Scalable distributed data structures for database management

Karlsson, S.J.

Citation for published version (APA):
Chapter 5

Database Systems

In this chapter we introduce the reader to various types of database systems (DBMSs); we then provide examples of the need of high-performance databases. The main architecture types of databases are explained. The important issue of scalability is then identified. We discuss the implicated need for scalable data structures, scalability both from an accessing and a processing point of view as well as for updates. In most practical cases, as will be seen, parallelism or distribution is mostly used as a means for implementing high-performance. The problem still remaining is that of scalability, i.e., the ability to grow/resize the application and the database to any unforeseen size.

5.1 The Need for High Performance Databases

Databases, do we need them? During the 90s, databases management systems have been extended to interface and access external data, legacy data and unstructured data (web). Many common user applications, e.g. Microsoft products, now permit databases, such as MS SQL, to access external data sources, for example email-files, diverse database formats, spread-sheet data, file-systems, and so on. However, there is another trend, using a different approach, at present mostly in the database research community, which is to specialize the DBMS to a specific application. Examples of this can be found in the area of Engineering Databases or Scientific Databases[FJP90], where a large amount of data is handled not by the application, but a database engine. For efficiency the database should be extensible with operations from the application domain and appropriate new types of indices. Computed Aided Engineering (CAE) systems are applications of interest for merging with databases, since they require advance modeling capabilities as well as advance queries; this is explored in the FEAMOS research prototype [Ors96]. It allows matrices to be used in the query language, and
equations can be solved by stating a declarative SQL-like query. Indications show that application programs become more efficient and more flexible, they are easier to build. A popular way to implement this merge is to embed the DBMS (code) into the application as a library. This then lets the application directly traverse and use the data using a so-called fast path interface. The database system can then also be extended to use, index, and query application data. DataBlades [SM96] is the Illustra concept of a packaging a collection of data types together with access methods, and related functions and operators into a module. Most other DBMS companies now develop similar concepts, under different names, but the idea is the same, modularity extensible databases systems that can handle new types of data.

The trend, to use database for more technical purpose, also draws interest from the telecommunication industry [Dou90]. High-performance reliable DBMSs, such as Clustra [Tor95], receives an increasing attention. The reliable DBMSs allows down rates of less than a few minutes a year. There are estimates of the needed rates of insertions, updates and also queries, and the number of such events is in the range of 10 000 per second. This high performance in combination with high reliability are not possible with available standard commercial database systems. TimesTen [Tim] uses main-memory database technology to achieve high transaction rates. Currently (1999) they can perform 10s of thousands transaction per second.

New applications areas for high performance database systems are in the telecom industry directory management, charging of calls, email-databases, and multi-media repositories. Internet/Web based application servers such as search engines, e-commerce web-sites benefit from using database systems. Many of these application databases are potentially not only huge in comparison with normal databases, but also require high throughput rates. These services are more and more often service by (database) systems, instead of specialized applications. They have to be able to handle increasingly larger data sets and data retrieval loads. One problem is that many database systems are static in their nature. That is, most simple database systems are static in nature, the ability to give the same per transaction performance when the amount of data doubles and/or the number of transactions doubles cannot be achieved. They do not scale to unforeseen workload and sizes. Optimally, a system would be able to deliver the same observed performance when the demands doubles iff the system is giving the double resources.

Using current technology only distributed and parallel (database) systems can build systems that cope with high storage demands. We now give an overview of the principles of such database systems.
5.2 Conventional Databases

Single-user databases are widely available and can be used on off-the-shelf hardware, such as single workstations or PCs.

Central Databases are then the most common type of databases for multi-user environments. They run on single (mainframe) computers. Banking databases, travel agency booking or corporate billing databases are examples of central databases. Several users can access the database using either the old style terminals or using client/server software. SQL is the most widely used and standardized query language, OQL (Object Query Language) and QBE (Query By Example) are other languages which are used.

In the end of the 1990s it has become common to have distributed database systems, often because of company merges. In the panel debate at VLDB'1999 [Day99] a representative from Oracle mentioned that the integration problems were so large that and cause so much more work that they were going back to more centralized solutions themselves, reducing the number of local databases in various formats and avoiding multi-database problems.

