A survey of Shared-Nothing Parallel Database Management Systems

[Comparison between Teradata, Greenplum and Netezza implementations]

Thomas Müseler
University of Applied Science Darmstadt
Haardtring 100
64295 Darmstadt, Germany
Thomas.Mueseler@gmail.com

ABSTRACT
Distributed database systems can be implemented in a many different ways. Mostly, they are customized for a special environment to handle big data problems. The data warehouse sector relies on these amounts, but has changed from a data storage to a real time management support during the last years [3]. The resulting increase of computation and storage capacity poses new requirements to the database systems. Previous approaches of a parallel database environment tried to solve this problem with shared disk and memory approaches.
The main contribution of this paper is the presentation of the current technology in the shared-nothing database sector. The concepts of the manufacturers Teradata, Greenplum and Netezza will be discussed for data warehouse requirements. Based on an architectural overview is a detailed insight of the index functionality given which is a crucial performance factor. Also data distribution algorithms of the manufacturers are analysed under data warehouse conditions.
At the end is a comparison to other shared concepts (shared-disk, shared-everything) given and the question raised, if the actual approach can be fulfilled by the manufacturers.

1. INTRODUCTION
Rising data rates and big data volumes are the new challenges for the today’s database systems. With computer unions is tried to counteract and therefore to distribute the computing loads to several instances.

An approach of distributed systems is the shared-nothing architecture in which each node can operate independently and separated from the other nodes. In contrast to the classical shared database concepts in which the main memory or hard disks are shared, each node has its own hardware components. In association with a high-speed network results a powerful computer network which is becoming increasingly popular through the excellent scalability.

This paper gives an analysis of the existing architecture concepts in shared-nothing database systems. It gives an conceptual insight of the manufacturer Teradata, Netezza and Greenplum in chapter 2. These three vendors are relatively small companies compared to the market leader (Oracle, IBM, SAP) and pursue interesting ideas in this sector.
A focal point of this paper is to the index implementation in chapter 4 which takes a decisive influence on the data distribution to each node. Based on the application range in the data warehouse environment (chapter 5), a comparison is made to other architectural models and the scaling of these kinds of networks. Furthermore is the question raised, if the shared-nothing can be adapted to other application fields and is therefore a good opportunity for future implementations.

2. ARCHITECTURE
The classification of distributed database systems in terms of their architecture can be done at different levels. The most widely used approach of Michael Stonebraker [18] builds the basis for further architectural views and will be extended. Basically, we distinguish between three main approaches. The shared-everything (SE), the shared-disk (SD) and the shared-nothing (SN) architecture are one of the basic concepts in a shared database environment. While SE-systems are sharing the processors (P) / memory resources (M) and thus constitute a closed circuit, require the SD / SN variants a communication network (N) to integrate their components.

Figure 1: Stonebraker Architecture with shared-everything, shared-disk, shared-nothing

Within the shared-nothing architecture, each processor uses its own main memory and disk. A high-speed network is connecting the various machines and is required for sharing and organization operations. This makes it more complex compared to the other two variants concerning the management of a large number of processing nodes.
The construction of a shared-nothing architecture can be divided into the following steps [14]:
1. Create a partitioning schema
2. Data distribution to the instances
3. Load balancing setup
4. Repeat the process in case of a re-partitioning
The required load balancing is a crucial success factor of a shared-nothing architecture and depends on the effectiveness of the data partitioning and load allocation systems. Furthermore, the addition of new database nodes is usually associated with a reorganization which should not limit the scalability. These points will be discussed with the implementation variants of the different manufacturers in the following chapters. Also the important part of the index implementation which is directly connected to the partitioning. The architectural point of view will be mapped to the data warehouse requirements at the end.

3. SYSTEM OVERVIEW AND PARTITIONING

In this section, the shared-nothing approaches and the architectural structure of the manufacturer are presented. One of the key points are the data distribution and partitioning over an extendable amount of computing nodes. The physical data structures will not be discussed in detail. The used sources for Teradata are [8][20][5], for Netezza [7][9][11] and Greenplum [6].

3.1 Teradata

The main components of the Teradata architecture are the Parsing Engine (PE), the access module processor (AMP) and the BYNET framework (Figure 2). The parsing engine is the user interface and responsible for the in- and out coming database queries. For these queries, an execution plan is created and the necessary AMPs determined. To create the optimized queries, the PE is aware about the number of connected AMPs and their associated structural information. A further task of the parsing engine is the right management. The entered user interactions are checked against the rights concept and accordingly passed over the redundant message bus (BYNET Framework).

