A Survey on Parallel and Distributed Data Warehouses

Pedro Furtado
Universidade Coimbra, Portugal

ABSTRACT

Data Warehouses are a crucial technology for current competitive organizations in the globalized world. Size, speed and distributed operation are major challenges concerning those systems. Many data warehouses have huge sizes and the requirement that queries be processed quickly and efficiently, so parallel solutions are deployed to render the necessary efficiency. Distributed operation, on the other hand, concerns global commercial and scientific organizations that need to share their data in a coherent distributed data warehouse. In this paper we review the major concepts, systems and research results behind parallel and distributed data warehouses.

INTRODUCTION

Decision support systems are important tools in the hands of today's competitive and knowledgeable organizations, and data warehouses (DW) are at the core of such systems. They store huge detailed and summarized historical data for decision makers to generate queries, make reports and perform analysis and mining that are the basis for their decisions and deeper knowledge. Users also need fast response times on complex queries in data warehousing, OLAP and data mining operations. Two major forces have contributed to the importance of parallel and distributed data warehousing: On one hand, the fact that data warehouses can be extremely large and highly resource demanding, while queries and analyses must be answered within acceptable time limits has led to a series of specialized techniques that were developed specifically for them, including view and cube materialization (Rousopoulos 1998), specialized indexing structures (O'Neil and Graefe 1995) and implementations on parallel systems, which we review along this paper. While all these specialized techniques and structures play an important role in the performing data warehouse, we focus on parallel systems in particular, which can provide top performance and scalability. Parallel processing answers satisfactorily the need to handle huge data sets efficiently, in both query processing and other concerns such as loading or creation of auxiliary structures; On the other hand, the evolution of the data warehouse concept from a centralized local repository into a broader context of sharing and analyzing data in an internet-connected world has given birth to distributed approaches and systems.

In this chapter we review important research and trends on these parallel and distributed approaches. Our approach is to introduce and illustrate the major issues first, and then to review some of the most relevant systems and research results on the field. Our first discussion is on parallel architectures, the physical infrastructure over which to store and process the data, with crucial implications on performance and scalability of the solutions. With this in mind, we then discuss types of parallelism and architectural issues in parallel database management systems. Then we discuss partitioning and allocation, one of the most fundamental enablers of intra-query horizontal parallelism. After discussing architecture and partitioning, we then turn our attention to parallel processing and optimization, including an illustration on how to process in horizontal intra-query parallelism. After reviewing the architectural, partitioning and processing basics of parallel data warehousing, we devote a section to the discussion of systems and research results on the subject of parallel data warehouses and another one on distributed data warehouses. Distributed data warehouse systems are a most relevant subject, since WAN-connected geographically distributed organizations share both data and analysis, and networking technology currently enables long distance collaboration.

Parallel and distributed data warehousing is an exciting field, and research in these issues is far from being exhausted. In a few words, autonomy, scalability, ubiquity and application contexts are some of the most fundamental issues that will certainly deserve a lot of attention in the future. We end the paper with conclusions and a brief discussion on these future trends.

PARALLEL ARCHITECTURES FOR DATA WAREHOUSING

Due to their high-demand on storage and performance, large DWs frequently reside within some sort of parallel system. In this section we review different base architectures that can be used to store and process the parallel data.

There is a whole range of architectures for parallelization, from shared-nothing to shared-disk and hybrid ones, as current state-of-the-art servers come with multiple processors. There are different nomenclatures for the basic models by which a parallel system can be designed, and the details of each model vary as well. Consider three basic elements in a parallel system: the processing unit (PU), the storage device (S) and memory (M). The simplest taxonomy defines three models, as described in (DeWitt and Gray 1992):

Shared Memory (SM): the shared memory or shared everything architecture, illustrated in Figure 1, is a system where all existing processors share a global memory address space as well as peripheral devices. Only one DBMS is present, which can be executed in multiple processes or threads, in order to utilize all processors;

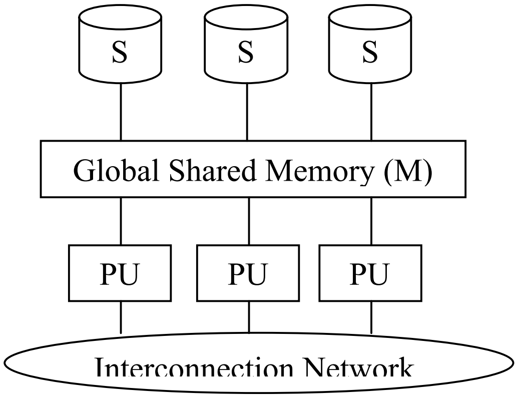


Figure 1. Shared memory Architecture

Shared Nothing (SN): the shared nothing architecture, illustrated in Figure 2, is composed of multiple autonomous Processing Nodes (PN), each owning its own persistent storage devices and running separate copies of the Database Management System (DBMS). Communication between the PNs is done by message passing through the network. A PN can be composed of one or more processors and/or storage devices.

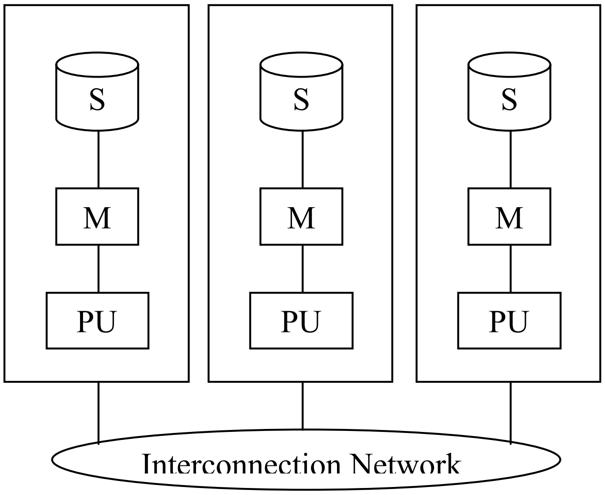


Figure 2. Shared nothing Architecture

Shared Disk (SD): the shared disk architecture, illustrated in Figure 3, is characterized by possessing multiple loosely coupled PNs, similar to SN. However, in this case, the architecture possesses a global disk subsystem that is accessible to the DBMS of any PN.

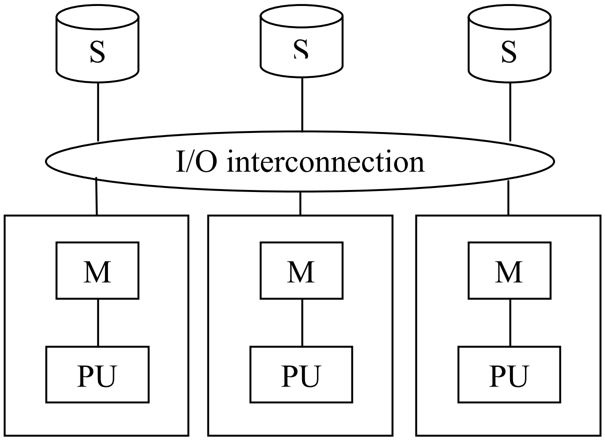


Figure 3. Shared disk Architecture

Another typical taxonomy for parallel system architectures categorizes them as multiprocessor systems, then further categorizes these into shared memory and distributed memory alternatives. In shared-memory (typically symmetric multiprocessors - SMP) systems, a number of processors – the processing units (PU) – share common memory and I/O. Distributed memory multiprocessor systems are systems that have multiple processors, but these do not all shared the same memory. This definition is so broad that it encompasses the shared nothing and shared disk models, depending on whether the storage sub-system or a significant part of it is shared or not, and it includes both Massively Parallel Processors, clusters of uni-processors and SMP processors.

