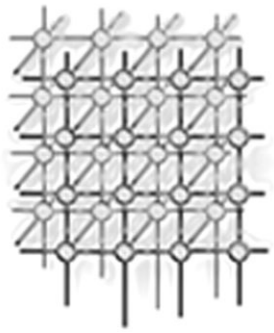
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:1169–1190 (DOI: 10.1002/cpe.786)

Dynamic query scheduling in parallel data warehouses



Holger Märtens[1,*,†], Erhard Rahm[2] and Thomas Stöhr[2]

[1] *University of Applied Sciences Braunschweig/Wolfenbüttel, Germany*
[2] *University of Leipzig, Institute of Computer Science, Augustusplatz 10–11, D-04109 Leipzig, Germany*

SUMMARY

Parallel processing is a key to high performance in very large data warehouse applications that execute complex analytical queries on huge amounts of data. Although parallel database systems (PDBSs) have been studied extensively in the past decades, the specifics of load balancing in parallel data warehouses have not been addressed in detail.

In this study, we investigate how the load balancing potential of a Shared Disk (SD) architecture can be utilized for data warehouse applications. We propose an integrated scheduling strategy that simultaneously considers both processors and disks, regarding not only the total workload on each resource but also the distribution of load over time. We evaluate the performance of the new method in a comprehensive simulation study and compare it to several other approaches. The analysis incorporates skew aspects and considers typical data warehouse features such as star schemas with large fact tables and bitmap indices. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: parallel database systems; query optimization; load balancing; data warehousing; Shared Disk architecture; disk contention

1. INTRODUCTION

For the successful deployment of data warehouses, acceptable response times must be ensured for the prevalent workload of complex analytical queries. Along with complementary measures such as new query operators [1], specialized index structures [2,3], intelligent data allocation [4], and materialized views [5,6], parallel database systems (PDBSs) are employed to satisfy the high performance requirements [7]. For efficient parallel processing, successful load balancing is a prerequisite, and many algorithms have been proposed for PDBSs in general. But we are not aware of load balancing

*Correspondence to: Dipl.-Inform. Holger Märtens, Fachhochschule Braunschweig/Wolfenbüttel, Department of Business Management, University of Applied Sciences, Robert-Koch-Platz 10–14, D-38440 Wolfsburg, Germany.
†E-mail: h.maertens@fh-wolfenbuettel.de

Contract/grant sponsor: Deutsche Forschungsgemeinschaft; contract/grant number: Ra497/10

Received 15 December 2002
Revised 2 April 2003
Accepted 26 May 2003

studies for data warehouse environments which exhibit characteristic features such as star schemas, star queries and bitmap indices.

This paper presents a novel approach to dynamic load balancing in parallel data warehouses based on the simultaneous consideration of multiple resources, specifically CPUs and disks. These are frequent bottlenecks in the voluminous scan/aggregation queries that are characteristic of data warehouses. Disk scheduling is particularly important here as the performance of CPUs develops much faster than that of disks. A balanced utilization of both resources depends not only on the *location* (on which CPU) but also on the *timing* of load units such as subqueries. We thus propose performing both decisions in an integrated manner based on the resource requirements of queued subqueries as well as the current system state.

To this end, we exploit the flexibility of the Shared Disk (SD) architecture [8] in which each processing node can execute any subquery. For scan workloads in particular, the distribution of processor load does not depend on the data allocation, allowing us to perform query scheduling with shared job queues accessed by all nodes. Disk contention, however, is harder to control than CPU contention because the total amount of load per disk is predetermined by the data allocation and cannot be shifted at runtime as for processors.

In a detailed simulation study, we compare the new integrated strategy to several simpler methods for dynamic query scheduling. They are evaluated for a data warehouse application based on the APB-1 benchmark comprising a star schema with a huge fact table supported by bitmap indices, both declustered across many disks to support parallel processing. Our experiments involve large scan/aggregation queries in which each fact table fragment must often be processed together with a number of associated bitmap fragments residing on different disk devices. This can lead to increased disk contention and thus creates a challenging scheduling problem. We particularly consider the treatment of skew effects which are critical for performance but have been neglected in most previous load balancing studies. As a first step in the field of dynamic load balancing for data warehouses, our performance study focuses on parallel processing of large queries in single-user mode, but the scheduling approaches can also be applied in multi-user mode.

Our paper is structured as follows. In Section 2, we review some related work from the literature. Section 3 outlines our general load balancing paradigm, whereas our specific scheduling heuristics are defined in Section 4. Section 5 describes the simulation system and the approaches to deal with skew. Section 6 presents the performance evaluation of the scheduling strategies for different data warehouse configurations. We conclude in Section 7.

2. RELATED WORK

We are not aware of any load balancing studies for parallel data warehouses. For general PDBSs, load balancing problems have been widely researched, for a variety of workloads and architectures [9–14]. Many of these approaches are of minor relevance for data warehousing because they rely on a costly redistribution of data—e.g. for hash joins or external sorting—that is usually too costly for a large fact table. Furthermore, most previous studies have been limited to balancing CPU load, sometimes including main memory [15,16] or network restrictions [17]. Even so, the need for highly dynamic scheduling has been emphasized [16, 18–20]. Conversely, load distribution on disks has largely been considered in isolation from CPU-side processing. Most of these studies have focused either on data

partitioning and allocation [13, 21–26] or on limiting disk contention through reduced parallelism [14]. Integrated load balancing of processors and disks considering the timing of potentially conflicting operations—as proposed in this paper—has not been addressed.

The SD architecture has been advocated due to its superior load balancing potential [11,14,27,28] especially for read-only workloads as in data warehouses, where common objections regarding the performance of concurrency and coherency control [29,30] can be ruled out. It also offers greater freedom in data allocation, e.g. for index structures [24]. But the research on how to exploit this potential is still incomplete. SD is also supported by several commercial PDBSs such as *IBM DB2 Universal Database for OS/390 and z/OS* [31], *Oracle9i Database* [32], and *Sybase Adaptive Server IQ Multiplex* [33]. Like other products addressing the data warehouse market—e.g. *IBM Informix Extended Parallel Server* [34], *IBM Red Brick Warehouse* [35], *Microsoft SQL Server 2000* [36] and *NCR Teradata* [37]—they support star schemas and bitmap indices[‡] as well as adequate data fragmentation and parallel processing. However, the vendors do not describe specific scheduling methods in the available documentation. Dynamic treatment of disk contention is limited to restricting the number of tasks concurrently reading from the same disk, similar to the *Partition* strategy used in our study (cf. Section 4.1). No special ordering of subqueries to avoid disk contention is mentioned, leading us to believe that such elaborate scheduling methods are not yet supported in current products.

3. DYNAMIC LOAD BALANCING FOR PARALLEL SCAN PROCESSING

This section presents our basic approach to dynamic load balancing, which is not restricted to data warehouse environments. We presume a horizontal partitioning of relational tables into disjoint *fragments*. If bitmap indices or similar access structures exist, they must be partitioned analogously so that each table fragment with its corresponding bitmap fragments can form an independent unit of processing. We focus on the optimization of scan queries and exploit the flexibility of the SD architecture that allows every processor to access any fragment regardless of its storage location.

For efficient parallel processing, database workloads must be distributed across the system as evenly as possible. For the extensive scan loads we consider in our study, this is true even within single queries (intra-query parallelism). As mentioned in the introduction, the two performance-critical types of resources are the processing nodes and the disks. (Main memory and network connections are not typically bottlenecks for the scan operations in question.) But the balance of CPU and disk load, respectively, depends on different conditions: CPU utilization is largely determined by *which* processor is assigned which fragments of the data. A balanced disk load, however, hinges on *when* the data residing on each device are processed because their location cannot be changed at runtime. As a consequence, we aim for a load balancing strategy that can view both resources in an integrated manner.

When a new query enters the system, it is randomly assigned a *coordinator* node that controls its execution and distributes the workload in the system. For the scan queries we consider, we provide two different *load granule*s based on the aforementioned horizontal fragmentation of tables: each subquery can comprise either a fragment or a *partition* of the relevant table, where a partition is defined as the

[‡]In *SQL Server* and *Teradata*, bitmap indices are used internally but cannot be defined by the user or DBA.