![Figure 2: Teradata Architecture](image)

If there are new messages on the BYNET, these are read from the Access Module Processor. Each AMP is connected with its own secondary storage and has only for those a write and read access. Teradata distributes the data to the associated AMPs and the required operations are performed locally. Data scans, index scans, aggregations, sorts and joins are executed separately by each AMP. The standalone file system provides journaling, caching and a recovery management. An optimum database performance is achieved by distributing the data across all AMPs. Teradata uses a hash partitioning schema to decrease the risk of access bottlenecks. The data is assigned to the AMPs based on a complex hash algorithm whereby the choice of the primary index plays an important role. As opposed to a random structure, the hash partitioning provides a direct access to the data which has a positive effect to the execution times. When a new node is added to the AMP framework, a reorganization of the tables is required. This is fulfilled by the hash function. The parsing engine is now taking the new node in the live environment which creates a linear scalability.

3.2 Netezza

By considering the Netezza architecture, there is no big difference compared to the Teradata architecture at the first view. The SMP host (Symmetric Multiprocessing) answers the requests of the various applications and passes them over a transport layer to the Snippet Processing Units (SPUs). During the passing process occurs the first difference. Since the parsing engine addresses the connected Teradata AMPs directly by a hash function, the SMP host forwards the request to all SPUs. A single write or read access enables therefore all machines even if only one of them performs the operation effectively. This may not be performant in individual operations, but it can achieve at high transaction volumes to a speed advantage. Netezza speaks about a two-stage architecture in this case. The operation requests will not be processed to the single nodes, but transferred to the physical layer with all involved SPUs.

![Figure 3: Netezza Architecture](image)

After the table data has been particularly loaded and distributed to all SPUs, an analysis process of the data is done. This includes a column-wise statistic on the lowest and highest data values and a few additional statistical values. This data is used as an input for the construction of the so-called "Zone Maps". Zone maps can be seen as a kind of partitioning method. The distributed records are based on a "Distribution Key" which can be set during the table creation. This key sets the partitions not fixed, but the number of allocated SPUs is predefined for each configuration. The distribution of the data is considered separately from the generation of the zone maps. If no specified distribution key is set, the data is randomly distributed. Unless there is a sorted load sequence, the SPUs also generate a zone map automatically. The dy-
namically partitioning of the “Zone Maps” is a different approach compared to the other vendors and should be explained in the following example:

A data warehouse analysis of telecommunication data will be performed over the time by phone calls. If the call data is sorted in the fact table according to a timestamp, Netezza automatically creates a zone map for this column. For further queries is thereby known exactly to which SPU sectors the data is provided.

If additional columns for the Distribution Key are required, they can be extended up to four additional columns. It should be noted that the data is not positioned in a bundle on one SPU. A join between two large tables for example would run therefore only on one single SPU which would be not effective in a parallel approach.

3.3 Greenplum

Also Greenplum uses a master-client architecture like the two previously presented models. The master server is responsible for the coordination of the requests, the load distribution and the resulting returns. The Greenplum database is based on PostgreSQL and the master server is the external view of an ordinary PostgreSQL database to the end user. This allows a faster transition since already implemented solutions for PostgreSQL, as the existing client programs such as psqi or APIs can be reused.

Data and index structures are distributed over the available segments and can be addressed exclusively by the master. The distribution of the data can be determined by two methods. The standard method is claimed in one or more distribution hash keys, similar to the Netezza method. If the hash values have the same values, the data is stored on the same segment. A wide dispersion of the data is achieved through the distribution by a primary key. The primary key or the first column of a table is also used when no distribution key is explicitly specified. The second variant is a random distribution of the data. According to the round robin method, the tuples are distributed sequentially to the segments.

The network layer provides the “Interconnect” as a gigabit Ethernet. UDP (User Datagram Protocol) is used as the standard protocol with an additional package validation by the Greenplum software. Featuring this method is the speed advantage as opposed to TCP (Transmission Control Protocol) persistent, without limiting the reliability. If TCP is used as the transmission protocol, the segment number is limited to 1000 nodes.