One important aspect of any parallel architecture is that the slowest components dictate performance. For this reason, when analyzing the advantages and disadvantages of each model, it is most important to identify the major bottlenecks that each architecture possesses:

Shared memory or SMP systems: these machines typically have a small number of processors, and the main advantage of this architecture is that the programming paradigm is the same as in uni-processor machines, and it is the responsibility of the operating system to handle the concurrency issues that result from the multiple parallel executions. These issues are typically shared memory concurrency and dynamic load balancing, which refers to dynamically distributing the tasks among the PUs. SMPs have a major drawback in their limited scalability, since there is a physical by-design limit on the number of processors, which is associated with limitations of the shared system bus and also I/O typical bottlenecks;

Shared nothing systems: from a hardware perspective, very cost-effective SN systems can be setup, since it is possible to make an SN with commodity computers connected by an ordinary switched network. Furthermore, a large number of PNs can be interconnected in this manner because, other than the network, no other resources are shared. On the other hand, the main disadvantages of SN architectures are the specialized software programming model, since it must take into account the architecture, and the fact that interconnections between processing units may become a bottleneck, since data needs to be exchanged between nodes. For this reason, data allocation, query processing optimizations and load-balancing issues are most relevant in SN. It is also important to note that SN systems are a very large family, since they range from the lowest-end cluster, built with commodity computing nodes and a standard switched network, to highly sophisticated top-performing clusters, possibly with high-end SM machines in some or all of the nodes;

Shared disk systems: in shared disk systems, the disk storage is shared. This has the advantage over shared nothing systems of avoiding the need to assign and store specific parts of the data in each node, as the data is all stored in the shared disk subsystem. However, it has the disadvantage of creating a possibly critical bottleneck and scalability limitations in the storage subsystem and interconnections, as all processing units share the same storage system;

One major advantage of shared memory systems over shared nothing ones is that, while they share the same memory, database and data sources in general, shared nothing systems need interconnecting middleware to distribute and synchronize tasks, for data exchange and load balancing. An interconnection network is necessary, and the faster the better. It may be anything from an external LAN to a high-speed internal switch. While this is a disadvantage of shared nothing systems, the degree of multiprocessors in shared memory systems is limited by design and by physical hardware limitations, while the degree of shared nothing nodes is theoretically unlimited and in practice can be adjusted to factors such as the data size and application needs.

In practice, hybrid architectures are usual: on one hand, dual and quad-processor computers are already becoming a commodity, and many such computers can be organized in a shared nothing environment with shared memory nodes; On the other hand, optimized software can take advantage of both centralized storage sub-systems and local storage units that may be used as very large disk caches for retrieved data. In practice, solutions range from high-end to low-cost ones. High-end, expensive solutions use proprietary, specialized and highly optimized software on high-end servers with powerful multiprocessor machines, high-end storage and I/O systems; Low-end inexpensive systems build a shared nothing environment with commodity low-end multiprocessors, open-source software and some middleware software.

These architectures are the infrastructure over which parallelism is implemented. In the next section we review types of parallelism and how the parallel architecture influences the design of a Relational Database Management System.

TYPES OF PARALLELISM AND DATABASE MANAGEMENT SYSTEM

There are different alternatives concerning parallel processing in database management systems. In this section we review those alternatives and also how the parallel architecture influences the design of the relational database management system (RDBMS).

Parallel processing refers to executing multiple threads (or processes) concurrently, with three main parallel processing approaches: inter-query, intra-query and hybrid parallelism alternatives.

Inter-query: in inter-query parallelism, which is illustrated in Figure 4, different threads/processing units handle different queries simultaneously. This has the potential of increasing throughput; On the other hand, it does not account for parallelizing operations such as a join, a sort and a selection, within a single query. This means that such an approach has the potential of producing significant speedups in contexts where multiple concurrent queries are submitted simultaneously;

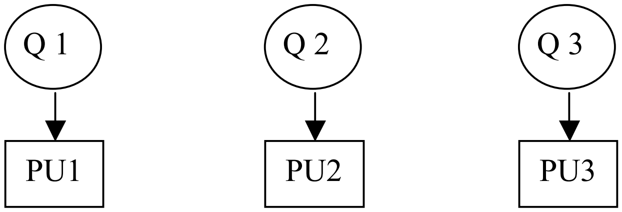
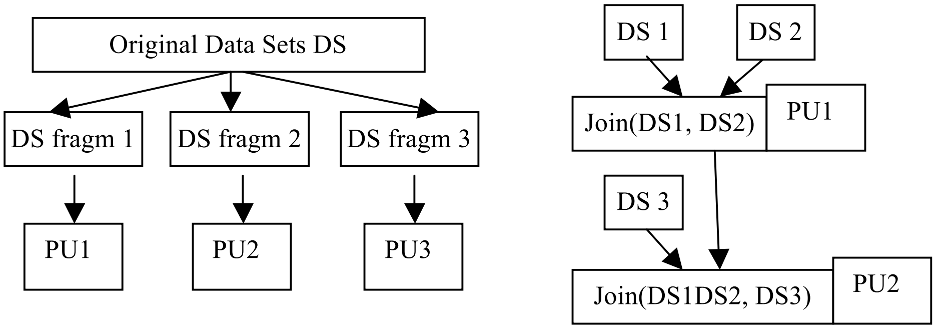


Figure 4. Inter-query Parallelism

Intra-query: intra-query parallelism parallelizes operations within a query, and it can be further decomposed into horizontal, vertical and hybrid parallelism: In horizontal parallelism, the data is divided into multiple pieces and the query is decomposed into a number of smaller sub-queries that will act on those pieces independently. If the data partitions are allocated different I/O devices, this can result in a significant speedup. Figure 5(a) illustrate this approach, whereby an original data set is partitioned into what is denoted as fragments or partitions, and each fragment is processed by a processing unit (PU); Vertical or pipelined parallelism assigns query operators to different processing units and feeds the results from one operator into the next one as soon as possible. Figure 5(b) illustrates this kind of parallelism. In that example three datasets are joined. The first two ones are joined in processing unit PU1, while the resulting intermediate data set is then joined with DS3 in PU2.

Hybrid parallelism combines both horizontal and vertical parallelism.



(a) Horizontal *(b) Vertical or pipelined parallelism*

Figure 5. Intra-query Parallelism

Another relevant issue in parallelism is how a parallel architecture influences the design of a Relational Database Management System (RDBMS) that is to run on that architecture. Depending on the hardware architecture, the RDBMS has certain specialized features.

RDBMS in shared memory architectures: since in this architecture all processors have access to all the data, the RDBMS design paradigm can be the same as in uni-processor machines. DBMS components in PUs executing SQL statements communicate with each other by exchanging messages and data through shared memory. Individual processing units may be processes or threads (threads incur in less overhead associated with context switches and achieve better resource utilization). Given the heavy data access characteristics of data warehousing contexts, this approach has hardware architecture inherent scalability limitations, since bottlenecks such as the system bus and I/O are crucial;

RDBMS in shared nothing architectures: in this context, the data must be partitioned into all disks and the parallel DBMS server is composed by a set of node DBMS components and a global component. The global component must parallelize the SQL queries and send the resulting subqueries to execute locally at each node. Nodes exchange data and messages through the interconnection network. This system offers scalability and significant speedup, but it raises two major issues: the RDBMS design paradigm is different from uni-processor machines, the software is complex and must implement added functionality such as repartitioning (on-the-fly data exchange for join processing); Problems such as data imbalance and excessive data transfer needs are latent in these systems and must be handled by all kinds of optimization features;

RDBMS in shared disk architectures – the database storage is shared by all RDBMS, and concurrency issues are handled by a distributed lock manager. This solution eliminates memory access bottlenecks of shared memory systems and is less dependent on the allocation of data partitions then shared nothing approaches. However, its scalability is limited by I/O bottlenecks, since the storage is shared.

In order to implement parallelism, it is necessary to divide data or processing into pieces that are to be executed in parallel. In the next section we discuss how data can be partitioned for parallelism.

PARTITIONING AND ALLOCATION

Data Partitioning refers to splitting huge data sets, such as data warehouse fact tables, into much smaller pieces that can be handled efficiently and enables horizontal parallelism. One of the objectives of partitioning is to reduce the amount of data that must be handled in order to answer a query. For instance, if the data is partitioned on an yearly basis, a query requesting the sales of the last year needs only process a partition that corresponds to last years' data.