5.3 Distributed Databases

A Distributed Database System (DDBS) can be defined as “a collection of multiple, logically interrelated databases distributed over a computer network” [ÖV91]. Further on they also define a Distributed Database Management System (DDBMS) as “the software system that permits the management of the DDBS and makes the distribution transparent to the users.” The important terms here are “logically interconnected”, “distributed over a computer network” and “transparent”. Then they give examples of what is not a DDBS:

- a networked node where the whole database resides
- a collection of files
- a multiprocessor system
- a shared-nothing multiprocessor system
- a symmetrical multiprocessor system with identical processors and memory components.
- a system where the OS is shared.

A DDBS is often heterogeneous with respect to hardware and operating systems. The data is physically stored at different sites in component databases, and the DDBMS is then the integration of these data into one
virtual "database". However, the same capabilities and software are usually part of the individual component database. The transparency is the most important feature of a DDBMS. The main difference compared to multidatabase systems, which we will discuss later, is that a distributed database distributes the data transparently over a number of nodes where each node uses the same DB software to manage its local data and where the nodes are coordinated through the DDBMS. Queries are can then be executed jointly and coordinated to provide efficient execution.

5.4 Federated Databases

A distributed database which allows for a mix of vendors for the individual DBMS products as well as different storage schemas is called a federated database. Data may be split and stored on different database systems. Using a gateway two DBMSs can access each others data.

The PEER system [ATWH94] is a federated object management system prototype that is "intended to represent and support complex data interrelation and information exchange in multi-agent industrial automation applications." [ATWH94]. Data consistency is aided by coexisting schemas at each site. A schema describes the part of information that a site has access to. Information is automatically administered to interrelate schemas and their derivations.

The Telegraph [HCS99] project proposes to focus more on storage and database management. The ideas involve a new storage manager which allows interfacing using numerous methods, such as querying through databases, normal files system access, HTTP (web) access, distributed persistent data structures for scalable internet services. They say that it should be based on a shared-nothing dataflow architecture that can balance load among the nodes. It will have an adaptive data layout system that automatically will handle fragmentation, replication, and migration of data within large clusters of disks. As Mariposa [SAP*96] it will be based on an economic federation system. Finally, the system will integrate methods of interactive visualization of query results allowing for data discovery, browsing and mining. Nice goals, however, it means integrating all the goodies from most of their previously fruitful projects and making them work together in a complex system.

5.5 Multidatabases

By contrast a Multidatabase System (MDBS) is built up from a number of autonomous DBMSs. Most of problems in the area of DDBMSs have their counterparts in multidatabase systems, too. However, the design is bottom-up: individual databases' already exist, and they have to be integrated to
form one schema. This involves translations of the different databases capabilities during query processing and data exchange. A multidatabase system has to cope with different variants of query languages, and perform all moves of data itself. It acts as a layer of software in-between the databases and the user, and the databases do not communicate with each other. Since multidatabase updates are a problem, they are usually not allowed online. Instead, they are executed locally, or in batch mode.

5.6 Data Servers

Another trend is to use the availability of powerful workstations and parallel computers for managing internal data in a DBMS. Such a computer dedicated for this purpose is called a Data Server. An example of this approach is shown in Figure 5.1, where several Terminals are connected to an Application Server that handles user input and data display, parses the query and calls upon the Data Server to execute it. The database itself is stored on a secondary storage media (disk). Data servers also seem to be becoming popular as storage sites of distributed databases[OV91].

On typical example is SAP’s 3-Tier Architecture [Mun99]. It employs one single database server for rudimentary storage. To handle large loads they moved much of the database work into application servers. The application servers have over the time been optimized using common database technology, it features an interpreter, monitor, locking, and table indexing and caching. The presentation layer uses front-ends that process actual requests from the users. Their system can use an SMP machine with more than 60 CPUs and more than 700 GB database sizes. However, in effect they do not benefit from new database features since they have retreated to using the lowest common features of the database systems. Scalability is achieved using faster machines, more main memory and more application and presentation servers.

