*Extraction Transformation and Load:* OLAP process that extracts data from various operational sources, filters it from inconsistencies and loads it into a data warehouse, 12, 17
FASMI	*Fast Analysis of Shared Multidimensional Information:* Rules defining an OLAP system, 10
FIFO	*First-In First-Out:* A principle of handling processes in the order they arrive, 62
GCC	*GNU Compiler Collection:* A collection of open source compilers, including: C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages, 75, 76
GIS	*Geographic Information System:* A system aimed to manage geographically referenced data, 21
GMP	*GNU Multiple Precision:* An open-source library to handle aribitrary precision arithmetic, 51, 73, 75, 91, 98, 100
GNU	*GNU's Not Unix:* A project sponsored by the Free Software Foundation to develop a broad spectrum of free software, 51, 76
GRS	*Goldstein Ramakrishnan and Shaft:* A compression method for data and indexes based on differential encoding of each attribute with respect to a frame of reference, 52–55, 58, 59, 71, 73, 83–85, 95, 97
HDC	*Hilbert Differential Coding:* A compression method where a tuple is represented as a difference between its Hilbert ordinal and the one from the previous tuple, 45, 46, 48, 49, 52, 55, 56, 58, 61, 64, 69, 70, 73, 79–82, 84–86, 91, 96, 97
HOLAP	*Hybrid OLAP:* A flavor of OLAP which aims to combine the advantages of ROLAP and MOLAP technologies, 19

HRC	*Hilbert R-tree Coding:* A compression method for R-tree indexes where hyper-rectangles are packed following a Hilbert curve, and dimension values are stored as differences with the hyper-rectangle pivot point, 55, 58, 59, 61, 73, 85, 97
I/O	*Input/Output:* A read or write operation generally involving access to secondary memory, 3, 5, 68, 71, 90, 96, 100
LAM	*Local Area Multicomputer:* An open source implementation of the MPI standard, 75, 76
LEDA	*Library of Effcient Data Structures and Algorithms:* C++ proprietary library for various data types and algorithms, including graph-network problems, geometric computations and combinatorial opimization, 76
LGPL	*Lesser General Public License:* A type of GNU license which allows both, commercial and non-commercial use of the software being licensed, 51
MOLAP	*Multidimensional OLAP:* A flavor of OLAP in which the data is stored a multidimensional array, 19
MPI	*Message Passing Interface:* De-facto standard specification for communications in parallel applications, 66, 75, 76
NLP	*Natural Language Processing:* A branch of Artificial Intelligence dealing with the generation and understanding of non-formal languages, 9
OLAP	*On-Line Analytical Processing:* Type of software used to obtain a variety of views of the data stored in a Data warehouse, 1, 2, 4–6, 10–13, 15, 18, 19, 21, 23, 31, 35–38, 40, 50, 52, 64, 68, 71, 73, 77, 78, 83, 89, 96, 97, 99, 101
OLTP	*On-Line Transaction Processing:* Type of software used to manage the data associated with the transactions produced by the operational day-to-day processes of an organization, 12, 19, 21, 38, 101
RDBMS	*Relational DBMS:* A DBMS which mantains data and indexes in table structures, following the relational paradigm, 19

RLE	*Run Length Encoding:* Compression technique where repeated symbols are replaced by a count of occurrences, 32, 33, 44, 93
ROLAP	*Relational OLAP:* A flavor of OLAP in which the multidimensional data is stored in a relational database, 2, 19, 40
SQL	*Structured Query Language:* Language used to manipulate data in a DBMS, 1, 18
STL	*Standard Template Library:* A C++ library providing parametrized basic algorithms and classes, 75, 76
TB	*Terabyte:* 10^{12} bytes, 5
TDC	*Tuple Differential Coding:* Database compression method where a tuple is represented as a difference between its lexicographical value and the one from the previous tuple, 3, 36, 42–45, 49, 50, 52, 54, 71, 73, 81–85, 91, 95, 97
VIM	*VI IMproved:* A version of the VI editor with menu and mouse support, 75, 76