TUM

INSTITUT FÜR INFORMATIK

Improving OLAP Performance by Multidimensional Hierarchical Clustering

Volker Markl Frank Ramsak Rudolf Bayer

TUM-I007
März 00

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-03-I007-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Improving OLAP Performance by Multidimensional Hierarchical Clustering

Volker Markl[*] *Frank Ramsak*[+] *Rudolf Bayer*[*+]

[*]Bayerisches Forschungszentrum
für Wissensbasierte Systeme
Orleanstraße 34
81667 München
Germany
☎ +49-89-48095-191
Fax +49-89-48095-203

volker.markl@forwiss.de

[+]Institut für Informatik
Technische Universität München
Orleanstraße 34
81667 München
Germany

{ramsak, bayer}@in.tum.de

Abstract

Data-warehousing applications cope with enormous data sets in the range of Gigabytes and Terabytes. Queries usually either select a very small set of this data or perform aggregations on a fairly large data set. Materialized views storing pre-computed aggregates are used to efficiently process queries with aggregations. This approach increases resource requirements in disk space and slows down updates because of the view maintenance problem. Multidimensional hierarchical clustering of OLAP data overcomes these problems while offering more flexibility for aggregation paths. In addition it has the potential to replace several bitmap indexes which are used to efficiently process high selectivity queries. We investigate query processing in OLAP environments and identify typical query patterns. Clustering is introduced as a way to speed up aggregation queries without additional storage cost. We also show the potential of multidimensional hierarchical clustering to reduce storage and maintenance cost for indexes. Clustering possibilities for OLAP data are investigated. The UB-Tree and the Tetris algorithm as physical storage structure and access method for clustered OLAP data are described. Performance and storage cost of our access method are investigated and compared to current query processing scenarios. In addition performance measurements on real world data for a typical star schema are presented.

1 Introduction

Data processing in data warehousing (DW) applications uses drill-down operations as well as slicing and dicing according to several dimensions. For this reason multidimensional data models, multidimensional query languages and even multidimensional DBMS (MDBMS) have been developed by the research community and implemented as commercial products. To a large extent, relational DBMS are used for decision support applications, since these systems are well researched and are reported to provide more efficiency for huge databases than MDBMS. Regardless, whether a multidimensional or relational paradigm is used to model and query OLAP data, queries result in multidimensional range restrictions in combination with sort operations and aggregations. Therefore any DBMS storing OLAP data must efficiently handle this typical query pattern.

Pre-computation, clustering and indexing are common techniques to speed up query processing. Precomputation results in the best query response time at the expense of load performance and secondary storage space. For DW applications, pre-computation is mostly discussed for aggregation operations

[CD97]. However, one requirement of DW is to efficiently deal with ad-hoc queries. Then, deciding which queries to pre-compute becomes extremely difficult. Pre-computation also leads to a view maintenance problem.

Indexing is used to efficiently process a query if the result set defined by the query restrictions is fairly small. Most OLTP applications use B-Trees as their standard indexing scheme. Favoring retrieval response time over update response time allows to build several indexes on one table or data cube of a DW. Bitmap indexes are widely discussed as an improvement over B-Trees for DW applications, since they efficiently evaluate queries with multi-attribute restrictions. However, the overall result set still must be relatively small. This is a major drawback of bitmap indexes, since usually a relatively large part of a cube must be accessed in order to calculate aggregated measures.

Clustering places data that is likely to be accessed together physically close to each other. The goal of clustering is to limit the number of disk accesses required to process a query by increasing the likelihood that query results have already been cached. Clustering has been well researched in the field of access methods. B-Trees, for instance, provide one-dimensional clustering. Multidimensional clustering has been discussed in the field of multidimensional access methods. See [GG97] and [Sam90] for excellent surveys of almost all of these methods.

The contribution of our paper is to apply multidimensional clustering and indexing using UB-Trees to a DW scenario. A clustering scheme for multiple hierarchical dimensions is defined, so that OLAP queries result in multidimensional range queries. The boundaries of each cluster in multidimensional space reflect the boundaries of a query with hierarchical restrictions. In contrast to usual multidimensional clustering schemes most queries will be handled locally in a cluster. Next to a cost analysis and comparison to other access methods we present measurements on a relational DW for a fruit juice company using a star schema with a fact table of 26 million records (an overall size of 7 GB). On this real-world data we experienced a performance increase up to a factor of ten compared to traditional techniques.

The rest of the paper is organized as follows: Section 2 surveys related work. In section 3 we describe our terminology and identify a standard query pattern for OLAP queries. Section 4 introduces *multidimensional hierarchical clustering*, the *UB-Tree* for a multidimensional partitioning and the *Tetris algorithm*, an efficient query processing method for OLAP queries. Chapter 5 gives a rigid cost analysis of our approach and compares it to traditional query processing scenarios. Section 6 presents performance measurements on real world data. Section 7 draws conclusions and gives an outlook on future work.

2 Related Work

The new requirements and research issues set by OLAP applications are summarized in [Wid95, WB97]. Besides the questions of data management (e.g., data cleansing, data maintenance) there are two issues of great importance. First, the question of providing a ‘good’ data warehouse architecture combining a conceptual, a logical, and a physical data model. An overview of the most popular models can be found in [BSH+98]. All these approaches have in common that they are based on a multidimensional data model. On the logical and physical level two main streams have been established – ROLAP that is based on the relational model and MOLAP that uses MDBMS.

The second important issue is the question of performance optimization. Due to the completely different query characteristics of OLAP applications in comparison to OLTP new questions have to be addressed here. The performance problem is heavily linked to the physical data model.

The index selection problem for ROLAP application is widely discussed in the research community [GHR+97, Sar97]. Especially bitmap indexes have been proposed to speed up ROLAP applications because of their compactness and support of star joins [CI98]. A common way of performance improvement is the usage of materialized views - often in combination with indexing methods [Moe98, WB98]. Due to the large number of possible views a selection problem exists besides the maintenance issue [Gup97, SDN+96, SDN+98]. Clustering of OLAP data plays a key role in providing good performance. Clustering has been well researched in the field of access methods. B-Trees [BM72], for instance, provide one-dimensional clustering. Multidimensional clustering has been discussed in the field of multidimensional access methods. See [GG97] and [Sam90] for excellent surveys of almost all of these methods. [ZSL98] addresses the issue of hierarchical clustering for the one-dimensional case.

3 Processing OLAP Queries

On the conceptual level a multidimensional (MD) view on the data models has been established by academia and the industry for OLAP applications [CD97]. In the MD model the numeric (quantitative) data (*measures*) (e.g., sales, cost) which is the focus of the analysis is organized along multiple *dimensions*. The dimensions provide categorical (qualitative) data (e.g., container size of a product), which determines the context of the measures. Therefore the measures can be seen as a value in a multidimensional space – one often refers to this model as a multidimensional *cube*. An important concept of OLAP data models is the notion of *dimension hierarchies*. Hierarchies are used to provide structure to the otherwise flat dimensions. Often the data in the dimensions can be categorized/classified according to some additional characteristics (e.g., shops could be classified according to their location). Usually OLAP users are not interested in the single measures but in some form of summarized data (e.g., sales in a certain area). Hierarchies provide an appropriate method of describing the level of aggregation for a dimension.

Typical OLAP operations are Drill-down, Roll-up and Slice-and-Dice [Kim96] and usually multiple dimensions are restricted at the same time. In general one can state that these operations in a MD model lead to range restrictions on the lowest hierarchy level of each dimension [Sar97].

3.1 The Physical Data Model

In the following we are concentrating on ROLAP where the conceptual MD model is mapped onto a relational database schema. The most established relational data models for OLAP applications are the *star schema* and the *snowflake schema*. In both approaches there is a central fact table that contains the measures and the dimension tables are situated around it. The connection between a fact tuple and the corresponding dimension members is realized via foreign key relationships. In the star schema the dimension tables are completely denormalized while in the snowflake schema they may be normalized. Queries usually contain restrictions on multiple dimension tables (e.g., only sales for specific customer group and for a specific time period are asked) that are then used as restrictions on the usually very large fact table. This operation (*star join*) is typical for such models. In ROLAP hierarchies are usually modeled implicitly by a set of attributes A_1, ..., A_n where A_i corresponds to hierarchy level *i*. We call such a sequence of attributes *hierarchically dependent*.

3.1.1 Running Example: The 'Juice & More' Schema

In this paper the following schema of the beverages supplier 'Juice & More', a real customer of one of our project partners[1] will serve as running example. In the data warehouse of 'Juice & More' data is organized along the following four dimensions: CUSTOMER, PRODUCT, DISTRIBUTION and TIME. Figure 3-1a shows the hierarchies over the dimensions (the number in parentheses specifies the maximal number of level members).

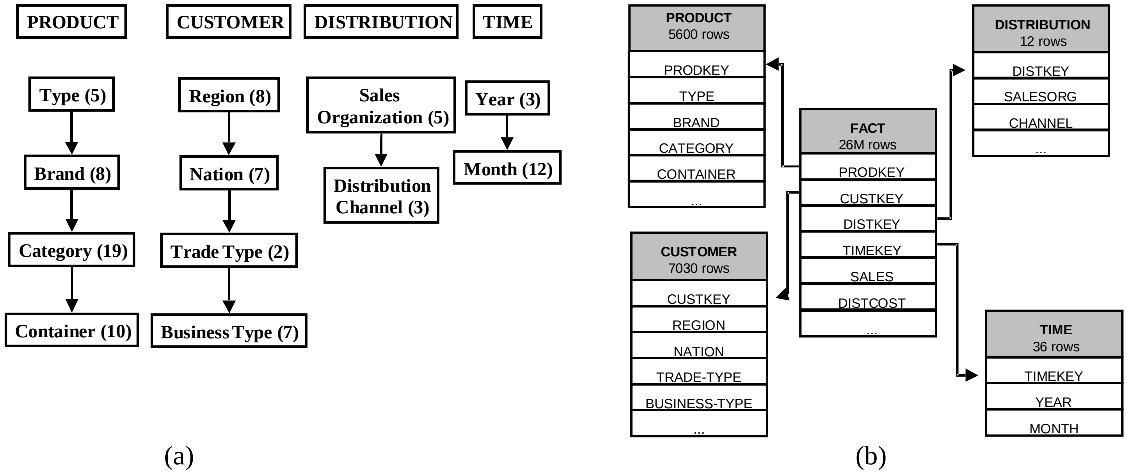


Figure 3-1 Hierarchies in the 'Juice & More' schema and the corresponding star schema

The ROLAP data model for the 'Juice & More' schema (Figure 3-1b) is a typical star schema with one fact table FACT and a table for each of the 4 dimensions. Let 'SALES' and 'DISTCOST' be some of the measures in the fact table.