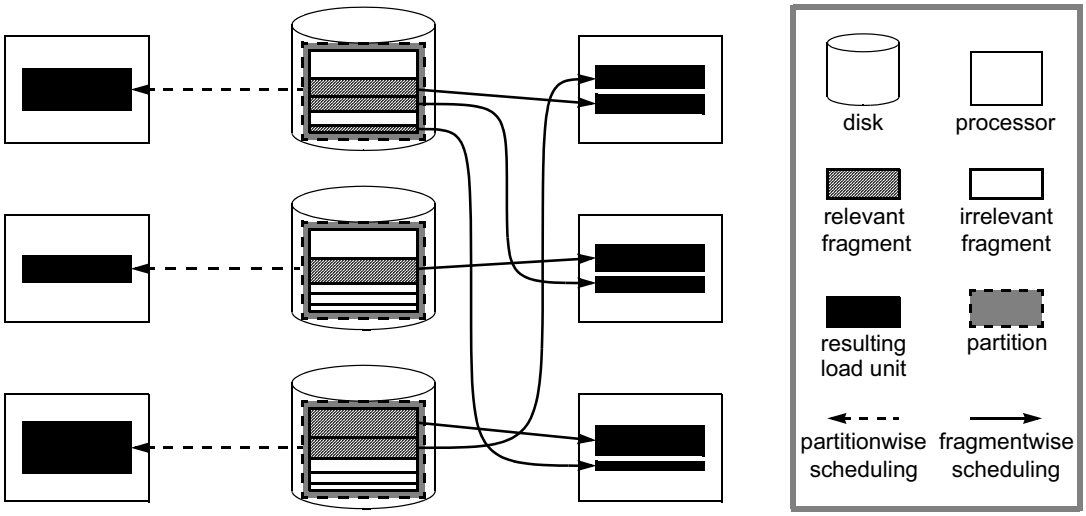


Figure 1. Partitionwise versus fragmentwise processing. Fragments provide a better load balance but require a larger number of subqueries (more arrows) and also incur disk contention.

union of all table fragments residing on the same disk (Figure 1). If the table is logically fragmented and its fragmentation attributes are also referenced in the query's selection predicate, some fragments may be excluded from processing because they are known to contain no hit rows. Subqueries will be generated only for relevant fragments or partitions, respectively. With either granule, we obtain independent subqueries that are uniform in structure and can be processed isolated from each other in arbitrary order, each on any one processing node. This gives us great flexibility in the subsequent scheduling step.

As fragments are much smaller than partitions, they permit a more even load distribution, especially in case of skew (Figure 1). Larger granules like partitions, however, require less communication between the coordinator and the other processing nodes and also reduce the overhead of scheduling itself. Furthermore, they minimize inter-subquery contention on the disks as no two nodes will process the same table partition, although subqueries may still interfere with each other when accessing bitmap indices (if present).

3.1. Scheduling

Presuming the voluminous queries we examine to work in full parallelism on all available processing nodes, we are left with the task of allocating subqueries to processors and timing their execution. We consider this *scheduling* step particularly important as it finalizes the actual load distribution in the system.

Based on the load granule, the query coordinator maintains a list of subqueries that are assigned to processors following the given ordering policy and processed locally as described below.

Each processor may obtain several subqueries up to a given limit t, and processing nodes are addressed in a round-robin manner, providing an equal number of subqueries per node (± 1). If there are more subqueries than can be assigned to the nodes, as is usually the case for larger queries, the remainder are kept in a central queue. When a processor finishes a subquery and reports the local result to the coordinator, it is assigned new work from the queue until all subqueries are done. Finally, the coordinator returns the overall query result to the user.

This simple, highly dynamic approach already provides a good balance of processor load. A node that has been assigned a long-running subquery will automatically obtain less load as execution progresses, thus nearly equalizing CPU load. Since no two subqueries address the same fragment, we may also achieve low disk contention by having different processors work on disjoint subsets of data. Specifically, tasks intensely addressing the same disk(s) should not be executed at the same time; consequently, they should be kept apart in the subquery queue. The specific ordering heuristics used in our study are detailed in Section 4.

3.2. Local processing of subqueries

When a node is assigned a fragment-sized subquery, it processes any required bitmap fragments and the respective table fragment simultaneously, minimizing memory consumption. Prefetching is used to read multiple pages into the buffer with each disk I/O. Furthermore, parallel I/O is employed for bitmap pages read from different disks. For the scan/aggregation queries we assume, the measures contained in the selected tuples are aggregated locally to avoid a shipping of large datasets, and the partial results are returned to the coordinator node at subquery termination. For partition-sized load granules, a node will process sequentially (i.e. in logical order) all the relevant fragments within its partition, simply skipping the irrelevant ones. Aggregates will be returned only for the entire partition to minimize communication costs. Multiple subqueries on the same processor coexist without any need for intra-node coordination. The maximum number of concurrent tasks per node, t, should correspond roughly to the performance ratio of CPUs to disks to achieve a good resource utilization (cf. Section 6.1).

4. SCHEDULING ORDER OF QUERY EXECUTION

As mentioned in the previous section, we regard the scheduling of subquery execution as the most important aspect of load balancing in our processing model as it determines the actual load distribution for both processors and disks. Based on the general scheme outlined above, the following subsections present scheduling policies based on either static (Section 4.1) or dynamic (Section 4.2) ordering of subqueries. Section 4.3 summarizes the final strategies compared in our simulation study.

4.1. Statically ordered scheduling

Our simpler heuristics employ a *static ordering* of subqueries. Note that even under these strategies, our scheduling scheme as such is still dynamic since the allocation of load units to processing nodes is determined at runtime based on the progress of execution.

4.1.1. *By partition number*

Subqueries are dispatched in a round-robin fashion with respect to the relevant table partitions, where each partition corresponds to one disk as defined in Section 3. Ideally, this means that no more than one subquery should work on a partition at any given time, unless there are more concurrent subqueries than partitions. In practice, subqueries do not necessarily finish in the same order in which they are started, so that disk load may still become skewed over time with fragment-sized load units. For partitionwise scheduling, of course, each table partition is guaranteed to be accessed by only one processor. With both granules, bitmap access (if required) can cause each subquery to read from multiple disks, so that a certain degree of access conflict may be inevitable. Still, we expect this strategy to achieve very low disk contention in all cases.

4.1.2. *By fragment number*

This heuristic applies only to fragment-sized load granules. It assigns subqueries in the logical order of the fragments they refer to. In the default case with a round-robin allocation of fragments to disks (cf. Section 5.3), this is equivalent to scheduling by partition number as long as a query references a consecutive set of fragments. Otherwise, the two will be similar but not identical. For different allocation schemes, the correlation of fragment numbers to partition numbers may be lost completely. This strategy was used in [4] and is included here as a baseline reference.

4.1.3. *By size*

This policy starts the largest subqueries first, using the expected number of referenced disk pages as a measure. (See Section 4.2 for its calculation.) It implements an LPT (*longest processing time first*) scheme that has been proven to provide good load balancing for many scheduling problems [38]. It does not consider disk allocation in any way but may be expected to optimize the balance of processor load which primarily depends on the total amount of data processed per node.

4.2. Dynamically ordered scheduling

The static ordering policies described so far tend to optimize the balance of *either* CPU *or* disk load. Implementing an integrated load balancing requires a more elaborate, *dynamic ordering* that reckons with both criteria: in order to distribute disk load over time and reduce contention, we estimate for every subquery its expected access volume on each disk and then try to execute concurrently those tasks that have minimum overlap in disk access. To simultaneously balance CPU load, we additionally consider the sizes of subqueries similar to the previous section.

4.2.1. *Disk access conflicts*

Before we can present our integrated scheduling method, we have to detail the calculation of disk access conflicts it incorporates. For simplicity of presentation, we will assume a load granule of single fragments, but all further considerations equally apply to partition-sized subqueries. Similarly, we will

refer to tables with supporting bitmap indices as used in our data warehouse application, although all concepts can be easily transferred to other data structures.