Data partitioning is also a pre-condition for horizontal (and hybrid) intra-query parallel processing. The data sets are divided into fragments or partitions, and the parallel processing software assigns different fragments to different processing units. Two main types of data partitioning are available (Ozsu and Valduriez 1999): Vertical and Horizontal partitioning. Vertical partitioning allows tables and materialized views to be decomposed into disjoint sets of columns. Note that the key columns are duplicated in each vertical fragment to allow "reconstruction" of an original table; Horizontal partitioning allows tables, materialized views and indexes to be divided into disjoint sets of rows (called fragments), physically stored and accessed separately. Most of today's commercial database systems offer native DDL (data definition language) support for defining horizontal partitions.

Partitioning involves the choice of partitioning criteria, that is, the specification of how the data set is to be partitioned. We define the following alternatives, based on whether values of attributes are involved in the partitioning decision or not:

Random and round-robin partitioning – these involve determining which partition will get each row randomly (random partitioning) or in a sequential round-the-table manner (round-robin). These are the simplest forms of partitioning, and typically lead to data-balanced placement. However, since the data was placed randomly, a search needs to go to all partitions, even if only a small portion of the data is to be accessed or if only rows with specific values, hash or ranges need to be accessed;

Value-wise (Hash, range, attribute-value wise) partitioning - a partitioning that is determined by the value, hash or range of one or a combination of attributes. With value-wise partitioning it is possible to locate a partition in a specific node if the search is based on the partitioning key(s). On the other hand, for this kind of partitioning criteria, the allocation algorithm must take data skew into consideration to avoid or decrease data unbalance;

For many query processing objectives, value-wise partitioning has a significant advantage over random or round-robin in that the database server can identify where a piece of data lies and therefore spare a lot of accesses: for instance, if sales data is partitioned by shop and product and a query requests a specific shop and product, only a partition needs to be processed to answer the query. Parallel hash-join algorithms also benefit immensely from hash-partitioning large relations into nodes in order to minimize data exchange requirements (Kitsuregawa, Tanaka and Motooka, 1983; DeWitt and Gerber, 1985). These strategies typically allocate a hash range to each node, so that joins can proceed in parallel in all nodes.

Given alternative partitioning options and the need to optimize the partitioning approach to handle huge data warehouses efficiently, many authors have investigated this optimization subject in different contexts. Early works include Hua and Lee (1990), which uses variable partitioning (size and access frequency-based) and concludes that partitioning increases throughput for short transactions, but complex transactions involving several large joins result in reduced throughput with increased partitioning. Williams and Zhou (1998) review five major data placement strategies (size-based, access frequency-based and network traffic based) and conclude experimentally that the way data is placed in a shared nothing environment can have considerable effect on performance. Some of the most promising partitioning and placement approaches focus on query workload-based partitioning choices (Zilio, Jhingram and Padmanabhan, 1994; Rao, Zhang and Megiddo, 2002). These strategies use the query workload to determine the most appropriate partitioning attributes, which should be related to typical query access patterns. Generic data partitioning that is independent of the underlying database server and targeted at node partitioned (shared nothing) data warehouses was discussed in (Furtado, 2004; Furtado, 2004b; Furtado 2004c ; Furtado, 2005). This approach allows most queries to be processed in parallel very efficiently, based on hash-based partitioning of large relations and copies of smaller ones. In the work on Multidimensional Hierarchical fragmentation (Stohr 2000) the authors considered star schemas and used workload-based value-wise derived partitioning of facts and in-memory retention of dimensions for efficient processing. They also introduced the use of join-bitmap indexes together with attribute-wise derived partitioning and hierarchy-aware processing for very efficient partition-wise processing over star schemas. The approach in (Bellatreche 2008) proposes a genetic algorithm for schema partitioning selection, whereby the fact table is fragmented based on the partitioning schemas of dimension tables.

We end this section by describing a typical partitioning and allocation scenario, to help illustrate how it works. In relational databases, Data Warehouses are frequently organized as star schemas (Chaudhuri and Dayal 1997), with a huge fact table that is related to several dimension tables, as represented in Figure 6. In such schema, the facts table (Sales fact in the Figure) stores data to be analyzed and pointers to dimensions, while dimension information is stored in dimension tables (the remaining relations).

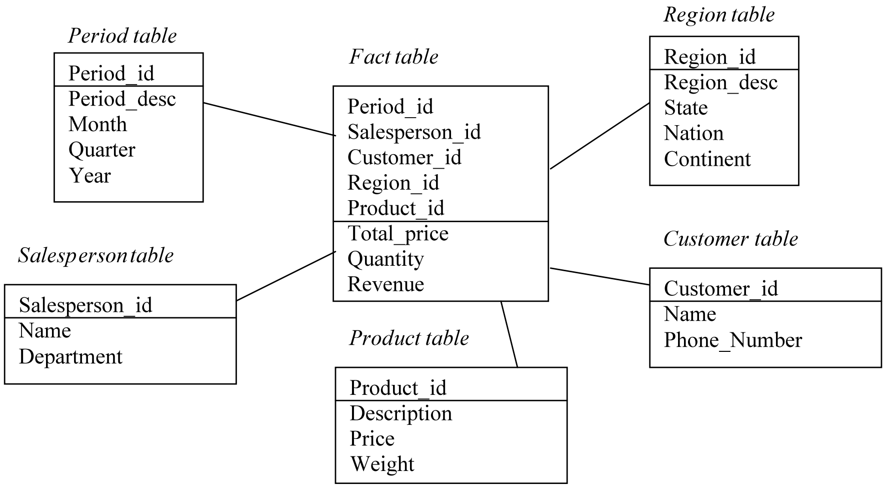


Figure 6. DW Sample Star Schema

It is frequent for dimension tables to be orders of magnitude smaller than the fact table. When considering a parallel environment, this means that it is worth to partition the central fact table into multiple pieces that can be processed in parallel, while the dimension tables are left complete. In a shared nothing environment, this means copying dimension relations into all nodes and dividing the fact throughout the nodes. This approach is very useful in what concerns parallel processing, since most operations can proceed in parallel (Furtado 2005), including processing of joins in parallel. More complex schemas may include bigger dimensions and multiple interconnected big relations that, for best performance, need to be partitioned, and it may happen that it is impossible to partition all for co-location, then a workload-based partitioning solution as the ones referenced above can be adopted, which tries to co-locate the relations that result in highest query processing gain.

Having discussed parallel architectures, types of parallelism and partitioning issues, we now turn our attention to parallel query processing. In the next section we discuss a parallel processing approach.

PARALLEL HORIZONTAL INTRA-QUERY PROCESSING

While in inter-query parallelization it is trivial to understand how a query is parallelized, for intra-query parallelism it is interesting to describe how individual queries are parallelized, in particular in the context of data warehouse schemas. In this section we illustrate how queries can be decomposed into a number of smaller sub-queries that will act on fragments independently, with significant speedup. This subject has been discussed in several works, which include (Akinde et al. 2003, Furtado 2005, Stohr 2000). Our illustration is based on the parallel query processing approach followed by the Data Warehouse Parallel Architecture (Furtado 2005, Furtado 2007), while we also discuss other works on the next section.

Figure 7 illustrates the basic architecture of DWPA for the shared nothing, node partitioned data warehouse, which can run in any number of computers interconnected by a LAN. It includes three major entities implemented as services: Submitter, Executor and the DWPA Manager. Submitters are simple services that may reside in any computer, do not require an underlying database server and submit queries to the system. The query may be submitted from a Submitter Console application or from other applications through an API. Once submitted, the query is parsed and transformed into high-level actions by a query planner. These actions are then transformed into Command Lists for each Executor service. Executors are services that maintain local database sessions and control the execution of commands locally and the data exchange with other nodes. Finally, the DWPA manager is a node which controls the whole system (it can be replicated for fault tolerance reasons), maintaining registries with necessary information for the whole system. When nodes enter the system, they contact the DWPA manager to register themselves and to obtain all the necessary information. In DWPA, any computer can assume any role as long as it runs the corresponding service.