3.1.2 Queries on the 'Juice & More' Schema

In the following we present typical queries that are taken from real applications for the schema given in the previous section. We will use these queries to illustrate our approach and we will present performance measurements for exactly these queries section 6.

[1] The company and the data presented here has been made anonymous.

Query 1 (Q1): This query computes the sales for a given product group (TYPE and BRAND specified) and a given customer group (NATION and REGION specified) for the months from October to December of 1993.

```
SELECT      SUM(SALES)
FROM        FACT F, CUSTOMER C, PRODUCT P, TIME T
WHERE       F.PRODKEY = P.PRODKEY AND F.CUSTKEY = C.CUSTKEY AND
            P.TYPE = $X_1$ AND P.BRAND = $X_2$ AND
            C.REGION = $Y_1$ AND C.NATION = $Y_2$ AND
            F.TIMEKEY = T.TIMEKEY AND T.YEAR = 1993 AND
            T.MONTH >= October AND T.MONTH <= December
```

Query 2 (Q2): This query calculates the cost of distribution of the products of type X for each distribution channel.

```
SELECT      SALESORG, CHANNEL, SUM(DISTCOST)
FROM        FACT F, DISTRIBUTION D, PRODUCT P
WHERE       F.DISTKEY = D.DISTKEY AND
            F.PRODUCTKEY = P.PRODUCTKEY AND
            P.TYPE = X
Group By    D.SALESORG,D.CHANNEL
```

Query 3 (Q3): This query restricts all dimensions on the first level of the hierarchies.

```
SELECT      SUM(SALES)
FROM        FACT F, DISTRIBUTION D, PRODUCT P, CUSTOMER C, TIME T
WHERE       F.DISTKEY = D.DISTKEY AND F.TIMEKEY = T. TIMEKEY AND
            F.CUSTKEY = C.CUSTKEY AND F.PRODKEY = P.PRODKEY AND
            P.TYPE = T AND D.SALESORG = S AND T.YEAR = Y AND
            C.REGION = R
```

4 Clustering OLAP data

Clustering places data that is likely to be accessed together physically close to each other. The goal of clustering is to limit the number of disk accesses required to process a query by increasing the likelihood that query results have already been cached. For a page size of *p* tuples this allows a speed up by a factor of *p* when queries retrieve large result sets. Thus for 30 tuples per page one can expect a clustered access to be up to 30 times faster than random access.

Since symmetrical clustering with respect to several dimensions is hard to achieve, most physical OLAP storage models either use non-clustering indexes like secondary B-Trees [Inf97] or cluster data with composite B-Trees [Red97]. The most prevalent OLAP data structure are bitmap indexes (e.g., [OQ97]). Bitmap indexes are useful, if multiple restrictions in low cardinality attributes like REGION or BRAND result in a very small selectivity (i.e., ratio of result set size and table size) for the conjunctive restriction. However, bitmap indexes are non-clustering secondary indexes which for small result sets may require a random access for every tuple. For large result sets (i.e., as soon as the selectivity of a query exceeds 1/*p*) they may require an access to every page of the table in worst case.

Figure 4-1 shows how bitmap indexes process a query that calculates the total sales of customers in Asia for distribution organization “TM”.

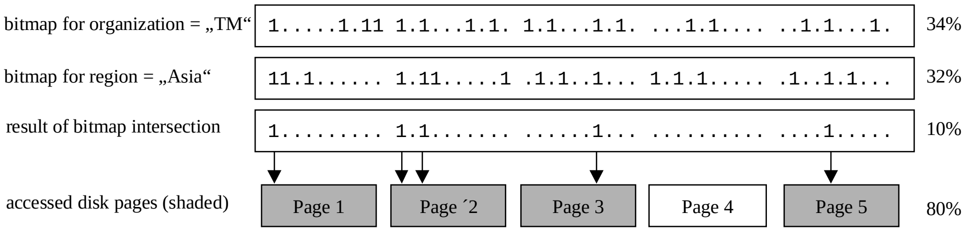


Figure 4-1 Bitmap Index Intersection

For each restriction, the bitmap is retrieved from the corresponding bitmap index. After intersecting these two bitmaps by a bitwise AND-operation the tuples corresponding to 1-bits are retrieved. In the figure we assume p = 10 tuples to fit on one page, thus ten consecutive bits correspond to the tuples on one disk page. The selectivities for both dimensions are 32% respectively 34%, resulting in an overall selectivity of 10%. Since the data is not clustered on the pages, the query needs to retrieve 80% of the fact table to retrieve 10% of the tuples.

In practice this ratio is even worse: Actual values for p range between 20 and 400 for 8kB pages. For the ‘Juice & More’ data warehouse the actual value is p = 30. Therefore bitmap index intersection might result in a full table scan already, when the conjunctive selectivity of a query exceeds 3,33%.

4.1 A Formal Description of Hierarchies on Dimensions

For our definition of multidimensional hierarchical clustering we use a set concept to formally define hierarchies: A dimension $\mathbb{D}$ consists of a base type having a set of values $\mathbb{V} = \{v_1,...,v_n\}$. A *hierarchy* of *depth h* over $\mathbb{D}$ is an ordered set of levels, i.e., $H=\{\mathbb{L}_0 ,..., \mathbb{L}_h\}$ (see Figure 4–2). Each *hierarchy level i* of H over $\mathbb{D}$ is a set of sets $\mathbb{L}_i = \{ m_1^i, ..., m_j^i \}$ with $m_k^i \subseteq \mathbb{V}$ for k=1,..,j. Each $m \in \mathbb{L}_i$ is a member set (or member) of the hierarchy of level i containing all members of a category. Usually a member m is assigned a name *label*(m) (e.g., ‘Orange Juice’ for m_1^1) instead of enumerating all values $v_k \in m$. The subset relationship $\subseteq$ between the members of two neighboring levels $\mathbb{L}_i$ and $\mathbb{L}_{i+1}$ defines a hierarchical relation (i.e., partial ordering) between the levels (e.g., the product ‘OJ0,7L’ is in the product category ‘Orange Juice’). Increasing the level of a hierarchy increases the *granularity* of the categorization, i.e., the data is classified according to finer categories.

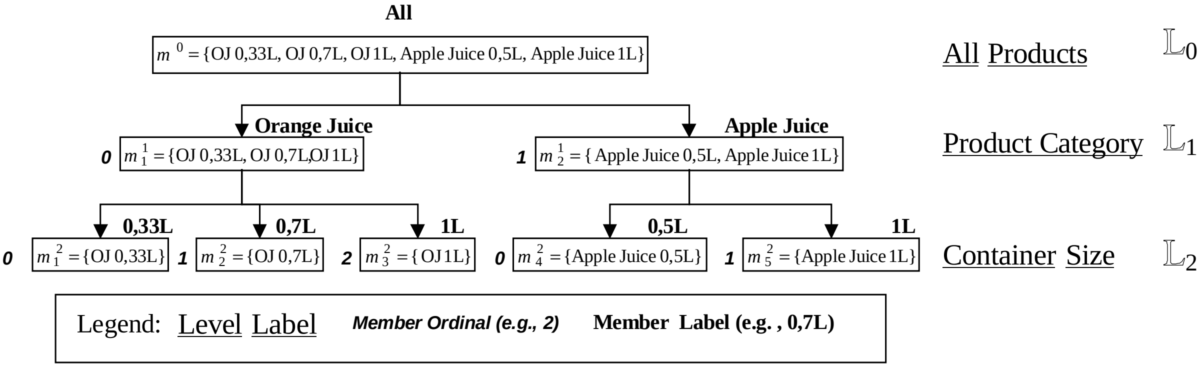


Figure 4-2 Example Hierarchy in Member Set Representation

With the base set $\mathbb{V}$ as the only member of level $\mathbb{L}_0$ (i.e., $\mathbb{L}_0 = \{\mathbb{V}\}$) a hierarchy H builds a hierarchy tree[2] with the root level $\mathbb{L}_0$. The nodes of H are the hierarchy members (or member labels) connected by edges which are defined by the subset relationship between members of neighboring levels. The *children* of a member m_k^i of level i are all members m_l^{i+1} of the lower level i+1 that are subsets of m_k^i, i.e., children(m_k^i)={ $m_l^{i+1} \in \mathbb{L}_{i+1} \mid m_l^{i+1} \subseteq m_k^i$ } (e.g., the set {{'Apple Juice 0,5L'},{ 'Apple Juice 1L'}} is the children set of 'Apple Juice'). The *parent* of a member m_k^i of level i then is the member m_l^{i-1} of the upper level i-1 that is a superset of m_k^i, i.e., *parent*(m_k^i)={ $m_l^{i-1} \in \mathbb{L}_{i-1} \mid m_l^{i-1} \supseteq m_k^i$ } (e.g. 'Orange Juice' is the parent of 'OJ 0,7L').

The bijective function ord_m defines a numbering scheme for the children of a member m of H. Ord_m assigns each subset (child) of m a number between 0 and the total number of children of m i.e.,

$$ord_m : children(m) \to \{0, \ldots, |\, children(m) \,| - 1\}$$

(see Figure 4–2 for an example).

Hierarchies should never relate members of different dimensions, since dimensions are independent and thus such a hierarchy could be split up in two separate hierarchies (see section 4.2.2).