One way to consider the current disk utilization is to constantly measure the actual load of all disks and periodically propagate it to the coordinator node. These statistics may be quickly outdated especially if the update interval is higher than the execution time of individual subqueries. To avoid these problems, we construct our own image of disk utilization based on estimated access profiles of single subqueries, the sum of which can be updated instantaneously in every scheduling step. First, we model the expected number of pages referenced per disk for each given subquery. For the relevant table fragment, this number is calculated from the subquery's estimated selectivity using an approximation of Yao's formula [39,40]. We further take into account the associated bitmap fragments as far as they are required for the query. The result is a *load vector*

$$p^s = (p_1^s, p_2^s, \ldots, p_D^s)$$

where p_d^s denotes the expected number of pages accessed on disk d by subquery s[§]. But since the degree of contention between subqueries *at a given point in time* does not depend on their total sizes, we are more interested in the distribution of load across the disks than in its absolute magnitude. Assuming that the execution time of s is approximately proportional to the total amount of data processed, we normalize the load vectors based on the total size of a subquery that computes straightforward as

$$\hat{p}^s = \sum_{d=1}^{D} p_d^s$$

(This value is also used for the scheduling-by-size heuristic above.) We divide each subquery's load vector by $\hat{p}^s$ to obtain its *intensity vector*

$$i^s = (i_1^s, i_2^s, \ldots, i_D^s)$$

which is now normalized to a total of 1, so that each coefficient i_d^s denotes the percentage of its load that subquery s puts on disk d.

We now define similar coefficients for a *set* of subqueries executed concurrently. The intensity vector of a subquery set S is given by the itemwise sum of the single vectors, re-normalized to 1 by division with the number of subqueries, $|S|$:

$$I^S = \sum_{s \in S} i^s / |S| \quad \text{with} \quad I_d^S = \sum_{s \in S} i_d^s / |S|$$

Analogous to single subqueries, it once again denotes the percentagewise load distribution across all disks, but this time for the entire subquery set. Note that we added the intensity vectors rather than the original load vectors of the subqueries because we are interested in the *current* load distribution (per time unit), so we would not want to weight subqueries by their sheer sizes at this point.

[§]For our star schema application, p^s will typically contain one large value representing the fact table fragment and several smaller ones for the required bitmap fragments; the remaining items will be 0.

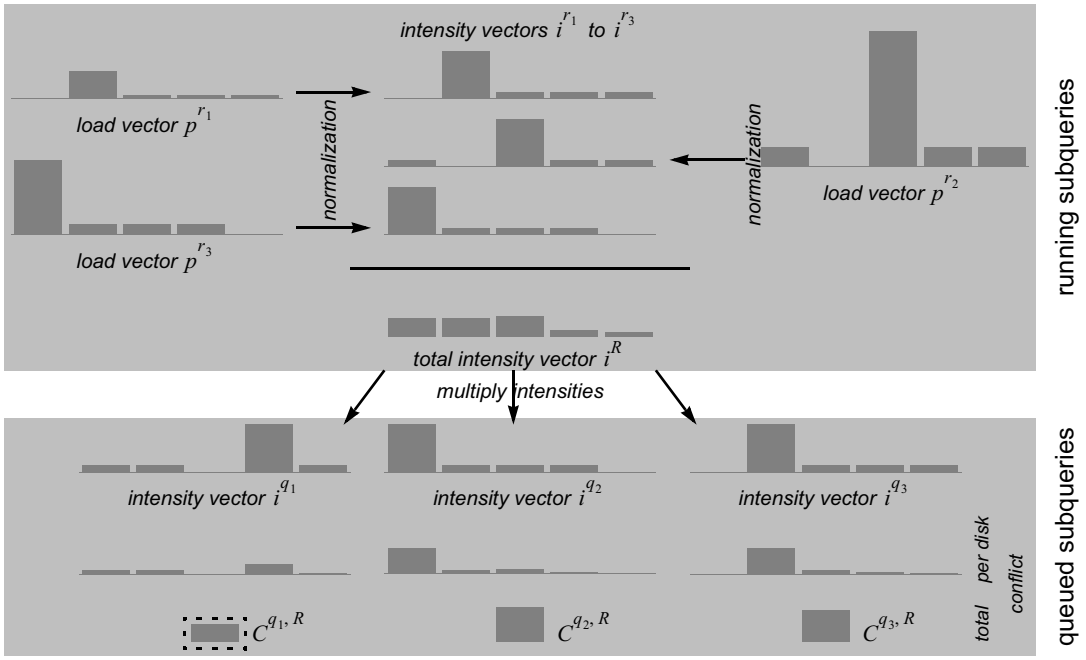


Figure 2. Example of conflict calculation. Three queued subqueries (q_1, q_2, q_3) are evaluated for concurrent execution with three running subqueries (r_1, r_2, r_3); q_1 has the lowest rate of conflict and is selected. The shape of the load vectors is typical of our application (one peak for a large table fragment, plus several small bitmap fragments).

Finally, we can define the *disk access conflict* between a single subquery s (to be scheduled) and a set of subqueries S (to be processed concurrently) as

$$C^{s,S} = \sum_{d=1}^{D} c_d^{s,S} \quad \text{with} \quad c_d^{s,S} = i_d^s \cdot I_d^S$$

This means that we first calculate the conflict of s and S *per disk* by multiplying their intensities locally, then add the results to obtain the total conflict rate. This total will have a value between zero (no conflict, i.e. s and S use disjoint disks) and one (maximum conflict, if s and S use the same single disk). The calculation of conflicts between concurrent subqueries is illustrated in Figure 2 using a five-disk example; each table fragment is assumed to have three associated bitmap fragments.

4.2.2. *Integrated scheduling*

Based on this notation, we can now identify in every scheduling step the subqueries that show minimum conflict with the set of subqueries currently running, R, enabling us to balance disk load over the

Table I. Scheduling strategies used in simulation experiments.

Notation	Load granule	Ordering
Partition	partition	static, by partition number
Logical	fragment	static, by fragment number
Size	fragment	static, by size
Integrated	fragment	dynamic, by conflict and size

duration of the query. To simultaneously address the distribution of CPU load, we also consider the size of a subquery as in the static ordering by size from the previous section. Specifically, we want long-running subqueries to have priority over—i.e. to be executed earlier than—shorter ones even if they incur a somewhat higher degree of disk contention. With Q denoting the set of queued subqueries, the final resulting policy can be phrased as

'select $q \in Q$ so that $C^{q,R}/\hat{p}^q$ is minimized'.

Since the intensity vector for the current disk load, I^R, will change with every subquery starting or finishing execution, it is now clear that the order of subqueries must indeed be determined dynamically. In Figure 2, subquery q_1 has minimal conflict with the subqueries already running, mainly because the table fragment it uses is on a different disk than those accessed by r_1 through r_3. It will thus be executed next unless it is significantly smaller than q_2 and q_3.

4.2.3. Variants

Numerous extensions to this strategy were evaluated in our study, but are omitted here for lack of space. These include the consideration of conflicts with other queued subqueries (to reduce contention in the future), different weightings of conflict against size (including the conflict criterion alone), as well as a second normalization of load vectors to compensate for a skewed overall load for the entire query.

4.3. Proposed strategies

Load granules and scheduling policies may, in principle, be combined arbitrarily, with the exception that partitionwise scheduling cannot sensibly be based on fragment numbers, as each partition contains several fragments. In the remainder of this paper, we will consider the four scheduling strategies listed in Table I with the indicated notations. We have, in fact, experimented on many more strategies, but will report in detail only on the more relevant ones.

5. SIMULATION SYSTEM AND SETUP

Our proposed strategies were implemented in a comprehensive simulation system for parallel data warehouses that has been used successfully in previous studies [4], extended with the query processing

Table II. System parameters used in simulations.