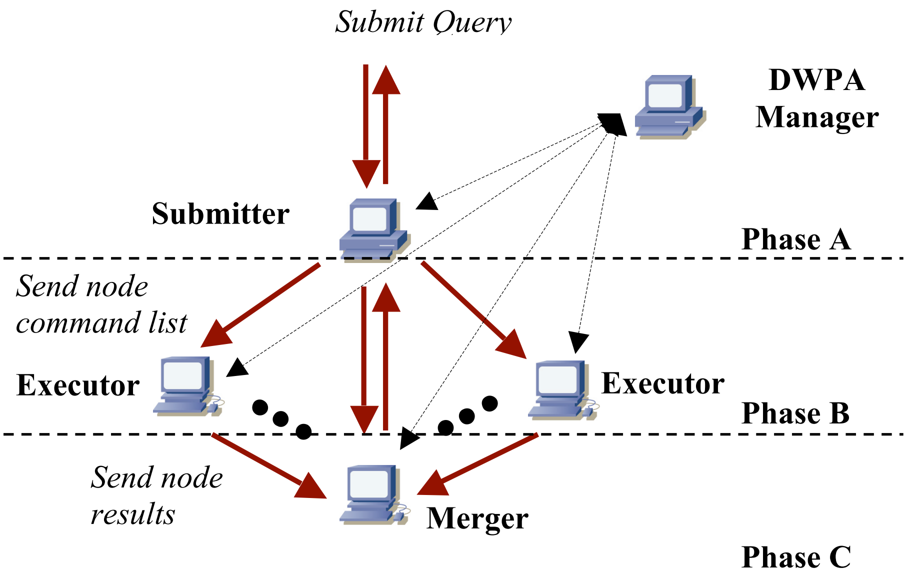


Figure 7: The DWPA Architecture

We will now describe basic query processing functionality. For simplicity, we start with the simplest possible example. Consider a single very large relation *R* partitioned into *n* nodes and a sum query over some attribute *x* of the relation. Formula (1) states that the sum of attribute *x* over all nodes is simply the sum of the sums of *x* in each node:

$$\sum x = \sum_{\text{all nodes}} \sum_{\text{over node } i}(x) \qquad (1)$$

The implementation of this very basic operation in DWPA involves the submitter parsing the initial query *sum(x) from R* and producing command lists for every node with the following operations:

1) a local query: *sum(x) as sumx from* R_{local}
2) data transfer commands for every executor node: send *sumx* to merger node
3) a merge query for the merger node: *sum(sumx) from partial_results*
4) signal the submitter to pull the results

The Merger node is an Executor that is chosen for merging the partial results if necessary. The query processing steps depend heavily on the placement layout of the data on the nodes. For instance, if relation *R* is replicated into all nodes or placed in a single node, the commands will be (executed in a single node):

1) a local query: *sum(x) as sumx from* R_{local}
2) signal the submitter to pull the results

Figure 8 shows a set of query processing steps that may be necessary in the processing of each query using DWPA (some queries may not require all these steps). Steps S1 to S4 represent the parsing and planning of queries, the generation of lists of commands for the executor nodes and the sending of those commands to Executors. Steps E1 to E4 represent the processing of the local queries within executor nodes, data exchanges between them and either sending the results to a merger node or signalling to the submitter that he can get the results. The merger node steps include a redistribution step EM3, which may be necessary for processing nested queries (for some queries containing subqueries, in which case more than one processing cycle may be required).

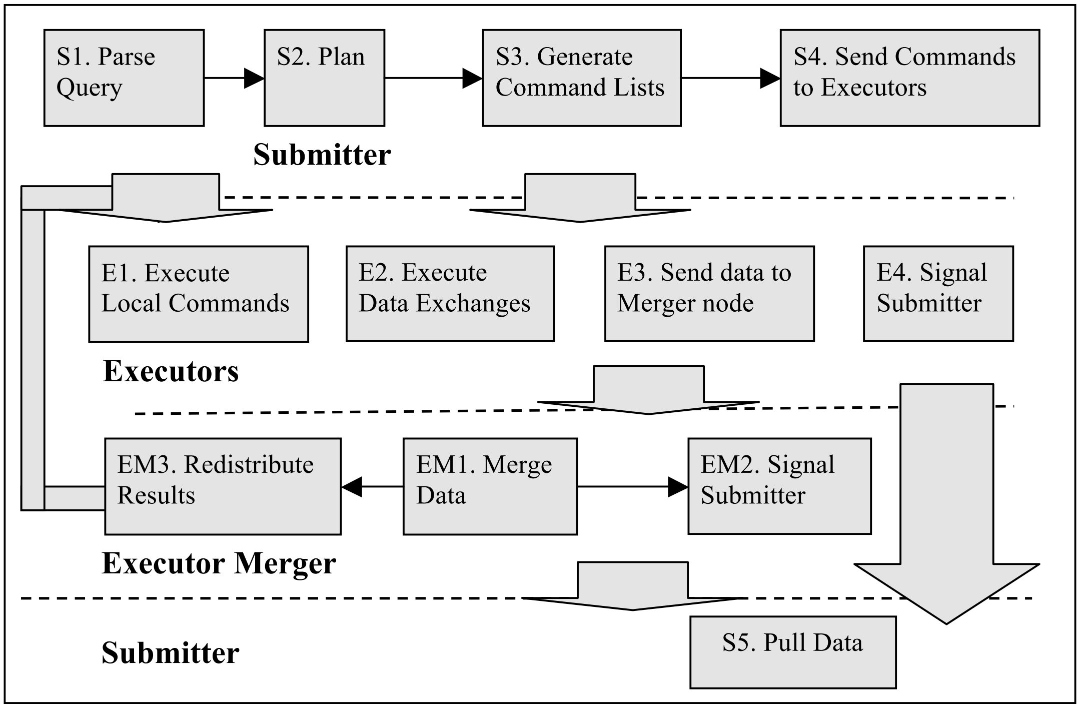

S1. Parse Query
S2. Plan
S3. Generate Command Lists
S4. Send Commands to Executors
Submitter
E1. Execute Local Commands
E2. Execute Data Exchanges
E3. Send data to Merger node
E4. Signal Submitter
Executors
EM3. Redistribute Results
EM1. Merge Data
EM2. Signal Submitter
Executor Merger
Submitter
S5. Pull Data

Figure 8: Query Processing Architecture (QPA) within DWPA

For instance, the following SQL query is from the TPC-H performance benchmark in (TPCC 2008) and computes the sales of each brand per month:

```
SELECT  p_brand, year_month, sum(l_quantity), count(*)
FROM    JOIN lineitem LI, part P, time T, supplier S
WHERE   year_month>= '1997' AND    supplier = ‘X’
GROUP BY   to_char(l_shipdate,'yyyy-mm'), p_brand, year_month;
```

This typical query contains group-by attributes that allow the aggregation to be determined for each group. This aggregation can be handled using the following scheme, which adheres to the diagram of Figure 7: each node needs to apply an only slightly modified query on its partial data, and the results are merged by applying the same query again at the merging node with the partial results coming from the processing nodes. Figure 9 illustrates this process for a simple sum query:

