Dynamic Query Scheduling in Parallel Data Warehouses

Holger Märtens, Erhard Rahm, and Thomas Stöhr

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

Abstract. Data warehouse queries pose challenging performance problems that often necessitate the use of parallel database systems (PDBS). Although dynamic load balancing is of key importance in PDBS, to our knowledge it has not yet been investigated thoroughly for parallel data warehouses. In this study, we propose a scheduling strategy that simultaneously considers both processors and disks while utilizing the load balancing potential of a Shared Disk architecture. We compare the performance of this new method to several other approaches in a comprehensive simulation study, incorporating skew aspects and typical data warehouse features such as star schemas.

1 Introduction

A successful data warehouse must ensure acceptable response times for complex analytical queries. Along with measures such as new query operators [8], specialized index structures [13, 19], intelligent data allocation [18], and materialized views [3], *parallel database systems (PDBS)* are used to provide high performance [5]. For effective parallelism, good load balancing is a must, and many algorithms have been proposed for general PDBS. But we are not aware of load balancing studies for data warehouses with characteristic features such as star schemas and bitmap indices.

In this paper, we evaluate a new approach to *dynamic load balancing* in parallel data warehouses based on the simultaneous consideration of both CPUs and disks. These are frequent bottlenecks in the voluminous scan/aggregation queries characteristic of data warehouses. 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 to perform 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 [16] in which each processing node can execute any subquery. For scan workloads, the balance of CPU load does not depend on the data allocation, permitting query scheduling with shared job queues for all nodes. Disk contention is harder to control because the total load per disk is predetermined by the data allocation and cannot be shifted at runtime.

In a detailed simulation study, we compare the new integrated strategy to several simpler methods of dynamic query scheduling. We use a data warehouse setting based on the APB-1 benchmark comprising a star schema with a huge fact table supported by bitmap indices, both declustered across many disks for parallel access. The large

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 321–331.

scan/aggregation queries we regard stress both disks and CPUs, creating a challenging scheduling problem. We particularly consider the often neglected but performance-critical treatment of *skew effects*. As a first step in the field, we focus on single-user mode, but our scheduling approaches can also be applied in multi-user mode.

In Section 2 of this paper, we briefly review some related work. Section 3 outlines our general load balancing paradigm, whereas our specific scheduling heuristics are defined in Section 4. Section 5 describes our simulation system and presents the performance evaluation of the scheduling strategies. We conclude in Section 6. Details omitted due to space constraints can be found in an extended version of this paper [11].

2 Related Work

We are not aware of any load balancing studies for parallel data warehouses. For general PDBS, load balancing problems have been widely researched, for a variety of workloads and architectures [4, 6, 9, 10, 16]. Many of these approaches rely on extensive data redistribution too costly in a large data warehouse. Furthermore, most previous studies have been limited to balancing CPU load, sometimes including main memory [14]. Even so, the need for dynamic scheduling has been emphasized [2, 14]. 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 [7, 15, 17] or on limiting disk contention through reduced parallelism [16]. Integrated load balancing as proposed in this paper has not been addressed.

The Shared Disk architecture has been advocated due to its superior load balancing potential especially for read-only workloads as in data warehouses [9, 12, 16]. It also offers great freedom in data allocation [15]. But the research on how to exploit this potential is still incomplete. SD is also supported by some commercial PDBS from *IBM*, *ORACLE*, and *SYBASE*. These and other data warehouse products (e.g., *INFORMIX*, *RED BRICK*, *MICROSOFT*, and *TERADATA*) support star schemas and (mostly) bitmap indices as well as adequate data fragmentation and parallel processing. But since no documentation is available on disk-sensitive scheduling methods, we believe that dynamic disk load balancing is 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 Shared Disk architecture.

The two performance-critical types of resources for scans are processing nodes and disks, but their respective load balance depends on different conditions: CPU utilization is largely determined by *how much* data each processor is assigned. A

balanced disk load, on the other hand, hinges on *when* the data residing on each device are processed because their location is fixed. We thus aim for an integrated view on both resources.

When a query enters the system, a *coordinator* node that controls its execution partitions the query into subqueries based on the presumed horizontal fragmentation. Each subquery scans either a fragment or a *partition* of the relevant table, where a partition comprises all table fragments residing on one disk. Fragments known to contain no hit rows are excluded. We thus obtain independent subqueries that can be processed on any processing node, yielding great flexibility in the subsequent scheduling step.