Parameter		Value	Parameter		Value
Processing nodes			Buffer manager		
total number (P)	general	20	page size		8 KB
	speed-up	2–50	buffer size	fact table	5000 pages
CPU speed		100 MIPS		bitmaps	5000 pages
subqueries per node (t)		varied		other	5000 pages
			prefetch size		8 pages
No. of instructions					
per query	initiate/plan	100 000	Disk devices		
	terminate	10 000	total number (D)	general	100
per subquery	initiate/plan	10 000		speed-up	20–100
	terminate	10 000	average seek time		8 ms
per I/O	read overhead	1500	average settle time	per access	4 ms
	+ per page	400	+ controller delay	+ per page	0.5 ms
per bitmap page	uncompressed	5000			
	compressed	20 000	Network		
per table row	extract	100	connection speed		100 Mbit/s
	aggregate	100	message size	small	128 B
per message	send	1000 + # bytes		large	1 page (8 KB)
	receive	1000 + # bytes			

methods detailed above. The following subsections describe the architecture and parameters of the simulated DBMS (Section 5.1), our sample database schema (Section 5.2), the modeling and treatment of skew effects (Section 5.3), and the query workload used in our subsequent experiments (Section 5.4).

5.1. System architecture and parameters

For this study, we simulate a generic SD PDBS and use the parameters given in Table II. The system realistically reflects resource contention by modeling CPUs and disks as servers. CPU overhead is reckoned for (sub-)query start-up, planning and termination; I/O initiation; page access; scanning of bitmaps; extraction and aggregation of fact rows; as well as communication overhead. Seek times in the disk modules depend on the location (track number) of the desired data within a disk. Each processor has an associated buffer module maintaining separate LRU queues for different page types (fact table, bitmap indices, permanent allocation). The network incurs communication delays proportional to message sizes but models no contention, so as to avoid specific network topologies unduly influencing experimental results.

5.2. Sample database schema and fragmentation

The data warehouse scenario in which we evaluate our load balancing methods models a relational star schema for a sales analysis environment (Figure 3) that was derived from the specification of the *Application Processing Benchmark (APB-1)* [41]. The denormalized *dimension tables Product*, *Customer*, *Channel* and *Time* each define a *hierarchy* (such as product divisions, lines, families,

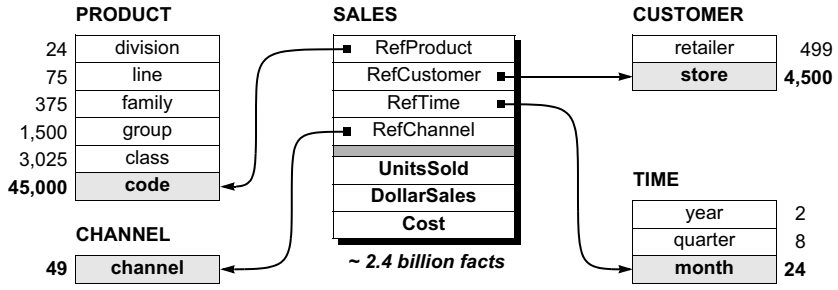


Figure 3. Sample star schema.

and so on). The *fact table Sales* comprises several *measure* attributes (turnover, cost etc.) and a foreign key to each dimension. With a *density factor* of 1%, it contains a tuple for 1/100 of all value combinations. A typical two-dimensional *star query* on this schema might, for instance, aggregate the turnover of a retailer over a single month (denoted by $Q_{\text{RetailerMonth}}$) which can be expressed in SQL as

```
SELECT SUM(DollarSales)
FROM Sales S, Customer C
WHERE C.Retailer = RETAILER
AND S.RefTime = MONTH
AND C.RefCustomer = C.Store
```

We incorporate common *bitmap join indices* [2] to avoid costly full scans of the fact table. We employ *standard bitmaps* for the low-cardinality dimensions *Time* and *Channel*, but use *hierarchically encoded bitmaps* [42] for the more voluminous dimensions *Product* and *Customer* to save disk space and I/O. With these indices, queries like the above can avoid explicit join processing between fact table and dimension table(s) in favor of a simple selection using the respective precomputed bitmap(s).

We follow a horizontal, multi-dimensional and hierarchical fragmentation strategy for star schemas (*MDHF*) that we proposed and evaluated in [4]. Specifically, we choose a two-dimensional fragmentation based on *Time.Month* and *Product.Family*. Each resulting fact table fragment combines all rows referring to one particular product family within one particular month, creating $n = 375 \times 24 = 9000$ fragments, according to the respective level cardinalities (cf. Figure 3). This can significantly reduce work for queries referencing one or both of the fragmentation dimensions by clustering hit rows and confining disk access to relevant fragments. It also supports both processing and I/O parallelism and scales well.

As demanded in Section 3, the fragmentation of bitmaps exactly follows that of the fact table, so that each fact table fragment can be matched with its associated bitmap fragments for parallel processing.

5.3. Skew effects and data allocation

One focus of our study is on skew effects, as these can pose a serious problem to effective load balancing. Specifically, the sizes of table fragments—and thus, of load units—can vary significantly,

potentially causing severe load imbalance. Apart from some skew inherent in the dimension hierarchies (for instance, 45 000 *Product Codes* cannot be split evenly into 3025 *Classes*), there is also *attribute value skew* [43] with respect to dimension values. With different products, customers, etc. occurring in the fact table at varying frequencies, single fact table fragments will deviate from the average density of 1% defined in Section 5.2. We represent such *density skew* by zipf-like frequency distributions on the bottom hierarchy levels of all dimensions, permuted across their respective domains so that neighboring attribute values need not have similar frequencies. These distributions are normalized to guarantee a constant total fact table size, which is essential to the interpretability of our simulation results.

To alleviate such skew effects, we compare two different methods of allocating table and bitmap fragments to disks. In addition to the round-robin pattern employed in [4], we also test an alternative method designed to create evenly sized disk partitions. Specifically, we employ a *greedy data placement algorithm* originally proposed in [25] for the declustering of files in parallel disk systems. This method allocates fact table fragments in *decreasing order of size* onto the least occupied disk at each time to keep disk partitions balanced. In both cases, the fragments of each bitmap are stored according to the allocation order of the associated fact table fragments: for a fact table fragment located on disk i, the k associated bitmap fragments are stored on disks i MOD $D, \ldots, (i+k-1)$ MOD D (where D denotes the number of disks), supporting parallel bitmap access.

Note, however, that a smart allocation scheme is merely a complement, not a replacement for intelligent scheduling techniques, as equal partition sizes are not sufficient to achieve a balanced system load at runtime.

5.4. Query workload and processing

A query generator module creates data structures representing queries and passes them to the processing module for execution. As our study regards single-user mode for the time being, queries are executed strictly sequentially. Focusing on fact table access, we assume simple aggregation queries similar to the example in Section 5.2 that do not require joins to the dimension tables. All queries within a single experiment are of the same type (e.g. Q_{Division}) but with random parameters (e.g. the specific division selected). Note, though, that different simulation runs will use the same sequence of queries, facilitating a fair comparison of results. The exact queries will be introduced in Section 6 as they are used.

In processing queries, we follow the algorithms detailed in Sections 3 and 4. In particular, we use the scheduling policies proposed in Section 4.3 with their corresponding notations.

6. SIMULATION EXPERIMENTS

We now discuss the results of several simulation series. Our first goal is to determine a sensible degree of intra-query parallelism, as this is a fundamental parameter to choose for all further experiments (Section 6.1). After that, we study in detail the performance of our scheduling schemes, especially the dependency between different types of queries and the ordering policies best suited for them (Section 6.2). In addition, we present speed-up experiments to test the stability of our load balancing approach in general and of our scheduling methods in particular for varying system configurations (Section 6.3).

6.1. Intra-query parallelism

Our first goal is to determine the degree of intra-query parallelism in which to process star queries. While it is straightforward to employ all processing nodes due to single-user mode (cf. Section 3), we have yet to select the number of concurrent subqueries per node, t, which must be large enough to fully utilize the resources. We test this issue on a skew-free database using the round-robin allocation method in order to get a good impression of resource utilization under ideal conditions, not distorted by any allocation or scheduling anomalies. For the same reason, we use long-running queries rather than shorter ones. Our scheduling policy is *Logical*, which performs reasonably well under these conditions (cf. Section 6.2).