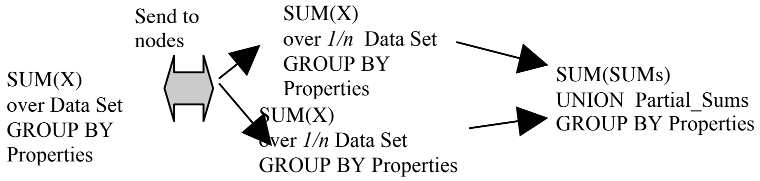


Figure 9: Typical SUM Query over DWPA

While the sum operation was unchanged in the query rewrite step of Figure 9, other aggregation operators need slight modifications. In practice simple additive aggregation primitives are computed in each node, from which the final aggregation function is derived. The most common primitives are: (LS, SS, N, MAX, MIN: linear sum LS = sum(x); sum of squares SS = sum(x^2); number of elements N, extremes MAX and MIN).Examples of final aggregation functions are:

$$COUNT = N = \sum\nolimits_{all\_nodes} n_{nodei} \tag{1}$$

$$SUM = LS = \sum\nolimits_{all\_nodes} LS_{nodei} \tag{2}$$

$$AVERAGE = \sum\nolimits_{all\_nodes} LS_{nodei} / \sum\nolimits_{all\_nodes} N_{nodei} \tag{3}$$

$$STDDEV = \sqrt{\frac{(\sum SS_{node_i} - \sum LS_{node_i}{}^2 / N)}{N}} \tag{4}$$

This means that the query transformation step needs to replace each AVERAGE and STDDEV (or variance) expression in the SQL query by a SUM and a COUNT in the first case and by a SUM, a COUNT and a SUM_OF_SQUARES in the second case to determine the local query for each node. Figure 10 shows an example of aggregation query processing steps, indicating the corresponding steps from Figure 8.

```
S1. Query submission:
Select      sum(a),      count(a),
average(a), max(a), min(a),
stddev(a), group_attributes
From data set
Group by group_attributes;
```

```
S3,S4.   Query    rewriting   and
distribution to each node:
Select sum(a), count(a), sum(a x
a),       max(a),        min(a),
group_attributes
From data set
Group by group_attributes;
```

```
E3. Results sending/collecting:
Create cached table
PRqueryX(node,    suma,    counta,
ssuma, maxa, mina,
group_attributes)
as <insert received results>;
```

```
EM1. Results merging:
Select sum(suma), sum(counta),
sum(suma)/          sum(counta),
max(maxa), min(mina)
(sum(ssuma)-
sum(suma)^2)/sum(counta), ga
From UNION_ALL(PRqueryX)
Group by group_attributes;
```

Figure 10: Basic Aggregation Query Steps

The query processing steps described above are not the only possibility for processing this query using intra-query horizontal parallelism. For instance, it would be possible to assign each group of the group-by clause of the example in Figure 10 to a specific node. Nodes would then have to exchange data so that each node would get the data pertaining to the groups that it should process. Given these and other query processing alternatives, the query optimizer must evaluate the costs of each one, including processing and data exchange costs, and determine the best alternative for each query.

While most SQL operations are readily parallelizable in a way that may allow nodes to process data with minimum data exchange requirements, the parallel join operator may incur in considerable data exchange overheads if the rows to be joined are not co-located or equi-partitioned (located in the same node). This is because the join must match rows from two data sets with the same value for a specific attribute. With careful hash-partitioning, it is possible that the two data sets are already placed in a co-located manner, that is, rows with matching attribute value may be co-located, but for schemas involving multiple large partitioned relations it is often impossible to co-locate all relations. In those cases some form of data exchange will be required. The following lists the parallel join alternatives:

Co-located or equi-partitioned join: the data is already partitioned according to the join hash values, so the join can occur in parallel with no data exchange requirements between nodes;

Redirected join: the data is not co-located but it is enough to re-locate the rows from one of the source data sets in order to co-locate them and proceed with a co-located join;

Repartitioned join: both source data sets need to be re-located in order to become co-located;

Broadcast join: one of the source data is broadcasted into all nodes to enable parallel joining with the other partitioned data set;

Once more, it is the task of a global query optimization to search for the best processing plans form a myriad of alternatives, and it is the task of a query processor with dynamic load balancing to counter potential imbalances, which include data storage, data exchange and data processing imbalances. In the next section we review work on parallel query optimizers.

THE PARALLEL OPTIMIZER

The parallel optimizer is a crucial element in any parallel database management system. In this section we review its concept and work on parallel query optimizers along time.

As database systems evolve, the need to develop query optimizers that are able to interpret their new features also arises. The evolution process has not been simple, as query optimizers have to conjugate database feature compatibility with reasonable search times.

The first globally accepted standalone query optimizer was presented by (Selinger et al. 1979) for the System R database engine. The optimizer was based on Dynamic Programming (DP), and tried to obtain the best execution plan by successively joining relations using the cheapest access paths and join methods. After each join iteration, the costliest plans were pruned leaving only the most eligible for the next phase. The exceptions to the previous rule were costlier plans considered to be interesting because they could benefit future join plans or database operations that had yet to be accounted for. The System R query optimizer had a major disadvantage, its space and time complexities are exponential, which limited the join cardinality of a submitted query.

In the early 1980's researchers began to tackle query optimization in parallel/distributed environments with communication costs. At the time standalone query optimizers were not equipped to handle features inherent to distribution, such as query segment node attribution or communication costs. Extensive work was done to adapt distributed concepts to the standalone optimizer domain. An example of this attempt is the R* (Lohman et al. 1985) which adds distribution features to the DP algorithm (discussed earlier). Kossman and Stocker (2000), develop a series of algorithms that adapt standalone DP to distributed environments. They reduce the inherent complexity, of such an adaptation, by introducing greedy algorithms that lessen the overall search space of a distributed query.

As discussed before, parallel architectures can exploit two types of parallel features, i.e. inter-query and intra-query. The first tries to simultaneously execute independent queries by assigning them to different sets of processors. Intra-query parallelism breaks down the operators that compose a query, conveying them to different processors for concurrent execution. We can further subdivide intra-query parallelism into inter-operator and intra-operator parallelism. The latter promotes the division of a specific operator into fragments, and subsequent concurrent execution of each fragment by a separate processor. Inter-operator parallelism allows two or more operators to be simultaneously executed by different sets of processors. The introduction of these features augments the search space of a parallel system. Thus it is important that parallel optimizers be able to reduce the search space without compromising plan generation.

One of the first parallel query optimizers to appear, the XPRS (Hong and Stonebraker 1993), created effective parallel plans using a two-phase technique. The first generated a traditional single processor execution plan, which was then scheduled for parallelization in the second phase. However the authors did not study the effects of processor communication, which limited its applicability on architectures that heavily rely on this resource, i.e. SN. The Join Ordering and Query Rewrite (JOQR) algorithm proposed in (Hasan 1996) also employed a two phase approach. However the first phase was different from that of the XPRS. JOQR rewrote the queries, submitted to it, by using a set of heuristics to diminish the search space, and took into account processor communication by using node colouring (Hasan and Motwani 1995). Node colouring applies a colour to represent PNs with relations partitioned by the same attribute. Joining relations with the same colour would account for less communication costs, and thus diminish the global response time. In (Shasha and Wang 1991), the authors propose a one-phase search algorithm that uses graph theory to solve the best order by which to join a set of relations. In their approach they apply a CHAIN algorithm to try and minimize the response times of the execution plans. CHAIN is based on DP, yet it can eliminate more plans if Kruskal's (1956) or Primm's (1957) spanning trees are used as a greedy heuristic. The authors focus on equi-join scenarios leaving out the rest of the join types, this limits the applicability of the algorithm.

With the ever-growing complexity of today's queries, researchers are focusing on methods that try to diminish the margin of error of current query optimizers. Special interest has arisen in the usage Robust Query Optimization (Babcock and Chaudhuri 2005, Chu, Halpern and Gehrke 2002) techniques. The methodology allows an optimizer to decide whether it wants to pursue a conservative or more aggressive plan to resolve a query. The conservative plan is likely to perform reasonably well in most situations, whilst the aggressive plan (traditional method) can do better, if the estimations are reliable, or much worse if these are rough approximations. Other researchers have studied Parametric Query Optimization (Ganguly 1998, Hulgeri et al. 2003, Ioannidis et al. 1997), which is based on the finding of a small set of plans that are suitable for different situations. The method postpones some of its decisions until runtime, using a subset of plans to decide how best to proceed to the next execution phase. Parametric query optimization does particularly well in scenarios where queries are compiled once and executed repeatedly, with possible minor parameter changes. However the subject that has captivated most of the recent interest is that of Adaptive Query Processing. The term classifies algorithms that use execution feedback as a means of resolving execution and optimization problems related to missing statistics, unexpected correlations, unpredictable costs and dynamic data. In (Deshpande et al. 2007) the authors conduct a reasonably complete survey on the issue.