Although there are quite some discussions about more complex hierarchy graphs in the research community [McGuff96a], we have not found n:m-relationships to be relevant for our project partners. The only exception to this is the issue of slowly changing dimensions, i.e., an object may change its hierarchy path over the course of time.

4.2 Multidimensional Hierarchical Clustering

For queries with large result sets one-dimensional clustering reduces disk accesses by a factor of p. Clustering of one-dimensional objects and single object hierarchies has been discussed to a large extent (e.g., [ZSL98], [BK89], [MS86], [Sal88]). However, OLAP queries often impose restrictions with respect to hierarchies over multiple dimensions. The result set satisfying these restrictions is usually quite large; for presentation it is grouped and aggregated or ranked. Clustering data with respect to multiple hierarchies can substantially speed up these operations.

If the order of dimensions during drill down is known in advance, clustering the data in this order will result in a good query performance. In principle, a concatenated clustering index (i.e., B*-Tree) on the hierarchy levels of all dimensions in *one* lexicographic order is maintained. However, with d dimensions with h_i hierarchy levels over dimension i, there are $(\Sigma_{i=1}^{d} h_i)! / \Pi_{i=1}^{d} (h_i!)$ possible lexicographic orderings. For the 4-dimensional 'Juice & More' cube (with 4 hierarchy levels for product, 4 for customer, 2 for distribution and 2 for time) there are 207900 possible orderings. Thus there is a high probability that the pre-defined clustering order will not be very useful for a particular query.

[2] We will explain how to deal with complex hierarchies (i.e., directed acyclic graphs) in section 4.2.2. Formally these hierarchies are modeled by dropping the requirement on H to be an ordered set of levels. Neighboring levels are then defined by coarsest refinement [Mar98].

MDBMS use multidimensional arrays to physically cluster data. However, for non-aggregated data this often leads to sparsity problems, which are discussed in more detail in section 4.3. Multidimensional access methods as commonly used in spatial DBMS provide multidimensional clustering in order to efficiently answer multidimensional range queries. In combination with a suitable hierarchy encoding scheme these methods can be used to significantly speed up OLAP queries.

4.2.1 Encoding Hierarchies by Surrogates

Many attributes in relational DBMS in general and in data warehouses in particular have an actual domain of a very small set of values. A typical example is the attribute REGION of the dimension table CUSTOMER of ‘Juice & More’, which has an actual domain of 8 values. However, a much longer character string is used to store the regions.

Definition 4-1 (enumeration type): We call the data type of an attribute to be an *enumeration type*, if its actual domain $\mathbb{A}$ consists of a relatively small finite set of values.

In order to maximize the entropy of an enumeration type $\mathbb{A}$ we define an order preserving one-to-one map f and its inverse function f^{-1}:

$$f : \mathbb{A} \rightarrow \mathbb{N}_0, \text{ so that for } a, b \in \mathbb{A}\text{: } f(a) <_{\mathbb{N}} f(b) \Leftrightarrow a <_{\mathbb{A}} b$$

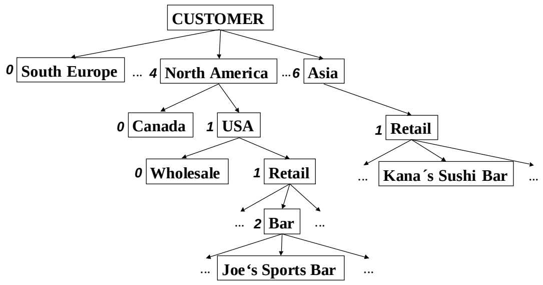
If there is no reasonable ordering on an enumeration type (e.g., it does not make sense to ask for REGION < “middle Europe”), we drop the requirement on f to be order preserving and merely require:

$$f : \mathbb{A} \rightarrow \mathbb{N}_0,\ f \text{ injective}$$

We call f a *surrogate function* for an enumeration type. For each value $a \in \mathbb{A}$ we call $f(a)$ the *surrogate* of a. For a very compact representation we number surrogates in sequential order. Figure 4–3a lists the values of the enumeration type REGION and the corresponding surrogates.

region	f(region)
South Europe	0
Middle Europe	1
Northern Europe	2
Western Europe	3
North America	4
Latin America	5
Asia	6
Australia	7

(a)



(b)

Figure 4-3 Surrogates for REGION and the entire Customer Hierarchy

To efficiently encode hierarchies, we introduce the concept of *compound surrogates* for hierarchies [Mar98]. Since we require hierarchies to form a disjoint partitioning, a uniquely identifying compound surrogate for each child node of a hierarchy member exists and can be recursively calculated by concatenating (∘) the compound surrogate of the member with the running number of the child node as calculated by the surrogate function *ord* from section 4.1. Thus, for a member m^i of hierarchy level i of hierarchy H we define its compound surrogate:

$$cs(H, m^i) = \begin{cases} ord_{father(m^i)}(m^i) & \text{, if } i = 1 \\ cs\left(H, \text{father}(m^i)\right) \circ ord_{father(m^i)}(m^i) & \text{, otherwise} \end{cases}$$

The hierarchy path North America → USA → Retail → Bar (Figure 4–3b) has the compound surrogate:

ord_{Customer} (North America) ∘ $ord_{\text{North America}}$ (USA) ∘ ord_{USA}(Retail) ∘ ord_{Retail}(Bar) = 4∘1∘1∘2.

The upper limit of the domain for surrogates of level i is calculated as the maximum fan-out (number of children) of all members of level i–1 of a hierarchy H, i.e.,

$$\text{surrogates}(H, i) = \max \{\text{cardinality}(\text{children}(H, m)) \text{ where } m \in \text{level}(H, i - 1)\}$$

A path Φ through a hierarchy of depth h is specified by a list of members $m^1, ..., m^h$, where m^i is a member of hierarchy level i. With $l_i = \lceil \log_2 \text{surrogates}(H, i) \rceil$ a *fixed length compound surrogate* can be stored in a very compact way by binary encoding.[3]

$$cs(H, \Phi) = \text{cs}(H, m^h) = ord_{father(m^1)}(m^1) + ord_{father(m^2)}(m^2) \cdot 2^{l_1} + ... + ord_{father(m^h)}(m^h) \cdot 2^{l_1 + l_2 + ... + l_{n-1}}$$

This formula leads to the compound surrogate $cs(H, \text{Bar}) = 1000011010_2 = 538$.

Usually growth expectations for a hierarchy are known well in advance. Often hierarchy trees are even static. Therefore it is possible to determine a reasonable number of bits for storing each surrogate of the compound surrogate of a hierarchy. Since hierarchies trees grow exponentially, the overall number of bits necessary to store a compound surrogate is relatively small. For instance, a hierarchy tree with four branches on 8 levels already represents $4^8 = 65536$ partitions and is stored by 16 bits.

The maximum length of the compound surrogates for 'Juice & More' can be computed from the maximum fan-out of the hierarchy levels given in Figure 3-1a. For any of the 4 hierarchies the maximum length of the compound surrogate does not exceed 15 bits and thus can be stored in a single integer value.

The lexicographic order on the hierarchy levels is preserved by this very compact fixed length encoding. Point restrictions on upper hierarchy levels result in range restrictions on the finest granularity of a hierarchy. For instance, the point restriction NATION = "USA" on the second level of the CUSTOMER hierarchy with f("North America") = 4 = 100_2 and f("USA") = 1 = 001_2 maps to the range restriction cs_{customer} between 528 = 1000010000_2 and 543 = 1000011111_2. Thus, a star join with this surrogate encoding for the foreign keys of a fact table results in a range restriction on each compound surrogate, if some hierarchy level of each dimension is restricted to a point (e.g., customer region = "USA"). In the same way intervals on the children of one hierarchy level result in a range of

[3] In general we use *variable length compound surrogates* that need $l_i(m) = \log_2 |\text{children}(m)|$ bits to store the surrogate for any child of m. However, since the hierarchy of 'Juice & More' is quite balanced (i.e., most hierarchy members have the same number of children), we chose *fixed length compound surrogates* for clustering the 'Juice & More' fact table.

the corresponding compound surrogates (e.g., year = 1998 and month between April and June). A star join on a schema with *d* dimensions creates a *d*-dimensional interval restriction on the fact table.

4.2.2 Dealing with complex Hierarchy Graphs

If two levels of a hierarchy graph are linked by several paths, there are several possibilities to define a hierarchy tree and therefore several ways to calculate the compound surrogates for physical clustering:

- If the order on the lowest level of granularity is identical for two hierarchy paths, then one path can be derived from the other path by an order preserving function on the lowest level of granularity. Then the clustering order for both hierarchy paths is identical. Thus, the clustering order for WEEK and MONTH in Figure 4–4a is identical. Both can be computed by an order preserving function from DAY, the lowest granularity level of the TIME hierarchy.
- If the query profile is known, the most useful path of the hierarchy graph used for restrictions, sort operations or grouping should be chosen. Thus, if in Figure 4–4b queries on CUSTOMER usually restrict REGION and NATION, this path should be chosen for clustering.
- If the query profile is not known, all paths of a hierarchy graph may be used for clustering, since hierarchies may be used for restrictions independently during drill-down. For clustering the different paths then can be considered to be independent dimensions. In the hierarchy graph of Figure 4–4b both the REGION hierarchy and the CUSTOMER hierarchy might be used for clustering. However, this approach increases the clustering dimensionality and thus should be used with care.

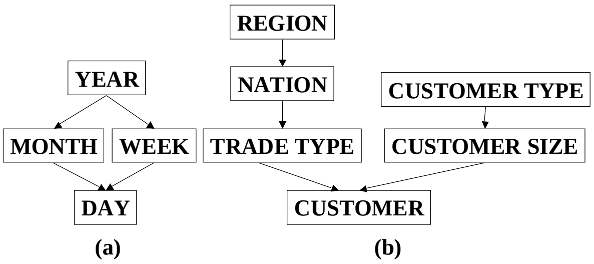


Figure 4-4 Complex Hierarchy Graphs

Other issues in the context of complex hierarchies are unbalanced hierarchies, slowly changing dimensions and multiple inheritance. Unbalanced hierarchies occur, if some hierarchy members have more child levels than others. This means, that the compound surrogates of Joe's Sports Bar and Kana's Sushi Bar in Figure 4–3b have different lengths. Using variable length compound surrogates or padding the shorter compound surrogate with zero bits solves this problem without any impact on clustering.

