

# An Architecture for Modular On-Line Analytical Processing Systems: Supporting Distributed and Parallel Query Processing Using Co-operating CORBA Objects

Andreas Rauber and Philipp Tomsich\*  
Institute of Software Technology  
Vienna University of Technology, Austria  
{andi, phil}@ifs.tuwien.ac.at

## Abstract

During the last few years, On-Line Analytical Processing (OLAP) has emerged as a valuable tool for the analysis, navigation and reporting of hierarchically organized data from data warehouses. Still, it remains a challenging task to implement and deploy an OLAP system, since no standardized architecture exists, which describes the common components and functionality of OLAP systems. Additionally, the formal models in use disregard the need for easily implemented and clearly defined interfaces between these components. This paper presents a model for OLAP engines, which permits the development of modular systems based on a simple data-representation using sets and vectors. The functional units of the query processor are implemented in CORBA as independent modules with firm interfaces and exchange data and messages communicate across a software bus.

## 1. Introduction

On-Line Analytical Processing (OLAP) is a technique for the analysis and navigation of data stored in multidimensional data warehouses [7]. It was introduced by E.F.Codd [4] with the aim of providing point-and-click simplicity in decision support systems. It particularly gained popularity as a building block of modern executive information systems. Most major database vendors offer at least one product based on this technology. These commercial OLAP products available today are proprietary systems, which can interface to only a few data warehouses. This lack of interoperability makes it impossible for the user to select freely from the available products, and to choose the components which best fit a given purpose.

Today, two dominant approaches for the implementation of OLAP systems exist: *relational OLAP*, which stores the data in tables, and *multidimensional OLAP*. The latter type uses multidimensional data structures (e.g., sparse arrays, grid files, etc.) to store tuples of data according to their position within the multidimensional data-space. Still, it remains a challenging task to implement and deploy an open system with clearly separated modules, as no *standardized*

*architecture* exists, which describes the common components and functional units present. While most research concentrates on the formal models, algorithmic foundations and data structures for OLAP, rather little attention is paid to the architecture and structure of OLAP systems.

Yet, the benefit of a well-defined architectural definition is self-evident: Clearly separated software modules simplify the development process considerably and accelerate the construction and deployment of applications. Different implementations of well-defined interfaces will remain interoperable, so that changes are kept local. Above all, simple components with limited functionality are easy to design and implement. Complex systems can then be composed of such small components.

The architecture which we propose in this paper fulfills several important, and often interrelated, goals:

**Modularity.** All modules, which adhere to a predefined interface, can interact seamlessly.

**Interoperability.** The system has to work with a wide array of databases and storage models. The integration of multiple database systems based on wrapper modules needs to be supported.

**Scalability.** OLAP requires consistent reporting performance, independent of the size of the underlying database or its dimensionality.

**Extendability.** We want to add additional modules without rebuilding the system.

Component-based OLAP systems offer a number of benefits both to the user and the developer. The user can choose from different suppliers and combine different query optimization strategies and query evaluation algorithms. Extensible OLAP systems offer a major benefit to developers, as well. We will show how to decompose an OLAP system into functional units, which can communicate using a “software bus” (e.g., CORBA). We propose a data model based on sets and vectors for the communication.

The remainder of this paper is structured as follows: Section 2 lists some related work on the structure and architecture for OLAP systems. Section 3 introduces an architecture for a modular OLAP system. We describe the modules and functional units from our architecture in Section 4 and present some conclusions in Section 5.

\*This work is supported by the European Union under Esprit/INCO grant no. 977071

## 2. Related Work

Although recent advances in data warehousing technology and computing power have led to a proliferation of data warehouses in use and to an increased interest in the architectural foundations of distributed and modular systems, rather little information is available on the architecture and structure of OLAP systems.

One of the first web-based OLAP systems was developed at our institute in the WWWEIS-DWH project [8]. Research on modeling data warehouses and meta-data using extended relational models is reported in [10]. [12] introduces the general architecture of data warehousing systems and treats the problem of modularizing data warehouses at a high level. The WHIPS data warehousing prototype at Stanford [13] uses CORBA objects to implement a scalable and modular system.