Fragments, being smaller than partitions, permit a more even load distribution especially in case of skew. Partition-sized subqueries, however, reduce the scheduling and communication overhead as well as disk contention as no two subqueries will process the same table partition, although some interference may still stem from index access.

Scheduling. Presuming full parallelism for the large queries we examine, 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. To this end, the coordinator maintains a list of subqueries that are dispatched following a given ordering policy (cf. Section 4) and processed locally as described below. All processors obtain the same number of subqueries (±1) up to a given limit roughly corresponding to the performance ratio of CPUs to disks; remaining tasks 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 depending on the order in which subqueries are dispatched. This aspect is elaborated in Section 4.

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 while exploiting prefetching and parallel I/O. 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 at subquery termination. For partition-sized subqueries, a node will process its partition sequentially, skipping irrelevant fragments. Multiple subqueries on the same processor coexist without any need for intra-node coordination.

4 Scheduling Order of Subquery Execution

Since we regard the scheduling of subquery execution as the most important aspect of load balancing in our processing model, we now present four scheduling policies based on either static (Section 4.1) or dynamic (Section 4.2) ordering of subqueries. Detailed calculations and some variant strategies can be found in the extended paper [11].

4.1 Statically Ordered Scheduling

Our simpler heuristics employ a *static ordering* of subqueries. Even under these strategies, however, our scheduling scheme as such is still dynamic as the allocation of workload to processing nodes is determined at runtime based on the progress of execution.

Strategy *LOGICAL*. This heuristic – taken from our previous study on star schema allocation [18] as a baseline reference – assigns fragment-sized subqueries in the logical or- der of the fragments they refer to. Since the allocation scheme applied here does not maintain this order (cf. Section 5.1), *LOGICAL* will not yield optimal performance.

Strategy *PARTITION*. Partition-sized subqueries are dispatched in a round-robin fashion with respect to their logical disk numbers. In single-user mode, this means that each table partition is accessed by only one processor at a time. However, bitmap access (if required) can cause each subquery to read from multiple disks, so that access conflicts may not be avoided completely. Still, we expect this policy to minimize disk contention.

Strategy *SIZE*. This method starts fragment-sized subqueries in decreasing order of size, based on the expected number of referenced pages. It implements an LPT (*longest processing time first*) scheme that provides good load balancing for many scheduling problems. It does not consider disk allocation but may be expected to optimize the balance of processor load based on the total amount of data processed per node.

4.2 Dynamically Ordered Scheduling

The static policies above tend to optimize the balance of *either* CPU *or* disk load. For an improved, integrated load balancing we reckon with both criteria based on a *dynamic ordering*. To distribute disk load over time and reduce contention, we try to execute concurrently subqueries with minimum overlap in disk access. To simultaneously balance CPU load, we also consider subquery sizes similar to the previous section. Figure 1 illustrates the following considerations using a 4-disk example with 4 subqueries.

Strategy *INTEGRATED*. We model the disk load characteristics of each subquery in the shape of a *load vector* ① containing the expected number of pages referenced on each disk. This number is calculated from the query's estimated selectivity and

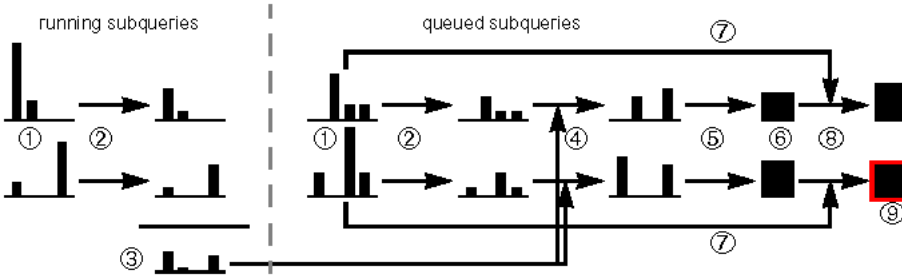

Fig. 1. Sequence of load vector calculation in strategy *INTEGRATED* (graph scaling varies).

includes both table and bitmap fragments. The load vector is normalized ② to represent the relative load distribution across the disks *at a given point in time* rather than its total magnitude.

