Multi-Dimensional Database Allocation for Parallel Data Warehouses

Thomas Stöhr Holger Märtens Erhard Rahm

University of Leipzig, Germany
{stoehr | maertens | rahm}@informatik.uni-leipzig.de

Abstract

Data allocation is a key performance factor for parallel database systems (PDBS). This holds especially for data warehousing environments where huge amounts of data and complex analytical queries have to be dealt with. While there are several studies on data allocation for relational PDBS, the specific requirements of data warehouses have not yet been sufficiently addressed. In this study, we consider the allocation of relational data warehouses based on a star schema and utilizing bitmap index structures. We investigate how a multi-dimensional hierarchical data fragmentation of the fact table supports queries referencing different subsets of the schema dimensions. Our analysis is based on realistic parameters derived from a decision support benchmark. The performance implications of different allocation choices are evaluated by means of a detailed simulation model.

1 Introduction

Data warehouses integrate massive amounts of data from multiple sources and are primarily used for decision support purposes. They have to process complex analytical queries for different access forms such as OLAP (on-line analytical processing), data mining, etc. In addition, successful data warehouses tend to be used by many users so that the concurrent execution of multiple complex queries must be supported. Ensuring short query response times in such an environment is enormously difficult and can only be achieved by a combination of different approaches, in particular the use of preaggregated data [12,37], special aggregation operators like cube [13], special index structures such as bitmap indices [22,24] and parallel query processing.

While the first three approaches have been investigated extensively in recent work, surprisingly, parallel query processing tailored for data warehouses has received very little attention in the research community. In particular, we are not aware of research results on data allocation for parallel data warehouses, which is the main focus of this paper. Of course, basic approaches of traditional parallel databases [7] can be employed for relational data warehouses as well. However, these approaches can only achieve suboptimal performance as they do not utilize specific characteristics of the database organization and query types of data warehouses. Hence, high performance data warehousing should be based not only on specialized index structures but also on tailored approaches for parallel database processing.

We focus on relational data warehouses based on a star schema [5]. The database thus consists of a huge fact table and multiple dimension tables. Dimensions are hierarchically structured, e.g., to group months into quarters, years etc. Queries typically perform aggregations on the fact table based on selections among the available dimension levels (e.g. sales of all products from product group *x* during quarter *y*). Such *star (join) queries* can be efficiently supported by bitmap indices but still involve substantial processing and I/O cost. The paper focuses on the design and evaluation of suitable data allocation methods for the fact table and bitmap indices to allow an efficient parallel processing of star queries.

While our data allocation methods are applicable to all major PDBS (parallel database system) architectures, due to space constraints we concentrate on the „Shared Disk“ approach [7], which is supported by several commercial DBMS (*IBM DB2/OS390, ORACLE*). Shared Disk is particularly attractive for data warehousing because the read-dominated workloads largely eliminate the need for concurrency and coherency control, which are performance-critical for OLTP [21,28]. Furthermore, there is a high potential for parallel query processing and dynamic load balancing since each processing node has access to all disks allowing it to process any query or subquery [28]. Data allocation for Shared Disk refers to the placement of tables and index structures onto disks; there is no fixed assignment of data partitions to processing nodes. The disk allocation must support parallel processing and load balancing while limiting disk contention.

We propose a multi-dimensional hierarchical fragmentation of the fact table based on multiple dimension attributes. Such an approach permits a significant reduction of processing and I/O overhead for many queries by restricting the number of fragments to be processed for both the fact table and bitmap data. Such savings are achieved not only for the fragmentation attributes themselves but also for attributes at different levels of a dimension hierarchy. The proposed data allocation and processing model also supports parallel I/O and parallel processing as well as

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

load balancing for disks and processors. To find a suitable fragmentation for a given star schema and workload, we determine and analyze critical parameter thresholds to be considered. Furthermore, we have developed a comprehensive simulator of a parallel database system allowing a detailed performance evaluation of the new data allocation methods. Results of several experiments are presented for database and query parameters obtained from the decision support benchmark APB-1. The study leads to several guidelines that can be used by a database administrator or implemented within a tool to determine a physical data warehouse allocation.

The remainder of the paper is organized as follows. In the next section, we mention related work on data allocation and look at the approaches of commercial PDBS. In Section 3, we introduce the APB-1 star schema that has been used in our simulation study and will help illustrate the data allocation methods. We also discuss how bitmap indices are employed for processing of star queries. Section 4 then presents our multi-dimensional data allocation approach and an analysis of major performance factors. After an overview of our simulation approach (Section 5) we present performance results of various experiments for the proposed multi-dimensional database allocation in Section 6. We conclude in Section 7.

2 Related work

In general relational PDBS, data allocation is based on a horizontal fragmentation of tables, typically *round robin*, *hash*, or *range fragmentation* [7]. Round robin simply distributes rows in their insert order, while hash and range fragmentations are based on a partitioning function applied on the values of a fragmentation attribute. Such a one-dimensional fragmentation permits queries on the fragmentation attribute, which may be a concatenation of several attributes, to be restricted to a subset of the fragments, thereby reducing work. A number of studies analyzed one-dimensional data allocation strategies for Shared Nothing systems, e.g. [4,8,19]. In [9], a multi-dimensional range fragmentation was proposed for scan processing in Shared Nothing environments. Furthermore, there are numerous approaches for multi-dimensional declustering and access methods for spatial data [11]. However, all of these proposals do not exploit the hierarchical structure of dimensions (fragmentation attributes) and the specifics of star queries.

In [34], a multi-dimensional partitioning strategy on fact tables is proposed to achieve load balancing for star queries. The strategy requires exactly 2^k processing nodes and is based on sorting and splitting the fact table on every dimension. Again, Shared Nothing is assumed and hierarchical properties of star schemas are not considered.

Except for our own work, data allocation for Shared Disk systems has hardly been addressed in research papers. In [28,30] we have outlined the increased flexibility of the Shared Disk approach for allocating tables and index structures. In particular, the number of data partitions per table does not impact the communication overhead for query processing as for Shared Nothing. This supports high degrees of declustering even for medium-sized tables. Furthermore, index structures such as B-trees may be partitioned differently from tables (or not at all) without performance loss. In [31], the impact of simple data allocations on scan performance in single- and multi-user mode was evaluated. In [18], a new approach for declustering large intermediate query results on shared disks was proposed and analyzed. These studies did not consider the specific aspects of data warehousing.

Numerous studies dealt with physical data allocation at the disk level in order to support I/O parallelism, e.g., within disk arrays [17,16,6,32]. Without additional index structures, physical declustering of data does not typically allow the query optimizer to predict which disks have to be accessed for a query. Hence, queries cannot be restricted to a subset of the partitions as for logical fragmentations based on attribute values. Furthermore, employing intra-query parallelism can lead to disk contention between concurrently running subqueries of the same query.

Data allocation in commercial PDBS

Most commercial PDBS have specific support for data warehousing such as bit index structures and processing of star queries, e.g., *ORACLE8i* [25], *IBM DB2 UNIVERSAL DATABASE (UDB)* [3], *RED BRICK WAREHOUSE* [29], and the *ADVANCED DECISION SUPPORT OPTION* of *INFORMIX DYNAMIC SERVER* [14]. *INFORMIX* and *ORACLE* allow choosing a variety of fragmentation options including a multi-dimensional range fragmentation [15,26]. However, it remains unclear to what extent multi-dimensional fragmentation is exploited to reduce query work. Furthermore, the hierarchical structure of fragmentation attributes is not utilized. None of the aforementioned vendors provide sufficient information or even tool support on how to determine an adequate data allocation for star schemas. Some systems such as *NCR TERADATA V2R3* [2] and *MICROSOFT SQL SERVER 7.0* [10] do not yet support bitmap indices.

Both *DB2 UDB* and *ORACLE8i* can dynamically create and evaluate bitmaps derived from B-tree indices on the fact table. Furthermore, *SYBASE ADAPTIVE SERVER IQ* [35] employs a completely different data allocation and processing strategy based on vertical partitioning of tables and possible bit-slicing of single attributes (called *BIT-WISE INDEXING*). Although these approaches have their merits, their consideration is beyond the scope of this paper.