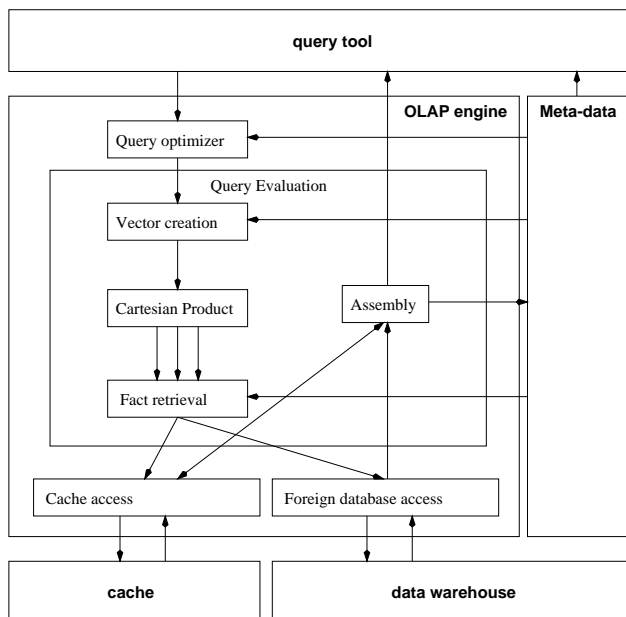
A number of approaches exist to the storage of data in data warehouses. [14] contrasts the performance of value-based (ROLAP) and multidimensional (MOLAP) implementations. The data-structures used include grid-files [5], B\*-trees, R\*-trees [1], X-trees, HB-trees [9], GiST [6], arrays and sets-based data-structures.

We argue in favor of using sets for the consistent representation of data within the OLAP system. Sets can be implemented efficiently both for serial and concurrent execution. Algorithms for the optimization and parallel execution of set operations (e.g., unions, intersections) and optimization techniques providing an optimal ordering of such operations are discussed in [3]. An implementation of parallel set operations using random binary balanced trees (treaps [11]) is presented in [2] and shows a very reasonable speed-up in multi-processor systems.

## 3. The Architecture

Figure 1 depicts the distinct phases of query evaluation, yet leaves the data flow unspecified. In this section we introduce an abstract model for the data representation and query evaluation in OLAP systems which justifies our choice of data-structures used for the communication between these modules. The computational model we use assumes a multidimensional data abstraction within the OLAP engine. We always treat the fact-base as a cube, even if it is only available in a relational data warehouse and needs to be converted into a multidimensional representation. This approach allows us to apply multidimensional operator like slice, dice, etc. to all intermediate query results and to use a single interface between the different phases of query evaluation.

The OLAP architecture proposed here is based on set-theory and allows simple and efficient implementation. A tuple  $T$  of values, as it is found in the relational database model, can be expressed as the Cartesian product of keys  $K_i$  with  $i \in \{1..n\}$  and values  $V$ :  $T = K_1 \times \dots \times$



**Figure 1.** An expanded view of the control flow for the evaluation of an OLAP query: The user generates a query using the graphical interface of a query-tool (e.g., a decision support system) and passes it on to the OLAP system. There, it first passes through a query optimizer which removes redundancies and rewrites the query to exploit parallelism, before it is evaluated. The evaluator creates an execution plan, which leads to the retrieval of a subset of the facts contained in the data cube. This execution plan consists of a set of position vector which denote the sub-space of the data-space relevant to the query. In order to create this set of vectors, the query is evaluated for each dimension in parallel (vector creation), before combining them through a Cartesian product. The facts specified within this execution plan can then be retrieved in parallel. The fact retrieval is controlled by meta-data indicating which tables within the data warehouse to consult and whether cached sub-cubes are available. The meta-data repository is read-only, except for the information of cached sub-cubes, which are written during the final assembly of the query results. The query result is added to the cache and returned to the client application.

$K_n \times V$ . Each  $K_i$  may contain multiple sub-keys, i.e.,  $K_i = K_i^1 \times \dots \times K_i^{j(i)}$ , which impose a hierarchical ordering. For example, if we consider a  $K_i$  which represents the time dimension and consists of a DAY, MONTH and YEAR attribute. These three key-attributes are the sub-keys  $K_i^1$ ,  $K_i^2$  and  $K_i^3$ . In this model, all dimensions  $K_i$  (e.g., time, space, product) are contained in  $K$  (e.g., time  $\times$  space  $\times$  product). Each dimension may consist of multiple sub-keys (e.g., YEAR, MONTH, DAY) for the dimension time, which structure the data hierchically.

