Hand-OLAP: a System for Delivering OLAP Services on Handheld Devices*

Alfredo Cuzzocrea, Filippo Furfaro,
DEIS, University of Calabria, 87030 Rende (CS), Italy
{cuzzocrea,furfaro}@si.deis.unical.it

Domenico Saccà
ICAR-CNR & DEIS, University of Calabria, 87030 Rende (CS), Italy
sacca@icar.cnr.it, sacca@unical.it

Abstract

The main drawbacks of handheld devices (small storage space, small size of the display screen, discontinuance of the connection to the WLAN, etc.) are often incompatible with the need of querying and browsing information extracted from the enormous amount of data which are accessible through the network. In this application scenario, the issues of compression and summarization of data have a leading role: data in a lossy compressed format can be transmitted more efficiently than original ones, and can be effectively stored in the handheld devices (setting the compression ratio accordingly). In this paper we describe a very effective compression technique for datacubes and the architecture of a system (based on this compression technique), called Hand-OLAP, which allows a handheld device to extract and browse compressed information coming from an OLAP server distributed on a wired network.

1 Introduction

Due to the growing interest in mobile computing, a great deal of research is investigating a series of problems which are crucial for the effectiveness of its applications. Such problems include the search of the most suitable data model, the definition of an easy-to-handle protocol of synchronization, and in general the analysis of every aspect which aim to make exchanging and consulting information feasible by means of wireless technology. The main drawbacks of handheld devices (small storage space, small size of the display screen, discontinuance of the connection to the WLAN, etc.) are often incompatible with the need of extracting information from the enormous amount of data which are accessible through the network. In this application scenario, the issues of compression and summarization of the information have a leading role as the large size of data sets represents a serious problem in designing efficient applications for managing and/or transmitting these data. A possible way to deal with such problems is certainly the application of approaches based on the *compression* of original data.

In the database context, a very effective lossy compression technique is the *histograms*-based one. Histograms summarize the contents of a relation R into a number of buckets. Buckets represent a partition of an attribute of R containing a number of aggregate values describing the occurrence of tuples of R within the corresponding range of that attribute. It corresponds to a succinct representation of the relation R. This compact representation of the relation can be directly used for estimating queries result and thus for fast (approximated) analysis of R. The issue of well approximating the original data distribution inside buckets is hence crucial. This has a direct counterpart on the kind of information we intend to keep within the bucket. Exploring approaches based on enriching classical bucket by addition information, like integrity constraints, small data structures, etc., is a very interesting research direction. Extending this approach to multidimensional data is another relevant topic. For multi-dimensional structures like datacubes this kind of compression is very useful since such structure are mainly used for making OLAP analysis. Therefore, besides of the improvement of performances of such applications thanks to the reduction of the data size, we can offer a natural support for them, since the typical goal of OLAP applications is extracting aggregate succinct even approximated information from data.

Hand-OLAP is a distributed Java-based system which allows to query and browse data stored in an OLAP server, belonging to a *wired domain*, using a handheld device, be-

*Work partly supported by a grant from COMPAQ srl, Italy.

longing to a *wireless domain*. The idea which the system is based on is the following: rather than querying the original data (being continuously connected to the WLAN), it may be more convenient for a mobile user (*m*-user) to maintain inside the handheld device a compressed view of the data he/she wants to deal with, and query them off-line.

Hand-OLAP works as follows (see Figure 1): after the *m*-user requests a bi-dimensional view from the available datacube sources, the requested view is extracted and materialized on a intermediate agent in the wired network; then the view is compressed in a lossy format and is transmitted to the handheld device. At this point, the *m*-user can query the compressed view locally (off-line) rather than the original datacube, thus obtaining approximate answers. Thus, Hand-OLAP enables an user of a handheld device to extract and browse information despite of the small size of the device display screen and even when he/she is not connected to the WLAN, making him/her free from waiting a long time to get a precision which is often not necessary. Indeed, the problem today is the abundance of information which is being accumulated at a pace that makes it no longer fit for direct human inspection. For these reasons, data compression is becoming imperative: indeed more and more often, the time and resources that need to be invested in order to gain access to information happens to be disproportionate to fruition time and value and defies the very purpose of accessing it.