3 Star schema and star queries

While our approaches can be used for any star schema, we make our discussions more specific by using the APB-1 schema, which is presented first. We then discuss the use of bitmap indices for query processing in the central case (no intra-query parallelism). Parallel query processing will be explained in Section 4.

3.1 Star Schema used for evaluation

Figure 1 shows our star schema based on the *Analytical Processing Benchmark APB-1* that was proposed by the OLAP Council to benchmark relational OLAP systems [1]. The schema provides a typical sales analysis environment with one fact table (*SALES*) and the four dimension tables *PRODUCT*, *CUSTOMER*, *CHANNEL*[1], and *TIME*. Every attribute of the dimension tables refers to a different hierarchy level.

1. We understand the APB-1 dimension *CHANNEL* to describe distribution channels.

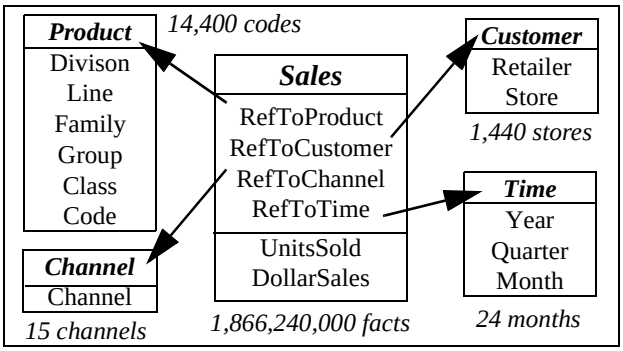

Fig. 1: Considered star schema (derived from APB-1

For instance, we have a 6-level product hierarchy differentiating several product divisions, each consisting of several product lines, each consisting of several product families, etc. Individual products are identified by product codes. As usual for star schemas, dimension tables are denormalized to reduce join overhead. The fact table *SALES* holds the measuring attributes *UnitsSold*, *DollarSales* and *Cost* for calculating aggregations. In addition, there is a foreign key per dimension each referring to the lowest hierarchy level. That is, each sales row refers to a specific product code, customer store, distribution channel and month.

APB-1 assumes a time frame of 24 months and scales the cardinality of the other dimension tables according to the number of channels. Our evaluations are based on a configuration of 15 channels resulting in the dimension cardinalities indicated in Figure 1. The cardinality of the fact table is determined by a *density factor* applied on the maximal number of possible value combinations (product of the dimension cardinalities). We used a density factor of 25% resulting in almost 2 billion fact rows. The benchmark also defines certain ratios between the attribute cardinalities within dimension hierarchies which we follow. Table 1 below shows the respective values for the *PRODUCT* dimension.

We assume typical star queries, also derived from APB-1, aggregating over one or multiple dimensions at different hierarchy levels. Expressed in SQL, a sample query called *1MONTH1GROUP* is

```
SELECT SUM(UnitsSold),SUM(DollarSales)
FROM Sales S, Product P
WHERE    S.RefToTime    = MONTH
AND      P.Group        = PRODUCTGROUP
AND      S.RefToProduct = P.Code
```

1MONTH1GROUP represents a two-dimensional star join query aggregating the measures *UnitsSold* and *DollarSales* for one product group within one month. One-dimensional queries used in our experiments are *1CODE*, *1MONTH* and *1STORE*, each referring to the specified hierarchy level.

3.2 Star query processing with bitmap indices

Such star queries involve a join between the fact table and one or more dimension tables. Standard join implementations such as hash join would require one or more full scans of the fact table. Due to the huge size of the fact table, such full scans are very costly and must be avoided whenever possible even when parallel scans can be utilized. This is also because for most queries, only a small fraction of the fact data is relevant.

Bitmap indices allow a much faster processing of star joins and selections on fact tables [22]. A standard bitmap index contains one bitmap per possible value of a given attribute; the bitmap consists of one bit per row to indicate whether or not that row matches the respective attribute value. A selection for a specific attribute value (e.g., *RefToTime=MONTH* in query *1MONTH1GROUP*) thus has to read only one bitmap to precisely identify all relevant fact rows. A variation are so-called *bitmap join indices* where each bitmap indicates which fact rows match an attribute value of the dimension table (via the respective foreign key). Hence, such bitmaps represent the precomputed result of a join between the fact table and a dimension table. For our sample query, a bitmap join index on product group would be beneficial. The join query can then be processed by reading and intersecting (AND-ing) the two bitmaps for *MONTH* and *PRODUCTGROUP*. The resulting bitmap represents the rows to be read from the fact table. Since the bitmaps are much more compact than the fact table and only the relevant fact rows are to be read, performance can be substantially improved compared to a full scan of the fact table. Of course, the sketched approach can be applied for an arbitrary number of query dimensions and multiple values per dimension; it is supported by several commercial DBMS. Note that more complex queries require additional processing steps (for instance, a join to the *PRODUCT* dimension may still be necessary if aggregation results are to be grouped by the product classes within a selected product group). This will not be considered further because the associated processing cost is typically much smaller than for fact table processing.

Simple bitmap indices become inefficient for high-cardinality attributes resulting in a large number of bitmaps and thus high storage overhead (which may be reduced by compressing the bitmaps). In this case, *encoded bitmap indices* [36] can help by encoding attribute values from a domain of size $|Dom|$ in approximately $\log_2|Dom|$ bits, thereby reducing the number of bitmaps necessary to represent the index. The trade-off is that selecting a single value in an encoded bitmap index requires finding a specific pattern in *all* of the bitmaps, rather than a single bitmap. This may, however, be ameliorated by an appropriate encoding. Specifically, [36] suggested an encoding scheme that represents individual hierarchy elements of star to support selections on the inner dimension levels.

For our study, we employ encoded bitmap join indices on the higher-cardinality dimensions *PRODUCT* and *CUSTOMER*. We utilize a hierarchical encoding that avoids a separate bitmap index per hierarchy level but only requires one index per dimension. This is illustrated in Table 1 for the *PRODUCT* dimension where we use separate bit sub-patterns to encode *DIVISION*s, *LINE*s *within DIVISION*s, *FAMILIE*s *within LINE*s etc. We only need 15 bits to identify a particular product code so that the index only consists of 15 bitmaps instead of 14.400 which would be needed for simple bitmaps. Locating all fact rows of a specific product code thus needs to evaluate 15 bitmaps (which we will access in parallel). The hierarchical encoding reduces the access cost for attributes at a higher level of the dimension hierarchy. For instance, *CODE*s (fact rows) belonging to the same *GROUP*

	DIVISION	*LINE*	*FAMILY*	*GROUP*	*CLASS*	*CODE*	total
#total elements	8	24	120	480	960	14,400	14,400
#elements within parent	8	3	5	4	2	15	
#bits for encoding ($\log_2$)	3	2	3	2	1	4	15
sample bit pattern	ddd	ll	fff	gg	c	oooo	dddllfffggcoooo

Table 1: Hierarchy representation in encoded bitmap join indices

share the same prefix (dddllfffgg) of the full bit pattern (dddllfffggcoooo) and can be precisely located with access to only 10 of the 15 bitmaps.

The encoded bitmap indices on *PRODUCT* and *CUSTOMER* need 15 and 12 bitmaps, respectively. For the low-cardinality dimensions *TIME* and *CHANNEL* we use simple bitmap indices consisting of up to 34 (24 for month, 8 for quarter, 2 for year) and 15 bitmaps each. This results in a maximum of 76 bitmaps for our configuration. As we will see, our multi-dimensional fragmentation permits eliminating some bitmaps, thus improving storage and access overhead.

4 Data allocation

In this section, we present our fragmentation and allocation strategy for star schemas supporting parallel query processing. We focus on the fact table and its bitmap indices. Dimension tables and their (B*-tree) indices usually cover only a very small fraction of the whole database so that they do not need special treatment. For instance, our four dimension tables only occupy 1 MB and can easily be stored on a single disk. Frequently accessed dimension data will automatically be cached in main memory.

Data allocation of the fact table and bitmap indices is determined in two steps. In the first step, we define a multi-dimensional horizontal fragmentation of the fact table resulting in *n* disjoint *fact fragments*. These fragments are our units for disk placement as well as for query processing. The fact table fragmentation is also to be applied to the bitmap indices meaning that each bitmap of any bitmap index is partitioned into *n bitmap fragments*. This ensures that the bits of a bitmap fragment refer to exactly one fact fragment and allows different fact fragments to be processed independently (in parallel). Of course, a bitmap fragment is much smaller than a fact fragment[2]. The second allocation step assigns all fragments onto disks. Typically, the number of fact fragments, *n*, is much higher than the number of disks, *d*, to support high degrees of parallelism and effective load balancing.