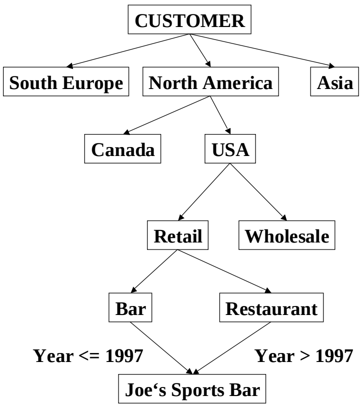


Figure 4-5 Change of a hierarchy over the time

Slowly changing dimensions can be addressed by marking each node of a hierarchy tree with a validity time interval. An object is physically clustered and retrieved with respect to its validity time. Reorganization of the physical clustering is not necessary: Even with a new classification upon a certain point of time the existing clustering should be correct from a historic perspective. If the business type of Joe's Sports Bar changes from bar to restaurant in 1998 (cf. Figure 4–5), all previously clustered data still is correct. The total sales over all bars in 1997 must include Joe's Sports Bar, whereas it is included in restaurants for 1998. However, each object of a hierarchy needs information about re-classification in order to correctly calculate the total sales to Joe's Sports Bar over the last years.

Multiple inheritance (e.g., Joe's Sports Bar is considered to be both a bar and a restaurant at the same time) is solved similarly to slowly changing dimensions: One of the several possible paths to a hierarchy node is chosen for clustering. The other paths of a hierarchy graph to that object then merely store a pointer to the sub-tree that actually stores the object. If multiple aggregation paths are possible, precautions must be taken that only one of these paths is used for aggregation.

4.3 Addressing Sparsity

Sparsity is defined as the percentage of a domain that is not existent in the actual domain. For a multidimensional data cube sparsity is the ratio between the number of cells not containing any data and the overall number of cells of a data cube. Some OLAP tools allow to mark dimensions to be sparsely populated and then specially handle them. However, a multidimensional cube is formed as the cross product over the domains of all dimensions. Therefore, even for non-sparse dimensions the sparsity of the entire cube becomes extremely high soon. The 'Juice & More' schema, for instance, is a star schema with four independent dimensions with a sparsity of 99,8%:

$$\text{sparsity(Juice \& More)} = 1 - \frac{26\,\text{Mio}}{7030 \cdot 5600 \cdot 36 \cdot 12} = 0{,}9984 \text{ with sparsity(star schema)} = 1 - \frac{|\text{Fact Table}|}{\prod_{i=1}^{d} |\text{Dim Table } i|}$$

To our knowledge sparsities of more than 99% are typical for data warehousing applications. The TPC-D benchmark [TPC97], for instance, can be regarded to be a snowflake schema with shared hierarchies consisting of three independent dimensions:

- part + supplier (combined dimension with 0.8 million records coming from 0.2 million parts from 10 thousand suppliers)
- customer + order (combined dimension with 1.5 million orders from 150 thousand customers)
- time (2557 records for seven years on the aggregation level of a single day)

For a fact table of 6 million records (a TPC-D scaling factor of 1) the resulting data cube has a sparsity of more than 99,99999%.

Thus, in practice sparsity forbids to materialize an entire data cube of raw data. Physical data organization in a multidimensional array is only feasible for highly aggregated data. However, serious decision support applications require a deep drill down into interesting areas of a data cube. Therefore it is necessary to have a physical representation of a sparsely populated data cube that allows efficient access to some part of that cube. With multidimensional hierarchical clustering drill down defines a subspace of a data cube by range restrictions in several dimensions. Therefore a method to cluster sparse data with respect to several dimensions in combination with an efficient range query and sort algorithm are necessary for efficient handling of drill down queries.

The surrogate calculating function of section 4.2.1 can use any multidimensional access method to implement multidimensional hierarchical clustering. However, using any variant of R-Trees [Gut84, BKS+90, BKK96] may result in a sub-optimal performance, since R-Trees may subdivide the universe into overlapping tiles, which may result in multiple accesses to one disk page. Therefore the most interesting candidates are Grid-Files [NHS84], hB-Trees [LS90] or space filling curves in combination with one-dimensional access methods [OM84, Jag90]. All of these methods provide a disjoint partitioning of multidimensional space. Because of its inherent hierarchical data space organization and its easy implementation, we use the UB-Tree [Bay96] for the ‘Juice & More’ data warehouse.

4.4 Concept of the UB-Tree

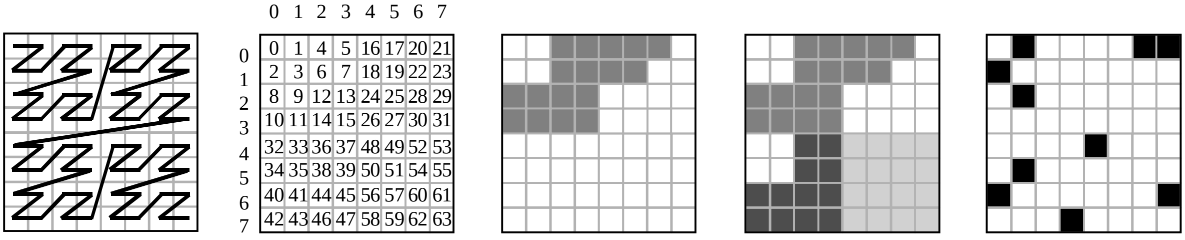
The UB-Tree [Bay96, Bay97a] is an access method for multidimensional point data and thus copes with sparsity without any additional overhead. It utilizes a space filling curve to create a hierarchical disjoint partitioning of a multidimensional universe while preserving multidimensional clustering. Using the Lebesgue-curve (Z-curve, Figure 4-6a) it is a variant of the zkd-B-Tree [OM84]. By the virtue of compound surrogates from section 4.2.1 for each dimension, the UB-Tree creates a multidimensional hierarchical clustering. This clustering is efficiently exploited by the UB-Tree range query algorithm [Bay96, Mar98] to answer queries with point or range restrictions in multiple hierarchies.

To define the UB-Tree partitioning scheme we need the notion of Z-addresses and Z-regions. We assume that each attribute value x_i of attribute A_i of a d-dimensional tuple $x = (x_1,...,x_d)$ consists of s bits[4] and we denote the binary representation of attribute value x_i by $x_{i,s-1}x_{i,s-2}...x_{i,0}$.

A *Z-address* $\alpha = \mathrm{Z}(x)$ is the ordinal number of a tuple x on the Z-curve and is calculated by interleaving the bits of the attribute values:

$$\mathrm{Z}(x) = \sum_{j=0}^{s-1} \sum_{i=1}^{d} x_{i,j} \cdot 2^{j \cdot d + i - 1}$$

A *Z-region* $[\alpha : \beta]$ is the space covered by an interval on the Z-curve and is defined by two Z-addresses α and β.



	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

(a) (b) (c) (d) (e)

Figure 4-6: Z-addresses and Z-regions

[4] Our implementation uses different lengths for the binary representation of attribute values [Mar98]. We just use identical lengths for an easy illustration.

For an 8×8 universe, i.e., $s = 3$ and $d = 2$, Figure 4-6b shows the corresponding Z-addresses. Figure 4-6c shows the Z-region [4: 20] and Figure 4-6d shows a partitioning with five Z-regions [0 : 3], [4 : 20], [21 : 35], [36 : 47] and [48 : 63].

The UB-Tree utilizes a B^*-Tree to partition the multidimensional space into Z-regions, each of which is mapped onto one disk page. At insertion time a full Z-region $[\alpha : \beta]$ is split into two Z-regions by introducing a new Z-address γ with $\alpha < \gamma < \beta$. γ is chosen so that the first half (in Z-order) of the tuples stored on Z-region $[\alpha : \beta]$ is distributed to $[\alpha : \gamma]$ and the second half is stored on $[\gamma : \beta]$. Thus a worst case storage utilization of 50% is guaranteed. Z-addresses are only used to organize the data and are not stored with every individual tuple, but only in the non-leaf nodes of the B^*-Tree. There is some freedom of choice for the Z-region split. For optimal query performance the split algorithm for UB-Trees tries to maintain rectangular regions and minimize fringes whenever possible. Assuming a page capacity of 2 points Figure 4-6e shows ten points which created the partitioning of Figure 4-6d.

The UB-Tree requires logarithmic time (in the number of actual values in the data cube) for the basic operations of insertion, point retrieval and deletion, and storage requirements are also linear. Range queries are processed by retrieving all Z-regions that intersect the query box and thus linearly depend on the result set size.

The problem of the zkd-B-Tree when handling tuples with identical leading bits in some attributes [LS90] does not occur for multidimensional hierarchical clustering: The leading bits of each dimension belong to the top level hierarchies and therefore partition the data space with respect to that dimension. Since the interleaving order of bit-interleaving hierarchically organizes the data space, the boundary of each Z-region exactly reflects the hierarchy over each dimension. The first hierarchical split levels correspond to the upper nodes of the hierarchy tree. Therefore a query box defined by hierarchical restrictions over a multidimensional hierarchically clustered UB-Tree will contain most Z-regions completely. Only very few Z-regions will be partly intersected by a query box (see Figure 4-7). Thus the retrieval overhead is minimal; almost all data being retrieved is part of the result set.

4.5 Processing OLAP Queries on Multidimensionally Clustered Data - The Tetris-Algorithm

Sort operations in combination with multi-attribute restrictions are the most important operations that are necessary to implement drill down queries in data warehousing applications. UB-Trees provide a single operator for efficiently processing this class of queries, the so-called Tetris algorithm [MB98].