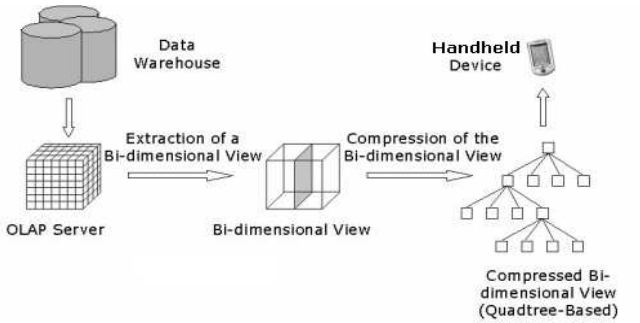

Figure 1. Extracting and compressing data

The paper is organized as follows. In Section 2 we briefly describe the related work about compression of datacubes in OLAP systems and summarization of information for mobile computing. In Section 3 we present the compression technique used for summarizing the selected information, which is based on the approach proposed by Buccafurri et al. in [3], and describe how compressed information can be queried. In Section 4 we introduce the architecture of the system and describe its main components in detail. In Section 5 we illustrate some functionalities of the client-side tool which allows to extract and browse the (summarized) OLAP data. Finally, in Section 6 we propose some extensions to the current architecture of the system.

2 Related work

Recent research is deeply investigating the problem of providing approximate answers to range queries in OLAP systems, in order to achieve better performances. Range queries consist of the application of a given aggregation operator (*count*, *sum*, *max*, *average*, etc.) over a multidimensional relation. Returning approximate answers is in many cases a necessary arrangement, since the amount of data is usually very large, and answering exactly to OLAP queries is often too complex and prohibitively expensive. This issue becomes critical when the range queries have to be delivered on handheld devices which cannot even transmit or accommodate large datacubes on their small storage space.

Many techniques for providing approximate answering to range queries are based on a compressed representation of the datacube in order to reduce the computational time: the queries are evaluated on such a compressed representation without accessing to the original datacube. Several compression models (like statistical [10], wavelet [11, 12], histograms [8, 9, 4], *synopsis* [7], and multivariate polynomials-based [1]), which had been originally defined and implemented in different contexts, have been used for compressing datacubes. In particular, the histogram-based techniques build a compressed datacube by partitioning the raw datacube into a number of blocks and storing for each block some aggregate data (i.e. the sum of the elements it contains). The answer to a query on a given range is computed by summing the values of all blocks which are wholly included in the range, and by performing suitable estimations of the pertinence values for the blocks which partially overlap the range. The sum of the elements of a range inside a block is estimated assuming that the values inside the blocks are uniformly distributed (*Continuous Values Assumption* - CVA). Thus, the estimation process is accurate if the boundaries of the blocks are defined in such a way that linear interpolation becomes effective (e.g., by avoiding that large value differences arise inside a block). Indeed, when the distribution of data inside a datacube is quite skewed, grouping a lot of cells into a few blocks makes the estimation process based on the CVA rather inaccurate, whatever is the technique used for partitioning the datacube.

The issue of datacube compression is crucial for handheld devices used as interfaces to OLAP systems. Indeed, in this case, pre-computation of a compressed view of a datacube is to be performed by a remote agent which fills the device with the reduced quantity of information that can be effectively handled on a reduced storage and a small screen, using a discontinuous and often slow connection to an OLAP server. Despite its relevance, this issue has not been so far addressed although data compression for hand-

held devices is being investigated in other contexts: e.g., for compressing mobile code (*Slim Binary Representation*)[6], based on the adaptive compression of syntax trees, and for *data summarization*[5], which is the extraction from a given text of a summary maintaining an equal informative content.

3 A compressed quad-tree representation of datacubes

We are given a bi-dimensional datacube D of size N. We want to construct a compressed representation of D using a size $M \ll N$. We adopt the technique introduced in [3] and based on a quad-tree partition of D which single-out a number of blocks and stores the the sums of measure values only for them. Figure 2 presents an example of compression process. In the upper part, the figure shows a bi-dimensional datacube whit initial sum of the measure values over the initial range equal to 256 and its partitioning process (the number reported inside the blocks is the sum of the measure values contained in the corresponding range); in the lower part, the figure shows the quad-tree which is obtained applying the partitioning process.

The compression process consists of a sequence of steps. We assume that the datacube contains non negative integers and that the sum of the measure values contained in the datacube can be stored using 32 bits. At the first step we store the sum of the elements contained in the whole datacube and split it into four blocks having the same size. The four blocks are obtained splitting each dimension of the block into two equal halves. Next, we take three of such generated blocks and, for each of them, we store the sum of their elements (using 32 bits) if the sum is not zero. Otherwise, if any of these sums is zero, we save 32 bits for each block containing only null values, and invest the saved space for splitting additional blocks.