In addition to the single load vectors for each subquery, we keep a global vector of current disk load, defined as the sum of the load vectors of all subqueries currently running ③. We can then compute an expected rate of access conflict between the current load and any queued subquery by comparing their respective load vectors. Specifically, the products of local intensities per disk ④, added over all disks ⑤, yield a measure of the total access conflict between each candidate and the current load ⑥. To integrate disk conflict estimates with the distribution of CPU load, we divide the expected disk access conflict for each subquery by its total size ⑦⑧, so that long-running tasks may be executed earlier than shorter ones even if they incur a slight increase in disk contention. The subquery that minimizes the resulting ratio ⑨ (thus optimizing the trade-off between both criteria) will be dispatched in the next scheduling step.

5 Simulation Study

We now present our simulation study, first introducing the simulation system used (Section 5.1), then discussing the performance of our scheduling schemes (Section 5.2), and finally testing the scalability of our methods in speed-up experiments (Section 5.3).

5.1 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 [18]. Simulating a Shared Disk PDBS with 20 processors and 100 disks, it realistically reflects resource contention by modeling both CPUs and disks as servers. CPU overhead is reckoned for all relevant operations, and seek times in the disk modules depend on the location (track number) of the desired data within a disk. Each processor owns a buffer module with separate LRU queues for fact table and bitmap

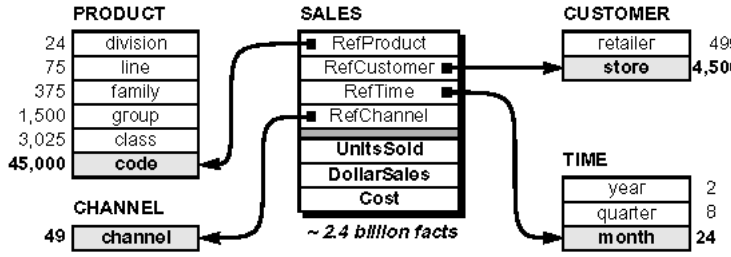

Fig. 2. Sample star schema.

access. The network incurs communication delays proportional to message sizes but models no contention, so as to avoid specific network topologies unduly influencing experimental results.

Our sample data warehouse is modeled as a relational star schema for a sales analysis environment (Figure 2) derived from the *APPLICATION PROCESSING BENCHMARK (APB- 1)* [1]. The denormalized *dimension tables PRODUCT, CUSTOMER, CHANNEL* and *TIME* each define a *hierarchy* (such as product divisions, lines, families, 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. We incorporate common *bitmap join indices* [13] to avoid costly full scans of the fact table. We employ *standard bitmap*s for the low-cardinality dimensions *TIME* and *CHANNEL*, but use *hierarchically encoded bitmaps* [19] for the more voluminous dimensions *PRODUCT* and *CUSTOMER* to save disk space and I/O. With these indices, queries 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 fragmentation strategy for star schemas that we proposed and evaluated in [18]. Specifically, we choose a two-dimensional fragmentation based on *TIME.MONTH* and *PRODUCT.FAMILY*. Each resulting fact table fragment thus combines all rows referring to one particular product family within one particular month, creating $375 \cdot 24 = 9000$ fragments. This can significantly reduce work for queries referencing one or both of the fragmentation dimensions; it also supports both processing and I/O parallelism and scales well. As demanded in Section 3, the fragmentation of bitmaps follows that of the fact table.

Since one focus of our study is on skew effects, we explicitly model *attribute value skew* in the fact table, using zipf-like frequency distributions with respect to dimension values. This leads to varying densities and sizes of table fragments, potentially causing severe load imbalance. To help alleviate such *density skew*, we employ a *greedy data allocation algorithm* similar to [17] which allocates fact table fragments in decreasing order of size onto the least occupied disk at each time to keep disk partitions balanced. Corresponding bitmap fragments of each bitmap are stored on adjacent disks to support parallel bitmap access. Note, however, that a smart allocation scheme is merely a complement, not a replacement for intelligent scheduling techniques employed at runtime.

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 that do not require joins to the dimension tables. All queries within a single

experiment are of the same type (e.g., $Q_{DIVISION}$, aggregating data from one product division) but with random parameters (e.g., the specific division selected). However, different simulation runs will use the same set of queries, facilitating a fair comparison of results.