A hierarchical ordering between these fields exists as well. A YEAR contains multiple MONTHS and so on (see Figure 2(a)). This hierarchy represents a hierarchy of data granularities. Starting with the least significant sub-key, sub-keys may be left *unbound* in queries to view the data at a coarser level of detail.

Multiple paths through the hierarchy may co-exist, as demonstrated in Figure 2(a), as long as there is only one entry without a predecessor ( $\top$ ) and only one without a successor ( $\perp$ ). In the example depicted, YEAR can be decomposed either into MONTH or WEEK. Still, because all paths

through the granularity tree reach the common element DAY ( $\perp$ ), all queries can be resolved by accessing the decomposition into this common level of detail (i.e., DAY). The requirement for a single root node can easily be satisfied by subsuming multiple roots in an artificial root node  $\top$ .

The logical organization of the data is position-based. The values (facts)  $v \in V$  (with  $V$  symbolizing the entire data warehouse) are associated with key-tuples  $k \in K$  and distributed across the  $n$ -dimensional data-space accordingly. For a key-tuple  $k$  and an associated value tuple  $v$ , the relation  $\phi : k\phi v$  ( $k = k_1 \times \dots \times k_n$  with  $k_i \in K_i$ ) holds if, and only if, a tuple  $t = k_1 \times \dots \times k_n \times v$  is contained in the fact-base. As a consequence, the spatial position of a value within the data cube along the  $i$ -th dimensional axis is determined by the value of the  $i$ -th sub-key  $k_i$ . This results in a  $n$ -dimensional hypercube, where each  $k_i$  gives the location of the value  $v$  along the  $i$ -th dimension.

The vector  $k$  with  $k\phi v$  is referred to as the position vector of a fact  $v$ , since it represents the spatial position of the fact within the multidimensional data space. Queries access the fact base by retrieving the facts associated with a given position vector. Sub-keys of any component of this position vector may remain *unbound* to any value to symbolize data at higher level of aggregation. If we want to represent the aggregated data for the YEAR 1999, we set the sub-key representing the YEAR to 1999 and leave the sub-keys for MONTH and DAY unbound.

The meta-data consists—amongst others—of semantical information (naming) to identify the  $i$ -dimensions and data regarding both the hierarchy (analogous to the granularities) of key-attributes (e.g., a YEAR consists of up to twelve MONTHS) and the hierarchy of instance-values of these key-attributes (e.g., the year 1999 contains a month March 1999). When inserting a new tuple into the fact-base, the attribute hierarchy is traversed to determine the spatial location of the new fact. If a specified key-attribute is not present, a new node is inserted. Additional meta-data may include information on data-types, ordering relations and integrity constraints.

Each node in this hierarchy is a tuple  $key \times ancestors \times descendants$ . The *ancestors*-set and *descendants*-set imply a part-of relationship on the keys: e.g., March 1998 is a part of 1998 (see Figure 2(b)). Two nodes may contain the same value for the key field, but identify distinct spatial locations (e.g., June1998 and June1999 both have June as their key attribute). Apparently, the equivalence of nodes can be determined only from their location within the hierarchy of nodes. Let  $\triangleright$  denote the *descendant* relation, such that  $a \triangleright b$  holds, if and only if  $b$  is a *descendent* of  $a$  (e.g., 1999  $\triangleright$  June1999). Now we can describe the position of any node  $n$  within the hierarchy as  $p(n) = \{e|e \triangleright^* n\}$  where  $\triangleright^*$  denotes the transitive hull over  $\triangleright$ . Two nodes can be tested for equivalence using their path:  $n_1$  and  $n_2$  are