Having discussed most aspects of interest to parallel data warehouse architectures and systems, we devote the next section to reviewing related work on parallel data warehouse systems and solutions.

PARALLEL DATA WAREHOUSE SYSTEMS

In this section we review works on parallel data warehouse systems, including both commercial systems and some of the most relevant research work on parallel data warehouses.

On the commercial side, one of the most popular commercial SN parallel database solutions is the IBM DB2 Parallel Edition (Baru and Fecteau 1995). Its partitioning strategy is based on hashing and nodegroups (NG), which allow a table to span multiple nodes. The relations are hash-partitioned, so as to form collocated data whenever possible. The application is very flexible, as it allows database administrators to introduce or remove processing nodes (PNs) into the system, without having to spend a lot of time reconfiguring the DW. It supports shipping of sub-queries and parallel joins. SQL statements are broken into fragments for each node, executing low-level operations. A cost-based optimizer takes the parallel settings, data partitioning and data exchange costs to optimize the execution plan. On the other hand, DB2 is by no means specialized for data warehouses or data warehouse schemas and workloads, and although it has an advanced cost-based optimizer, it is not specifically adapted to heterogeneous and non-dedicated environments. As with most proprietary application, it requires the acquisition of the entire software bundle together with a fast hardware architecture, which are both usually costly.

Rao (Rao et al. 2002) seeks to automate the process of data partitioning in the IBM DB2 shared nothing architecture. Given a workload of SQL statements, the objective is to determine automatically how to partition the base data across multiple nodes to achieve overall optimal performance for that workload. Instead of using only heuristics as previous approaches did, their solution is tightly integrated with the optimizer and uses it both to recommend candidate partitions for each table that will benefit each query in the workload, and to evaluate various combinations of these candidates.

Another popular commercial solution is Oracles' Real Application Cluster (RAC 2008). The RAC assumes a hybrid parallel architecture. The PNs get their data from a high-end disk arrays such as a Network Attached Storage (NAS) or a Storage Area Network (SAN), meaning that the application is partially built on SD. Once the data is loaded onto the PNs, a cache fusion technology is activated, which basically allows the PNs to query each others' cache so as to obtain needed data without resorting to the disk array. Oracle classifies the cache fusion as a form of shared everything. The main disadvantages of the RAC are its necessity for high-end hardware such as a disk array and a fast network interconnect for inter-node and disk array communication. Thus, a fair amount of expenditure is needed to acquire both the hardware and software to build this system.

Research work and prototypes have proposed relevant solutions to speedup processing over both SN and SD architectures. We review some of those next.

Akal (Akal et al. 2002) investigates processing OLAP queries in parallel, where a coordination infrastructure decomposes a query into subqueries and ships them to appropriate cluster nodes. In the second phase, each cluster node optimizes and evaluates its subquery locally. Their proposal involves efficient data processing in a database cluster by means of full mirroring and creating multiple node-bound sub-queries for a query by adding predicates. This way, each node receives a sub-query and is forced to execute over a subset of the data, called a virtual partition. The advantage of virtual partitions is that they are more flexible than physical ones, since only the ranges need to be changed. The authors of (Lima et al. 2004) further improve the previous solution. They claim that the solution is slow because it does not prevent full table scans when the range is not small, and propose multiple small sub-queries by node instead of just one (Lima et al. 2004b), and an adaptive partition size tuning approach.

The Data Warehouse Parallel Architecture (DWPA) (Furtado 2004, Furtado 2005, Furtado 2007) is an environment adaptable and cost-effective middleware for parallel DWs, which places no requirements in either RDBMS software or hardware. Its services are aimed at SN and mixed SN-Grid environments mostly, because these are very scalable and their cost acquisition is relatively low. In SN environments, the DW is partitioned amongst the PNs so that each DBMS instance can directly access data from its local partition. Partitioning is justified by the fact that PNs do not have enough storage space to accommodate the entirety of the DW, and that by subdividing the relations into various PNs the system is able to parallelize query execution. The DWPA architecture partitions a schema by using a partitioning strategy called Workload Based Placement (WBP) (Furtado 2004). In short, WBP hash-partitions large relations based on the schema and workload characteristics of the DW, whilst small relations are replicated throughout the PNs. A relation is considered to be small if it can fit comfortably in physical memory, and the operations involving that relation are not significantly slower than those of a partitioned alternative. WBP tries to equi-partition relations whenever possible, so as to maximize query throughput. To equi-partition or co-locate relations is to divide two or more relations using the same attribute and hash function. In doing so each PN is able to access locally and join the equi-partitioned relations, which avoids any unnecessary data exchange among the PNs. DWPA can be organized into a set of node groups (NG) (Furtado 2007). Each NG is composed of a set of PNs. NGs are usually created for reasons related to availability, performance enhancement or geographic locality. The usage of NGs benefits parallel query execution in various manners as they may contain any subset of the data warehouse data. The creation of multiple NGs, each containing the entirety of the DW, can also allow the execution of simultaneous queries in local or networked contexts, without these interfering with each others response times. Another application of NGs is to foment the use of bushy trees in query processing: if the DW is subdivided into two or more NGs then it is possible to segment a query and execute portions of it in separate NGs. Replication for dynamic load-balancing and availability is also considered in (Furtado 2005b, Furtado 2007), where instead of node partitions the data set is divided into a much larger set of chunks and those are divided into the nodes in alternative ways that confer the desired load and availability balancing characteristics.

The Multidimensional Hierarchical Fragmentation approach (Stohr et al. 2000) proposes a data allocation solution for parallel data warehouses that takes into account the data warehouse structure, including its multidimensional and hierarchical characteristics. The idea is that a multi-dimensional hierarchical fragmentation of the fact table with in-memory copies of all dimensions supports well queries referencing different subsets of the schema dimensions. Fragments are determined based on dimension attribute values, once a specific combination of attributes is chosen as partitioning criteria. The system then uses hierarchy information to determine which fragments should be accessed to answer a specific query. Bitmap indexes are also added to speedup processing when small parts need to be accessed from specific fragments.

Stöhr (Stöhr et al. 2002) investigates dynamic load-balancing for parallel data warehouses focused on shared disk systems, proposing a scheduling strategy that simultaneously considers both processors and disks, while utilizing the load balancing potential of a shared disk architecture, also catering for skew.

Aguilar-Saborit et al. 2005 looks at the problem of ad-hoc star join query processing in cluster architectures, and propose the Star Hash Join (SHJ), a generalization of Pushed Down Bit Filters (Aguilar-Saborit et al. 2003) for clusters. The objectives of the technique are to reduce the amount of data communicated, the amount of data spilled to disk during the execution of intermediate joins in the query plan, and the amount of memory used by auxiliary data structures such as bit filters.

In (Raman, Han and Narang 2005), the authors argued for the advantages of a shared disk solution (SD) with no pre-partitioning, with the argument that it allows a completely dynamic allocation of processing to the processing units. They presented DITN, an approach where partitioning is not the means of parallelism. In their approach, data layout decisions are taken outside the scope of the DBMS and handled within the storage software. Query processors see a "Data In The Network" image. The authors argue that repartitioning (on-the-fly data exchange between nodes that is sometimes necessary during the processing of partitioned data sets) is unsuitable for non-dedicated machines because it poorly addresses node heterogeneity, and is vulnerable to failures or load spikes during query execution. DITN uses an alternate intra-fragment parallelism, where each node executes an independent select-project-join-aggregate-group by block, with no tuple exchange between nodes. This method handles heterogeneous nodes cleanly, and adapts during execution to node failures or load spikes. Initial experiments suggest that DITN performs competitively with a traditional configuration of dedicated machines and well-partitioned data for up to 10 processors at least, while at the same time giving significant flexibility in terms of gradual scale-out and handling of heterogeneity, load bursts, and failures.