5.2 Scheduling Strategies

Since the performance of our strategies will depend in part on the type of query being processed, we consider both disk-bound and CPU-bound workloads, as well as borderline cases that shift between categories. In our case, the *selectivity* of a query *within* the relevant fact table fragments determines the ratio of CPU to I/O load. Our CPU-intensive queries each have a 100% selectivity within the fragments they access. I/O-bound loads, in contrast, select only some of the tuples in each fragment, causing less CPU work per I/O, and use bitmap indices, which are also cheap to process on the CPU side.

All queries are tested for our four scheduling strategies under varying degrees of skew on the two fragmentation dimensions, *TIME* and *PRODUCT*, using the same degree of skew to both dimensions. Under the zipf-like distributions we employ, the skew parameter may range from 0 (no skew) to values around 1 (heavy skew).

Disk-Bound Queries. In Figure 3, we show simulation results for two disk-bound queries, $Q_{CHANNEL}$ and Q_{STORE}. With our greedy allocation scheme, balanced disk partitions can be processed in constant time regardless of skew under a proper scheduling method. *PARTITION* achieves the best response times as would be expected for disk-bound workloads, keeping disks optimally loaded at nearly 100%. It also minimizes the inevitable disk contention caused by concurrent access to 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 analysis it performs is similarly effective to avoid disk contention as a strict separation of partitions, despite the additional size criterion.

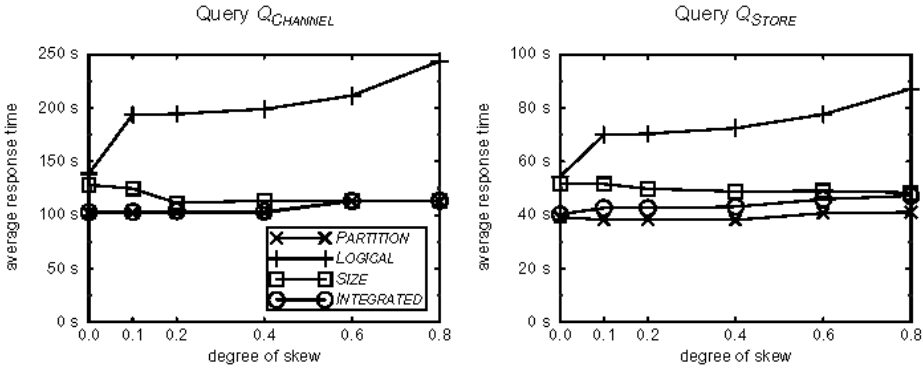

Fig. 3. Disk-bound queries

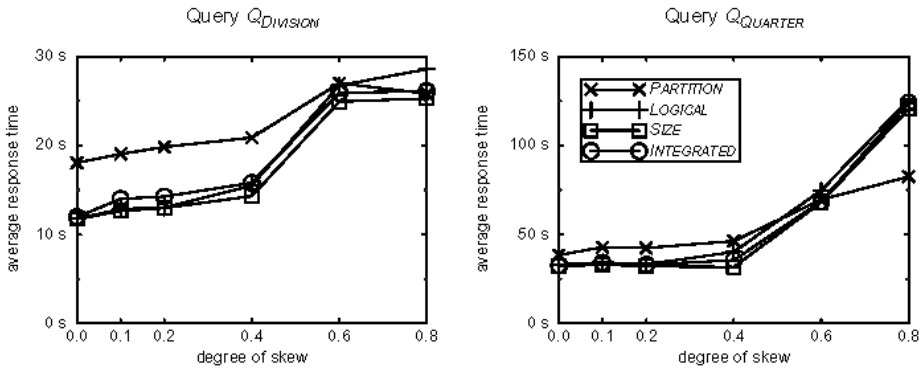

Fig. 4. CPU-bound queries

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. Still, it cannot contend with the near-optimal *PARTITION*, with differences of up to 35%.

CPU-Bound Queries. The CPU-bound queries $Q_{DIVISION}$ and $Q_{QUARTER}$ perform a selection on the skewed fragmentation dimensions *PRODUCT* and *TIME*, respectively, and thus respond markedly to skew effects (Figure 4). Although partition sizes are well balanced for the database as a whole, this is not the case for single product divisions or calendar quarters and the largest fragment *within* such a subset can dominate the query's response time. This can be corrected by data allocation only to a limited extent.

The best results are achieved by *SIZE* as it balances the sheer amount of data processed per node, which is essential for CPU-bound queries. *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 under low disk utilization. The other two strategies achieve good success; *INTEGRATED* approximates *SIZE* most closely with only 10% deviation, demonstrating good performance for CPU-bound workloads as well.