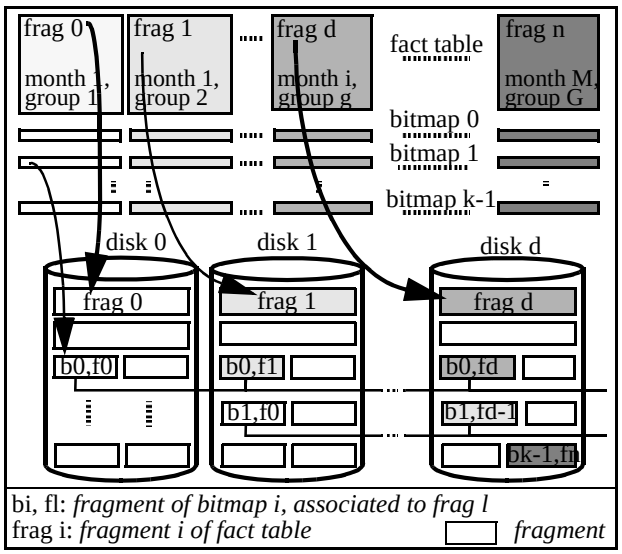

Fig. 2: Fragmentation and allocation of a star schema

Figure 2 illustrates our data allocation approach. A simple round robin allocation of fact fragments to the disks is used. Each of the *k* bitmaps (from all indices) is partitioned into *n* bitmap fragments. We place the *k* bitmap fragments belonging to the same fact fragment onto consecutive disks to enable intra-query parallelism for bitmap processing. For instance, if fact fragment *frag i* is placed on disk *j*, the associated bitmap fragments of all *k* different bitmaps are placed on disk $j, j+1, .. j+k-1 (\text{modulo} d)$ [3].

In Section 4.1, we will introduce our approach of multi-dimensional hierarchical fragmentation, followed by a discussion how it reduces query work and bitmap requirements (Section 4.2). After a brief discussion of our parallel processing strategy in Section 4.3, we present some basic thresholds for fragmentation parameters in Section 4.4. Section 4.5 quantifies the I/O cost for different types of queries, and Section 4.6 discusses the physical allocation of table and index fragments to disks. Finally, Section 4.7 lists a set of guidelines for multi-dimensional star schema declustering derived from these considerations.

4.1 Multi-dimensional fragmentation of the fact table

To reflect the inherent multi-dimensional and hierarchical organization of star schema data and queries, we propose a ***multi-dimensional hierarchical fragmentation*** called ***MDHF*** for partitioning the fact table. It allows choosing multiple *fragmentation attributes* from different *hierarchy levels* of the dimension tables. Each fragmentation attribute refers to a different dimension. For each fragmentation attribute – and thus for each dimension – a range partitioning can be specified consisting of disjoint value ranges for the attribute's domain. For completeness, the union of the value ranges must cover the whole domain. As in general multi-dimensional range fragmentation [9], a fragment then consists of all (fact) tuples belonging to one value range per fragmentation attribute.

For simplicity, we will focus on „point fragmentations" where each value range consists of exactly one attribute value of a fragmentation attribute. This approach is feasible for data warehousing due to the well-defined domains of

2. Bitmaps store 1 bit per fact tuple. Therefore, the size of a fact fragment is $(8 \cdot SizeFactTuple)$ times the size of bitmap fragments. (*SizeFactTuple* denotes the size of a fact tuple in bytes)
3. For Shared Nothing, the bit fragments would have to be allocated to the same processing node as the fact fragment thus restricting the number of disks for allocating bitmaps.

dimensional attributes. Point fragmentations eliminate the need to define value ranges for fragmentation attributes and allow a high number of fragments of reduced size. The number of fragments is simply determined by the product of the fragmentation attributes' cardinalities. Note that point fragmentations still result in range fragmentations at lower levels of the dimension hierarchy because for each value at a specific dimension level, we have an associated value range at the lower levels. For instance, a specific product group covers a range of product classes and product codes.

We denote an *m*-dimensional (point) fragmentation *F* by specifying its fragmentation attributes f_i in the form

$F = \{f_1, f_2, ..., f_m\}$ with

f_i = '$Dimension_i$::$Hierarchy\text{-}level_{i,j}$'. i = 1..*m*

Every fact fragment obtained for such a fragmentation *F* contains all fact rows with one particular value per fragmentation attribute[4]. For instance, the fragmentation

$F_{MonthGroup}$ = {*time::month, product::group*}

refers to a two-dimensional fragmentation on product group and month. Each fact fragment of this fragmentation combines all fact rows referring to one particular product group and one particular month. Based on the cardinalities given in Section 3, $F_{MonthGroup}$ results in $24 \cdot 480 = 11{,}520$ fact fragments. The order in which the fragmentation attributes are specified is irrelevant for the contents of fact fragments. However, we use a specific logical ordering of the dimensions for placing the fragments to disks. For instance, Figure 2 shows an allocation for $F_{MonthGroup}$ for *M* (= 24) months and *G* (= 480) groups where we first allocate the *G* fragments for the first month, followed by the *G* fragments of the second month etc.

4.2 Reducing query work and bitmap requirements

MDHF not only allows queries on the fragmentation attributes to be confined to a subset of the fragments but also many other query types by utilizing the hierarchical structure of dimensions. We can distinguish the following four basic cases or query types Q*i* for which such an improvement is possible. Each case has several subcases depending on whether all fragmentation dimensions are involved in the query or only a subset, and on whether attributes not belonging to a fragmentation dimension are additionally to be evaluated.

- *Q1: Queries on the fragmentation attributes*
  Queries referencing all fragmentation attributes can be confined to the minimal number of fragments. For exact-match predicates on all fragmentation attributes we have only 1 fragment to process. This holds for our query *1MONTH1GROUP* from Section 3 and the above fragmentation $F_{MonthGroup}$. Note that every fact row of the selected fragment is relevant for this query so that there is no need to use bitmaps for the query attributes.
  Queries referencing a subset of the fragmentation attributes still can be confined to comparatively few fragments but more than in the previous case. For exact-match queries the number of fragments to be processed increases by the cardinality of the fragmentation attributes not accessed. For instance, if we want to aggregate all facts for one product *GROUP* - over all 24 months - for fragmentation $F_{MonthGroup}$, we have to process 24 fragments. Again, there is no need to access bitmaps for the query attribute because we have to completely process the fragments.
  Bitmap access is only needed for additional query attributes not belonging to any fragmentation dimension. For instance, to aggregate over 1 product *GROUP* and 1 *STORE* we have to process 24 fact fragments but can use a bitmap index on *CUSTOMER* to restrict processing to the relevant fact rows.
- *Q2: Queries on „lower-level" attributes of the fragmentation dimension*
  Queries accessing attributes from the dimension of a fragmentation attribute but below in the hierarchy can also be restricted to the minimal number of fragments. This is because each value of such an attribute belongs to exactly one value of the fragmentation attribute and is thus confined to a small number of fragments. Ideally, only 1 fragment needs to be accessed when all fragmentation dimensions are involved (e.g. a query *1CODE1MONTH* aggregating over 1 product *CODE* and 1 *MONTH* for $F_{MonthGroup}$). Queries not referencing all fragmentation dimensions need to process a correspondingly larger number of fragments as in the previous case (e.g. query *1CODE* aggregating for 1 product *CODE* and all *MONTH*s involves 24 fragments).
  In contrast to Q1, only a subset of the fragment rows is relevant even if only attributes from the fragmentation dimensions are accessed. Bitmaps may be used within the fragmentation dimension to select the relevant rows (e.g., for product code) and to allow Boolean operations with other bitmaps.
- *Q3: Queries on „higher-level" attributes of the fragmentation dimension*
  Queries accessing attributes from the dimension of a fragmentation attribute but higher in the hierarchy can also be restricted to a subset of the fragments but to more than in the cases before. This is because each value of such an attribute has multiple associated values of the fragmentation attribute and thus a correspondingly higher number of fragments. For instance, if we want to aggregate a product *GROUP* over a *QUARTER* we have to access three fragments rather than one. Again, the number of fragments increases if only some fragmentation dimensions are involved. For instance if want to aggregate for one *QUARTER* – over all product *GROUP*s – we have to process $480 \cdot 3$ fragments (one eighth of all fragments).
  As in case Q1, all tuples of the selected fragments are relevant (no bitmap access for fragmentation dimension).
- *Q4: „Mixed" queries on fragmentation dimensions*
  For queries referencing at least two fragmentation dimensions, mixed cases are possible involving both attributes at a lower (or equal) and at a higher (or equal) level than a fragmentation attribute. An example for this case, is a query for a specific product *CODE* and *QUARTER* *under*

