9 Database Scalability

What the reader will learn:

- that the number of concurrent users is one important aspect of scalability
- and that volumes of data stored and accessed is also important, particularly in the era of Big Data
- that scalability brings its own issues for DBAs, such as the cost and performance implications
- that cloud computing's "use when needed" model can be helpful in handling scaling issues
- that there are many approaches to tackling scalability issues and that each has strengths and weaknesses

9.1 What Do We Mean by Scalability?

As with most chapters in this book we will start by defining what we mean by scalability. And, as with some other chapters there are several possible answers.

Dictionary.com defines it as:

> the ability of something, especially a computer system, to adapt to increased demands.

But even this is debatable. In its literal sense scale can go up (increased demands), but it can go down.

It is far from only computing that uses the term. Here is a definition from a paper in the healthcare area:

> The ability of a health intervention shown to be efficacious on a small scale and or under controlled conditions to be expanded under real world conditions to reach a greater proportion of the eligible population.

But this is a database book, so let us narrow this down to what it means for database professionals. Even here, however, there are possible areas of confusion.

Hoskins and Frank (2002) define scalability in terms of processing power:

"the ability to retain performance levels when adding additional processors"

In this MSDN Dunmall and Clarke (2003) article about load testing, scalability is described in terms of:

"… the number of concurrent users anticipated at peak load in production"

In the era of Big Data the focus is often more to do with the volumes of data that an organisation needs to store. This is 10-Gen's take on scalability from their MongoDB website:

Auto-sharding allows MongoDB to scale from single server deployments to large, complex multi-data center architectures.

In actual fact these three elements are often closely related: concurrent user count; processing loads; data volume. It is often the case that more data or users will cause more processing, for example. So, for this book, we will describe scalability as:

The ability of a database system to continue to function well when more users or data are added.

We will split this chapter into two main parts as we first examine strategies for coping with increases in the number of users of a system, and then look at the alternative approaches to handling the growth of data being stored and queried.

9.2 Coping with Growing Numbers of Users

As with many Microsoft products, ease of use is a real plus-point for Access. Some relational database concepts are difficult for non-technical users, and yet people are able to use the Wizards to help them create complex databases with apparent ease. And, compared to licenses for products like SQL Server and Oracle, Access is cheap.

So why doesn't everyone just use Access? Well we can get into a war between vendor's claims and counter-claims if we are not careful, but the more general point in a chapter about scalability is that even Microsoft only claim 255 as the maximum number of users it can cope with. Anecdotal evidence, and the author's experience, leads to the belief that considerably fewer than 255 may be the sensible maximum number of concurrent users.

If you are storing information about your immediate family's addresses, or you are creating an application for the administrators of a small company, then this limit may not matter. However, if you are hoping to be an Amazon, with thousands of concurrent users, Access will clearly not do the job! According to Complete.com (http://www.compete.com/us/) Amazon.com had 129 million users in the month before this chapter was written. Even if every user logged on at different times of day and was logged on for less than a minute, this is roughly 3000 concurrent users at any point in the day. Naturally their systems designers will have to build in resources to cope with peak times, so this figure will actually be much, much greater than 3000.

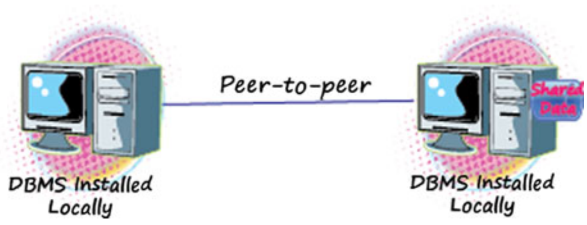


Fig. 9.1 Peer-to-peer database system

9.2.1 So Why Can't Access Cope with Lots of Users?

Well, because it was never designed to be truly multi-user. Access is a file-server solution. It relies heavily on the operating system of a single file-server (usually a PC) on which it runs and stores its data. Databases like SQLServer and Oracle are Client-Server databases. Architecturally they were designed from the start to be capable of allowing many users to access the system. Client-server solutions can manage their own processes and data placement, including the use of multiple servers to share the load. This is not to say that Access's architecture is second best—it is just different; designed to meet different needs.

Typically the file-server database is designed with a single user in mind. Many Access users will use the PC they sit at to install and run Access. In that unconnected mode, multiple users are never going to be an issue.

Before Access became the success it currently is there was an trend for computers to become connected to one another in a peer-to-peer way. This meant that the Access-like databases on the PC mentioned above could now be available to other PCs in a workgroup. But what actually happens is that Access is installed on all the PCs that are permitted to use the database, and a copy of the data is taken to work on the second PC. This scenario, and the stand-alone PC are both referred to as one-tier architecture (Fig. 9.1).