At the following steps we choose the block containing the least uniform distribution of data, split it into four sub-blocks with the same size and store the sums corresponding to three of them. The sum corresponding to the fourth sub-block can be obtained by subtracting the sums of the other three (stored) sub-blocks from the value associated to the parent block. Again, we store only non zero sums and invest the saved space for splitting more blocks. In the quad-tree of Figure 2, the nodes with gray color represent stored blocks with not-zero sum and the nodes with white color represent not-stored blocks with not-zero sum obtained from the sums of parent and brother nodes.

The number of splits executed in the compression process depends on the amount of storage space which is available in the handheld device, as will be further explained.

The uniformity of the distribution of elements inside a block is measured by evaluating its variance. Such a greedy criterion in choosing the block to be split aims to build a

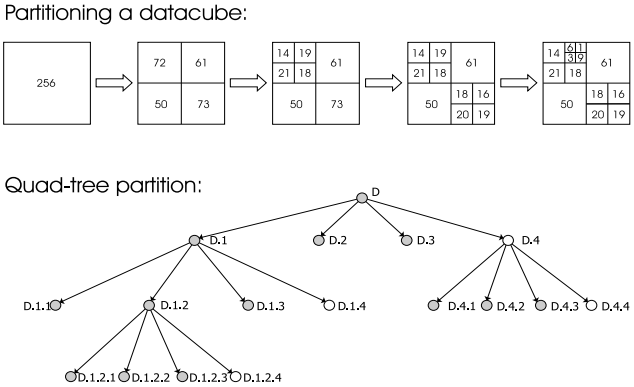

Figure 2. Quadtree Compression

partition of the cube whose blocks contain elements with *small* differences.

At the end of the described process, we obtain a summarized representation of the datacube, hierarchically organized according to a quad-tree. Each node of the quad-tree corresponds to the sum of a block generated during the partition, and the root is associated to the sum of the whole datacube. If the sum of the whole datacube exceeds $2^{32}-1$, 32 bits don't suffice for storing its value. Thus, we have to split the datacube and, if necessary, its sub-blocks until each resulting block contains a sum value which can be represented using 32 bits. In such a case, the compressed representation of the datacube is a forest of quad-trees. The roots of each quad-tree are associated to distinct ranges of the raw datacube. Without loss of generality, in the following we will assume that the compressed representation of the datacube consists of a unique quad-tree.

The storage space occupied by the compressed representation of the cube (organized as an unique quad-tree) consists of: (*i*) the space used for storing the exact values of the non zero sums, (*ii*) the space used for storing the structure of the quad-tree partition. Figure 3 shows the arrays representing, respectively, the sums and the structure of the quad-tree in Figure 2.

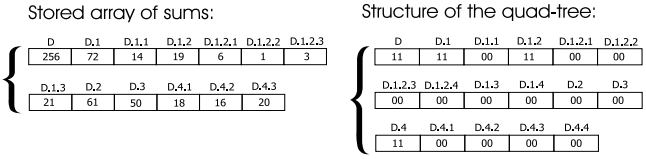

Figure 3. Array representation of a quad-tree

Assuming that every blocks generated by the partition process have non zero sums, and denoting the number of splits as t, the number of sum values which must be stored is $3 \cdot t + 1$. Since we use 32 bits for representing each sum, the representation of every sums occupies $96 \cdot t + 32$ bits.

The structure of the quad-tree representing the partition of the bi-dimensional view can be stored using two bits for each node of the quad-tree. The value of the first of such bits indicates whether the corresponding node is a leaf or not. The other bit indicates whether the associated sum is zero or not (if the sum is zero, it is not represented in the array of the sums). Thus, for storing the structure of the partition we need as many bits as twice the number of nodes of the quad-tree, that is: $2 \cdot (4 \cdot t + 1)$.

To conclude, assuming that every blocks generated by the partition process have non zero sums, the storage space occupied by the compressed representation of the datacube is $96 \cdot t + 32 + 2 \cdot (4 \cdot t + 1) = 104 \cdot t + 34$ bits.

The number of splits which can be done during the compression process is determined by the amount of available storage space. Denoting the available storage space as S, the number of splits is given by: $(S - 34)/104$. Indeed, this value represents the minimum number of splits which can be performed by the compression process. Generally, since datacubes are very sparse, a lot of blocks contain only null elements, and the saved storage space can be invested for further splits.

Now we explain how to perform a range query on the compressed datacube.