4. More formally, a fragment of fact table T, dimension tables D_i, and fragment attributes f_i contains the result of the following relational expression for a specific combination of attribute values w_i ($i = 1 \ldots m$):
$\pi_{T\text{-}attributes}\ (\ldots((T \bowtie \sigma_{f_1 = w_1}(D_1)) \bowtie \sigma_{f_2 = w_2}(D_2)) \ldots \bowtie \sigma_{f_m = w_m}(D_m))$

$F_{MonthGroup}$. This query can be restricted to 3 fragments because 1 product *CODE* and 3 *MONTH*s are involved. As for Q2, only a subset of the fact rows of the selected fragments is relevant.

Thus, all queries referencing at least *one attribute of any fragmentation dimension* can be confined to a subset of the fragments. Furthermore, MDHF implies that for selections on fragmentation attributes and on higher-level attributes of a fragmentation dimension all tuples of a fact fragment are relevant so that there is no need to use bitmaps for these attributes (cases Q1 and Q3). This allows us to completely eliminate bitmaps for these attributes (because they would only contain „1“ bits) resulting in substantial storage and processing savings. For our fragmentation $F_{MonthGroup}$, we do not need any bitmaps for the *TIME* dimension because for each query on *MONTH*, *QUARTER*, or *YEAR* all rows of the selected fact fragments are relevant. For the product dimension, we do not need bitmaps for product *GROUP* and higher levels, thus saving 10 bitmaps (cf. Table 1). Compared to the maximum of 76 bitmaps (Section 3.2), for $F_{MonthGroup}$ at most 32 bitmaps are thus to be maintained.

4.3 Parallel processing of star queries

We employ two levels of intra-query parallelism for star queries. For each fragment to be processed, we assign a subquery processing the fact fragment and the associated bitmap fragments. Within each subquery, I/O parallelism can be used, e.g. to concurrently access multiple bitmaps. The fragmentation only defines the maximal number of subqueries. The actual degree of intra-query parallelism and the assignment of subqueries to processing nodes is determined by the scheduling and load balancing strategy (see Section 5). To improve sequential I/O performance for both the fact table and bitmaps, we read multiple consecutively stored pages per I/O. Such *prefetch granules* typically range from 1 to 8 pages.

The following steps are performed for processing a star query with respect to a given fragmentation *F*:

1. Determine fact fragments to be processed based on the query's attributes and the fragmentation attributes of F
2. For each query attribute q_i, determine all associated bitmap fragments. Bitmap access is needed for a q_i, iff
   - the dimension of q_i is not represented in *F*, or
   - the dimension of q_i is represented in *F*, but on a higher hierarchy level.
3. Assign a subquery to each fact fragment and its corresponding bitmap fragments
4. For each subquery scheduled for execution
   a) access and process a set of consecutive pages of all relevant bitmap fragments to determine hit rows.
   b) access fact pages containing hits and process aggregation

   Iterate steps 4a and 4b until all pages of the fragment are processed.

4.4 Fragmentation thresholds

As we have seen, both fragmentation and bitmap indices allow identifying which fact data is relevant for a query, thereby reducing processing work. While bitmap indices are effective in many cases they require a substantial storage and processing overhead. For our configuration, each bitmap occupies 223 MB. Since our fragmentation approach eliminates the need for bitmaps for all fragmentation attributes and higher-level attributes of the respective dimension, it seems desirable to choose a fine-grained fragmentation. However, choosing a fragmentation with too many fragments causes a substantial administration overhead and deteriorates I/O performance as we will discuss. On the other hand, the number of fragments must not be too small in order to support a larger number of disks and processors. In particular, there should be at least 1 fragment per fact table disk.

The finest possible fragmentation would be to use all dimensions at the lowest level, i.e. *{time::month, product::code, customer::store, channel::channel}*. This would eliminate all bitmaps but result in more fact fragments (7.5 billion) than fact tuples. The four-dimensional fragmentation *{time::quarter, product::group, customer::retailer, channel::channel}* reduces the number of fact fragments to about 9 million plus about 440 million bitmap fragments. The administration overhead for maintaining such a number of fragments is considered prohibitive. Ideally, the size of the fragmentation information should be small enough to be cached in main memory.

As our simulation experiments have revealed, a high number of fragments can be even more harmful with respect to I/O performance. This holds particularly for queries based on bitmap processing (e.g., for attributes not belonging to a fragmentation dimension). This is because even for huge fact tables a high number of fragments can reduce the average size of a bitmap fragment under the size of a prefetch granule (or even under 1 page). This strongly increases the number of bitmap I/Os and deteriorates I/O times. To ensure a minimal bit fragment size of *PrefetchGran* pages (size of prefetch granule) an upper threshold for the number of fragments n_{max} should be observed with

$$n_{max} = N/(8 \cdot PgSize \cdot PrefetchGran)$$

In this formula, *N* denotes the number of fact tuples and *PgSize* the page size (in Bytes). For instance, with *PrefetchGran* = 4 and *PgSize* = 4K we get n_{max} = 14,238. For a fact tuple size of 20 B, this corresponds to a minimal fragment size of 2.5 MB.

This is an important threshold eliminating already a substantial number of fragmentation choices. For our sample schema, there are 168 possible fragmentations. As can be seen in Table 2, 1/2 to almost 3/4 of these options can be ruled out by demanding a specific minimal bitmap fragment size. In particular, of the 36 possible four-dimensional fragmentations only 1 results in a bitmap fragment size of at least one page and none guarantees at least a size of four pages.

# fragmentation dimensions	**minimum bitmap fragment size**			
	any	**≥ 1 page**	**≥ 4 pages**	**≥ 8 pages**
1	*12*	*12*	*12*	*11*
2	*47*	*37*	*31*	*27*
3	*72*	*22*	*13*	*9*
4	*36*	*1*	*–*	*–*
total	*167*	*72*	*56*	*47*

Table 2: Number of fragmentation options under size constraints

4.5 I/O cost introduced by a fragmentation

Minimizing the I/O requirements and I/O time of a query is of prime importance for achieving a suitable response time. If a fragmentation *F* restricts the number of fragments to be processed for a query *Q*, it as well reduces the number of fact table and bitmap pages that need to be accessed. This is because in this case, all relevant hit rows are co-located within a smaller subset of all pages, increasing the number of hits per page and improving prefetch efficiency. Furthermore, as we have seen, a fragmentation can avoid bitmap access, e.g. if all rows of a fact fragment are relevant for a query.

In order to analytically quantify the I/O performance of different fragmentations, we have developed a set of mathematical formulas estimating the *number of fact table pages containing hit rows* and the *number of bitmap pages* to be accessed for a query. Details on this are provided in [33]. For simplicity, the estimates assume a uniform distribution of query hits within each relevant fragment and page. Furthermore, it is assumed that all pages of a fragment are stored consecutively on disk.

Based on the cases Q1 – Q4 discussed in Section 4.2, we roughly distinguish two classes of queries with respect to their I/O overhead for a given fragmentation. We denote these *I/O overhead classes* as *IOC1* and *IOC2*. In the following, we characterize the I/O behavior of these classes and quantify 2 extreme cases. With $Dim(S)$ we denote the dimensions represented in a set *S*, $hier(h)$ determines the hierarchy level of an attribute *h*, $card(h)$ denotes the cardinality of an hierarchy attribute *h*. Finally, f_q denotes a fragmentation attribute of the *same dimension* of a query attribute *q*.