3.4 Architecture summary

The architecture analysis shows that all three manufacturers have no major differences. The requests will be accepted from one centralized client and distributed over a network layer to the individual server instances. The original Stonewinder architecture refers to a communication between the database instances and does not consider the central client. Further subdivisions in cluster architectures (clustered everything, clustered disk see [17]) are not explicitly described by the manufacturers. However, multiple clients can achieve such a subdivision. This is also dependent on the scalability,

as only a limited number of sessions/server instances can be managed. The database manufacturers implement data partitioning in different ways. Teradata relies on a distribution by means of a hash function which is closely linked to the index functionality. Netezza offers an automatic creation of its zone maps to support the user. If there is no defined distribution specified. With a distribution key, the partitions can be also created by itself. Such a distribution key is also implemented in Greenplum as a partitioning method and stored in the table definition. A further variation is the random distribution using the round robin method, which is not supported by Teradata and Netezza.

4. INDEX STRUCTURES

Optimized access structures are a crucial factor for the relation between the CPU and the write/read processing. The existing access gap by a factor of 10^5 between memory and disk storage [16] requires an intelligent data access via a buffer management. The usage of indices can contribute for this database optimization and defined primary and secondary keys can prevent full table scans. This can, for example, standard reports in the data warehouse environment lead to a better performance.

However, distributed database systems have to be also variable for ad-hoc requests and short processing times to full table scans. The higher the selectivity of these queries, the more computing power is required. In these cases, an intelligent index selection is crucial. Mass handling and selective queries constitute thus the challenges for the access system.

Through the different architectural concepts, the manufacturers of distributed shared-nothing databases can also notice a number of different index mechanisms. In the following section, these concepts will be considered in more detail. The sources are the user manuals of the manufacturer Teradata [8], Netezza [7] and Greenplum [6]. Additional resources for the Teradata architecture [10] and the Teradata indices [19].

4.1 Teradata

Based on a parallel database architecture uses Teradata the hashing method for the indices construction. This is considered as a more effective method compared to the “raw-data storage” and is used (with the exception of join indexes) for all types. Teradata offers a multitude of possible indexes. Here is the generation of the primary index in the

---

2. PostgreSQL console application
3. Application Programming Interfaces
foreground. The following list gives a rough overview about the available index types:

- Primary Index (UPI, NUPI, PPI, NPPI)
- Secondary Index (USI, NUSI)
- Join Index (Single, Aggregated, Multitable)

Each table of an AMP sets automatically a primary index. Generally, a distinction is made between two types. Besides the Unique Primary Key (UPI) which implies the uniqueness of tuples, the Non Unique Primary Index (NUPI) is for non-unique tuples.

As described in the architecture section, the hash value of the primary key is responsible for the distribution through the AMPs. This is generated from the primary index value of the hash function and is always unique to that value. A hash map contains the different buckets that represent the range of values of the current in the system available AMPs.

In four connected AMPs are four different buckets. The generated hash value points to one of the buckets, and thus determines the location of the entry. This calculation is performed in the parsing engine. The size of the bucket number can be specified with 16 or 20 bits, which corresponds to a size of 65,536 or 1,048,576 buckets. The other bits are reserved for the overflow (modulo).

The file system stores the hash value and other column values directly in the table. In case of a NUPI, the hash value is not always unique and another 32-bit value ("uniqueness value") is added. This represents the 64-bit row ID, which is also unique for a NUPI. In case of a same hash value, the uniqueness value is increased by 1. For unique indexes, the value remains according to 1.

In addition to the partitioning of the hash value, it is possible to define the partitions by themselves. The Partitioned Primary Indexes (PPI) defines the partitions already at the creation of the table. There is also a "nonunique" version available (NPPI).

The definitions of secondary and join indexes should not be further discussed at this point. This can be consulted at [1].

As described in the architecture section, the hash value of the primary key is responsible for the distribution through the AMPs. This is generated from the primary index value of the hash function and is always unique to that value. A hash map contains the different buckets that represent the range of values of the current in the system available AMPs.

In four connected AMPs are four different buckets. The generated hash value points to one of the buckets, and thus determines the location of the entry. This calculation is performed in the parsing engine. The size of the bucket number can be specified with 16 or 20 bits, which corresponds to a size of 65,536 or 1,048,576 buckets. The other bits are reserved for the overflow (modulo).

The file system stores the hash value and other column values directly in the table. In case of a NUPI, the hash value is not always unique and another 32-bit value ("uniqueness value") is added. This represents the 64-bit row ID, which is also unique for a NUPI. In case of a same hash value, the uniqueness value is increased by 1. For unique indexes, the value remains according to 1.

In addition to the partitioning of the hash value, it is possible to define the partitions by themselves. The Partitioned Primary Indexes (PPI) defines the partitions already at the creation of the table. There is also a "nonunique" version available (NPPI).

The definitions of secondary and join indexes should not be further discussed at this point. This can be consulted at [8].