The query engine navigates the quad-tree searching for: (*i*) all *maximal* blocks which are completely involved in the query; (*ii*) all blocks which are partially included in the range of the query and which are not split (such blocks correspond to leaves of the quad-tree). The former blocks do not introduce any approximation, since for each of them the exact value of the sum inside the corresponding range is stored. The latter ones, on the contrary, introduce approximation, since we cannot re-construct the exact distribution of the original data contained in the corresponding ranges. The answers to the query in Figure 4 is approximate, since its lower boundaries do not coincide with the boundaries of any blocks of the compressed representation. The answer to the query in Figure 4 is the sum of the value associated to the block $D.1$ with a "contribute" of the block $D.3$. Such a contribute is obtained by performing a linear interpolation, assuming that the elements inside $D.3$ are uniformly distributed. For instance, assuming that the query involves an half of the block $D.3$, and denoting the sum of $D.3$ as $S(D.3)$, the estimated contribute is: $S(D.3)/2$.

3.1 Adding indices to leaf blocks

A problem with the compression process described in the previous section is that data distributions inside blocks are not guaranteed to be uniform enough to be well-approximable by linear interpolation. As a consequence, the estimation error risks to be intolerable. A way for facing the above problem is keeping, beside the overall sum

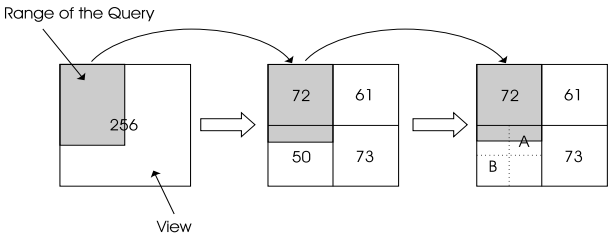

Figure 4. A query with approximate answer

of the element occurring in each block, further information for improving the accuracy in reconstructing range queries inside the blocks.

Experience acquired in compression of mono-dimensional [4, 2] and bi-dimensional cubes [3] suggests us to associate indices to the blocks corresponding to the leaves of the quad-tree for describing approximately their internal distribution of data — the index stores the approximate representations of the sums of internal sub-blocks inside the corresponding block. In Figure 5 we show how an instance of index (2/LT index) is built for the leaf block D.3 of the compressed datacube shown in Figure 2. The index is obtained as follows: the terminal block is partitioned into four sub-blocks and, in turn, each of the four sub-blocks into other four sub-sub-blocks. The index stores approximate aggregate data about both the generated sub-blocks and sub-sub-blocks. Such aggregate data consist of the sums of the elements contained in the regions which are colored in grey in Figure 5. The values of the sums are stored using less than 32 bits, introducing some approximation. The number of bits used for each stored value depends on the size of the corresponding sub-block. That is, referring to Figure 5, we use 6 bits for both A and B (which have the same size), and 5 bits for C, whose size is an half of A and B. Analogously, we use 4 bits for D and E (whose size is an half of C), and so on.

We point out that saving one bit for storing the sum of C w.r.t. A can be justified by considering that, on average, the value of the sum of the elements inside C is an half of the sum corresponding to A, since the size of C is an half of the size of A. Thus, on the average, the accuracy of representing A using 6 bits is the same as the accuracy of representing C using 5 bits.

Denoting the value of the sum of the whole block associated to the index as $S(Block)$, the approximate representation of the sum $S(A)$ of the region A in the index of Figure 5 using b bits is given by: $L_{S(A)} = (S(A)/S(Block)) \cdot (2^b - 1)$. The approximate value of $S(A)$ which can be retrieved from $L_{S(A)}$ is given by: $\tilde{S}(A) = (L_{S(A)}/(2^b - 1)) \cdot S(Block)$. Analogously, the approximate representation of the sum inside the region C using b bits is given by: $L_{S(C)} = (S(C)/S(A)) \cdot (2^b - 1)$, and consequently