***IOC1*: Clustered hits, no bitmap access.** A query of this class achieves near-optimal I/O conditions by not requiring bitmap access and finding all hits optimally located (clustered) within pages of the fact fragments. This is achieved for query types Q1 and Q3 above when only attributes from the fragmentation hierarchies of F are to be accessed. Therefore, in mathematical terms, a query Q is assigned to IOC1, if

$Dim(Q) \subseteq Dim(F) \wedge \forall q \in Q: hier(q) \geq hier(f_q)$.

A $Q \in IOC1$ has to process all pages of the determined fragments. In the optimal case (subclass *IOC1-opt*), where

$Dim(Q) = Dim(F) \wedge \forall q \in Q: hier(q) = hier(f_q)$,

queries only have to process one fragment (query type Q1, restricted to *F*-dimensions). Every dimension *f* of *F* that is not referenced in *Q* increases the number of fragments to be processed with the factor $card(f)$. Accessing an attribute *q* with $hier(q) > hier(f_q)$ on average increases the number of fragments by a factor of $card(f_q)/card(q)$. An increased number of fragments also reduces the number of hits per page, thereby increasing the number of I/O operations.

***IOC2*: Spread hits and bitmap I/O.** This class contains all remaining queries performing *bitmap access* to determine the hit rows within the fragments. This covers queries of types Q2 and Q4 as well as queries accessing dimensions not represented in *F*. For these queries, hit rows are spread across more fragments than for *IOC1*. This results in a reduced number of hits per page and prefetching granule introducing worse I/O efficiency and overhead. In the worst case, called *IOC2-nosupp*, a query is not supported at all by the fragmentation, i.e., it does not reference any fragmentation dimension. Hence, all bitmap fragments of all referenced dimensions have to be processed. Assuming more hits than the number of fragments and uniform distribution of hits, every fact fragment has to be accessed.

Quantitative comparison

The formulas developed in [33] allow determining the number of fragments to be accessed as well as the number of fact table and bitmap I/O operations for a given fragmentation and query type. They can thus be used within a tool to quantify the I/O performance of different fragmentation choices for a given query mix to help determine a good fragmentation. Table 3 illustrates the differences in I/O work for the one-dimensional sample query *1STORE* as determined with these formulas. The query belongs to *IOC1-opt* for the optimal fragmentation F_{opt} = *{customer::store}* and to *IOC2-nosupp* for, e.g., $F_{nosupp} = F_{MonthGroup}$ = *{time::month, product::group}*. We assume a prefetching granule of 8 pages on fact fragments and 5 pages on bitmap fragments (the bitmap fragment size is 4.9 pages for F_{nosupp}). With F_{nosupp}, only every 7th page in a fact fragment contains hits, thus strongly reducing prefetch efficiency. The table shows that a suitable fragmentation permits improvements in I/O performance by several orders of magnitude. Our simulation system allows more detailed performance predictions, in particular with respect to response time, by considering processor and disk contention as well as other factors.

4.6 Physical allocation

Having found a suitable fragmentation, the resulting fact and bitmap fragments have to be allocated onto disks. With respect to the *degree of declustering*, our simulations have confirmed that a full declustering of the fact table utilizing all available disks is the best approach as it supports the maximal degree of parallelism and load balancing. While the minimal number of disks is determined by the capacity requirements to store the fact table, bitmaps and other data, a typically much larger number of disk has to be used – within a reasonable economic range – in order to provide a high degree of I/O rates and I/O bandwidth.

As already indicated in Fig. 2, we use a special round robin allocation called *staggered round robin* to allocate fact fragments and their associated bitmap fragments. In this approach, the bitmap fragments of a fragment are allocated onto consecutive disks so that all bitmap fragments needed for a query can be accessed in parallel during a subquery.

In Section 6.2, we will analyze to which degree this affects query response times. Additionally, fact fragments could also be declustered to support I/O parallelism. We

	F_{opt}	F_{nosupp}
#fragments to be processed	*1*	*11,520*
#fact table I/O [pages]	*795*	*5,189,760*
#bitmap I/O [pages]	–	*691,200*
total I/O size [MB]	*25*	*31,075*

Table 3: I/O characteristics for query *1STORE*

store fact tables and bitmap data onto the same disks to allow all disks to be used for the fact table.

As we observed in our simulation experiments, care must be exercised with round robin in order to not artificially restrict parallelism for certain query classes. This is because the p fragments to be accessed by a query can get allocated onto *less than p* disks, introducing sequential disk work. For instance, assume our fragmentation $F_{MonthGroup}$ and $d = 100$ disks. To place the 11,520 fragments on disk, we have to determine an *allocation order* on the fragmentation attributes. Assume that we first assign the 480 fragments for month 1 consecutively, then the next 480 fragments for month 2 and so on. Processing query type *1CODE* requires access to 24 of these fragments (1 per month) which correspond to every 480th fragment. Due to 480 and 100 having a *greatest common divisor (gcd)* of 20, all relevant fragments for *1CODE* are located on only 5 disks, thus reducing possible parallelism by a factor of 4.8. If we decide to allocate the other way round, *1CODE* is optimally supported while, e.g., *1MONTH* queries are restricted to 25 disks ($gcd = 4$). Hence, we have to find an allocation that reduces the probability of such a clustering as far as possible. A solution is to choose a prime number for the degree of declustering or to use a modified allocation scheme introducing certain gaps to avoid such a clustering.

4.7 Guidelines for data allocation

The conclusion to be drawn from this section is that fragmentation and allocation have a large impact on I/O and query performance. To find a suitable fragmentation, a number of guidelines can be formulated:

- Exclude all possible fragmentations which break at least one of our three thresholds (i) minimal bitmap fragment size, (ii) maximum number of fragments to administer, (iii) maximum number of bitmaps to be materialized. Values for (ii) and (iii) depend on the main memory or disk storage space that can be utilized in a practical environment.
- Limit the dimensionality of fragmentation based on the dimensions typically referenced in the query profile. A broad query mix normally favors a higher number of declustering dimensions. On the other hand, one- or two-dimensional fragmentations may have too few fragments to even use all available disks, which is of course unacceptable.
- Analyze the I/O load introduced by the remaining fragmentations using the analytical formulas of [33]. If no query type is favored, choose a fragmentation that achieves the minimum total amount of I/O work performed by all query types. Otherwise, consider all fragmentations which optimize the favored queries and proceed as above for the rest of the queries on the remaining fragmentation candidates.

5 Simulation system and setup

To test and evaluate the allocation and processing methods described above and to verify our analytical considerations, we designed and implemented a comprehensive simulation system named *SIMPAD* (*Sim*ulation of *Pa*rallel *D*atabases). It is written in C++ and based on the *CSIM* simulation library [20]. Through a modular design, *SIMPAD* supports the evaluation of different PDBS and hardware architectures, algorithms, database allocations and workloads. For brevity, we only mention aspects relevant to the study at hand. Major simulation parameters and their settings used in the experiments are listed in Table 4.

For this study, we have modelled a Shared Disk PDBS with a variable number of processing nodes and disks. Processors and disks are explicitly modeled as servers to realistically capture access conflicts and delays. CPU overhead is accounted for in all major query processing steps and communication (Table 4). The disk model calculates varying seek times based on track positions rather than giving constant or stochastically distributed response times. This allows realistic testing of multi-user and parallel processing. A simple buffer manager is used supporting LRU page replacement and prefetching. We maintain separate buffers for tables and indices. An idealized contention-free network model is employed with communication delays proportional to message sizes, so as not to bias simulation results due to a specific choice of a network topology.

Our database model is based on the star schema detailed in Section 3, with a flexible parameterization for the dimension hierarchies and cardinalities as well as the fact table density. Bitmap join indices are provided on the fact table for all dimension keys, with a possible choice of either standard or encoded bitmaps. The dimension tables have B*-tree indices, but these are not relevant to our experiments. The data allocation is determined separately for all tables and indices, each of which can be assigned to an individually defined group of disks. The evaluated fragmentation and allocation of the fact table and its bitmap indices follow the description in the previous sections.

A query generator creates a series of query structures that are passed to the processing module. In this initial study, we restrict ourselves to single-user mode, so queries are issued sequentially with a new query starting as soon as the previous one has terminated. For a single simulation, all queries are of the same type (e.g., *1STORE*), but specific parameters are chosen at random (e.g., the actual *STORE* selected).