4.2 Greenplum

Greenplum implemented the B-tree index as the default preference. Furthermore, the variants GiST and bitmap are available. Hash and GIN indexes are not supported.

GiST [13]:

The GiST (Generalized Search Tree) is a balanced tree structure that is used as a template for a wide variety of index implementations. Based on an object-oriented approach is an extensibility for new data types and specific queries supported. B-Trees, R-Trees and other index structures can be implemented.

The implementation of new index structures requires a very detailed database knowledge. For this reason, the GiST interface represent a higher level of abstraction, which is primarily concerned with the semantics of the data types. Concurrent processes, logging and search operations are implemented by the GiST Layer.

The support of range predictors (<, =, >) can in B-trees, or a more accurate data comparison may not be sufficient for specific data formats. An image database has to cover different other tasks like "Find all black and white photographs". To provide such functions, seven implementation methods are provided (consistent, union, compress, decompress, penalty, pickup split, same). Each method should not be further discussed at this point. These can be found in chapter 51.3 at [1].

A restriction to store large amounts of data is currently a lack of a missing bulk-load support. This makes the indexing relatively slow compared to the creation in a B-tree. First approaches are a generalization of the R-tree and described under [2].

Bitmap [4][21]:

To use a B-tree index for a complete table, can cause, that the indexes consume more space than the table data itself. If the columns of the table have also a low cardinality, the B-tree implementation reaches no gains for the performance. It is therefore recommended to use bitmap indexes at this point.

The bitmap index consists in its simplest form of a vector B per attribute value and the corresponding identifier. If an attribute is assigned to a corresponding value from the vector, this place is occupied with a "1". The remaining positions are filled with zeros.

<table>
<thead>
<tr>
<th>π(R)</th>
<th>B⁺</th>
<th>B⁻</th>
<th>B⁺</th>
<th>B⁻</th>
<th>B⁺</th>
<th>B⁻</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Value-List Index

A query can thus be realized through simple binary operations and provides a speed advantage in comparison of multiple WHERE terms. A mapping function converts the values afterwards and passes them as a result.

In a small value range obtains the bitmap index compression a higher benefit (in terms of memory requirements) compared to the B-tree. The optimal value range is given by Greenplum from 100 to 100,000 values in a column. For lower values, an index is not useful in most of the cases. For larger values, the B-tree implementation achieves more efficiency again.

4.3 Netezza

The index function in Netezza is taken from those already described in Section 3.2 zone maps. For this reason, no classical index structures exist. A new database query addresses only the affected zone maps respectively SMPs and not all available machines. Based on an ordered load sequence, range queries such as in a data warehouse environment, can achieve a high performance. Unlike indices, zone maps will be automatically created and restructured with no administrative effort.

The system generates zone maps for all table columns with the data types INTEGER, DATE, and TIMESTAMP. In addition, zone maps are available in materialized views, assumed the columns are ordered (ORDER BY) and the data
type is not greater than 8-byte. For larger columns (such as CHAR/VARCHAR), only the first 8 bytes are used for the generation. A detailed list of data types, refer to "zone maps and materialized views" at [7].

4.4 Index structure summary
In contrast to the architecture models, the implementation methods in terms of index structures are far apart of the various manufacturers. Teradata augur a fast data addressing by using hash indices. The only manufacturer of a B-tree implementation is Greenplum. If this method proves to be inefficient, it is referred to the bitmap storage. The GiST index is given for specific data types or individual realization. Netezza doesn’t support the classical index method and allows with its zone maps an automatic analysis and structuring of the data. The alignment and optimization of the data warehouse market is identified with this approach.

This shows that one single indexing approach does not exist. Since the critical factors such as structure, volume and operating range are specific, they cannot be generalized. A deeper insight to this topic will be given in Chapter 5 “Application field”.

Generally, manufacturers recommend the use of indexes by the default settings. Based on that, more detailed and optimized configurations can be made.

5. APPLICATION FIELD
The shared-nothing architecture should be compared and discussed to other approaches based on the data warehouse environment. Furthermore, the associated scaling and data distribution is explained on a lower implementation level. Other application fields are not in the scope of this paper.

5.1 Data Warehouse
Considering the recommended uses of shared-nothing databases is the data warehouse industry at the top. Large amounts of data have to be managed, in which the petabyte border has already fallen. This contrasts with the changing nature of the markets. Companies have to be flexible and able to make decisions in a short time. For these challenges, a data warehouse support is beneficial. The requirements for the database market are therefore shorter analysis cycles and a higher number of processable ad-hoc requests.