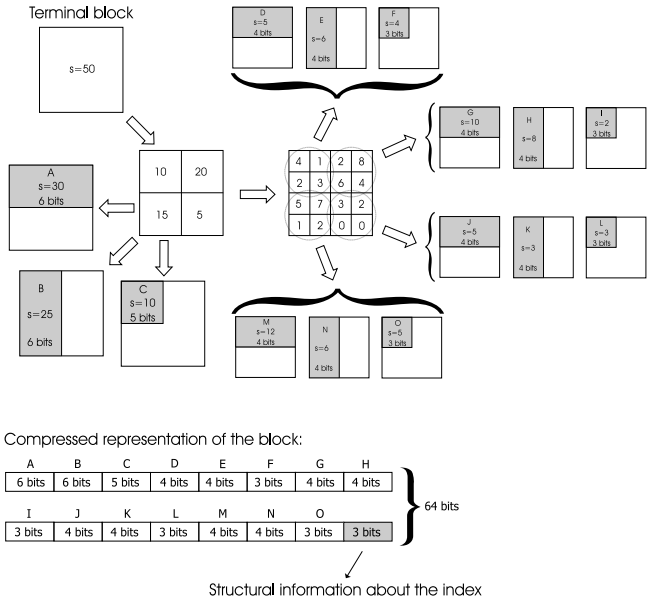

Figure 5. Building an index

the approximate value of $S(C)$ which can be retrieved from $L_{S(C)}$ is given by: $\tilde{S}(C) = (L_{S(C)}/(2^b - 1)) \cdot S(\tilde{A})$. Thus, the approximate representation of the sum inside a range R_i of the indexed block is evaluated on the basis of the approximate representation of the smallest range R_j which belongs to the same index and incorporates R_i. That is, $S(A)$ and $S(B)$ are evaluated using the sum of the whole block, $S(C)$ is estimated using $\tilde{S}(A)$, $S(D)$ and $S(E)$ are estimated using $\tilde{S}(C)$, and so on. Such a computation aims to minimize the average error (see [4]).

The above described index is based on a balanced quad-tree partition of a block. Different types of index can be used, based on different partitions. For instance, we can build an index based on an unbalanced quad-tree partition. Such an index is more suitable for a block where the elements are distributed *heterogeneously*, i.e. blocks consisting of some regions containing very skewed data distributions and other regions with rather uniform distributions. The detailed description of these indices can be found in [3].

The evaluation of a query inside an index is analogous to the evaluation process over a compressed datacube. That is, the query engine navigates the quad-tree equivalent to the index (which is made of all sub-blocks investigated by the index) and searches for: (*i*) all *maximal* sub-blocks which are completely involved in the query; (*ii*) all sub-blocks which are partially included in the range of the query and which are not split (such blocks correspond to leaves of the quad-tree).

Different types of index have been designed with the purpose of suitably approximating different kinds of data distributions. We follow thus the approach of selecting the most suitable index for a block on the basis of the actual distribution of data inside the block. That is, we measure the approximation error carried out by the index, and select the index which provides the best accuracy. For measuring the approximation error of an index I applied to a given block q we use the following metrics:

$$\epsilon_q(I) = \sum_{i=1}^{64} (sum(b_i) - sum_{nLT}(b_i))^2 \qquad (1)$$

where b_i represents the $i-$th (among 64 ones) sub-block of q obtained by dividing its sides into 8 equal-size ranges, and $sum_I(b_i)$ represents the estimation of the sum of elements occurring in b_i which can be done by using the index I and the knowledge of $sum(q)$ (the estimation of such sums can be done as explained above).

For a block q, we choose the index which "generates" the minimum value of ϵ_q^{nLT}.

3.2 A compressed quad-tree representation of datacubes using indices

Let us non describe the technique introduced in [3] to combine the usage of indices with the quad-tree based representation. We associate indices to the blocks corresponding to the leaves of the quad-tree, except the ones having zero sum or containing very uniform distributions of data as for such blocks the use of indices does not improve accuracy .

The storage space occupied by the compressed representation consists of: (*i*) the space used for storing the exact values of the non-zero sums, (*ii*) the space used for storing indices, and (*iii*) the space used for storing the structure of the quad-tree partition. Assuming that every block generated by the partition process contains a non zero sum, denoting the number of splits as t, the number of sum values which must be stored is $3 \cdot t + 1$. Since we use 32 bits for representing each sum, the representation of every sums occupies $96 \cdot t + 32$ bits. In the worst case (i.e., all leaves are indexed), the number of indices is equal to the number of leaves of the quad-tree: $3 \cdot t + 1$ (the number of leaves is equal to the difference between the number of nodes in the quad-tree and the number of intermediate nodes, i.e. $(4 \cdot t + 1) - t$). Since we use 64 bits for each index, we need $192 \cdot t + 64$ bits for storing all indices.