The Tetris algorithm is a generalization of a multidimensional range query algorithm that efficiently combines sort operations with the evaluation of multi-attribute restrictions. The basic idea is to use the partial sort order imposed by a multidimensional partitioning in order to process a table in some total sort order. Essentially a plane sweep [PS85] over a query space defined by restrictions on a multidimensionally partitioned table is performed. The direction of the sweep is determined by the sort attribute. Initially the algorithm calculates the first Z-region that is overlapped by the query box, retrieves it and caches it in main memory. Then it continues to read and cache the next Z-regions with respect to the requested sort order, until a complete thinnest possible slice of the query box (in the sorting dimension) has been read. Then the cached tuples of this slice are sorted in main memory, returned in sort order to the caller and removed from cache. The algorithm proceeds reading the next slice, until all Z-regions intersecting the query box have been processed. Only disk pages overlapping the query space are accessed. With sufficient, but modest, cache memory each disk page is accessed only once (see section 5).

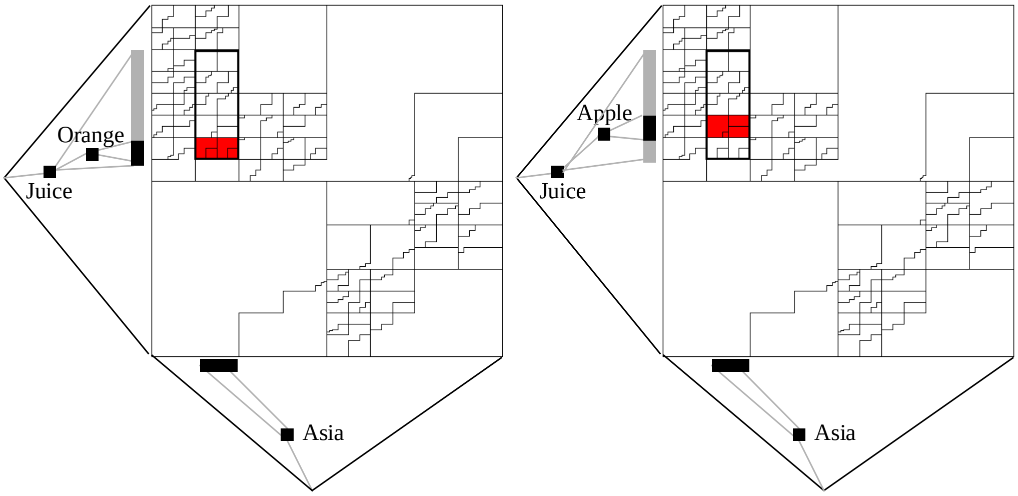


Figure 4-7: Processing a query box in sort order with the Tetris algorithm

Figure 4-7 illustrates how the Tetris algorithm processes a hierarchically clustered relation to calculate the total sales for each different fruit juice for all customers in Asia. The restriction of the NATION to 'Asia' results in an interval in the CUSTOMER dimension. The same holds for the restriction to 'Juice' for PRODUCT. The boundaries of each query interval correspond to Z-region boundaries and thus minimize the number of Z-regions only partly not being whollycontained in the query box. The query box is read in sort order from bottom to top; the aggregates for each juice type are calculated on the fly. The part of each Z-region from which tuples are cached is shaded. When all Z-regions intersecting the 'Orange Juice' slice have been read, this slice is sorted and aggregated. In the same way the next slices ('Apple Juice', 'Cherry Juice', etc.) are processed. This continues until the entire product interval defined by the restriction to 'Juice' has been handled. We named this algorithm *Tetris algorithm,* since the visualization of the Z-region processing order reminds us of the Tetris computer game.

The Tetris algorithm avoids external sorting, since only sublinear temporary storage is required with respect to the number of tuples to sort. In addition, I/O-time is linear in size of the query result set. In contrast to a standard merge-sort algorithm sorting is no longer a blocking operation. Thus aggregations can be calculated on-the-fly and allow better interactive response times. In addition, the Tetris algorithm efficiently processes iceberg queries for ranking [FSG+98], if the desired measure is used as a further dimension of the UB-Tree. The Tetris algorithm then does not read the entire query box in the sorting dimension, but terminates after processing the first slices. A detailed description and analysis of the Tetris algorithm can be found in [MZB98].

4.6 Materializing Aggregates

Multidimensional hierarchical clustering is not only applicable to the raw data itself, but can also be used to organize views with materialized aggregates. Higher aggregation levels result in a UB-Tree with shorter compound surrogates or reduced dimensionality. It makes sense to store pre-computed aggregates for the highest aggregation levels with restrictions in only one dimension, e.g., the total sales on a yearly basis. However, multidimensional hierarchical clustering allows to derive many aggregates efficiently from the raw data. This avoids materialization of many aggregation levles and thereby reduces the view maintenance problem for summary tables to a large extent.

5 Performance Analysis

For retrieving or sorting a relation in combination with multidimensional hierarchical restrictions we define cost functions for response times and intermediate temporary storage. Our analysis considers a UB-Tree, a composite secondary index (CSI, clustering B^*-Tree) over all attributes (foreign keys of each dimension and measure attributes) of a fact table, a single secondary index (SSI, non-clustering B^*-Tree) on the attribute with the least selectivity and a full table scan (FTS). In addition we analyze the performance of bitmap index intersection (BII), which combines the bitmaps of each restricted attribute to determine the result set of the query.

5.1 Cost Functions for Retrieval with Multi-Attribute Restrictions

An FTS to answer multidimensional range queries with selectivity s_j in dimension j can exploit prefetching techniques to reduce the number of random page accesses at the expense of having to read the entire table. Using a CSI with a composite B^*-Tree in lexicographic order A_1, ..., A_d allows to use the index for the restriction in A_1 at the expense of having a random access for each page. A SSI on A_j requires a random page access for each tuple satisfying the restriction in A_j, since no clustering of the tuples is available. The number of random accesses of a SSI is limited to P, if the row identifiers of the SSI are sorted and then processed in physical page order for data page retrieval. For point restrictions on the index attribute (e.g., REGION = ‘Asia’), sorting of row identifiers may even be avoided: index pages for tuples with identical index attributes may be organized in the physical order of the row identifiers. Then point restrictions will get a list of row identifiers sorted according to the physical location of the tuple. This makes a SSI not to degenerate and behave similarly to an FTS in worst case.

BII requires a random access for each tuple satisfying the restrictions in all attributes. In addition the corresponding part of each bitmap index has to be retrieved. In analogy to a SSI the result of BII is a bitmap which is used to access data pages in physical order. Thus multiple random accesses to one data page will not occur.

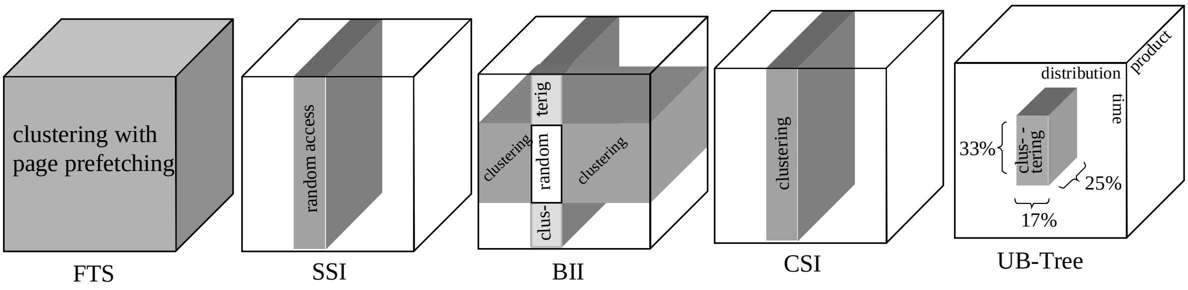


Figure 5-1: Access Methods and Clustering

The shaded part of each cube in Figure 5-1 shows the part of a three dimensional database which is retrieved by the corresponding access method to answer a query to compute the sales for one year (s_{TIME} = 33%) for all fruit juices (s_{PRODUCT} = 25%) sold by direct marketing ($s_{\mathrm{DISTRIBUTION}}$ = 17%): an FTS retrieves the entire database exploiting clustering and prefetching. In contrast to that a SSI will rarely utilize any clustering benefits for small result sets. BII retrieves each bitmap by clustered access, whereas the data itself will often be spread over many data pages and then must be retrieved by random access to each page. However, for larger result sets the probability rises that prefetching might be applicable for bitmap indexes. This means, that BII will not be much less efficient than an FTS in worst case. A CSI with the DISTRIBUTION dimension as first attribute in concatenation order utilizes clustering but only exploits the 17% restriction on DISTRIBUTION. In contrast to that the UB-Tree utilizes the restrictions of all dimensions and retrieves the data in a clustered way.

In accordance with [HR96] we use a cost model that takes random page accesses and page transfers into account. Let t_π be the (average case or worst case) positioning time of a hard disk, t_τ be the transfer time of a hard disk and t_ξ the CPU time[5] spent for page processing after retrieval. We assume that the prefetching strategy of the file system reads a physical cluster of C consecutive pages from disk with one random access into the read-ahead cache. This takes time $t_\pi + (t_\tau + t_\xi) \cdot C$. Reading k pages in consecutive order therefore takes

$$\mathrm{c}_{\mathrm{scan}}(k) = \lceil k/C \rceil \cdot t_\pi + \max(k, C) \cdot (t_\tau + t_\xi)$$

Using that cost model we calculate the cost of processing a fact table consisting of T tuples stored on P pages restricted by a multidimensional interval $Q = [[y, z]] = [y_1, z_1] \times ... \times [y_j, z_j] \times ... \times [y_d, z_d]$ with a selectivity of s_j in attribute A_j. For UB-Trees we assume d-dimensional hierarchical clustering of the table. For secondary indexes we assume B_j to be the size in pages of a secondary index on A_j. Figure 5-2 displays cost formulas for these access methods as derived in [Mar98]. With increasing result set size the function prefetch(T,P,s_1,...,s_d) in the cost function for BII will grow from 1 to C (see [Mar98] for details).