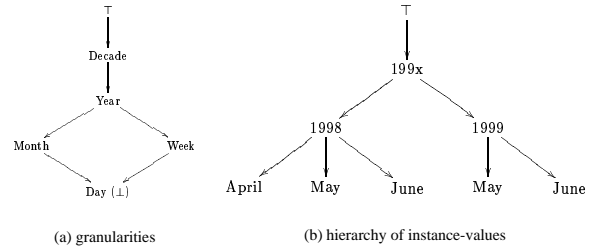


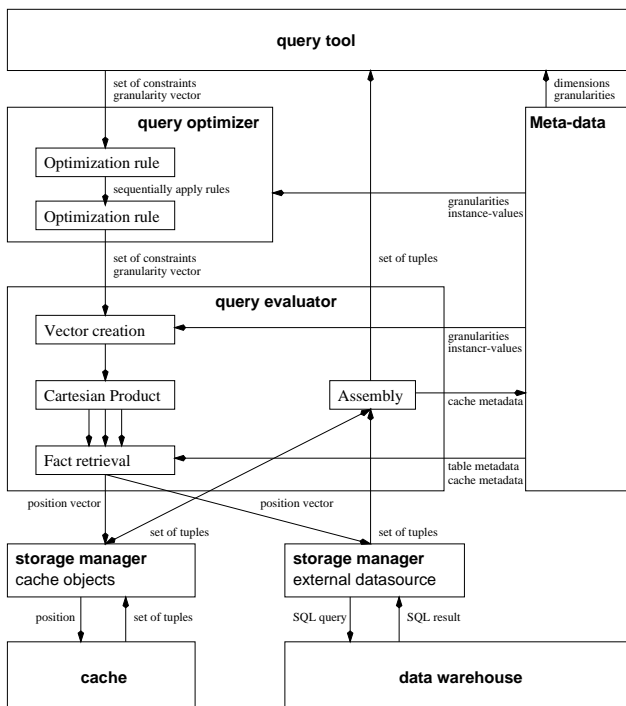
Figure 2. Example of a time dimension: The arrows denote a “contains”-relationship.

the same node, if and only if,  $p(n_1) = p(n_2)$ . For example,  $p(\text{June1998}) = \{\top, 199x, 1998\}$  while  $p(\text{June1999}) = \{\top, 199x, 1999\}$ . Using the *descendant* relation, we can define the contents of the *descendants* and *ancestors* fields of a node as well:  $n.\text{descendants} = \{e|n \triangleright e\}$  and  $n.\text{ancestors} = \{e|e \triangleright n\}$ . As a result, the node for 199x is represented by the tuple  $\langle 199x \times \top \times \{1998, 1999\} \rangle$ . Since the nodes 1998 and 1999 contain a set of descendant nodes again, this definition apparently leaves us with a structure of nested sets.

We want our queries to select a part of the data cube at a selectable granularity. Thus we need to constrain the extent of the cube along each of its dimensions by introducing constraints on the key-attributes. For example, if we constrain the YEAR to 1999 and view the cube at the level of detail corresponding to YEARS, only a single slice of the cube will result. Apparently the level of detail (granularity) at which we view the cube is the other influencing factor for the results of a query. For this reason, we have to include information on the desired granularity for each dimension. For an  $n$ -dimensional cube an  $n$ -dimensional *granularity vector* is necessary.

Query evaluation uses term substitution to expand the hierarchically nested sets of keys, while applying constraints to them, in order to obtain a set of vectors with spatial positions at a given level of detail. For example, if a query requests a sub-cube, which is constrained to data from the month June and the year 1999, the algorithm starts with the top ( $\top$ ) node in its selected set (or permissible set). It will then perform an *expansion* by replacing every selected node with all its descendant nodes (e.g., the years 1998 and 1999). After the expansion the constraints are applied to eliminate nodes outside the query result. In this example, the constraint on the year field will be applied, which will leave the node 1999 as the only member of the query result/set. This expansion and application of constraints needs to be repeated until all constraints have been applied once and the desired level of detail is reached.

To sum this up, we need four distinct data structures for the representation of queries and query results: **granularity vectors**, which specify the desired level of detail in *queries*,



**Figure 3.** Data-flow between the modules of our OLAP architecture: A user constructs a query using the graphical user-interface of a query-tool. During query construction (particularly for *ad-hoc* queries), the query-tool may read meta-data. This query is transformed into a set of constraints and a granularity vector. These constraints first pass through a pipeline of query optimizers, which rewrite the query to eliminate redundancies and exploit parallelism. The rewritten set of constraints then enters the query evaluator, together with the granularity vector. The query evaluator uses the meta-data to transform the query into position vectors, which correspond to the spatial locations of the facts to be retrieved from the data warehouse. The position vectors are passed in parallel to storage managers to retrieve the necessary data. These storage managers are wrapper objects which provide a multidimensional interface to data warehouses and cached sub-cubes. The results of the query are reassembled in the query evaluator, transformed into a list of tuples and passed back to the user-interface application for presentation. A cache object is created and a reference to this object stored in the meta-data repository.