In this and all further experiments, we distinguish disk-bound from CPU-bound queries, which often show quite different behavior depending on the utilization of both resources. For our application, the 'boundness' of a query depends on its *selectivity within* the fact table fragments it reads as this determines the amount of CPU work performed per I/O. In this simulation series, we use the CPU-intensive queries Q_{FullScan} and Q_{Year} which have a 100% selectivity within the fact table fragments they access (although Q_{Year} selects only half of the fragments). We contrast them with Q_{Channel} and Q_{Store}, which are I/O-bound because (i) they select only some of the tuples in each fragment, causing less CPU work per I/O, and (ii) they perform their selection by means of the bitmap indices, which are also cheap to process on the CPU side. Q_{Store} is more strongly disk-focused with a selectivity of just $1/4500$ and access to 13 (encoded) bitmaps, whereas Q_{Channel} selects $1/49$ of the data and reads just one (standard) bitmap. These selectivities are based on the hierarchy level cardinalities as shown in Figure 3.

The results seen in Figure 4 show that a good value of t is in the range 4–6, depending on the query type. Q_{FullScan} and Q_{Year} can fully load each processor with just four subqueries, which apparently suffices to avoid idle cycles during I/O of single subqueries. Q_{Channel} and Q_{Store}, however, need more concurrent subqueries in order to keep all disks busy at all times. Not accidentally, the optimum is reached at $t = 5$, which corresponds to our ratio of 100 disks to 20 (single-CPU) processing nodes. Thus, the total number of subqueries equals the number of disks.

A higher-than-optimal value of t does not seem to cause any delays due to excessive contention. As a consequence, we can simply use $t = 5$ for all further experiments. We refrain from using even higher values, however, as these would reduce flexibility in scheduling by binding subqueries to specific processors earlier than necessary.

6.2. Scheduling strategies

As discussed in Section 4, we expect the optimal scheduling strategy to depend in part on the type of query to be processed, because our various policies tend to optimize the utilization of different resources. Consequently, we conduct simulations for both disk-bound and CPU-bound queries. Furthermore, we identify borderline cases where queries can shift from one category to the other. All queries are tested for our four scheduling strategies under varying degrees of skew, for both round-robin and greedy allocation. In these experiments, we model density skew on the two fragmentation dimensions, *Time* and *Product*, to create differently sized fragments. Under the zipf-like distributions we employ, the skew parameter may range from zero, denoting the absence of skew, to values around one, signifying very heavy skew. Note that we apply the same degree of skew to both dimensions,

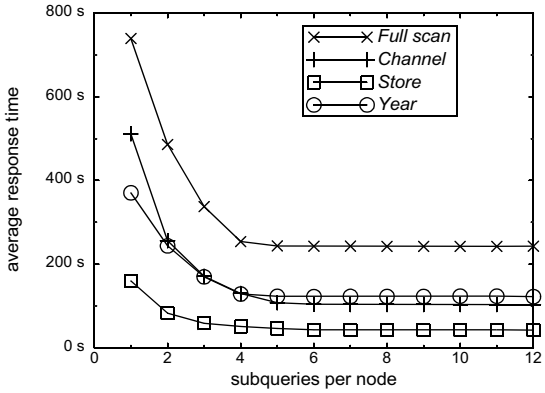


Figure 4. Subqueries per note (various queries—round-robin allocation).

leading to an even stronger combined skew effect. In contrast, the skew in fragment sizes is not as extreme as on single attribute values, as each fragment comprises a range of values especially for the *Product* dimension.

6.2.1. *Disk-bound queries*

In Figure 5, we show simulation results for the two disk-bound queries introduced above, Q_{Channel} and Q_{Store}. The first observation is that for round-robin allocation, response times increase sharply with density skew up to a factor of 4.6 (Q_{Channel}). This comes as no surprise as this allocation strategy is unable to balance disk partitions when fragment sizes differ. As a consequence, the largest fact table partition will determine the duration of processing with little chance to remedy the situation by intelligent scheduling. The greedy allocation scheme, however, creates balanced disk partitions that can be processed in constant time irrespective of skew, if a proper scheduling method is selected. Even under less successful scheduling schemes, overall response times are about 50% lower than for round-robin.

With greedy allocation, *Partition* achieves the best response times as would be expected for disk-bound workloads, because disks are optimally loaded at nearly 100%. This strategy also minimizes the inevitable disk contention caused by concurrent access to both fact table and bitmap fragments.

Integrated performs equally well as *Partition* for the Q_{Channel} query with only 1% deviation; it is only slightly worse on Q_{Store} with at most 15% response time increase[¶]. Apparently, the conflict

[¶]An exception occurs for the Q_{Store} query with hierarchy skew only (i.e. zero density skew) under round-robin allocation. Here, the *Integrated* method is trapped by several larger fragments allocated to a small number of disks and having exactly equal

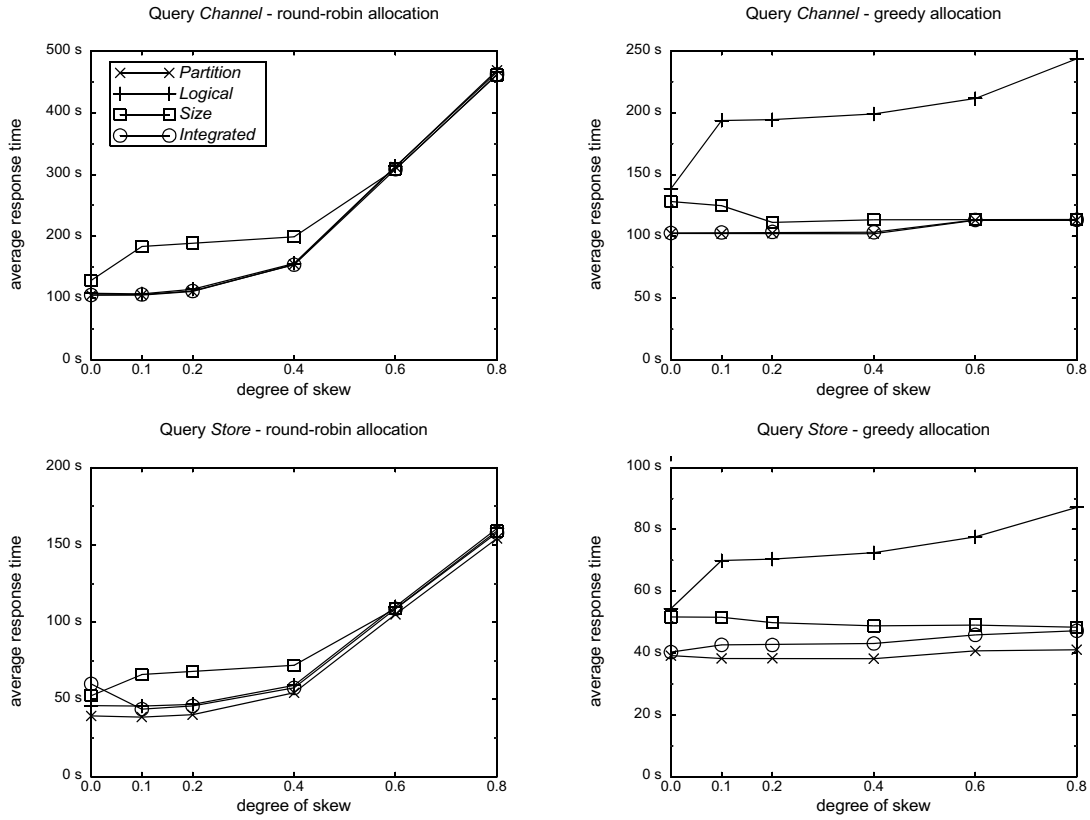


Figure 5. Disk-bound queries.

analysis it performs is similarly effective to avoid disk contention as a strict separation of partitions, despite the addition of a second criterion.

The other strategies are less successful here as they do not respect disk allocation to the same degree. The worst case is *Logical*, which processes fragments in their logical order that is unrelated to their disk location under the greedy scheme, more than doubling the response time. *Size* mimics partitionwise scheduling to some extent because it processes fragments in the same size-based order in which they were allocated, leading to minimal disk contention at least for the crucial 100 largest fragments. Still, it

sizes and conflict ratings. This anomaly disappears with even the slightest degree of density skew and would thus not exist in a real database.