[5] Note that t_ξ heavily depends on the specific query. For complex restrictions like multidimensional intervals or IN-clauses of SQL t_ξ may be considerably higher than for simple single-attribute restrictions.

$$c_{\mathrm{FTS}}(d,P,C)=\left(t_\pi\cdot\frac{1}{C}+t_\tau+t_\xi\right)\cdot P$$

$$c_{\mathrm{CSI}}(d,P,s_1,\ldots,s_d)=\left(t_\pi+t_\tau+t_\xi\right)\cdot s_1\cdot P$$

$$c_{\mathrm{SSI\ on\ dimension\ }i}(d,T,s_1,\ldots,s_d,B_i)=\left(t_\pi+t_\tau+t_\xi\right)\cdot(\min(T\cdot s_i,P)+B_i)$$

$$c_{\mathrm{BII}}(d,t,s_1,\ldots,s_d,B_1,\ldots,B_d)=\underbrace{\left(t_\pi\cdot\frac{1}{C}+t_\tau+t_\xi\right)\cdot\sum_{i=1}^{d}\left(s_i\cdot B_i\right)}_{\text{bitmap retrieval \& intersection}}+\underbrace{\left(\frac{t_\pi}{\mathrm{prefetch}(t,p,s_1,\ldots,s_d)}+t_\tau+t_\xi\right)\cdot\min\left(T\cdot\prod_{i=1}^{d}s_i,P\right)}_{\text{tuple retrieval by random access (clustered access for large result sets)}}$$

$$c_{\mathrm{UB}}(d,P,y,z)=\left(t_\pi+t_\tau+t_\xi\right)\cdot\prod_{j=1}^{d}n_j(d,P,y_j,z_j)$$

Figure 5-2: Cost functions for retrieval of a multidimensional interval

For a table of P pages of uniformly distributed data partitioned by a d-dimensional UB-Tree and a query box Q = [[y, z]] the cost $c_{\mathrm{UB}}(d,P,y,z)$ is the product of the number of Z-regions intersecting the multidimensional interval [[y, z]] in each dimension as derived by [Mar98]. For each attribute A_j the cost function c_{UB} requires the values y_j and z_j to be normalized to [0, 1]. The formula

$$n_j(d, P, y_j, z_j) = n(y_j, z_j, l_j(d, P)) + ((n(y_j, z_j, l_j(d, P) + 1) - n(y_j, z_j, l_j(d, P))) \cdot p_j(d, P)$$

to calculate the number of Z-regions intersected by the restriction [y_j, z_j] in attribute A_j of a d-dimensional UB-Tree consisting of P pages requires the following auxiliary functions:

- $l_j(d,P)$ (actual number of completed recursive splits with respect to A_j)

$$l_j(d,P)=\begin{cases} l_{j\downarrow}(d,P)+1 & ,\text{if } \lfloor\log_2 P\rfloor \bmod d \le j \\ l_{j\downarrow}(d,P) & ,\text{otherwise}\end{cases} \qquad \text{where } l_{j\downarrow}(d,P)=\left\lfloor\frac{\log_2 P}{d}\right\rfloor$$

- $p_j(d,P)$ (probability of an incomplete recursive split in A_j)

$$p_j(d,p)=\begin{cases}\dfrac{P}{2^{\lfloor\log_2 P\rfloor}}-1 & ,\text{if } j=\left(\lfloor\log_2 P\rfloor \bmod d\right)+1 \\ 0 & ,\text{otherwise}\end{cases}$$

- $n(y_j, z_j, l_j)$ (number of Z-regions for l_j completed splits in A_j)

$$n(y_j,z_j,l_j)=\begin{cases}1 & ,\text{if } z_j=y_j \\ \left\lfloor z_j 2^{l_j}\right\rfloor-\left\lfloor y_j 2^{l_j}\right\rfloor+1 & ,\text{otherwise}\end{cases}$$

Our measurements have shown that this rather complicated cost function describes the actual behavior of the UB-Tree very accurately [Mar98].

5.2 Simulation of Response Times for Queries with Multi-Attribute Restrictions

Current operating systems usually prefetch C = 16 pages with one random access. We assume t_π = 10 [ms], t_τ = 0,6 [ms] and t_ξ = 0,4 [ms].

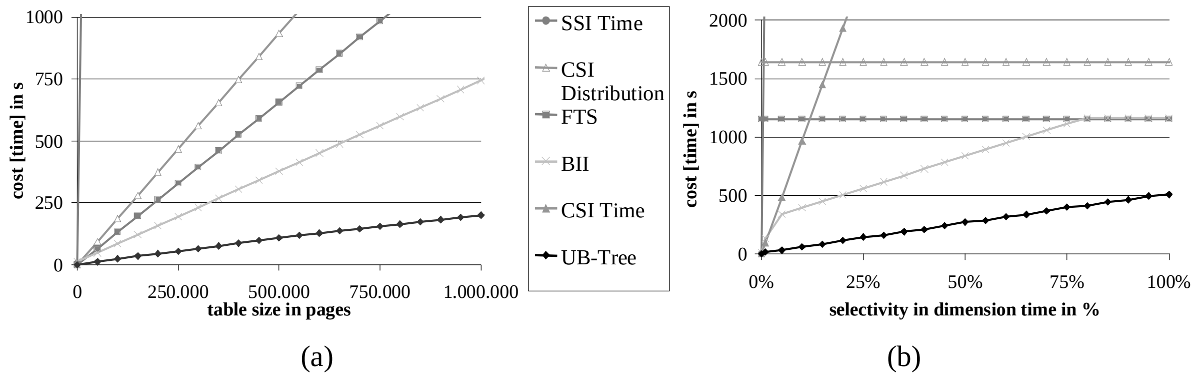


Figure 5-3: Response times for queries with multi-attribute restrictions on 'Juice & More' (simulation)

Using our cost functions Figure 5–3a shows the cost [in s] of the sales query (s_{TIME} = 33%, $s_{\text{DISTRIBUTION}}$ = 17%, s_{PRODUCT} = 25%, s_{CUSTOMER} = 100%, see section 5.1) for 4-dimensional hierarchical clustering compared to other access techniques. The table size is varied from one page up to one million pages.

Varying the selectivity of the restriction in TIME for a table size of *P* = 878k pages (about 7GB for a page size of 8kB) shows that multidimensional clustering with the UB-Tree is superior to both a SSI and a CSI on the time dimension, since these access methods cannot exploit any restriction but the one on time. UB-Trees can exploit the restrictions on DISTRIBUTION and PRODUCT in addition to the restriction in TIME. Thus UB-Trees are also superior to an FTS and to BII using bitmap indexes for all four dimensions (Figure 5–3b). For an overall selectivity of 75% · 17% · 25% · 100% = 3,1875 % an FTS is already preferable to BII. Since bitmap indexes do not cluster the data, the result set defined by the restrictions in all dimensions must be sufficiently small for BII to be competitive.

5.3 Cost Functions for Sort Operations

For the following considerations we assume a merge sort algorithm using a main memory of *M* pages and a merge degree of *m*. We divide the sort process in a retrieval phase (which retrieves the data to create initial runs for the merge-sort) and a sort phase (which actually performs the merge-sort). Because of the multi-attribute filtering of the retrieval phase the data set to be sorted is usually smaller than the entire table. With s_j denoting the selectivity of the restriction in attribute A_j and independence of the attributes, $P \cdot \Pi s_i$ pages need to be sorted. The cost functions of section 5.1 can be used to calculate the cost of the retrieval phase to create the initial runs for the sort operation. If an access method does not return the tuples in the requested sort order, sorting with the cost of c_{sort} takes place. If $M > P \cdot \Pi s_i$, sorting takes place in main memory. c_{sort} then is the cost of an internal sort operation. c_{sort} is zero, if a CSI with A_1 as first attribute is used for the retrieval phase and the sorting attribute is A_1, since the data then is already retrieved in the desired sort order.

$$c_{\text{sort}}(d,P,C,m,M,s_1,\ldots,s_d) = \begin{cases} t_\xi \cdot P \cdot \prod_{i=1}^{d} s_i \cdot \log\left(P \cdot \prod_{i=1}^{d} s_i\right) & \text{, if } M > P \cdot \prod_{i=1}^{d} s_i \\ \underbrace{\left(t_\pi \cdot \frac{1}{C} + t_\tau + t_\xi\right)}_{\text{consecutive access}} \cdot \underbrace{2}_{\text{read \& write}} \cdot \underbrace{\left(P \cdot \prod_{i=1}^{d} s_i\right)}_{\text{pages to sort}} \cdot \underbrace{\log_m\left(\frac{P}{M} \cdot \prod_{i=1}^{d} s_i\right)}_{\text{number of merge phases}} & \text{, otherwise} \end{cases}$$

As shown in the section before, SSI and CSI on restricted attributes are only efficient for fairly small result sets. In this case sorting would take place in main memory. One can expect that as soon as external sorting is necessary, SSI and CSI are not efficient for the retrieval phase anymore.

The Tetris algorithm has to sort each cached slice. Since the algorithm reads $n(y_j, z_j, l_j)$ slices, the overall cost of internal sorting according to A_j is

$$c_{\text{Tetris}}(d,P,y,z,j) = t_\xi \cdot n(y_j, z_j, l_j) \cdot cache_{\text{Tetris}}(d,P,y,z,j) \cdot \log cache_{\text{Tetris}}(d,P,y,z,j)$$

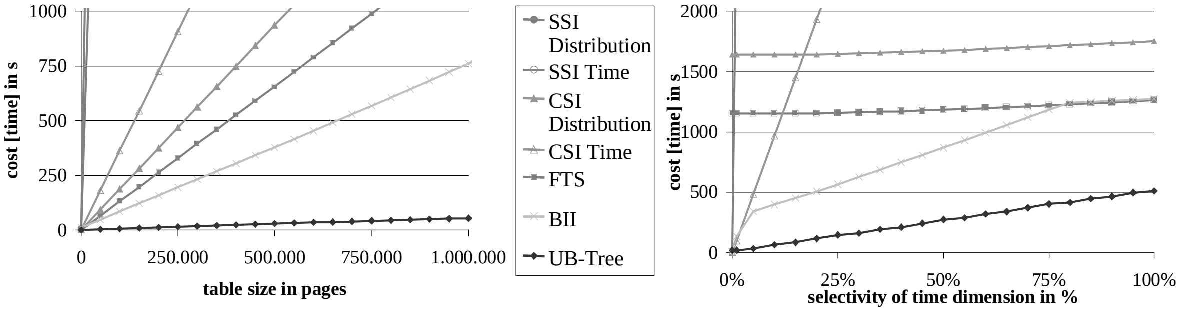
The Tetris cache is considerably smaller than the temporary storage of $P{\cdot}\Pi s_i$ required by the merge-sort algorithm that is necessary after the data has been retrieved by an FTS or any index on a restricted attribute. To sort A_j the Tetris algorithm just requires to cache one slice, i.e.,

$$cache_{\text{Tetris}}(d,P,y,z,j) = \prod_{\substack{i=1,\ldots,d \\ i \neq j}} n_i(d,P,y_i,z_i)$$

For a two-dimensional UB-Tree the above formula results in a square root function of the number of Z-regions overlapping the query box, i.e., $cache_{\text{Tetris}}(2, P, s_1, s_2, j) = \sqrt{P \cdot s_1 \cdot s_2}$.