The performance of a particular system is limited. All high-performance database systems rely therefore on multiple processor and memory units. Since these combinations have also a limitation in their capacity, the extension of a system environment is a crucial factor.

The three main architectures of distributed database systems (shared-disk, shared-everything, shared-nothing) have been described in the previous sections. In this chapter, based on the scalability of a data warehouse are the advantages and disadvantages of each model presented. The shared-everything architecture is again divided into a shared-memory in order to clarify the differences. The used sources are [12] and [14].

Multiple processor units are sharing a single memory in the shared-memory architecture. Complex lock-algorithms at the drive level are not needed at this point, as long as the buffer and resource management is located at the storage system. The access to the connected processors can therefore be easily managed, but holds equally a scaling limitation. All write and read operations have to run on a common bus. The resulting bottleneck requires a complex hardware to keep to the cache memory of the processor consistent. Consequently, networks of more than 8 or 16 processors are rare.

Similar problems occur in the shared-disk architecture. The RAM is directly assigned to the processors and typically coupled via a SAN$^4$ or NAS$^5$ to the devices. The now independent storage system can no longer serve as a central buffer and resource management which increases the complexity of access control.

On the other side, the shared disc access could generate a bottleneck for the file system. In most cases is a "shared cache" implemented to counteract the already lower access speeds. Database requests imply therefore an additional validation of the local and shared cache. Only if no cache memory contains the requested page, it will be read from the hard disk. For typical data warehouse queries that run sequentially on a fact table, this method is not particularly suited to the shared-cache and creates an additional overhead.

With a horizontal partitioning of a shared-nothing system (Figure 5), the database tuples are shared across multiple nodes. The management of allocated resources does not require an interprocess synchronization at an individual system. This architecture is designed for data warehouse databases very well because it accesses with a large parallelism the data and therefore reduces the bandwidth on the bus system. A higher JOIN-performance between the fact table and the associated dimension tables is the result.

In order to achieve the profitable parallelism, a structured data sharing is necessary. This is in most cases controlled by the indices or a distribution key (see Chapter 3). In a star schema, the partitioning of the fact table is usually profitable. A general decision cannot be made in case of the data dependency and needs to be resolved individually. Often is a time or functional dependency given, which has to be detected, for example, by Netezza automatically. The relatively small dimension tables are distributed to different

---

$^4$Storage Area Network
$^5$Network Attached Storage
nodes to reach the greatest possible parallelism.

The distribution of lock and buffer management tables on each instance is a good prerequisite for the expansion of the database system. That makes it less restrictive compared to the shared-disc architecture and it is possible to run several thousand nodes at the same time. This must not necessarily represent high-performance computers, so that economic aspects are encouraged as well.

6. Conclusion and Prospect

The analysis of the shared-nothing approaches has shown that even the data warehouse field has great potential for this architecture. Especially the good scalability is a major advantage against the other architectures. The distribution of the data has a large influence on the performance of a system and determines where the execution will be performed. Relatively simple operations on a star schema can be implemented thereby as a high-performance system. A bigger problem, however, is the effective processing of complex database queries. Especially variable loads can cause the data distribution to the fact that already busy machines get further requests and an uneven distribution of resources occurs. Shared-disk systems can deal with such inquiries regarding the dynamic allocation in a better way and are therefore prioritized in such applications (for example OLTP). The raised question of the introduction "shared-nothing databases are the future?" cannot be generalized for all database sections. The data warehouse sector is a good developing field for the shared-nothing architecture and will become more and more important relating to the rising amounts of data. Other future markets, such as Cloud Computing[15], raises very similar requirements to a database. In particular, the dynamic addition of new nodes plays again a major role. An automated partitioning achieves thereby still not the same quality as a manual. It is a big challenge to figure out a profitable partitioning only based on a data analysis in widespread data structures. The shared-disk partitioning concept is independent from this problem and is well established in this area nowadays.

To integrate the shared-nothing architecture even more into these areas, an improved load-balancing and less complex partitioning is required. A lot of dependencies are related to the data structures, what is also related to the previously large shared-nothing cloud databases. SimpleDB (Amazon), Big-Table (Google) or PNUTS (Yahoo!) are largely proprietary developments that have been optimized for a specific application. The respective architectures have their advantages and disadvantages and must be optimized for the appropriate application. A dynamic approach (compared to Netezza) has to be asserted by future requirements.

7. References

[1] PostgreSQL 8.3.16 Documentation.