**sets of constraints**, which determine the subset of the data cube in a *query*, **position vectors**, which describe the spatial positions of facts, and **sets of tuples**, which encode the query results.

#### 4. Modules

The data flow between the modules of our OLAP architecture is given in Figure 3. In this section we describe these modules, which communicate using a software bus, in more detail. The modules are treated in approximately the same order as they are used in the processing of queries.

##### 4.1. Query Generation/User-Interface

A graphical user-interface allows the user to perform the customary OLAP operations: drill-down (zooming in), roll-up (zooming out) and pivoting (changing the dimensions displayed at the axes). Ad-hoc queries are supported using a simple navigator interface.

The queries are transformed into a set of constraints and

a granularity vector according to the following rules:

**Drill-down.** A component of the granularity vector is replaced with one of its successors. A constraint may be added at the same time to reduce the visible domain.

For example, given the granularities from Figure 2(a) and the key-attributes from Figure 2(b): Assume, the user currently views data at the DECADE level and notices an anomaly within the aggregated data for the decade 199x. To investigate further, a drill-down occurs to the next more detailed granularity (YEAR). To evaluate this query, the time-component of the granularity vector, which initially points to DECADE, is replaced with YEAR. Now imagine, the user sees that the anomaly is caused by some event in the year 1998. To track the problem down, only the months of 1998 are required. The granularity will be adjusted to MONTH and an additional constraint YEAR equals 1998 is introduced. Or the user may choose to display the data for all JUNE's of a given DECADE. Such a query may start at the DECADE granularity, with a constraint on the value of the DECADE only. When this drill-down is issued, JUNE is added to the set of constraints and the relevant component of the granularity vector is set to MONTH. As a result of this operation, the values for DECADE and MONTH are constrained, while YEAR is unconstrained.

**Roll-up.** All constraints on the current granularity in the dimension, where the roll-up occurs, are deleted. The component of the granularity vector corresponding to this dimension is replaced with one of its predecessor according to a traversal history or default traversal paths. A roll-up from YEAR to DECADE clears all constraints set for YEAR and set the time-component in the granularity vector to DECADE.

**Pivot.** One dimension is replaced with another. Neither the granularity vector, nor the set of constraints are modified. The pivoting operation affects solely the way data is presented to the user requiring no reevaluation of the query.

##### 4.2. Query Optimizer

Every query first passes through the *query optimization* module where a set of constraints is mapped to a set of optimized constraints. Actually, multiple of these modules can exist concurrently—in that case, the query is piped through all of them and they act as filters making it simple to evolve optimization strategies.

##### 4.3. Query evaluator

The *query evaluator* module is responsible for evaluating queries, i.e. for creating position vectors from the constraints and the meta-data, which can be used as input to the storage managers. For every query, an new instance of this module can be instantiated. These modules may run on different machines improving the overall scalability of the system as queries can be processed in parallel.

A number of algorithms offer themselves for query evaluation. We use an algorithm, which closely follows the

model described in Section 3. We initialize a set of permissible instances with the  $\top$ -element. Then the contents of this set are replaced with the successors of the elements within the set which satisfy all given constraints. This expansion process is repeated, until only elements at the level of detail which was specified by the relevant component of the granularity vector remain. The Cartesian product over the permissible instances of all dimensions is the set of position vectors. The position vectors are passed on to the storage managers. Which storage manager is to be used, is determined using the meta-data repository. If the query result is cache-able, a cache object is created and a reference to it is stored in the meta-data repository. For example: we can assume a query which requires a granularity of MONTH in the time dimension and has a constraint “YEAR equals 1999”. During expansion, we first replace the  $\top$ -element with the parts it consists of (which will have the type DECADE). In our case that is just 199x (the value of DECADE is unconstrained). Now we iterate this replacement process, first yielding a set of YEARS and finally the requested MONTHs. This set will contain 1999 and finally May 1999 and June 1999, respectively.