The structure of the quad-tree representing the partition of the bi-dimensional view can be stored using two bits for each node of the quad-tree. The values of these bits indicate whether the corresponding node is a leaf or not, whether its sum is zero and whether it is associated to an index. Thus, for storing the structure of the partition we need as many bits as twice the number of nodes of the quad-tree, that is: $2 \cdot (4 \cdot t + 1)$. To conclude, the storage space occupied by

the compressed representation of the bi-dimensional view is $96 \cdot t + 32 + 192 \cdot t + 64 + 2 \cdot (4 \cdot t + 1) = 296 \cdot t + 98$ bits.

The number of splits in the compressed representation is determined by the amount of available storage space. Denoting the available storage space as S, the number of splits is given by: $(S - 98)/296$. Indeed, this is the minimum value of t for a given storage space, since generally a lot of blocks contain zero sums and not every leaves must be associated to an index. Thus, the saved space can be invested for further splits.

We point out that, using the same storage space, the number of splits which can be done by the combined compression technique is less than the number of splits which can by done by the compression technique using no indices. Nevertheless, as indices provide details on the data distribution inside the leaves, these benefits overcome the drawback of having a smaller number of blocks.

4 Hand-OLAP: a system for delivering OLAP services on handheld devices

In this section we describe the architecture of the system Hand-OLAP (see Figure 6) whose goal is to provide compressed (bi-dimensional) views of a datacube coming from an OLAP server to handheld devices, for enabling *m*-users to browse and query (approximately) the desired information locally, even when the connection to the WLAN is off. The system is multi-tier type, and every software layer corresponds to a specific application logic.

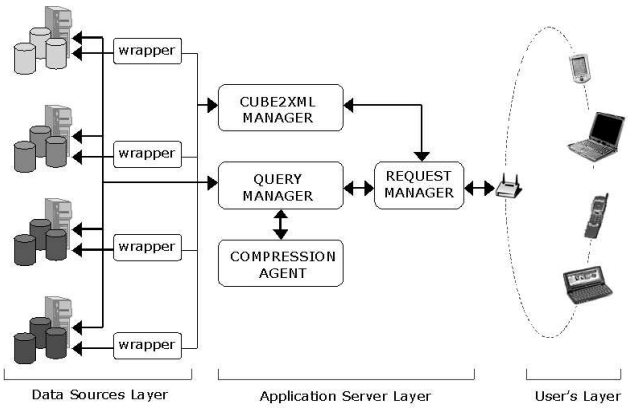

Figure 6. System overview

The (software) layers of the system are the following: (*i*) *Data Sources Layer*: it is the collection of OLAP servers from which the desired information can be retrieved, and of the wrappers which extract meta-information about the available datacubes as well as the actual data; (*ii*) *Application Server Layer*: it is the layer which elaborates the user's request, interacts with the OLAP servers, computes the compressed representation of the extracted view and sends it to the handheld device; (*iii*) *User Layer*: it includes the client-side tool which allows a handheld device to acquire and elaborate the desired information. As shown in Figure 6, exchanging information between the wired network and the wireless environment is allowed by an access point. The communication between the handheld devices and the access point is supported by an ad-hoc transport binary-based protocol based on the standard IEEE 802.11.

The system works as follows. When the handheld device is connected to the WLAN, the client side tool allows an *m*-user to request (or refresh) a collection of XML meta-data describing the information he can retrieve from the wired network, i.e. the information stored in the OLAP server which can be accessed. The XML meta-data are hierarchically organized (according to the usual data model for OLAP technology) and contain several details about the structure of the available information, such as the names of the dimensions and their number, the description of the measure attribute, etc.

An *m*-user may define a portion of the available data containing the information he/she is interested in, and request it. Such a portion is a bi-dimensional view defined over the available data. The user can also specify which OLAP server must be queried for retrieving information. The *m*-user's request is processed by the Application Server Layer, which extracts the requested information from the data sources it is connected to. Next, the retrieved information are summarized and sent to the *m*-user. Further details about how the Application Server Layer works are given in Section 4.1. The compressed representation of the requested information is eventually stored in the handheld device. Therefore, an *m*-user can browse and query the received data off-line, obtaining approximate answers.

We point out that the information is elaborated in the *wired* domain and delivered to the *wireless* domain, accordingly to the common development pattern of the wireless applications.

The steps of the extraction process are summarized in Figure 1. The system is currently in development phase. The server-side and the client-side components are being developed using Java. In particular, for the server-side components we are using the *Java2 Enterprise Edition* (J2EE) platform, and for the client-side tool we are using the *Java2 Micro Edition* (J2ME) platform. The compression libraries for the *Compression Agent* have been implemented using C++, for achieving better performances in the process execution. The Compression Agent interacts with the Java server-side components using *Java Native Interface API* (JNI), which supports inter-operability between Java code and external libraries.