By dedicating the computer for a data server, it is then easier to tune the memory management algorithms. Usually the database systems have more knowledge than the operating system as to how and when it uses what data. In the 1970s the idea of dividing the database management system into two parts, a host computer part and a back-end computer, appeared [CRDHW74]. Later the terms application server and data server were used respectively. Figure 5.1 shows the main idea.

5.7 Parallel Data Servers

Parallel computers are nowadays becoming more and more widespread. During the 1990s PC file servers commonly use 2 or more processors, and it is becoming affordable even for single workstations. For such hardware DDBS
Figure 5.1: Data and application servers.
5.8. DATABASE MACHINES

Technology is used in implementing parallel data servers. A parallel data server is essentially implemented on a parallel computer and makes extensive use of the advantages of the parallelism in data management that then can be gained. Often, support for distributed databases is part of the implementation. The data managed is automatically fragmented or de-clustered, making the system self-balancing. The work on parallel data servers is related to the work on Database Machines, which will be discussed in the next section. However, since special parallel hardware computers are expensive and current technology is advancing fast, the trend is to use a number of networked mainstream machines for implementing the parallel data servers. Ronström [Ron98] used a special fast interconnect network in building clusters of machines (network multicompurers). The network used was the open standard Scalable Coherent Interface, SCI [IEE92], which has received considerable attention during the 1990s.

5.8 Database Machines

Related to the work on parallel data servers is the earlier work done in the framework of Database Machines. Below we explain the term and present a short overview of some selected systems. In the next chapter we then go further into details of how large amounts of data are managed in very large systems.

The first mention of a Database Machine was in [CRDHW74], later the terms Database Computer, or Data Server have been used for a DBMS-dedicated machine. A dedicated machine has become a natural choice in a distributed environment [ÖV91]. In such a machine there is not an operating system in the ordinary sense. Hence, the DBMS has specially tailored operating system services; minimally this means just dedicated device drivers and a monitor. This is in contrast to a more typical DBMS environment on a general-purpose computer with some operating system. The reason for having a dedicated machine with more specialized software and hardware is to overcome the I/O limitations [BD83] of the von Neumann computer architecture and other restrictions. Another reason is to be able to use technology that is not yet available off-the-shelf. One way to overcome I/O limitations is to keep the whole database in stable main memory [LR85] or I/O bandwidth can be increased by using parallel I/O [Du84]. Multiprocessor computers have been studied for performance and data availability.

There are mainly two types of parallel computer architectures. The Shared-Everything type of computers provide high performance but are not scalable to any larger sizes. All the nodes share memory, disks and all other resources are typically communicated via shared buses. Examples include the Sequent Computers, Sun SPARC/Center machine, SGI Origin 2000.

It is widely known that this architecture limits the size of an efficient
system to around 32 processors. However, it is relatively easy to program. The alternative, the Shared-Nothing computer type, requires extensive programming to share any information, and to perform any kind of work jointly using the available resources. Often new algorithms have to be engineered, and much research is concerned with finding algorithms to use the power of the shared-nothing computers. The benefits are that, if one succeeds in programming the shared-nothing computer in a scalable way, the application can scale to a large number of processors.

5.9 Overview of Some Data Servers

In *Parallel Database Systems: The Future of High Performance Database Systems* [DG92] there is an overview of state of the art commercial parallel systems. Teradata is a shared-nothing parallel SQL system that shows near-linear speed-up and scale-up to a hundred processors. The system acts as a server back-end and the front-end application programs run on conventional computers. The Tandem NonStop SQL, currently (1999) owned by Compaq, system uses processor clusters running both server and application software on the same operating system and processors. The Gamma system, too, shows near linear speed-up and scale-up for queries; it runs on Intel’s iPSC/2 Hypercube with a disk connected to each node. An implementation of Oracle runs on a 64-node nCUBE shared nothing, with good price-performance measures: also it was the first system to provide 1000 transactions per second.

Examples of shared-nothing databases are Bubba [BAC+90], Teradata DBC/1012 [Cor88], Gamma [DGG+86] and the Tandem Nonstop SQL [Tan87]. Examples of shared-memory database systems are XPRS [SKP088], and the Sequent machine.