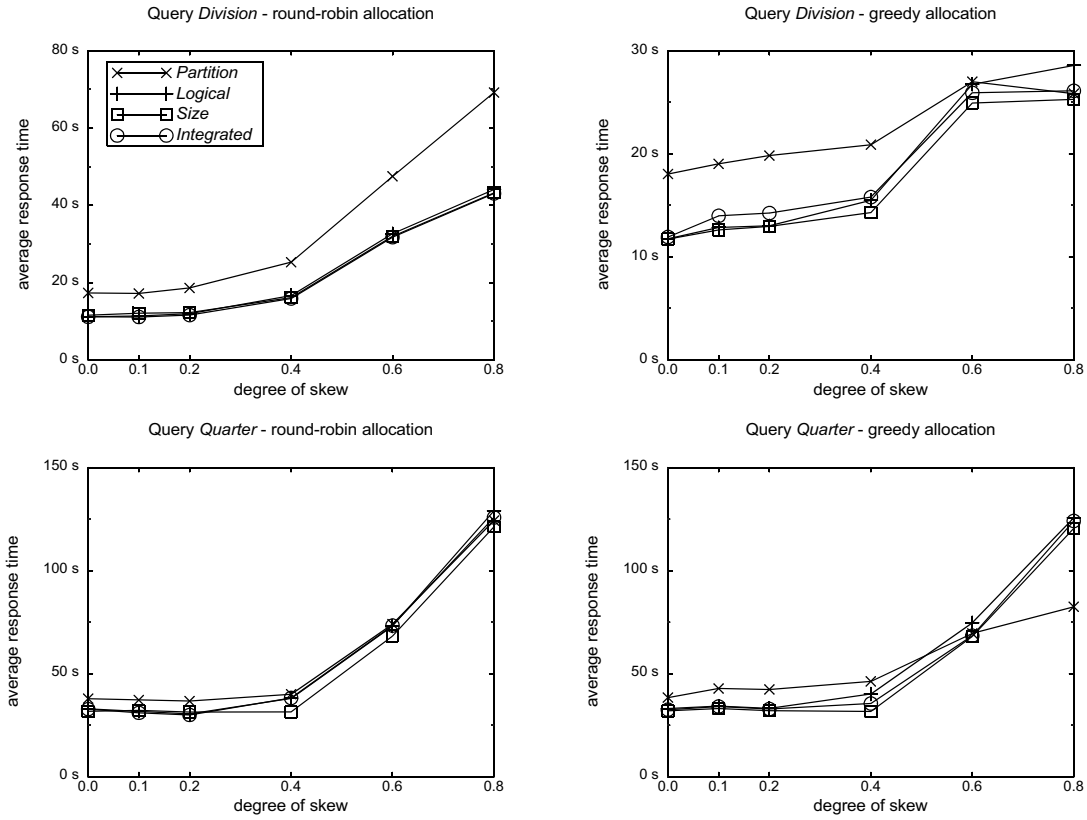


Figure 6. CPU-bound queries.

cannot contend with the near-optimal *Partition*, with differences of up to 25% (Q_{Channel}) and 35% (Q_{Store}), respectively.

6.2.2. *CPU-bound queries*

Figure 6 presents simulation results for CPU-bound queries. Rather than the unlikely full scan used for calibration in Section 6.1, we now consider somewhat smaller queries Q_{Division} and Q_{Quarter} that are more likely to respond to skew effects. This is because they perform a selection on the skewed fragmentation dimensions *Product* and *Time*, respectively. Thus, they process only a subset of fragments but access all tuples within the selected ones.

With CPU-bound queries, the processing nodes must be utilized as much as possible for good response times. This is best achieved by *Size* as it balances the sheer amount of data processed per node,

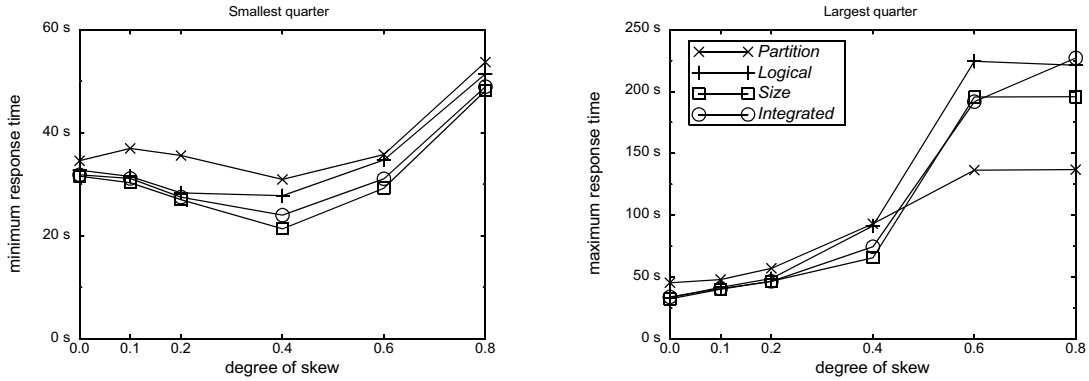


Figure 7. Shift from CPU- to disk-bound (query *Quarter*—greedy allocation).

which is the essential objective in this case. *Partition* performs worst (up to 58% for Q_{Division} and 46% for Q_{Quarter}) because it does not permit more than one processor to access the same disk even when the disks are underloaded. The other two strategies achieve good success; *Integrated* approximates *Size* most closely with no more than 10% deviation, demonstrating good performance for CPU-bound workloads as well.

Note that, in contrast to the previous section, response times increase with skew even under greedy allocation. The reason is that although the greedy scheme balances partition sizes for the database as a whole, this may be impossible for single product divisions or calendar quarters because the largest fragment *within* such a subset can be so huge that it requires a disproportionate amount of time for processing. In our sample database, the largest fragment of all comprises as much as 1/90 of all data under extreme skew[‖]. Clearly, it represents an even higher share of the respective quarter or division, so that reading it dominates the response time. This can be corrected by data allocation only to a limited extent.

Furthermore, increasing skew changes the ranking of scheduling methods in favor of *Partition*. This is most pronounced in Q_{Quarter}, where *Partition* becomes by far the best strategy for extreme skew, now offsetting *Size* by 46%. That is because the aforementioned suboptimal allocation due to skewed fragment sizes causes the query to become *locally disk-bound*, i.e. a single disk (or possibly a small subset of disks) becomes the bottleneck even though the query as a whole is CPU-bound!

This situation is analyzed in detail in Figure 7, which shows the response time development for the 'smallest' and the 'largest' query of the Q_{Quarter} type, respectively. More precisely, we consider the processing of the least densely and most densely populated quarters for each given degree of skew. For small queries, we observe two conflicting tendencies. First, response times tend to decrease as

[‖]This is because we have 9000 fragments of 1% average density. The largest fragment can achieve 100% density so that it contains $100 \times 1/9000 = 1/90$ of the entire database.

stronger skew produces greater differences between quarters, so that the smallest quarter is smaller with high skew than with low skew. Second, response times increase due to the growing imbalance in fragment sizes. From a skew degree of about 0.4, the second effect dominates the first, but the queries remain CPU-bound for the entire range as indicated by the superior performance of *Size* in contrast to *Partition* (10–45% over the full range).

For large quarters, response times increase monotonously as both the size of the respective quarter and the fragment imbalance increase with growing skew. It is only these queries that shift from CPU-bound to locally disk-bound so that *Partition* wins out by 43% for high skew.

6.2.3. Discussion

The results show that there is no single scheduling scheme that is absolutely optimal for all situations. For (globally or locally) disk-bound queries, minimal response times are normally achieved under the *Partition* heuristic, whereas CPU-bound workloads are best processed using *Size*. The correct choice of the truly best strategy then depends on the 'boundness' of a query, as determined by its selectivity and utilization of bitmap indices, the data allocation, the degree of skew, the number and relative performance of disks and CPUs, as well as page and tuples size and more. A cost-based query optimizer of a PDBS might make a sensible decision by comparing the total (estimated) processing cost on the CPU and disk side, respectively, although locally disk-bound queries may be hard to detect.

Alternatively, our dynamic scheduling scheme based on the *Integrated* heuristic was able to adapt to different types of queries and thus performed near-optimally in most experiments. Using this strategy thus promises to be more robust for complex workloads and avoids the need to select among different scheduling approaches based on error-prone cost estimates. Especially in a multi-user environment, we expect such an adaptive method to react more gracefully to the inevitable fluctuations in system load. In contrast, the correct selection of a simple method (*Partition* or *Size*) will become very difficult against a continually changing background load, especially when the latter alternates between CPU-bound and disk-bound states. This aspect, however, needs to be investigated in future studies.