Figure 7 shows the system architecture from a software components point of view. In particular, the architec-

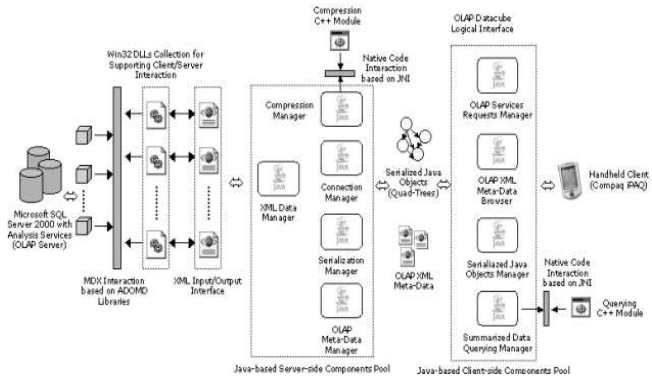

Figure 7. Hand-OLAP software architecture

ture shown is the first prototype of Hand-OLAP we have developed: it uses *Microsoft Analysis Services* as OLAP server over a *Microsoft SQL Server 2000* DBMS server and *Compaq iPAQ* as mobile client. We have developed a set of Win32 libraries based on the *Microsoft ADOMD API*, the Microsoft API for multidimensional data management, and a XML-based protocol which performs both the OLAP server meta-data extraction (about the server, data sources, catalogs, datacubes, dimensions, measures, etc.) and the interaction between OLAP server (*COM-compliant*) and Application server (*J2EE-compliant*). As shown in Figure 7, the server-side and the client-side components are completely Java-based. For example, the *Summarized Data Querying Manager* implements an ad-hoc algorithm for browsing and querying summarized OLAP data according to the data model described in Section 3.2.

4.1 The Application Server Layer

The Application Server Layer is the software layer where the Java server-side application logic resides. This layer implements the main functionalities of the system: request management, datacube/XML wrapping, data querying/viewing, compression. This logic is based on a pool of lightweight server-side software components, as shown in Figure 7.

The Application Server Layer consists of three components which cooperate to fulfil the *m*-user request (see Figure 6): (*i*) *Request Manager*: it is the component which receives the request of the *m*-user, and translates it either into a request to the *Cube2XML Wrapper* for retrieving meta-information about the content of the datacubes, or into a request to the *View Manager* for retrieving a compressed representation of the view defined by the user; (*ii*) *Cube2XML Wrapper*: it is the component that extracts meta-information about the OLAP server it is connected to, and returns them in an XML format; (*iii*) *View Manager*: it is the component that extracts from the selected datacube the bi-dimensional view defined by the *m*-user, uses the Compression Agent for summarizing it, and returns its compressed representation; (*iv*) *Compression Agent*: it is the component that receive a bi-dimensional view from the View Manager, and return its compressed representation. The View Manager sends the extracted bi-dimensional view to the Compression Agent together with the value of the desired compression ratio. Such a value depends on both the amount of storage space of the handheld device and the size of the view (see Section 3).

In the future developments of the system, the above described components (in particular the View Manager) will also deal with the problem of *integrating* data coming from heterogeneous OLAP servers, obtaining an unique datacube after *logically* integrating a set of datacubes.

4.2 The User Layer

The User Layer consists of a client-side tool which supplies an *m*-user with instruments for: (*i*) scanning the wired network and selecting an OLAP server to extract information from; (*ii*) asking for XML meta-data about the information contained in the accessible data sources, and browsing them; (*iii*) defining a bi-dimensional view over the selected datacube, containing *interesting* information; (*iv*) downloading the compressed representation of the selected view, after negotiating the compression ratio; (*v*) *refreshing* an already downloaded compressed view; (*vi*) executing range-sum queries on the compressed representation, without accessing to the original data.

5 Hand-OLAP in action

The main purpose of the system Hand-OLAP is to allow a handheld device to request a bulk of information coming from an OLAP server distributed on a wired network, and store the received (compressed) data locally, in order to query the received information off-line. The request of an user consists of a *bi-dimensional window* defining the range of data which has to be extracted. The request issued on a handheld device is processed by an Application server which queries the OLAP server. After receiving all the replies, it creates a view containing the range of data which the user needs, compresses it and sends it to the user. Thus, the reply to an user is a compressed representation of the range of data he requested. After receiving it, the user can store it locally and query it off-line, obtaining approximate answers. The approximation is a necessary arrangement, due to the small storage space and the small size of the display screen in handheld devices. Moreover, an user with a handheld device is typically more interested in querying and browsing approximate data without being connected to

any WLAN, rather than obtaining exact answers after being connected to the OLAP server for a long time.