*Bubba* [BAC+90] started out in 1984. The aim was to design a scalable, high-performance and highly available database system that would cost less per performance unit than the mainframes in the 1990s. At the beginning the Bubba project was mostly concerned with parallelizing the intermediate language, FAD. FAD was used for LDL [CGK+90] compilation. The FAD language has complex objects, OIDs, set- and tuple-oriented data manipulators and control primitives. Both transient and permanent data are manipulated the same way. The FAD program was translated into the Parallel FAD language extension, in combination with the Bubba Operation System they built. In the project data placement, process and dataflow control, interconnection topology, schema design, locking, safe RAM and recovery were studied. They later regretted including all these features and functions that limited the complete study of the complex systems. In their first prototype they learned quite a few “lessons”, as they say. Parallelism, for example, gives rise to extra costs in terms of processes, messages and delays. They
found that dataflow control was another important issue. They redesigned the language and the implementation. Another identified problem was their usage of three different storage formats for objects (disk, memory and message). Their use of the C++ environment did not make things easier. So in their second system they used the C language. The second, and perhaps more realistic prototype, was rewritten from scratch. There were several reasons for this, including C++, new programmers, and serious robustness problems. Since this was not to be a commercial system, not all important features of the Bubba system were implemented. In the new system only one type of object representation was used, and it was the same for disk, memory and messages. For the Bubba Operating System, the AT&T UNIX was used, with some extensions. Their conclusions at the end of their final Bubba prototype are that "Shared nothing is a good idea (but has limitations)"; "dataflow seems better than remote procedure calls (RPCs) for a shared-nothing architecture"; "More compilations and less run-time interpretation", "Uniform object management", and that for fault tolerance it is better to replace a failing node than trying to make the nodes fault tolerant. Apart from this, they mention that there was some trouble finding a commercially-available hardware platform for their work. Even though the hardware and software were bought, there were both software (operating system) and hardware bugs, but eventually the system functioned properly.

Another system was the PRISMA/DB system [AvdBF+92], which was a parallel, memory relational DBMS. It was built from scratch using easily available hardware at the time. It was built on the POOMA shared-nothing machine. Each of the 100-nodes (68020) had 16 Mbytes of memory and they were interconnected in a configurable way. The Parallel Object Oriented Language (POOL-X) was developed, that featured processes, dynamic objects, and synchronous as well as asynchronous communication. Some of its specialities were that it could create tuple types on the fly and conditions on them could be compiled into routines. This helped to speed up scanning, selections and joins. The project was not as successful as one might expect from a main-memory system. It was not really a magnitude better than disk-based DBMSs. The problems included the facts that the hardware did not run in full speed, that the hardware was outdated when the project was evaluated, and that the compiler of the experimental programming language was not fully optimized. But among the positive results they found that by using their language they managed to build a fully functional DBMS from scratch, and the project could then be finished on time.

The XPRS (eXtended Postgres on Raid and Sprite) DBMS [SKPO88] was aimed at high availability and high performance for complex ad-hoc queries in applications with large objects. It was optimized for either a single CPU system or a shared-memory multiprocessor system. The aim was to show that a general purpose operating system can also provide high transaction rates and that custom low-level operating systems are not a necessity.
They were much concerned with removing hot spots in data accesses. This was done by reducing the time the locks are being held using a new locking schema, and by running DBMS commands in a transaction in parallel. A fast path schema was proposed to achieve high performance, as opposed to the common method of stripping out high-level functionality (such as query optimizations and views) from the DBMS. For better performance when I/O-ing large objects they built a two-dimensional file system. This achieved reduction of the mean time to failure (MTTF) then cured by using RAID [PGK88] or striping techniques that provides fault tolerance. These techniques keep a bit parity block for $N$ disk blocks (on different disks). This block can be used to reconstruct any of the $N + 1$ blocks from the $N$ other blocks. Thereby the overhead is reduced to $1/N$.

DBS3 [BCV91] used the assumption that the success of RISCs lies in simplicity and high performance compilers, that large main memories will be available, and that one should rely on advanced OSs. The first implies that a good optimizer and simple basic DB units are more important than having a complex design and a complex language to program it. Furthermore the whole database can be entirely stored (cached) in main memory. This simplifies optimization and cache management. Lastly, portability is now important and most newer OS include better means for memory management (cache tuning, virtual memory, mapped I/O) and transactions (threads).