#### 4.4. Meta-data repository

The *meta-data repository* is a catalog of information, describing the structure of the data warehouse. For relational data warehouses, it basically contains a description of the scheme of each relation and information on how to map the different data granularities to tables in the data warehouse. Apart from other metadata, we keep information on the currently cached data cubes together with their location in the meta-data storage. Basically, our meta-data repository maintains a key-value table which provides read-only access for all modules and write access for the query evaluator only to register newly created cache entries.

Currently, every machine maintains a read-only replica of the meta-data storage. Cache meta-data represents a special type of meta-data, which can be written at will, but remains local to every workstation. Our near-term plans include a new communication model, which will allow us to maintain multiple meta-data repositories and notify all of them, when a new (remote) cache object becomes available.

#### 4.5. Storage Managers and Cache objects

The OLAP engine needs to access data from various data sources like relational data warehouses, multidimensional data warehouses and the local cache. We create wrapper modules to access all these different data sources in a uniform way by serving as *storage managers*. These provide multidimensional abstraction to the actual data source used. A request to retrieve data is always expressed as a position vector and the result is always encoded as a set of tuples. This position vector denotes the spatial location of the requested value within the data cube. The storage manager

transforms the position vector into a query which is meaningful to the actual storage method used (e.g., a SQL query).

Caching is an integral part of our architecture as we cache all query results to exploit the temporal locality of reference in OLAP applications during user-interaction. We represent each cached cube by a separate *cache object*. Basically, such an object is simply another storage manager. It wraps around a previous query result and presents itself to the query evaluator according to the generic interface.

#### 4.6. Presentation

The query evaluator returns the query result as a set of tuples to the application which provides the user interface and constructs queries from user interaction. The user-interface is primarily concerned with the presentation of the data received from the OLAP engine and supporting navigation and data exploration.

### 5. Summary and Conclusion

We presented an architecture for modular OLAP systems with parallel and distributed query evaluation using CORBA objects. We distribute multiple queries across multiple processors/workstations by dynamically starting additional query evaluation modules. Within the query evaluator, both the constraint-application process as well as the fact retrieval support parallelization. The constraints are applied concurrently to each dimension and fact retrieval is carried out in parallel. The architecture we presented is well suited for distributed and parallel query optimization and reflects a formal model of evaluating OLAP queries. This model is based on sets, which allows the use of efficient, parallel algorithms. It is used in a prototype system currently under development at our institute.

### References

- [1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R\* tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD'90*, pages 322–331, 1990.
- [2] G. E. Blelloch and M. Reid-Miller. Fast Set Operations Using Treaps. In *Proc. 10th Annual ACM SPAA*, 1998.
- [3] L. Böszörményi and H. Kosch. High performance sets. In *Proc. High Performance Computing 98*, pages 972–975, 1998.
- [4] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: An IT-Mandate. Technical report, E. F. Codd and Associates, 1993.
- [5] M. Freeston. The BANG file: A new kind of grid file. In *Proc. SIGMOD'87*.
- [6] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of the Int. Conf. on VLDBs*, pages 562–573, 1995.
- [7] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Datawarehouses*. John Wiley & Sons, Inc., 1996.
- [8] A. Kurz and A. M. Tjoa. Integrating executive information systems and data warehouses. In *Proc. Int. Conf. On Business Information Systems*, 1996.
- [9] D. B. Lomet and B. Salzberg. The HB-tree: A multi-attribute indexing method with good guaranteed performance. In *Proc. ACM Symposium on Transactions of Database Systems*, volume 15, pages 625–658, 1990.
- [10] O. Mangisengi, A. M. Tjoa, and R. R. Wagner. Metadata for data warehouses using extended relational models. In *Third IEEE Meta-Data Conf.*, 1999.
- [11] R. Seidel and C. R. Aragon. Self-adjusting binary trees. *Algorithmica*, 1996.
- [12] J. Widom. Research problems in data warehousing. In *Conference on Information and Knowledge Management*, 1995.
- [13] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Workshop on Materialized Views*, pages 26–33, 1996.
- [14] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proc. SIGMOD'97*, pages 159–170, 1997.