Chen (Chen et al. 2004) discusses parallel solutions for handling the data cube, arguing that pre-computation of data cubes is critical to improving the response time of On-Line Analytical Processing (OLAP) systems and can be instrumental in accelerating data mining tasks in large data warehouses. They present a parallel method for generating ROLAP data cubes on a shared nothing multiprocessor. Since no (expensive) shared disk is required, their method can be used on low cost clusters consisting of standard PCs with local disks connected via a data switch.

These research works have shown how allocation and query processing can be optimized in different ways for handling data warehouses and cubes effectively in both shared nothing and shared disk parallel architectures. In the next section we briefly discuss distributed data warehouses.

DISTRIBUTED DATA WAREHOUSES

Many organizations have physically distributed databases with extremely large amounts of data. Traditionally the data warehouse would be seen as a centralized repository, whereby data from all sources would be imported into that large centralized repository for analysis. Nowadays the speed and bandwidth of wide-area computer networks enables a distributed approach, whereby parts of the data may reside in different places, parts being cached and/or replicated for performance reasons, and the system functions to the outside world as a single global access-transparent repository. As the amount of data and number of sites grow, this distributed approach becomes crucial, as a single centralized data warehouse importing data from all the sources has obvious scalability limitations.

Grid technology is another useful element in distributed computing platforms, and naturally also for distributed data warehouses. The computational grid offers services for efficiently scheduling jobs on the grid, and for grid-enabled applications where data handling is a most relevant part, the data grid becomes a crucial element. It typically builds on the concept of files, sites and file transfers between sites. These use services such as GRID-ftp, plus a Replica Manager to keep track of where replicas are located. The multi-site, grid-aware data warehouse is a large distributed repository sharing a schema and data concerning scientific or business domains. However, differently from typical grid scenarios, the data warehouse is not simply a set of files and accesses to individual files, it is a single distributed schema and both localized and distributed computations must be managed over that schema.

Some authors have dealt with allocation and processing of data warehouses in distributed and grid environments. In (Akinde et al. 2003) the authors discuss query processing in a distributed data warehouse, consisting of local data warehouses at each collection point and a coordinator site, with most of the processing being performed at the local sites. In that paper they consider the problem of efficient evaluation of OLAP queries over the distributed data warehouse, and propose the Skalla system for this task. Skalla translates OLAP queries, specified as certain algebraic expressions, into distributed evaluation plans which are shipped to individual sites, and the approach operates in a manner that reduces the amount of data that needs to be shipped among sites.

In (Costa and Furtado 2006) the authors investigate Grid-Dwpa, an efficient architecture to deploy large data warehouses in grids with high availability and good load balancing. They propose both efficient data allocation, partial replication strategies and scheduling solutions that maximize performance and throughput of the grid-enabled architecture for OLAP. The replication strategies provide adequate guarantees that site availability problems do not impair the system and result in only small system slowdown. In (Costa and Furtado 2008) the authors propose scheduling for efficient query processing in the Grid-Dwpa environment. The system generates site and node tasks, forecasts the necessary time to execute the task at each local site, estimates total execution times, and assigns task execution to sites accordingly.

Another work on grid-aware data warehouses is presented in (Lawrence Rau-Chaplin 2006). The OLAP-Enabled Grid considers the scenario where the data of a single organization is distributed across a number of operational databases at remote locations. Each operational database has capabilities for answering OLAP queries, and access to a possible variety of other computational and storage resources which are located close by. Users who are interested in doing OLAP on these databases are distributed over the network. Their proposal considers the following entities:

OLAP Server - A machine which has sole control over an operational database. It may maintain some materialized views and may also act as a computational or storage resource. The OLAP servers all have the same schema, but each maintains a partition of the total data available to the users;

Computational Resource - A machine which offers cycles for performing tasks on the behalf of other entities in the Grid;

Storage Resource - A machine which offers disk space for storing data on behalf of other entities in the Grid;

Resource Optimizer - There is exactly one resource optimizer for each site. A resource optimizer has the information necessary to perform scheduling and allocation of computational and storage resources, and to carry out queries. It may also have some cache space for storing common query results for queries generated in its site;

User - Users submit ad-hoc queries to resource optimizers and may enter and leave the network at will. Each user has an amount of cache space for caching query results.

The proposal itself is for a two-tiered grid-based data warehouse. The first tier is composed by local (cached) data. Remote database servers are in the second tier. Each submitted query is evaluated in order to verify if it can be answered with data from the local site. Then, if it cannot be entirely answered locally, the query is re-written into a set of queries. Some of those are executed locally (with the existent data) and the others, which access the data that is missing at the first tier, are executed at the second tier (database servers). The use of cached data is also considered at the database server level.

Wehrle et al (2007) also deal with a distributed, grid-aware environment. They apply the Globus Toolkit together with a set of specialized services for grid based data warehouses. Fact table data is partitioned and distributed across participant nodes. Dimension tables data is replicated. A *local data index service* provides local information about data stored at each node. A *communication service* uses the *local data index* service from the participant grid's nodes to enable that remote data is accessed. The first step in query execution is to search for data at the local node (using the local index service). Missing data is located by the use of the communication service and accessed remotely.

The distributed and grid-aware data warehouse context is still an evolving one, as community data warehouses come into play in current and future systems and concerning different application scenarios.

CONCLUSIONS AND FUTURE TRENDS

There has been a significant amount of work during the last two decades related to parallel and distributed data warehouses, and those works have contributed to increasing significantly our knowledge of those systems, issues and solutions, and it has also brought some maturity to the field. In this paper we reviewed the main concepts, works and trends on parallel and distributed data warehouse architectures and systems. We first described parallel architectures, types of parallelism, partitioning and allocation. Then we described how parallel horizontal intra-query processing works and we also reviewed the parallel optimizer. We then turned our attention to some of the most relevant works on parallel and distributed data warehouse systems.

Work in parallel and distributed data warehouses in the future is expected to advance the concepts and systems to new levels of autonomy, scalability and ubiquity. It will answer questions such as how the systems will be able to adapt automatically to very heterogeneous environments, in either parallel or wide-area distributed environments. It will provide answers to the issue of how to completely automate and optimize allocation and mixes of base data, materialized views, cubes and indexes in either parallel or distributed settings for optimal performance. We will also increasingly see applications of data warehouse and grid technologies to distributed and collaborative applications and problems.

REFERENCES

Akal F., Böhm K., Schek H.-J.(2002). OLAP Query Evaluation in a Database Cluster: a Performance Study on Intra-Query Parallelism, East-European Conf. on Advances in Databases and Information Systems (ADBIS), Bratislava, Slovakia, 2002.

Akinde, M. O., Bhlen, M. H., Johnson, T., Lakshmanan, L. V. S. and Srivastava, D. (2003) " Efficient OLAP query processing in distributed data warehouses", Information Systems 28, pp. 111-135, Elsevier, 2003.

Babcock B. and Chaudhuri S. (2001). Towards a robust query optimizer: a principled and practical approach. In SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pages 119–130, New York, NY, USA, 2005.

Baru C. and G. Fecteau (1995). An overview of db2 parallel edition. In SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, pages 460–462, New York, NY, USA, 1995.

Bellatreche L. (2008). A Genetic Algorithm for Selecting Horizontal Fragments, Encyclopedia of Data Warehousing and Mining - 2nd Edition , John Wang, 2008

Bellatreche L. (2008). Bitmap join indexes vs. Data Partitioning, Encyclopedia of Data Warehousing and Mining - 2nd Edition , 2008.

Chan C.-Y. and Ioannidis Y. E. (1998). Bitmap index design and evaluation. Proceedings of the International Conference on the Management of Data, pages 355-366, 1998.