Permanent data is stored apart from temporary data. This two-level storage divides data so that permanent data is stored on disk and temporary data is managed in (virtual) main memory. It can make use of fragmented relations, both temporal and permanent. Zero or more indices can be used for each fragment. The transaction processing is aimed at online transactions and decision-support queries. Using their parallel execution model, they finally achieved good intra-query parallelism, using pipelining and declustering.

As part of the Sequoia 2000 Project at the University of California, the Mariposa project [SAP+96] proposes a micro-economic model for Wide Area Distributed Database Systems. Their system uses terms from the market economy: sellers, bidders, brokers, budget, purchase, advertisements, yellow pages, coupons, bulk contracts and the term “greedy”. The idea is to set up an economy and means for trade and then let the “invisible hand” guide the actual trading of resources.

Worth mentioning are also the commercial systems that includes DB2, and Microsoft SQL Server.

5.10 DB History

In the beginning of the 1980s designing special hardware was very popular; nowadays one tries to use ordinary high-performance workstations. New extensions in the area of memory management are beginning to emerge in the
operating systems (UNIX, Windows NT) that will allow more and better control over the system's resources. And instead of parallel computers the networked parallel computer is becoming more widespread. Special interconnect, such as SCI [IEE92], set standard in the 90s for high-performance network computers.

During the end of the 1990s “hot areas” for database systems include the usage of database systems for storing and generating web-pages, such as search engines. These systems more than ever requires scaling for growing amounts of data and user queries. Especially interesting is database systems that keep (most) data in main memory a trend allowing for much faster and higher throughput than disk based database systems. Research prototypes existed for many years, these have recently started to be commercialized by for example TimesTen [Tim] database system.

5.11 Conclusions

The need for “big” database servers will always be here; distributed cooperative solutions arise, but neither the less local solutions will dominate.

My impressions of the experience gained from the projects mentioned above can be concluded as follows:

- In a shared-nothing architecture dataflow is considered a better paradigm than RPC for query processing.
- The system should be self-managing and self-balancing.
- Fault tolerance is provided by replacement of a faulty node rather than making a node fault tolerant. Replications allows fast switch-over.
- Do not build your own hardware, it will become outdated before the project is finished.
- New hardware and operating systems are error-prone; move to a stable platform.
- Do not write your own experimental implementation language.
- Do not write a compiler, you will not be able to optimize it completely.
- Shared-memory is easier\(^1\) to program than shared nothing; it does, however, not scale to more than a certain number of nodes, currently

---

\(^1\)My honored Licenciate opponent Øystein Torbjørnsen, had a bit different view on the matters. He meant that the only way to master/simplify/make efficient implementations on a shared memory machine is to program it as a shared-nothing machine. Erlang, is a programming language that promotes the same idea, by not apparently sharing data. The system can however of for efficiency share data in its implementation.
substantially less than thousands. Newer parallel computers are likely to have shared-memory (everything) processor nodes connected into a shared-nothing multi-computer.

- Use the same format for all storage of the same data: on disk, in memory, in buffers, in messages.
- Use fast-path access to data instead of stripping high-level functionalities from the DBMS.
- Parallel I/O-systems give high-performance.
- Do not implement all features and functions in all possible variants.

5.12 Properties of Data Structures for Parallel Data Servers

In this section we will give an introduction to required properties of data structures that are used in distributed data applications, such as Parallel/Distributed Data Servers.

5.12.1 The Problem

Modern systems manage high volumes of data, and if they implement data access paths (indices) at all, they are often hard-coded with the application's data. Data is indexed through some key identifying the data; this can efficiently be implemented by using hashing algorithms or some tree structure that keeps the data sorted. It is now well-known that most systems mainly use variants of Linear Hashing [Lit80] or B(+)-Trees [BM72] for their access paths. For example, DB2 only supports the B-tree style index. IBM is reluctant to add other indices into the DB core of DB2, instead new indices are implemented by mapping their structure down to B-trees [?]. Other examples of indices include R-trees (spatial), AVL-trees (main memory sorted index) and SpiralStorage.