Increasing skew changes the ranking in favor of *PARTITION*. With $Q_{QUARTER}$, *PARTITION* becomes by far the best strategy for extreme skew, now offsetting *SIZE* by 46%. This is because the skewed fragment sizes turn the query *locally disk-bound*, i.e., a single disk becomes the bottleneck even though the query as a whole is CPU-bound!

This situation is analyzed in detail in Figure 5, which shows the response times of queries referencing the least densely and most densely populated quarters for each given degree of skew. The smaller queries remain CPU-bound for the entire range because density skew is less severe toward the lower end of our zipf-like distribution curve. For large quarters, however, 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.

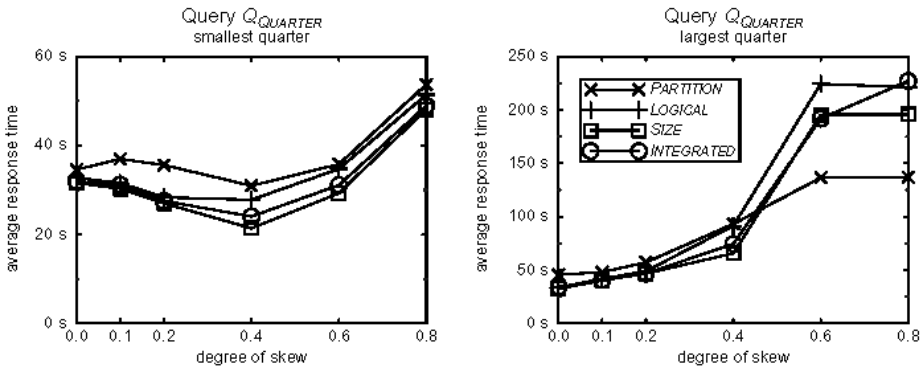

Fig. 5. Shift from CPU-bound to disk-bound

Discussion. The results show that no single scheduling scheme is 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 choice of the truly best strategy then depends on the 'boundness' of a query, as determined by its selectivity and index utilization, the degree of skew, and a number of other parameters. 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.

On the other hand, our dynamic scheduling scheme based on the *INTEGRATED* heuristic was able to adapt to different types of queries and 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 between *PARTITION* and *SIZE* will be very difficult against a continually changing background load alternating between CPU-bound and disk-bound states. This aspect, however, needs to be investigated in future studies.

5.3 Speed-Up Behavior

In this simulation series, we test the scalability of our query processing and scheduling strategies with varying numbers of disks and processors. For each configuration, we run the queries $Q_{QUARTER}$ and $Q_{RETAILERMONTH}$ under a medium skew degree of 0.4 and against skewless data, respectively. Results are shown in Figure 6.

Since $Q_{QUARTER}$ is CPU-bound, we test its speed-up in relation to the number of processors, using *SIZE* as the scheduling strategy according to the previous results. Against skewless data, $Q_{QUARTER}$ shows linear speed-up until the disks of the system become bottlenecks and speed-up with respect to processors is no longer achievable. With skew (dashed graph), the curves decline earlier because response times are

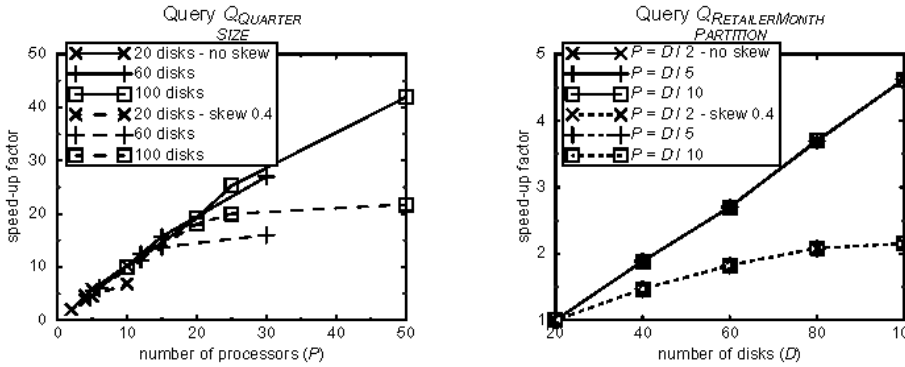

Fig. 6. Speed-up behavior of queries $Q_{QUARTER}$ and $Q_{RETAILERMONTH}$

dominated by the work on the largest fragment, causing locally disk-bound processing.