Parallel processing and load balancing

New queries are first assigned to a randomly selected coordinator node that is responsible for parallelizing the query and scheduling its execution. This coordinator creates a *task list* of all subqueries to be performed, each comprising one fact fragment and its associated bitmap fragments. Based on the query type and fragmentation, the scheduler considers only the relevant fact fragments and bitmaps as outlined in Section 4. The list is sorted in the order in which the fragments were allocated to disks, so that consecutive subqueries can be expected to access different disks. The coordinator assigns subqueries from the task list to available processors in a round-robin manner, where each node receives a maximum of t concurrent tasks (t being a system parameter). As described in 4.2, each subquery processes the required bitmap fragments, reads the associated fact table pages, extracts hit rows, and locally aggregates the measures (e.g., *DOLLARSALES*) found there. When a node finishes a subquery, it returns the partial aggregate to the coordinator and is assigned another task. When all rele-

Parameter	settings	Parameter	settings	Parameter	settings
disk devices		**no. of instructions**		**buffer manager**	
number (d)	100	initiate/plan query	50,000	page size	4 KB
speed-up experiments	1 – 100	terminate query	10,000	buffer size fact table	1000 pages
avg. seek time	10 ms	initiate/plan subquery	10,000	buffer size bitmaps	5000 pages
avg. settle time +	per access 3 ms	terminate subquery	10,000	prefetch size fact table	8 pages
controller delay	+ per page 1 ms	read page	3,000	prefetch size bitmaps	5 pages (var.)
processing nodes		process bitmap page	1,500		
number (p)	20	extract table row	100	**network**	
speed-up experiments	1 – 50	aggregate table row	100	connection speed	100 Mbit/s
CPU speed	50 MIPS	send message	1,000 + #B	message size (small)	128 B
subqueries per node	var.	receive message	1,000 + #B	message size (large)	1 page (4 KB)

Table 4: Parameters settings used in simulations

vant data has been processed, the query is terminated and the overall aggregate gathered by the coordinator is returned to the user.

The simulation system considers the CPU and message overhead introduced by this scheduling. Early simulation experiments showed that the overhead is very small compared to the actual processing and the coordinator node can is also be used for processing subqueries. We do, however, count coordination as one task so that the coordinator node will only process $t - 1$ subqueries at a time.

We have devised several additional scheduling techniques that will be explored in a future study.

6 Simulation results

In this section, we present first results of simulation experiments we performed to verify and study our data allocation approach from Section 4. There are three basic simulation series: speed-up tests to verify the scalability of our approach (6.1); experiments on the impact of parallel subqueries and bitmap I/O (6.2); simulations for different fragmentation and allocation strategies (6.3).

6.1 Speed-up experiments

The first series of simulations was designed to investigate the scalability of the allocation approach and fragment-oriented query processing. We used the star schema configuration introduced above with fragmentation $F_{MonthGroup}$ (11,520 fragments) on different hardware configurations as listed in Table 5. The number of disks, d, was varied from 20 to 100; the number of processors, p, ranges from $1/20$ to $1/2$ of the number of disks, resulting in 1 – 50 processors. For each configuration, we tested two simple query types, *1STORE* and *1MONTH*, to study both disk- and CPU-bound workloads.

We first discuss the performance for query *1STORE*. This query is not supported by the chosen fragmentation and thus requires access to all fragments as well as to all the bitmaps of its encoded index. Figure 3 shows the response time and speed-up curves for this query. We use a fixed number of subqueries per node $t = d/p$ so that the total number of subqueries equals the number of disks which proved sufficient to utilize all disks. As can be seen, response times depend solely on the number of disks used because *1STORE* is heavily disk-bound. The only exception is the data point for 20 disks and 1 processor. That particular experiment suffers from the fact that the single node is also its own coordinator and can only process $t - 1$ (i.e., 19 rather than 20) subqueries at a time (cf. Section 5). The chosen data allocation, processing model and scheduling strategy allow for linear improvement of response times with an increased number of disks. In fact, speed-up with respect to d is slightly superlinear, due to reduced seek times when there is less data on a single disk.

number of disks (d)	number of processors (p)				
20	*1*	*2*	*4*	*5*	*10*
60	*3*	*6*	*12*	*15*	*30*
100	*5*	*10*	*20*	*25*	*50*

Table 5: Hardware parameters for speed-up experiments

The *1MONTH* query, on the other hand, is optimally supported by our fragmentation. It is confined to the 480 fragments associated with the *MONTH* selected and need not access any bitmap. This query is CPU-bound; as can be seen in Figure 4, its response times depend on the number of processors rather than disks. Again, we achieve optimal speed-up, this time with respect to the number of processors p. The only exception is for the configuration with 100

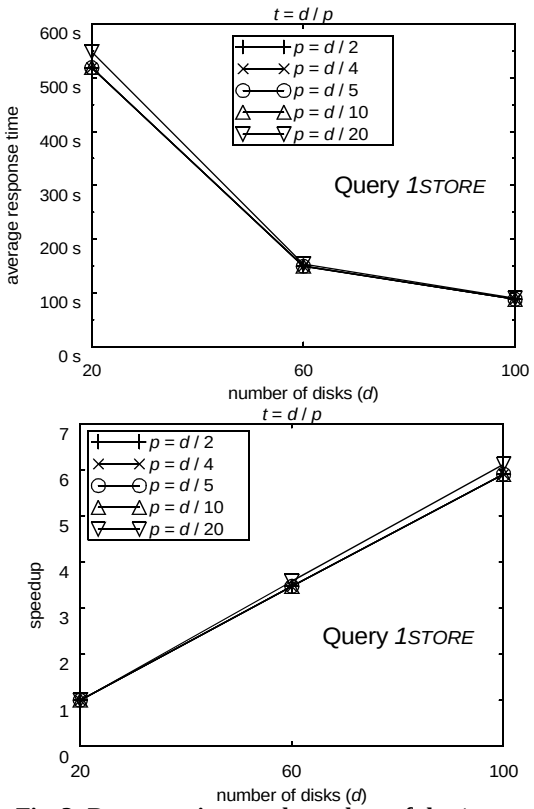

Fig. 3: Response times and speed-up of the *1STORE* query

disks and 50 nodes in the upper end of the curve. Here, a discretization problem occurs: With $t = 4$, which is optimal for most other settings, subqueries will be executed in "batches" of 200 ($t \cdot p = 4 \cdot 50$). For the 480 fragments to be processed, this produces three batches of 200, 200, and 80, the last one being inefficient to process due to the reduced parallelism. Amending the parameter to $t = 5$, we obtain two batches of 250 and 230, which are processed much more efficiently even though a single batch will take longer. Using this result, we can re-establish linear speed-up, as represented by the dotted line in the speed-up graph of Figure 4.

The results confirm that the approaches chosen permit optimal utilization of I/O and processing parallelism. It was demonstrated that linear speed-up can be achieved with respect to either the number of disks or the number of processors, depending on whether the workload is disk- or CPU-bound. In the remaining experiments, we use a fixed hardware configuration of 100 disks and 20 processing nodes.

6.2 Parallel subqueries and parallel bitmap I/O

In this experiment, we investigate the impact of the number of subqueries and the effectiveness of parallel bitmap I/O within a subquery in more detail. The latter is supported by our "staggered" data allocation assigning the bitmap fragments of a fact fragment to separate disks (Section 4.5). A potential problem is increased disk contention, which may harm other subqueries that have to access the same disks. We evaluated this trade-off using the I/O-intensive *1STORE* query type that has to access 12 bitmap fragments for each fact table fragment. Based on the 100-disk, 20-node configuration determined above, we tried both parallel and non-parallel bitmap I/O for varying numbers of concurrent subqueries per processor, obtaining the results of Figure 5.