6.3. Speed-up behavior

In this simulation series, we test the scalability of our query processing and scheduling strategies. We simulate two different star queries on several hardware configurations shown in Table III. The number of disks, D, is varied from 20 to 100; the number of processors, P, ranges from $1/10$ to $1/2$ of the number of disks, resulting in 2–50 processors. With the number of subqueries per processing node set to $t = 5$ as determined in Section 6.1, this leads to a total of 10–250 concurrent subqueries. For each configuration, we run the queries Q_{Quarter} and $Q_{\text{RetailerMonth}}$ under a medium skew degree of 0.4 and against skewless data, respectively. Results are shown in Figure 8.

Since Q_{Quarter} is a CPU-bound query, we test its speed-up in relation to the number of processors rather than the number of disks. For the same reason, we use *Size* as the scheduling strategy, according to the results obtained in Section 6.2. In addition, we try the *Integrated* policy that we proposed for both CPU- and disk-bound workloads. The results, however, are equivalent for both methods.

Processed against skewless data, Q_{Quarter} exhibits linear speed-up. The graphs drop only when the disks of the system become bottlenecks and speed-up with respect to processors is no longer achievable.

Table III. Hardware parameters for speed-up experiments.

Number of disks (D)	Number of processors (P)			
20	2	4	5	10
40	4	8	10	20
60	6	12	15	30
80	8	16	20	40
100	10	20	25	50

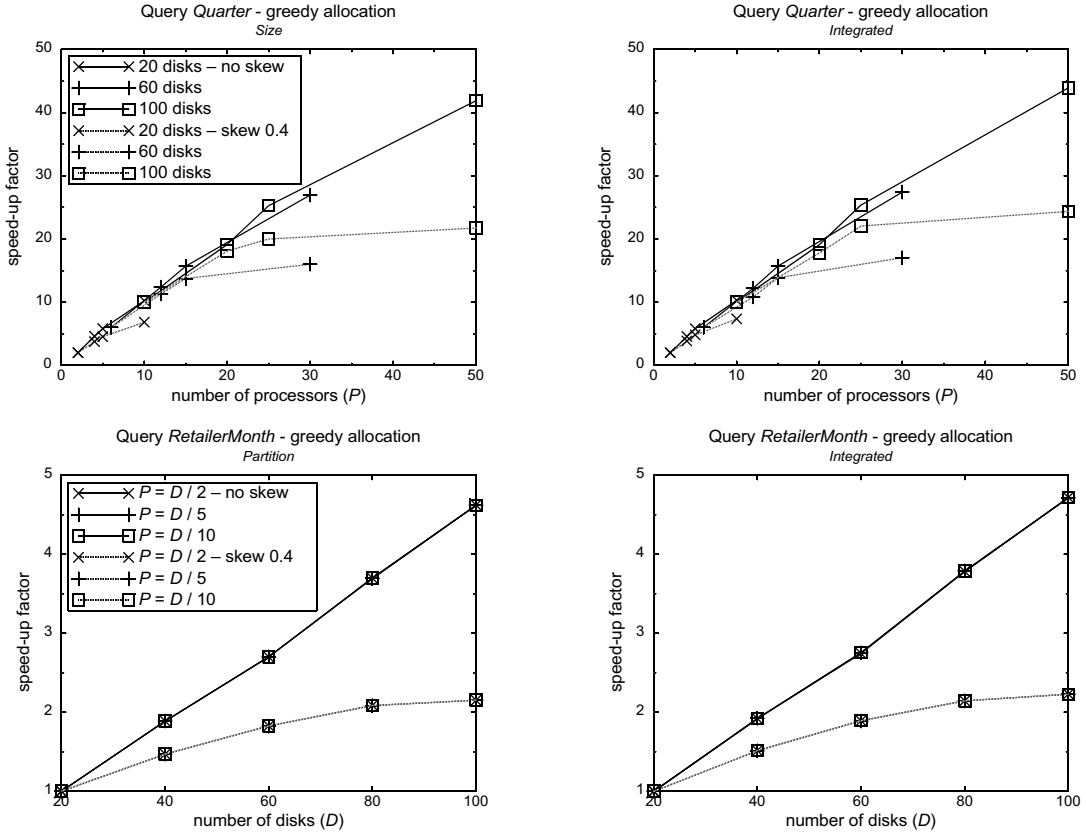


Figure 8. Speed-up behavior of queries Q_{Quarter} and $Q_{\text{RetailerMonth}}$.

This occurs at the last points of each curve, which were obtained with a disk-to-processor ratio of only 2:1.

With skew (dashed graph), the curves decline somewhat earlier because response times are dominated by the work to be done on the largest fragment as discussed before. This leads to increasing load imbalances and sub-optimal response times as the load per single resource (processor or disk) decreases.

To test the speed-up for disk-bound queries, we use $Q_{\text{RetailerMonth}}$ rather than Q_{Channel} and Q_{Store} as above. This is because the latter do not respond to skew to the same extent, as they perform no selection on a skewed dimension. In contrast to Q_{Quarter}, $Q_{\text{RetailerMonth}}$ is scheduled using *Partition* as well as *Integrated*. Speed-up is evaluated in relation to the number of disks.

Our results are similar to the previous case. For both policies, speed-up is again near-linear with skewless data but limited by the largest fragment in case of skew. The effect is even stronger this time as skew is more pronounced on lower hierarchy levels (months) than on higher ones (quarters; cf. Section 6.2).

Overall, our load balancing method scales very well for all relevant scheduling policies; limitations due to skewed fragment sizes are not caused by scheduling and must be treated at the time of data allocation.

7. CONCLUSIONS

In this paper, we have investigated load balancing strategies for the parallel processing of star schema fact tables with associated bitmap indices. We found that simple scheduling heuristics—most notably, *Partition* and *Size*—can be very effective; they are also easy to implement. But the selection of the appropriate method depends on the load properties of the query (CPU- or disk-bound) which can be difficult to determine in some cases, especially under skew conditions. As an alternative, we proposed a more complex, dynamically ordered scheduling approach (*Integrated*) that yields only slightly poorer performance but naturally adapts to different query types. Although our study was conducted on a SD architecture, most of the results can be transferred to other environments, in particular, Shared Everything systems. For Shared Nothing, our methods could be adapted to manage multiple disks attached to a single processing node. In this case, disk-sensitive heuristics such as *Partition* can be deployed locally to achieve an even disk utilization with low contention between subqueries. Still, Shared Nothing systems will remain inferior for CPU-bound workloads as they do not support strategies like *Size* or *Integrated*.

Our scheduling methods can be applied not only to star schemas but to arbitrary data sets, provided that the latter are horizontally partitioned into independent fragments. Among others, this may include tables with tree-based indices (instead of bitmaps) as well as materialized views. A strategy similar to *Integrated* has been successfully applied to object-relational joins [44], where it was used to manage concurrent access to intermediate results stored on disk. Operators without disk access (e.g. in-memory joins) are outside the scope of our algorithms, but the CPU load they cause is implicitly considered in our strategies as it hinders the execution of disk-based operations.

As mentioned in Section 6.2.3, the extension of our findings to multi-user mode is not trivial. In particular, the simple heuristics *Partition* and *Size* may no longer be sufficient as the processing of one query depends on the system load state caused by others executed concurrently. In our future

work on this subject, we thus expect our integrated strategy to gain importance, although its stability in these cases is yet to be ascertained.

ACKNOWLEDGEMENT

Part of this work was sponsored by the Deutsche Forschungsgemeinschaft under grant no. Ra497/10.