The compression technique is very fast (i.e., the compressed representation can be computed efficiently) and very effective: the application server replies rapidly to the *m*-user's request, and the *m*-user is allowed to re-construct the original information with smaller approximation than other compression techniques.

The *m*-user can extract information from the requested compressed view using two alternatives: (i) either he/she can navigate the structure of the view by selecting a block (corresponding to a node of the quad-tree) and zooming in and out to traverse the quad-tree; or (ii) he/she can just submit a specific range query and receive the estimated answer.

6 Future works

Possible future extensions of Hand-OLAP are: (*i*) the implementation of a component of the Application Server which acts as a mediator for integrating data extracted from heterogenous datacubes at the Data Sources Layer; (*ii*) adding amenities for browsing meta-data using a compressed global view of the datacubes of interest; (*iii*) making the compression process *adaptive*, that is, using the degree of user interest to portions of data as an alternative (or additional) criterium for splitting a block; (*iv*) moving from the fixed number of 4 partitions for a block, as required by quad-trees, to a variable number as dictated by a semantic division of each dimension range, e.g., by extending paginated structures like *R-trees* which determine the maximum number of partitions on the basis of the available page size; (*v*) extending the system to deal with datacubes with more than two- dimensions.

The last issue is rather complex because of the limited perceptual bandwidth of the *m*-user, in addition to the traditional refractoriness of human beings to deals with hyperspaces. The approach that we intend to work on is to investigate representation models and interaction tools which help the *m*-user to construct bi-dimensional view from high-dimension datacubes.

References

[1] R. Agrawal, A. Gupta, S. Sarawagi, Modeling Multidimensional Databases, Proceedings of the *IEEE International Conference on Data Engineering*, Birmingham, UK, April 1997

[2] F. Buccafurri, F. Furfaro, G. Lax, D. Saccà, Binary-Tree Histograms with Tree Indices, Proceedings of the *International Conference on Database and Expert Systems Applications*, LNCS 2453, Springer, September 2002.

[3] F. Buccafurri, F. Furfaro, D. Saccà, C. Sirangelo, A Quad-Tree Based Multiresolution Approach for Compressing Datacubes, *ICAR-CNR TR*, December 2002.

[4] F. Buccafurri, L. Pontieri, D. Rosaci, D. Saccà, Improving Range Query Estimation on Histograms, Proceedings of the *IEEE International Conference On Data Engineering*, San Josè, CA, USA, February 2002

[5] O. Buyukkokten, H. Garcia-Molina, A. Paepcke, Seeing the Whole in Parts: text Summarization for Web Browsing on Handled Devices, Proceedings of the *International World Wide Web Conference*, Hong Kong, May 2001

[6] M. Franz, T. Kistler, Slim Binaries, *Communications of the ACM*, 40:12, pp. 87-94, December 1997

[7] P. B. Gibbons, V. Poosala, S. Acharya, Y. Bartal, Y. Matias, S. Muthukrishnan, S. Ramaswamy, T. Suel, AQUA: System and Techniques for Approximate Query Answering, *Bell Labs TR*, February 1998

[8] V. Poosala, Y. E. Ioannidis, Selectivity Estimation Without the Attribute Value Independence Assumption, Proceedings of the *International Conference on Very Large Databases*, Athens, Greece, August 1997

[9] V. Poosala, V. Ganti, Fast Approximate Answers to Aggregate Queries on a Datacube, Proceedings of the *IEEE International Conference on Scientific and Statistical Databases Management*, Cleveland, OH, USA, July 1999

[10] J. Shanmugasundaram, U. Fayyad, P.S. Bradley, Compressed Datacubes for OLAP Aggregate Query Approximation on Continuos Dimensions, Proceedings of the *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA, USA, August 1999

[11] J. S. Vitter, M. Wang, B. Iyer, Datacube Approximation and Histograms via Wavelets, Proceedings of the *ACM SIGIR-SIGMIS International Conference on Information and Knowledge Management*, Bethesda, MD, USA, November 1998

[12] J. S. Vitter, M. Wang, Approximate Computation of Multidimensional Aggregates of Sparse Data using Wavelets, Proceedings of the *ACM SIGMOD International Conference on Management of Data*, Philadelphia, PA, USA, June 1999