5.4 Simulation of Response Times for Queries with Multi-Attribute Restrictions and Sort Operations

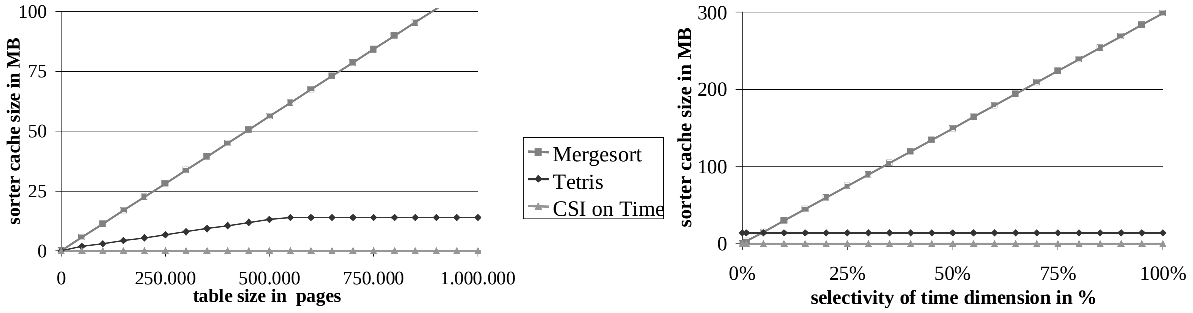
Using the same parameters as in section 5.2 and additionally using a main memory cache of 32 MB and a merge degree of m = 2 for the merge sort algorithm Figure 5–4 shows the cost [in s] for sorting the result set of a fact table defined by restrictions in multiple hierarchical dimensions.



(a) (b)

Figure 5-4: Sorting the restricted ‘Juice & More’ fact table according to the TIME dimension (simulation)

Figure 5–4a shows the cost [in s] for a query that computes the monthly sales over all customers (s_{CUSTOMER} = 100%) for one year (s_{TIME} = 33%) for all fruit juices (s_{PRODUCT} = 25%) sold by direct marketing ($s_{\text{DISTRIBUTION}}$ = 17%). This query requires sorting the 4-dimensional query box by month to calculate the monthly sales. Again the tables size is varied from one page to one million pages. The speed up of the Tetris algorithm for UB-Trees grows superlinearly with increasing table size, since all other access methods require an external merge sort to calculate the monthly groups. Varying the selectivity of the restriction in TIME for a table size of P = 878k pages in Figure 5–4b shows the superiority of the Tetris algorithm, since hierarchical clustering allows to exploit multi-attribute restrictions to reduce the number of random accesses and at the same time avoids an expensive external sort operation. Sorting with Tetris takes place in main memory as long as this memory suffices to hold one slice of the query box. Figure 5–5(a, b) shows that the temporary storage for the merge sort algorithm used by FTS, BII, CSI distribution and SSI distribution soon exceeds the main memory sorter cache of M = 32 MB when processing the queries of Figure 5–4(a, b). In contrast to that sorting with Tetris never requires more than 14 MB of cache for one slice and thus sorting can take place in main memory.



(a) (b)

Figure 5-5: Temporary storage required for the sorter cache (simulation)

A CSI or SSI on TIME does not require any sorter cache. The tradeoff of these two access methods is the inability to use restrictions in multiple dimensions. Overall, the Tetris algorithm for UB-Trees outperforms any access method either with respect to response time or with respect to both response time and temporary storage requirements.

5.5 Further Analysis

Using our cost functions we found out that multidimensional hierarchical clustering and the Tetris algorithm are superior to one-dimensional access methods, unless a strongly preferred sort order exists or the restrictions are not selective enough to make up the tenfold speed of an FTS. Another limitation of our technique is the number of dimensions: Increasing dimensionality exponentially reduces the potential of multidimensional space partitioning to create a total sort order in one dimension. Our theoretical and practical analysis shows that multidimensional indexes of up to 6 dimensions are handled very well with table sizes larger than 1 GB. These dimensionalities are typical for data warehousing applications. and in particular for the ‘Juice & More’ schema. This dimensionality is also sufficient for the TPC-D benchmark [MZB98]. With larger table sizes even further attributes could be added to the UB-Tree in order to speed up queries with restrictions or sorted processing in this attribute.

6 Performance Measurements

In this section we present measurements performed on the ‘Juice & More’ schema with our prototype implementation of the UB-Tree on top of the commercial Oracle8 Server. For the evaluation of our clustering technique we defined a benchmark with 36 queries. In comparison we also conducted measurements with native Oracle access methods: full table scan (FTS) and bitmap indexes (BII). For these measurements we used a completely denormalized fact table, that is, no additional joins had to be performed to answer the queries. The bitmap indexes were created on each hierarchy level. We did not include secondary indexes in our comparison measurements because earlier experiments showed that they are neither competitive to the UB-Tree nor to FTS or BII [MZB98].

6.1 Measurement Environment

The measurements were performed on a SUN Enterprise with four 300 MHz UltraSPARC processors and 2 GB RAM under Solaris 2.6. As secondary storage a partition on a SPARCstorage Array with Raid-Level 0 (6 disks striping, 5-6 MB/s transfer rate per disk) was used. All measurements were done in a single-user environment.

It is important to note that our implementation still causes significant overhead due to the fact that we have implemented the UB-Tree on top of a DBMS and not in the kernel itself. First, the number of SQL statements that have to be processed (UB: 1 statement for each page in the result set, Oracle methods: 1 statement in total) leads to extensive inter-process communication (about 30% of the total processing time) and DBMS overhead (e.g., parsing of statements). Second, our table is larger than the one for the FTS and the bitmap indexes due to unimplemented compressing techniques in the UB-Tree (for 8 KB pages: UB: 878362 pages, FTS: 723539 pages, BII: FTS+31134 pages).

6.2 Results

Table 6–1 shows the result set sizes and Figure 6–1 the response times of the three example queries (section 3.1.2). Q1 shows that the UB-Tree with multidimensional clustering is over 2 times faster as BII even for very small result sets. Q3 which is processed by the unoptimized UB-Tree at least 10 times faster than with any other access method undermines this observation.

Query	Loaded Tuples	Percentage of Database
Q1	8160	0,03%
Q2	1696416	6,52%
Q3	19752	0,08%

Table 6-1 Result Set Sizes

The result set of query Q2 is quite large but the almost perfect clustering factor of the UB-Tree (in average more than 29 out of 30 tuples/page belong to the result set) still leads to a speed up of more than 30 % in comparison to BII. The time for FTS for Q2 differs from the times for Q1 and Q3 due to the less complex WHERE clause of the statement. The number of comparison operations is therefore much smaller than for the other queries which causes the faster execution.

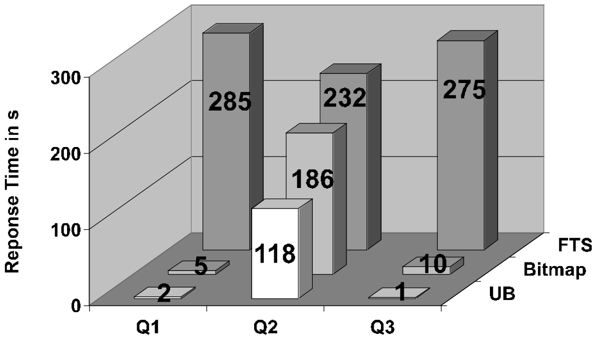


Figure 6-1 Response Times on Real Data

All these results on real data show how well the multidimensional hierarchical clustering with UB-Trees works in practice and the accuracy of our theoretical cost model. In total more than 77% of all benchmark queries (28 out of 36) showed a speed up between a factor of 1.3 and 10 over traditional techniques. In addition, the resource requirements of the UB-Tree were significantly less than these of native access methods (see [MZB98]) which makes it especially valuable in multi-user environments.

7 Conclusions and Future Work

We have defined an encoding scheme for hierarchical dimensions that allows to cluster the data with respect to multiple hierarchical dimensions. The proposed hierarchy encoding scheme relies on Z-ordering and ensures that the Z-region boundaries created by the hierarchical splitting of a UB-Tree exactly reflect the hierarchy over each dimension. The first hierarchical split levels correspond to the upper nodes of a hierarchy tree. Multidimensional hierarchical clustering reduces the number of random accesses to the fact table for star joins and other queries with restrictions in multiple hierarchies by a factor of *p*, where *p* is the page size in tuples. In addition, sort operations as necessary for grouping and aggregation are performed on the fly without additional I/O. For dimensionalities typical for data warehousing only I/O-time linear in size of the result set prior to aggregation and sublinear temporary storage are necessary to aggregate parts of a data cube. Thus secondary storage space and pre-computation time for many aggregates and bitmap indexes can be avoided. In addition the widely discussed view maintenance problem is minimized. The benchmark results for typical queries of a 7 GB real world retail data warehouse confirmed our analytical expectations and showed significant speedups up to factor 10 in response time. Depending on the query, temporary storage requirements for sorting are reduced by several orders of magnitude. Our clustering approach also holds not only for ROLAP but also for MOLAP implementations of a DW since both ROLAP fact tables and MOLAP data cubes can be clustered in this way.