Chaudhuri, S. and Dayal, U. (1997). An overview of data warehousing and OLAP technology. SIGMOD Rec. 26, 1 (Mar. 1997), 65-74.

Chen Y., Dehne F., Eavis T., Rau-Chaplin A. (2004). Parallel ROLAP Data Cube Construction On Shared-Nothing Multiprocessors. In Distributed and Parallel Databases, Volume 15, Number 3, May 2004, pages 219-236.

Chu F., Halpern J., and Gehrke J. (2002). Least expected cost query optimization: what can we expect? In PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 293–302, New York, NY, USA, 2002.

Costa R. and Furtado P (2008). Optimizer and QoS for the Community Data Warehouse Architecture, in New Trends in Database Systems: Methods, Tools, Applications", Eds. D. Zakrzewska, E. Menasalvas, L. Byczkowska-Lipiñska1, Springer-Verlag, 2008.

Costa R. and Furtado P. (2006). Data Warehouses in Grids with High QoS. In A. M. Tjoa and J. Trujillo, editors, DaWaK, volume 4081 of Lecture Notes in Computer Science, pages 207–217. Springer, 2006.

Deshpande A., Z. Ives, and V. Raman. (2007). Adaptive query processing. Foundations and Trends in Databases, 1(1):1–140, 2007.

DeWitt D. J. and Gray J. (1992). Parallel database systems: The future of high performance database systems. Commun. ACM, 35(6):85–98, 1992.

Furtado P. (2004). Workload-based Placement and Join Processing in Node-Partitioned Data Warehouses. In proceedings of the International Conference on Data Warehousing and Knowledge Discovery, 38-47, Zaragoza, Spain, September 2004.

Furtado P. (2004). Experimental Evidence on Partitioning in Parallel Data Warehouses. Proceedings of the ACM DOLAP 04 - Workshop of the International Conference on Information and Knowledge Management, Washington USA, Nov. 2004.

Furtado P. (2005). Hierarchical aggregation in networked data management. In Euro-Par, volume 3648 of Lecture Notes in Computer Science, pages 360–369. Springer, 2005.

Furtado P. (2005). Replication in Node-Partitioned Data Warehouses. DDIDR2005 Workshop of International Conference on Very Large Databases, 2005.

Furtado, P. (2005). "Efficiently Processing Query-Intensive Databases over a Non-dedicated Local Network". Proceedings of the 19th International Parallel and Distributed Processing Symposium, Denver, Colorado, USA, May 2005.

Furtado P. (2007). Efficient and Robust Node-Partitioned Data Warehouses", in "Data Warehouses and OLAP: Concepts, Architectures and Solutions, ISBN 1-59904365-3 eds. R. Wrembel and C. Koncilia, Ideas Group, Inc, chapter IX, pp. 203-229, 2007.

Ganguly S. (1998). Design and analysis of parametric query optimization algorithms. Proceedings of the 24rd International Conference on Very Large Data Bases, pages 228–238, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

Golfarelli M., Maniezzo V., and Rizzi S. (2004). Materialization of fragmented views in multidimensional databases. Data & Knowledge Engineering, 49(3):325–351, June 2004.

Hasan W. (1996). Optimization of SQL queries for parallel machines. PhD thesis, Stanford, CA, USA, 1996.

Hasan W. and Motwani R. (1995). Coloring away communication in parallel query optimization. In VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases, pages 239–250, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

Hong W. and Stonebraker M (1993). Optimization of parallel query execution plans in xprs. Distributed and Parallel Databases, 1(1):9–32, 1993.

Hulgeri A. and Sudarshan S. (2003). Anipqo: almost non-intrusive parametric query optimization for nonlinear cost functions. In Proceedings of the 29th international conference on Very large data bases, pages 766–777.

Ioannidis Y., R. T. Ng, K. Shim, and T. K. Sellis (1997). Parametric query optimization. VLDB J., 6(2):132–151, 1997.

Kossmann D. and Stocker K. (2000). Iterative dynamic programming: a new class of query optimization algorithms. ACM Trans. Database Syst., 25(1):43–82, 2000.

Kruskal J. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 7(1):48–50, Feb. 1956.

Lawrence M. and Rau-Chaplin A. (2006). The OLAP-Enabled Grid: Model and Query Processing Algorithms" in Proceedings of the 20th International Symposium on High Performance Computing Systems and Applications (HPCS'06), IEEE, Eds. R. Deupree, St. Johns, Canada, May 2006.

Lima, A. A. B., Mattoso, M., Valduriez, P. (2004). OLAP Query Processing in a Database Cluster, Proc. 10th Euro-Par Conf., Pisa, Italy, 2004.

Lima, A. A., Mattoso, M., Valduriez (2004). P. Adaptive Virtual Partitioning for OLAP Query Processing in a Database Cluster", 19th Brasilian Simposium on Databases SBBD, 18-20 October 2004, Brasília, Brasil.

Lohman G., Mohan C., Haas L., Daniels D., Lindsay B., Selinger P., and Wilms P. (1985). Query processing in r*. In Query Processing in Database Systems, pages 31–47. Springer, 1985.

O'Neil P. and Graefe G. (1995). Multi-Table Joins Through Bitmapped Join Indices. SIGMOD Record, 24(3):8-11, 1995.

Ozsu M. T. and Valduriez P. (1999). Principles of Distributed Database Systems : Second Edition. Prentice Hall, 1999.

P. Furtado. Replication in node partitioned data warehouses. In VLDB Workshop on Design, Implementation, and Deployment of Database Replication (DIDDR), 2005.

Prim R. (1957) Shortest connection networks and some generalizations. The Bell System Technical Journal, 3:1389–1401, 1957.

Raman V., Han W., Narang I. (2005). Parallel querying with non-dedicated computers, in Proceedings of the 31st international conference on Very large databases, Trondheim, Norway, 2005.

Rao, J., Zhang c., Megiddo n., Lohman G. (2002). "Automating Physical Database Design in a Parallel Database". Proceedings of the ACM International Conference on Management of Data, 558-569, Madison, Wisconsin, USA, June 2002.

Rousopoulos R. (1998). Materialized Views and Data Warehouses. SIGMOD Record, 27(1):21-26, 1998.

Saborit J., Muntés-Mulero V., Larriba-Pey J (2003). Pushing Down Bit Filters in the Pipelined Execution of Large Queries. Euro-Par 2003: 328-337.

Saborit J., Muntés-Mulero V., Zuzarte C, Larriba-Pey J. (2005). Ad Hoc Star Join Query Processing in Cluster Architectures. DaWaK 2005: 200-209.

Sanjay A., Narasayya V. R., and Yang B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 359–370, June 2004.

Sanjay A., Surajit C., and Narasayya V. R. (2000). Automated selection of materialized views and indexes in microsoft sql server. Proceedings of the International Conference on Very Large Databases, pages 496–505, September 2000.

Selinger P., Astrahan M., Chamberlin D., Lorie R., and Price T. (1979). Access path selection in a relational database management system. In SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data, pages 23–34, New York, NY, USA, 1979.

Shasha D. and Wang T.-L. (1991). Optimizing equijoin queries in distributed databases where relations are hash partitioned. ACM Trans. Database Syst., 16(2):279–308, 1991.

Stöhr, T, Märtens, H.; Rahm E. (2000). Multi-Dimensional Database Allocation for Parallel Data Warehouses, Proc. 26th Intl. Conf. on Very Large Databases (VLDB), Cairo, Egypt, 2000.

Stöhr, Märtens T, Rahm H., Erhard (2002). Dynamic Query Scheduling in Parallel Data Warehouses Proceedings of Euro-Par 2002 Conference, Paderborn, August 2002.

Yu, C. T. and Meng W. (1998). Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, 1998.

TPC (2008). Transaction processing council benchmarks - http://www.tpc.org/.

RAC (2008) - Oracle real application clusters, http://www.oracle.com/technology /products/database/clustering/index.html.