REFERENCES

1. Gray J *et al.* Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, vol. 1, Fayyad U, Mannila H, Piatetsky-Shapiro G (eds.), 1997; 29–53.
2. O'Neil P, Graefe G. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record* 1995; **24**(3):8–11.
3. O'Neil P, Quass D. Improved query performance with variant indexes. *Proceedings of the ACM SIGMOD Conference*, Tucson, AZ, 1997. ACM Press, 1997.
4. Stöhr T, Märtens H, Rahm E. Multi-dimensional database allocation for parallel data warehouses. *Proceedings of the 26th VLDB Conference*, Cairo, 2000. Morgan Kaufmann: San Francisco, CA, 2000.
5. Baralis E, Paraboschi S, Teniente E. Materialized view selection in a multidimensional database. *Proceedings of the 23rd VLDB Conference*, Athens, 1997. Morgan Kaufmann: San Francisco, CA, 1997.
6. Gupta A, Mumick IS. Maintenance of materialized view: Problems, techniques, and applications. *Data Engineering Bulletin* 1995; **18**(2):3–18.
7. Chaudhuri S, Dayal U. An overview of data warehousing and OLAP technology. *SIGMOD Record* 1997; **26**(1):65–74.
8. DeWitt DJ, Gray J. Parallel database systems: The future of high performance database systems. *Communications of the ACM* 1992; **35**(6):85–98.
9. Bouganim L, Florescu D, Valduriez P. Dynamic load balancing in hierarchical parallel database systems. *Proceedings of the 22nd VLDB Conference*, Bombay, 1996. Morgan Kaufmann: San Francisco, CA, 1996.
10. DeWitt DJ, Naughton JF, Schneider DA, Seshadri S. Practical skew handling in parallel joins. *Proceedings of the 18th VLDB Conference*, Vancouver, 1992. Morgan Kaufmann: San Francisco, CA, 1992.
11. Lu H, Tan KL. Dynamic and load-balanced task-oriented database query processing in parallel systems. *Proceedings of the 3rd EDBT Conference*, Vienna, 1992 (*Lecture Notes in Computer Science*, vol. 580). Springer: Berlin, 1992.
12. Manegold S, Obermaier JK, Waas F. Load balanced query evaluation in shared-everything environments. *Proceedings of the 3rd Euro-Par Conference*, Passau, 1997 (*Lecture Notes in Computer Science*, vol. 1300). Springer: Berlin, 1997.
13. Nodine MH, Vitter JS. Deterministic distribution sort in shared and distributed memory multiprocessors. *Proceedings of the 5th ACM SPAA*, Velen, 1993. ACM Press, 1993.
14. Rahm E, Stöhr T. Analysis of parallel scan processing in parallel shared disk database systems. *Proceedings of the 1st Euro-Par Conference*, Stockholm, 1995 (*Lecture Notes in Computer Science*, vol. 966). Springer: Berlin, 1995.
15. Brown KP, Mehta M, Carey MJ, Livny M. Towards automated performance tuning for complex workloads. *Proceedings of the 20th VLDB Conference*, Santiago, 1994. Morgan Kaufmann: San Francisco, CA, 1994.
16. Rahm E, Marek R. Dynamic multi-resource load balancing in parallel database systems. *Proceedings of the 21th VLDB Conference*, Zurich, 1995. Morgan Kaufmann: San Francisco, CA, 1995.
17. Spiliopoulou M, Freytag JC. Modelling resource utilization in pipelined query execution. *Proceedings of the 2nd Euro-Par Conference*, Lyon, 1996 (*Lecture Notes in Computer Science*, vol. 1123/1124). Springer: Berlin, 1996.
18. Avnur R, Hellerstein JM. Eddies: Continuously adaptive query processing. *Proceedings of ACM SIGMOD Conference*, Dallas, TX, 2000. ACM Press, 2000.
19. Märtens H. Skew-insensitive join processing in shared-disk database systems. *Proceedings of the 3rd IDPT Conference*, Berlin, 1998. Society for Design and Process Science: Austin, TX, 1998.
20. Zhou X, Orlowska ME. Handling data skew in parallel hash join computation using two-phase scheduling. *Proceedings ICA^{3}PP Conference*, Brisbane, 1995. IEEE Computer Society Press: Washington, DC, 1995.
21. Chen PM, Lee EL, Gibson GA, Katz RH, Patterson DA. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys* 1994; **26**(2):145–185.
22. Ghandeharizadeh S, DeWitt DJ, Qureshi W. A performance analysis of alternative multi-attribute declustering strategies. *Proceedings of the ACM SIGMOD Conference*, San Diego, CA, 1992. ACM Press, 1992.
23. Mehta M, DeWitt DJ. Data placement in shared-nothing parallel database systems. *VLDB Journal* 1997; **6**(1):53–72.
24. Rahm E, Märtens H, Stöhr T. On flexible allocation of index and temporary data in parallel database systems. *Proceedings of the 8th HPTS Workshop*, Asilomar, 1999. http://research.microsoft.com/~gray/hpts99/.

25. Scheuermann P, Weikum G, Zabback P. Data partitioning and load balancing in parallel disk systems. *VLDB Journal* 1998; **7**(1):48–66.
26. Wu K-L, Yu PS, Chung J-Y, Teng JZ. A performance study of workfile disk management for concurrent mergesorts in a multiprocessor database system. *Proceedings of the 21st VLDB Conference*, Zurich, 1995. Morgan Kaufmann: San Francisco, CA, 1995.
27. Rahm E. Parallel query processing in shared disk database systems. *Proceedings of the 5th HPTS Workshop*, Asilomar, 1993.
28. Reuter A. Load control and load balancing in a shared database management system. *Proceedings 2nd ICDE Conference*, Los Angeles, CA, 1986. IEEE Computer Society Press: Washington, DC, 1986.
29. Mohan C, Narang I. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. *Proceedings of the 17th VLDB Conference*, Barcelona, 1991. Morgan Kaufmann: San Francisco, CA, 1991.
30. Rahm E. Empirical evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. Database Systems* 1993; **18**(2):333–377.
31. International Business Machines Corp. *IBM DB2 Universal Database for OS/390 and z/OS Administration Guide Version 7*. Part No. SC26-9931-02, October, 2002. IBM Corp.: White Plains, NY, 2002.
32. Oracle Corp. *Oracle9i Data Warehousing Guide*. Part No. A96520-01, March. Oracle Corp.: Redwood Shores, CA, 2002.
33. Sybase, Inc. *Sybase Adaptive Server IQ 12.4.3 Administration and Performance Guide*. Document ID 38152-01-1243-01, May, 2001. Sybase Corp.: Dublin, CA, 2001.
34. International Business Machines Corp. *Performance Guide for IBM Informix Extended Parallel Server Version 8.40*. Part No. 000-9083, August, 2002. IBM Corp.: White Plains, NY, 2002.
35. International Business Machines Corp. *IBM Red Brick Warehouse Version 6.20 Query Performance Guide*. Part No. 000-9045, August, 2002. IBM Corp.: White Plains, NY, 2002.
36. Berenson H, Delaney K. *Microsoft SQL Server Query Processor Internals and Architecture*. White Paper, Microsoft Corp. Microsoft Corp.: Redwood, WA, 2000.
37. NCR Corp. *Teradata RDBMS Performance Optimization V2R4.1*. Part No. B035-1097-061A, October, 2001. NCR Corp.: Dayton, OH, 2001.
38. Graham RL. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics* 1969; **17**(3):416–429.
39. Gardy D, Némirovski L. Urn models and Yao's formula. *Proceedings of the 7th ICDT Conference*, Jerusalem, 1999 (*Lecture Notes in Computer Science*, vol. 1540). Springer: Berlin, 1999.
40. Yao SB. Approximating block accesses in data base organizations. *Communications of the ACM* 1977; **20**(4):260–261.
41. *APB-1 OLAP Benchmark, Release II*. OLAP Council, November 1998. http://www/olapcouncil.org.
42. Wu M-C, Buchmann AP. Encoded bitmap indexing for data warehouses. *Proceedings 14th ICDE Conference*, Orlando, FL, 1998. IEEE Computer Society Press: Washington, DC, 1998.
43. Walton CB, Dale AG, Jenevein RM. A taxonomy and performance model of data skew effects in parallel joins. *Proceedings of the 17th VLDB Conference*, Barcelona, 1991. Morgan Kaufmann: San Francisco, CA, 1991.
44. Märtens H. On parallel join processing in object-relational database systems. *Proceedings of the 9th BTW Conference*, Oldenburg, 2001. Springer: Berlin, 2001.