In our future work we are particularly interested in doing tests in multi-user environments where we expect even more significant speedups. We are in the process of integrating the UB-Tree into a DBMS kernel to reduce the overhead of the current implementation. In addition we are investigating a methodology for query optimization with multidimensional indexes, both for heuristics-based and cost-based query optimizers.

Acknowledgments

We thank our project partners SAP, Teijin Systems Technology, NEC, Hitachi and the European Commission for funding this research work. In particular we thank our master student Roland Pieringer for his effort in doing the performance measurements reported in this paper and SAP for providing the real data and the test environment.

References

[BM98] Bayer, R.; Markl, V.: The UB-Tree: A Multidimensional Index and its Performance on Relational Database Management Systems, Technical Report, FORWISS 1998, submitted paper

[DKO+85] DeWitt, D.J.; Katz, R.H.; Olken, F.; Shapiro, L.D.; Stonebraker, M.R.; Wood, D.: Implementation Techniques for Main Memory Database Systems, Proc. ACM SIGMOD Intl. Conf. on Management of Data, 1984, pp. 1-8

[FNP+79] Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R.: Extendible Hashing – a fast access method for dynamic files. ACM Transactions on Database Systems 4(3), 1979, pp. 315 – 344

[Gra93] Graefe, G.: Query Evaluation Techniques for Large Databases, ACM Computing Surveys 25, pp. 73-170

[Gün93] Günther, O.: Efficient computations of Spatial Joins, Proc. 9th Int. Conf. on Data Engineering, Vienna, 1993

[HNK+90] Harada, L.; Nakano, M.; Kitsuregawa, M.; Takagi, M.: Query Processing Methods for Multi-Attribute Clustered Relations, Proc. 16th Int. Conf. on Very Large Databases, 1990, pp.59-70

[IBM97] IBM Corporation. IBM DB2 Universal Database for UNIX Documentation, IBM Corporation, 1997.

[Jag89] H.V. Jagadish. Incorporating Hierarchy in a Relational Model of Data. Proc. Of the 1989 ACM SIGMOD Intl. Conference on Management of Data, Portland, Oregon, 1989.

[JL98] Jürgens, M.; Lenz, H.J.: The R*a-Tree: An improved R*-Tree with Materialized Data for Supporting Range Queries on OLAP-Data, DWDOT Workshop, Vienna, 1998

[ME92] Mishra, P.; Eich, M.H.: Join Processing in Relational Databases, ACM Computing Surveys, Vol. 24 No.1, 1992, pp. 194-211

[Mer81] Merret, T.H.: Why Sort-Merge gives the best Implementation of the Natural Join, SIGMOD Rec??ord 13, 1981, pp. 39 – 51

[Ora97] Oracle Corporation. Oracle 8 Documentation, Oracle Corporation, 1997

[Rot91] Rotem, D.: Spatial Join Indices, Proc. Int. Conf. on Data Engineering, 1991, pp. 500-509

[Bay96] R. Bayer. The universal B-Tree for multidimensional Indexing. Technical Report TUM-I9637, Institut für Informatik, TU München, 1996

[Bay97a] R. Bayer. The universal B-Tree for multidimensional Indexing: General Concepts. - In: World-Wide Computing and Its Applications '97 (WWCA '97), Tsukuba, Japan, 10-11, Lecture Notes on Computer Science, Springer Verlag, March, 1997

[Bay97b] R. Bayer. UB-Trees and UB-Cache – A new Procesing Paradigm for Database Systems. Technical Report TUM-I9722, Institut für Informatik, TU München, 1997

[BK89] E. Bertino and W. Kim. Indexing Technique for Queries on Nested Objects. IEEE Transactions on Knowledge and Data Engineering, pages 196-214, 1989.

[BKK96] S.D. Berchtold, D. Keim, and H.-P. Kriegel. The X-Tree: An Index Structure for high-dimensional Data. Proc. 22nd Int. Conf. on Very Large Data Bases, 1996

[BKS+90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust Access Method for Points and Rectangles. ACM SIGMOD Intl. Conference on Management of Data, pp. 322 – 331. 1990.

[BKS93] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins using R-Trees, ACM SIGMOD Intl. Conference on Management of Data, pp. 237-246. 1993.

[BM72] R. Bayer and E. McCreight, E.: Organization and Maintainance of large ordered Indexes. Acta Informatica 1, 1972, pp. 173 – 189

[BSH+98] Blaschka M., Sapia C., Höfling G., and B. Dinter. Finding Your Way through Multidimensional Data Models. Proc. Intl. Workshop on Data Warehouse Design and OLAP Technology, Vienna, August 1998.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technologies. ACM SIGMOD Record 26(1),1997

[CI98] C. Chan and Y. Ioannidis. Bitmap Index Design and Evaluation. Proc. ACM SIGMOD Intl. Conf. On Management of Data, 1998.

[FSG+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing Iceberg Queries Efficiently. Proc. of the VLDB 98, pages 299 – 310. 1998.

[GG97] V. Gaede and O. Günther. Multidimensional Access Methods. Humboldt Universität, Berlin, 1997.

[GHR+97] H. Gupta, V. Harinarayan, A. Rajaraman, and D. Ullman. Index Selection for OLAP. Proc. Intl. Conf. on Data Engineering, 1997

[Gup97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. Proc. of the Intl. Conference on Database Theory, Athens, Greece, January 1997.

[Gut84] A. Guttman. R-Trees: A dynamic Index Structure for spatial Searching. ACM SIGMOD Intl. Conference on Management of Data, pp. 47 – 57. 1984.

[HR96] E.P. Harris and K. Ramamohanarao. Join algorithm costs revisited. VLDB Journal, 5, 1996.

[Inf97] Informix Software Incorporation. A New Generation of Decision Support Indexing for Enterprisewide Data Warehouses.

Http://www.informix.com/informix/corpinfo/zines/whitpprs/wpxps.pdf, 1997.

[Jag90] H.V. Jagadish. Linear Clustering of Objects with multiple Attributes. ACM SIGMOD Intl. Conference on Management of Data, pp. 332 – 342. 1990.

[Kim96] R. Kimball. The Data Warehouse Toolkit. John Wiley & Sons, New York. 1996.

[KKD89] W. Kim, C.Kim, and A. Dale. Indexing techniques for object-oriented database. In Object-Oriented Concepts, Databases, and Applications, pages 371-394. Addison-Wesley, 1989.

[LS90] D. Lomet and B. Salzberg. The hB-Tree: A Multiattribute Indexing Method with good guaranteed Performance. ACM TODS, 15(4), pp. 625 – 658. 1990.

[Mar98] V. Markl. The UB-Tree: A multidimensional access method and its application to relational database management systems. Ph.D. thesis draft, TU München, 1998.

[MB98] V. Markl and R. Bayer. The Tetris-Algorithm for Sorted Reading from UB-Trees. In “Grundlagen von Datenbanken”, 10th GI Workshop, Konstanz, 1998.

[McG96a] F. McGuff. Data Modeling Patterns for Data Warehouses. Comprehensive Systems Inc. July 1996.

[Moe98] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. Proc. of the 24th VLDB Conference. New York, USA, 1998.

[MS98] Microsoft Corporation. OLE DB for OLAP Programmer’s Reference. http://www.microsoft.com/data/oledb/olap. February, 1998.

[MZB98] V. Markl, M. Zirkel, and R. Bayer. Processing Operations with Restrictions in Relational Database Management Systems without external Sorting. Submitted paper, 1998.

[NHS84] J. Nievergelt, and H. Hinterberger, and K. C. Sevcik. The Grid-File. ACM TODS, 9(1), pp. 38-71, March 1984.

[OLA98] The OLAP Council. MDAPI TM The OLAP Application Program Interface Version 2.0 Specification January 1998.

[OM84] J. A. Orenstein and T.H. Merret. A Class of Data Structures for Associate Searching. Proc. ACM SIGMOD Intl. Conf. on Management of Data, Portland, Oregon, pp. 294-305, 1984.

[OQ97] P. O´Neill and D. Quass. Improved Query Performance with Variant Indexes. ACM SIGMOD Intl. Conf. On Management of Data, Tucson, Arizona, pp. 38-49,1997.

[PS85] F. P. Preparata and M. I. Shamos. Computational Geometry: An Introduction. Springer-Verlag, New York, 1985.

[Red97] Redbrick Systems. Star Schema processing for Complex Queries.

http://www.redbrick.com/rbsg/whitepapers/starJoin.pdf, 1997.

[Sal88] B. Salzberg. File Structures: An Analytic Approach. Prentice Hall, 1988.

[Sar97] S. Sarawagi. Indexing OLAP data. Data Engineering Bulletin 20 (1), pp. 36-43, 1997.

[SDN+96] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. Proc. of the 22nd VLDB Conference. Mumbai (Bombay), India, 1996.

[SDN+98] A. Shukla, P. Deshpande, and J. Naughton. Materialized View Selection for Multidimensional Datasets. Proc. ACM SIGMOD Intl. Conf. On Management of Data, 1998.

[TPC97] Transaction Processing Performance Council. TPC Benchmark D (Decision Support). Standard Specification, Revision 1.2.3. http://www.tpc.org. June, 1997.

[WB97] M.C. Wu, and A.P. Buchmann. Research Issues in Data Warehousing. BTW’97. 1997.

[WB98] M.C.Wu and A.P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. ICDE, Orlando, 1998.

[Wid95] J. Widom. Research Problems in Data Warehousing. Proc. of 4th CIKM, November 1995.

[ZDN+98] Y. Zhao, R. Deshpande, J. Naughton, and A. Shukla. Silmutaneous Optimization and Evaluation of Multiple Dimensional Queries. Proc. ACM SIGMOD Intl. Conf. On Management of Data, 1998.

[ZSL98] C. Zou, B. Salzberg, and R. Ladin. Back to the Future: Dynamic Hierarchical Clustering. Proc. of the ICDE 1998: 578 – 587, 1998.