To test the speed-up for disk-bound queries, we use $Q_{RETAILERMONTH}$ which is more responsive to skew than $Q_{CHANNEL}$ and Q_{STORE} used above. $Q_{RETAILERMONTH}$ is scheduled using *PARTITION*, and speed-up is evaluated in relation to the number of disks.

As in the the previous case, speed-up is 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).

For both types of workload, the *INTEGRATED* policy we proposed achieved equivalent results to the above (not shown here). 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.

6 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 like *PARTITION* and *SIZE* can be very effective. But the selection of the appropriate method depends on whether a query is disk-bound or CPU-bound, which can be difficult to determine especially under skew conditions. As an alternative, we proposed a more complex, dynamically ordered scheduling approach (*INTEGRATED*) that yields only slightly worse performance but naturally adapts to different query types.

While we assumed a Shared Disk environment, most of the results can be transferred to other architectures, in particular, Shared Everything. Shared Nothing systems are restricted to strategies similar to *PARTITION*, which we found to be non-optimal. This demonstrates the benefits of Shared Disk and justifies our architectural choice. The extension of our findings to multi-user mode is not trivial. As the simple heuristics *PARTITION* and *SIZE* may no longer be sufficient, we expect our integrated

strategy to gain importance. Verifying this assumption will be a focus of our future work.

References

1. APB-1 OLAP Benchmark, Release II. OLAP Council, Nov. 1998.
2. R. Avnur, J.M. Hellerstein: Eddies: Continuously Adaptive Query Processing. Proc. ACM SIGMOD Conf., Dallas, 2000.
3. E. Baralis, S. Paraboschi, E. Teniente: Materialized View Selection in a Multidimensional Database. Proc. 23rd VLDB Conf., Athens, 1997.
4. L. Bouganim, D. Florescu, P. Valduriez: Dynamic Load Balancing in Hierarchical Parallel Database Systems. Proc. 22nd VLDB Conf., Bombay, 1996.
5. S. Chaudhuri, U. Dayal: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record 26(1), 1997.
6. D.J. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri: Practical Skew Handling in Parallel Joins. Proc. 18th VLDB Conf., Vancouver, 1992.
7. S. Ghandeharizadeh, D.J. DeWitt, W. Qureshi: A Performance Analysis of Alternative Multi- Attribute Declustering Strategies. Proc. ACM SIGMOD Conf., San Diego, 1992.
8. 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, 1997.
9. H. Lu, K.-L. Tan: Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems. Proc. 3rd EDBT Conf., Vienna, 1992.
10. S. Manegold, J.K. Obermaier, F. Waas: Load Balanced Query Evaluation in Shared-Everything Environments. Proc. 3rd Euro-Par Conf., Passau, 1997.
11. H. Märtens, E. Rahm, T. Stöhr: Dynamic Query Scheduling in Parallel Data Warehouses. Techn. report, University of Leipzig, 2001. `http://dol.uni-leipzig.de/pub/2001-38/en`
12. 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. 17th VLDB Conf., Barcelona, 1991.
13. P. O'Neil, G. Graefe: Multi-Table Joins Through Bitmapped Join Indices. ACM SIGMOD Record 24 (3), 1995.
14. E. Rahm, R. Marek: Dynamic Multi-Resource Load Balancing in Parallel Database Systems. Proc. 21st VLDB Conf., Zurich, 1995.
15. E. Rahm, H. Märtens, T. Stöhr: On Flexible Allocation of Index and Temporary Data in Parallel Database Systems. Proc. 8th HPTS Workshop, Asilomar, 1999.
16. E. Rahm, T. Stöhr: Analysis of Parallel Scan Processing in Parallel Shared Disk Database Systems. Proc. 1st Euro-Par Conf., Stockholm, 1995.
17. P. Scheuermann, G. Weikum, P. Zabback: Data Partitioning and Load Balancing in Parallel Disk Systems, VLDB Journal 7(1), 1998.
18. T. Stöhr, H. Märtens, E. Rahm: Multi-Dimensional Database Allocation for Parallel Data Warehouses. Proc. 26th VLDB Conf., Cairo, 2000.
19. M.-C. Wu, A.P. Buchmann: Encoded Bitmap Indexing for Data Warehouses. Proc. 14th ICDE Conf., Orlando, 1998.