5.12.2 Scalability

In DBMSs the need for scalable data structures is more obvious than for specialized programs. Whatever arbitrary upper limit is set on the amount of data a data structure or a DBMS can handle, it will probably be exceeded at some future time. A scalable data structure can be characterized by the following:

- Insert and retrieval time is independent of the number of stored elements (i.e., it is more or less constant).
5.12. PROPERTIES OF STRUCTURES FOR SERVERS

- It can handle any amount of data, there is no theoretical upper limit that degrades the performance.
- Furthermore, it is desirable that it grows and shrinks gracefully, not having to reorganize itself totally (as some hashing structures do total rehashing of all stored elements), but rather incrementally reorganizes itself during normal processing.

Linear Hashing [Lit80] is an example of a scalable data structure, which is an algorithm for managing random access data that can dynamically grow or shrink in size. It is based on ordinary hashing schemes and has therefore the advantage of direct access, but not the limitations of a fixed array of buckets. The array is allowed to grow when the data structure reaches a certain saturation limit, or shrink when it decreases below some limit. This is achieved through splitting and merging of individual buckets. Other variants include Spiral Storage and Extensible Hashing. The access cost for hashing structures is approximately constant.

B-trees [BM72] are another example. This algorithm maintains a set of ordered data. The data is stored in leaf nodes that are allowed to store a minimum and a maximum number of elements. When the number of elements exceeds the maximum or is below the minimum, the leaf is either split or merged with other leaves. The index is maintained in the tree nodes using a similar principle, providing in the end not linear access cost but logarithmic. Variants of scalable ordered data structures include AVL-trees, 2-3-trees, and others.

5.12.3 Distribution

Sometimes the amount of data is larger than what can efficiently be managed or used by a single workstation. Even if this amount of data could be connected physically to one workstation, the processing capabilities of the workstation would not be enough for searching and processing the data. Then, instead, one can employ a distributed data structure that distributes the data over a number of nodes, i.e. workstations. Such a data structure can then be used to keep very large amounts of data online. One way to do this is to apply a hash function on the keys that partitions the data into fragments. The simplest idea is that each fragment stores the data of one bucket, one or several fragments are then stored on each node. Some other schemas require a central directory that is visited before each retrieval or insertion of a data item to get the address of the node storing it. This solves the problem of finding where the data resides if some of it has been moved (because of reorganization). However, the directory can easily become a hot spot when many clients are accessing it. Solutions using hierarchies of distributed directories can then be used, they can then cache results of earlier requests to improve performance. This is a schema similar to the well-known internet
DNS service [Ste94]. Another, simpler distribution strategy is to store one field of a record for all records in a file entirely in one node and other fields on other nodes. However, if the amount of data grows fast, this is no scalable solution. We notice that these distributed data structures will also have to be scalable over any number of storage sites. This is a relatively new concept SDDS Scalable Distributed Data Structures.

### 5.12.4 Availability

Sometimes distribution is used in combination with some redundancy to achieve high availability. High availability is necessary, for example in banking and telecom [Tor95] applications, but also in other areas with online transactions, or where the information is of such importance that the extra down time of reading backups cannot be allowed. Using a high availability schema, disk crashes as well as some other sources of read or write errors can then be recovered from. The classical variant here is RAIDs, Redundant Array Independent Disks [PGK88] where a number of disks are connected to one or several computers. One of the disks is used for storing a parity page; this page is calculated by xor-ing a disk-page from each disk and storing the result on the parity disk. Each time a write is being performed, this page is updated. If one of the disks fails or a page on one of the disks fails, it can be recovered (reconstructed) from the other disks. Using more disks for parity can ensure recovery from more “errors” and thus gives higher availability.

### 5.12.5 Conclusions

New applications of databases put requirements on the memory management. Among these requirements we identified that high performance requires scalability, distribution and high availability. Data structure access has to hide the implementation of these above aspects and make the access transparent. In this chapter we discussed these three (orthogonal) aspects, but not the important joint usage of them. Scalable solutions will by necessity be distributed and will require high availability and new data structures, such as the SDDSs, are important for achieving this.