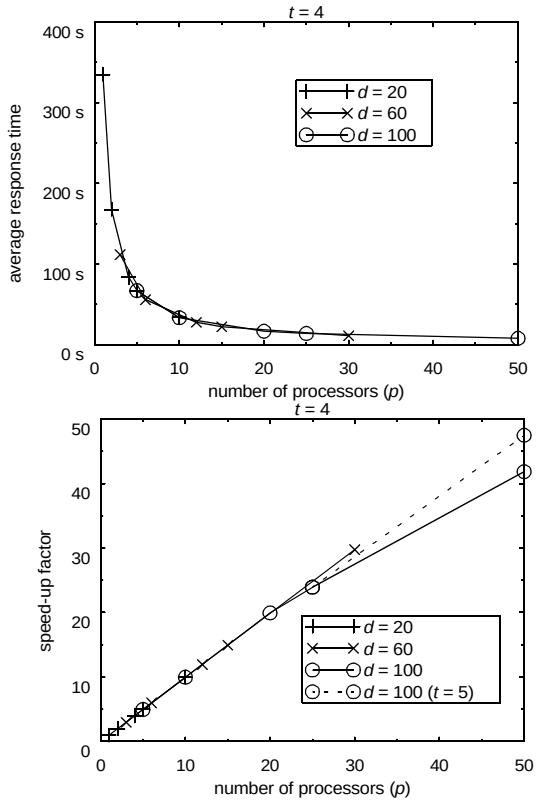

Fig. 4: Response times and speed-up of the *1MONTH* query

The figure illustrates the importance of using multiple subqueries per node for this I/O bound query type in order to fully utilize all disks. We are able to linearly improve response times up to about 5 subqueries per node where the total number of subqueries reaches the number of disks (100). A higher number of subqueries has only little impact on response time (also influenced by single-user mode). This almost ideal behavior again confirms the scalability and good load balancing achieved with our Shared Disk-oriented scheduling strategy.

Parallel bitmap I/O delivers noticeable response time improvements of up to 13 % despite the concurrent access of subqueries to the same disks and although a larger part of the response time is caused by fact table I/O. The improvements are especially pronounced for a smaller number of concurrent subqueries. For many subqueries, performance of the two alternatives becomes similar due to increased disk contention. However, even in this case parallel bitmap I/O remains slightly ahead.

The conclusion we draw is that parallel bitmap I/O is a good default setting mostly improving system performance. For mixed CPU- and I/O-bound workloads, we expect additional improvements for parallel bitmap I/O due to reduced disk traffic compared to purely I/O-bound cases such as for *1STORE*. Further improvements are likely by utilizing parallel I/O on fact fragments which are larger than the bitmap fragments.

6.3 Implication of the fragmentation strategy for query processing

In this experiment, we quantify our results of Sections 4.4 and 4.5, where we outlined the impact of the fragmentation strategy on query processing. We observed that fine-grained fragmentations allow many query types to be confined to few fragments, thereby increasing I/O efficiency. On the other hand, I/O problems can be introduced if bitmap fragment sizes fall below the size of a prefetch granule. For our experiment on these effects we choose two query types *1STORE* and *1CODE1QUARTER* on three two-dimensional fragmentations $F_{MonthGroup}$, $F_{MonthClass}$ and $F_{MonthCode}$, based on the time and product dimensions. The fragmentations only differ in the selected hierarchy level of the product dimension. Table 6 shows the resulting number of frag-

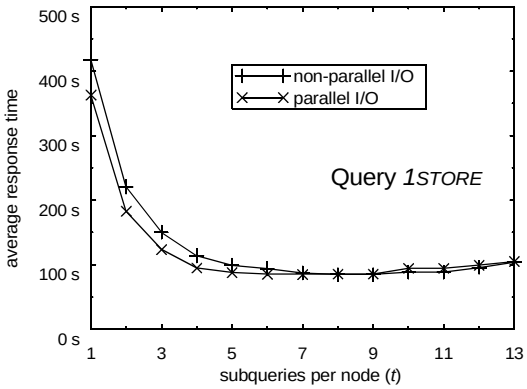

Fig. 5: Response time effects of parallel bitmap I/O

	$F_{MonthGroup}$	$F_{MonthClass}$	$F_{MonthCode}$
number of fragments	*11,520*	*23,040*	*345,600*
bitmap fragment size [pages]	*4.9 (5)*	*2.5 (3)*	*0.16 (1)*

Table 6: Fragmentation parameters for experiment 3

ments and bitmap fragment sizes (numbers in parantheses denote the prefetch granule size). According to Section 4.5, *1STORE* belongs to I/O class ***IOC2-nosupp*** in any considered case, because the customer dimension is not represented in any of our sample fragmentations, forcing the query to access all fragments. *1STORE* is assigned to the worst case class ***IOC2-nosupp*** because the customer dimension is not represented in any of our sample fragmentations forcing the query to access all fragments. Due to its query selectivity of 1/1440, and since there are about 200 tuples per fact table page, only 1 in 7 pages of every fragment contains a hit. Query type *1CODE1QUARTER* accesses exactly 3 fragments (one for each month of a quarter), residing on 3 disks regardless of the fragmentation. It has to process only 16,200 rows in total with a sensible processing parallelism of at most 3. For $F_{MonthClass}$ and $F_{MonthGroup}$, bitmap access is introduced so that the query type belongs to class ***IOC2*** in these cases. For $F_{MonthGroup}$, I/O class ***IOC1*** is given because the fragments contain only relevant fact rows.

Figure 6 shows the response time behavior of the two queries types for the three fragmentations. The x-axis refers to the total number of subqueries over all (20) processing nodes. For both queries, I/O dominates response times. While *1STORE* has about 80 times more hit tuples than *1CODE1QUARTER* its response time is more than 300 times worse for lower degrees of parallelism. Only for the two better fragmentations and at least 100 subqueries can this query type obtain a response time that is about 80 times higher than for *1CODE1QUARTER* which achieves its optimum for only 3 subqueries. While this leaves many resources unused, in multi-user mode this can be advantageous for other queries.

1CODE1QUARTER benefits from the chosen fragmentations. Within a product group, the selectivity is 1/30 for a certain product. Therefore, every fact page (each containing 200 tuples) of the 3 fragments contains hits regardless of the granule of fragmentation varying from group to code. For fragmentation $F_{MonthClass}$ fragment size halvens compared to $F_{MonthGroup}$ resulting in a corresponding response time improvement because every fragment page is to be read. The best response times are achieved for $F_{MonthCode}$ for which no bitmap access is necessary and fragments only contain relevant tuples (***IOC1***).

1STORE exhibits the inverse behavior with respect to the fragmentations. In particular, the fine-grained fragmentation $F_{MonthCode}$ results in the worst performance. This is especially because the bitmap fragment size drops to only 1/6 of a page resulting in an extreme number of bitmap pages (more than 4 million) to be read for the 12 bitmaps.

This indicates that a fragmentation such as $F_{MonthCode}$ must be avoided, which can be achieved by considering the fragmentation threshold introduced in Section 4. A possibility to improve efficiency for finer fragmentations is to cluster together multiple bitmap and fact fragments, respectively, and subsequently assign such *granules* of clustered fragments to consecutive pages and *to one single subquery.* This can especially help to avoid unacceptable bitmap fragment sizes.

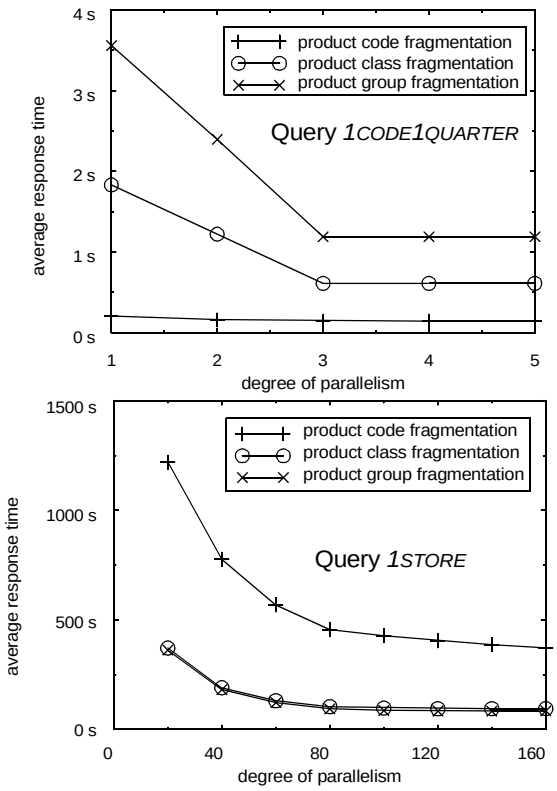

Fig. 6: Response times of *1STORE*, *1MONTH1QUARTER* for different fragmentations

7 Conclusions

In this study, we have developed a multi-dimensional hierarchical fragmentation and allocation method for star schemas in a parallel data warehouse environment. The approach called MDHF allows all star queries referencing at least one attribute from any fragmentation dimension to be confined to a subset of the fact table fragments. This clusters hit rows within fewer pages, thereby supporting fewer I/O operations and effective prefetching. Moreover, we can eliminate bitmap indices on the fragmentation dimensions either completely or partially, further saving disk space and I/O load. Our technique uses an analogous fragmentation of fact tables and their associated bitmap indices to enable simultaneous, fragmentwise processing that can be parallelized effectively.

We developed a number of guidelines for finding an appropriate fragmentation. As outlined, we must avoid very fine fragmentations to limit the administration overhead and to avoid only partially filled bitmap index pages that sharply increase I/O load. The guidelines were verified using a detailed simulation model based on the APB-1 decision support benchmark. Together with our analytical formulas calculating star query I/O costs [33], they can be used within a tool to automatically determine suitable fragmentation candidates for a given query mix.

Our approaches assume a Shared Disk PDBS but can be applied to other architectures with minor modifications. The simulation results demonstrated that the flexibility of Shared Disk architectures permits efficient load balancing based on a round robin allocation scheme in combination with intra-processor parallelism. This approach exhibits near-linear scalability with respect to the number of disks and processors. Furthermore, it enables parallel I/O for all bitmap fragments accessed within a given subquery.

We believe that our fragmentation and allocation approaches are directly applicable to commercial PDBS with comparatively little effort. In future studies, we will elaborate on the load balancing properties of star schema processing and examine data skew effects as well as the consequences of multi-user mode. Furthermore, we want to explore how our multi-dimensional hierarchical partitioning can be exploited for the complementary allocation decisions associated with materialized views and caching of query results.

References

[1] *APB-1 OLAP Benchmark, Release II*. OLAP Council, Nov. 1998. www.olapcouncil.org/research/bmarkly.htm

[2] C. Ballinger: *Teradata Database Design 101*. White Paper, NCR Corporation, 1998.

[3] S. Brobst, B. Vecchione: *DB2 UDB: Starburst Grows Bright*. Database Programming & Design, 1998.

[4] G. Copeland et al.: *Data Placement in Bubba*. Proc. ACM SIGMOD Conf., Chicago, 1988.

[5] S. Chaudhuri, U. Dayal: *An Overview of Data Warehousing and OLAP Technology.* SIGMOD Record 26(1), 1997

[6] P. M. Chen et al.: *RAID: High-Performance, Reliable Secondary Storage.* ACM Computing Surveys 26 (2), 1994.

[7] D. J. DeWitt, J. Gray.: *Parallel Database Systems: The Future of High Performance Database Systems*. Comm. ACM 35 (6), 85 - 98, 1992.

[8] S. Ghandeharizadeh, D. J. DeWitt: A *Multiuser Performance Analysis of Alternative Declustering Strategies*. Proc. 6th Int. Conf. on Data Engineering, 1990.

[9] S. Ghandeharizadeh, D. J. DeWitt, W. Qureshi: *A Performance Analysis of Alternative Multi-Attribute Declustering Strategies.* Proc. ACM SIGMOD Conf., 29 - 38, 1992.

[10] G. Graefe, J. Ewel, C. Galindo-Legaria: *Microsoft SQL Server 7.0 Query Processor.* White Paper, Microsoft, 1998.

[11] V. Gaede, O. Günther: *Multidimensional Access Methods*. ACM Comp. Surv. 30 (2), 170 - 231, 1998.

[12] A. Gupta, I. S. Mumick: *Maintenance of Materialized View: Problems, Techniques, and Applications*. Data Eng. Bulletin 18 (2), June 1995.

[13] J. Gray et al.: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*. In: U. Fayyad, H. Mannila, G. Piatetsky-Shapiro: Data Mining and Knowledge Discovery 1, 29 - 53, 1997

[14] Informix Corporation: *Informix Decision Support Indexing for the Enterprise Data Warehouse*. White Paper, 1998.

[15] Informix Corporation: *INFORMIX-OnLine Dynamic Server: Administration Guide*. http://www.informix.com/answers/oldsite/answers/pubs/pdf/811xpsu/7624.pdf (June 2000)

[16] R. H. Katz, W. Hong: *The Performance of Disk Arrays in Shared-Memory Database Machines*. Distr. and Parallel Databases 1 (2), 167 - 198, 1993.

[17] E. K. Lee, R. Katz: *An Analytic Performance Model of Disk Arrays*, Proc. ACM SIGMETRICS Conf., 1993.

[18] H. Märtens: *On Disk Allocation of Intermediate Query Results in Parallel Database Systems*, Proc. EURO-PAR Conf., Toulouse, LNCS 1685, Springer 1999. http://dol.uni-leipzig.de/pub/1999-24

[19] M. Mehta, D. J. DeWitt: *Data Placement in Shared-Nothing Parallel Database Systems*. VLDB Journal 6 (1), 1997.

[20] Mesquite Software Inc.: *User's Guide CSIM18 Simulation Engine*. Manual, 1996.

[21] C. Mohan, I. Narang: *Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment.* Proc. VLDB Conf., 193 – 207, 1991.

[22] P. O'Neil, G. Graefe: *Multi-Table Joins Through Bitmapped Join Indices*. ACM SIGMOD Record 24 (3), 1995.

[23] P. O'Neil: *Model 204 Architecture and Performance*. Proc. 2nd HPTS Workshop, Asilomar, 1987.

[24] P. O'Neil, D. Quass: *Improved Query Performance with Variant Indexes*. Proc. ACM SIGMOD Conf., 1997.

[25] Oracle Corporation: *Star Queries in Oracle8*. White Paper, 1997.

[26] Oracle Corporation: *Oracle 8i Administrator's Guide*. http://www.irm.vt.edu/oracle_816_docs/server.816/a76956/index.htm (June 2000)

[27] E. Rahm: *Empirical Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems.* ACM TODS 18 (2), 333 – 377, 1993.

[28] E. Rahm: *Parallel Query Processing in Shared Disk Database Systems*. Proc. 5th HPTS Workshop, Asilomar, 1993. http://www.informatik.uni-leipzig.de/ifi/abteilungen/db/abstr/Ra93.HPTS.ps

[29] Red Brick Systems, Inc.: *Star Schema Processing for Complex Queries*. White Paper, 1998.

[30] E. Rahm, H. Märtens, T. Stöhr: *On Flexible Allocation of Index and Temporary Data in Parallel Database Systems.* Proc. 8th HPTS Workshop, Asilomar, 1999. http://dol.uni-leipzig.de/pub/1999-23

[31] E. Rahm, T. Stöhr: *Analysis of Parallel Scan Processing in Parallel Shared Disk Database Systems*. Proc. EURO-PAR Conf., LNCS 966, Springer 1995. http://dol.uni-leipzig.de/pub/1995-22

[32] P. Scheuermann, G. Weikum, P. Zabback: *Data Partitioning and Load Balancing in Parallel Disk Systems*, VLDB Journal 7 (1), 48 – 66, 1998.

[33] T. Stöhr: *Analytical Evaluation of a Multi-Dimensional and Hierarchical Allocation Strategy for Parallel Data Warehouses.* Technical Report, Univ. of Leipzig, Germany, 2000 (to appear)

[34] J. Sun, W.I. Grosky: *Dynamic Maintenance of Multidimensional Range Data Partitioning for Parallel Data Processing*. Proc. First ACM Intl. Workshop on Data Warehousing and OLAP (DOLAP), Washington D.C., 72 – 79, 1998

[35] Sybase, Inc.: *Adaptive Server IQ*. White Paper, 1997.

[36] M.-C. Wu, A. P. Buchmann: *Encoded Bitmap Indexing for Data Warehouses.* Proc. 14th Proc. Int. Conf. on Data Engineering, Orlando, 1998.

[37] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom: *View Maintenance in a Warehousing Environment*. Proc. ACM SIGMOD Conf., San Jose, 1995.