Allowing multiple users to read data is a relatively trivial task for a database. However, one of the most significant issues for multi-user databases is for the management system to control the write processing that is needed when users want to insert, delete or update data. And one of the control mechanisms used is that of locking rows, or tables—preventing other users from having access to a row whilst it is being changed. We cover locking in more detail in Chap. 3. Locks are often a problem because, to the users who do not own the lock, this just looks like they are having to wait for the data. They tend to see it as a system performance problem.

As with any system that relies exclusively on the resources of one PC, things can go wrong. Information about Access locks is stored in a.ldb file. These files are cleaned up as the system looses users, but occasionally they are left behind and need to be cleared up by the systems administrator.

These problems may not be an issue for smaller systems with few users. However, the more users there are, the more susceptible the system becomes to things going wrong. And this means we can say that Access does not scale well.

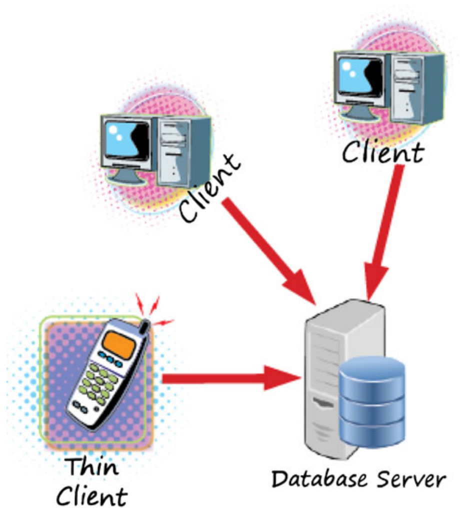


Fig. 9.2 A client/server database system

9.2.2 Client/Server

Largely in response to the scalability problems associated with the peer-to-peer approach, computer science began to look to a different model. What became apparent was that connecting computers together and allowing them to share processing in some way was a good use of resource. It was also clear, particular at a time when disk, cpu and ram were very expensive, that having some sort of specialisation for particular machines was the best way of maximizing the usage of these resources.

So evolved powerful PC-like computers that became known as Servers. They "served" services to PCs that connected and requested such services. The latter became known as the Client of the Server. The model of high performing machine serving one or many client machines the became known as Client/Server (or Client-Server) (Fig. 9.2).

There are all sorts of variations on this theme. The World Wide Web is a form of client/server, for example. Even mainframes, it could be argued, are Client/Server systems with central processes carrying out the work and then returning the results to dumb terminals.

One of the decisions for designers of client/server systems is where to do the processing. In the mainframe example all the processing is carried out at the server end. What made the client/server revolution different to mainframes was that the clients tended to be relatively intelligent PCs. In database terms, the decision then becomes—do you use the server to serve-up the data and then do the manipulation (in the form of sorting, filtering, etc) at the client end? Or do you do everything at the server end and just send the finished dataset, but use the PC's graphical abilities to present it to the user?

Another pair of oft-used terms here are "*thin client*" or "*fat client*". The former can almost be thought of as a more graphically pleasing version the dumb terminal. All the processing is carried out at the server end. At the opposite end we have all the processing carried out by high performance client PCs, and the server merely "serving" requests for data.

There are advantages and disadvantages to both the fat and thin approaches. Perhaps the biggest reason for the client to be a fat client PC is that of user expectation. Users now expect to be able to have some control over their own computing. In addition thick clients allow sometimes lengthy data transfer to happen in the background whilst the user works on other tasks. Simple tasks like screen refreshes and paging through data can be processed at the PC, whereas, with a thin client, network traffic would be increased significantly as the keystrokes and results pass back and forth.

Amongst the advantages of thin client, perhaps the two most significant are those of cost savings and greenness. Without the need for powerful computing power, thin clients can be better than half the price of a PC to purchase, and can use five- or six-times more electricity in use.

In addition thin client supporters claim it to be easier be easier to manage (software upgrades are easy since they just apply to one machine, for example) and more secure (since the applications are entirely housed in the data center, where strict rules and policies can be applied).

Whichever approach is taken it was clear that the client/server model allowed for much better scaling than did peer-to-peer. Handling more users could be as simple as adding more RAM or upgrading the CPU on the server.

However, as pressure rose for more and more users to be able to access an organisation's data, the single server solution (known as two-tier) showed weaknesses and the next step was to split server-side processing over two servers. Typically this would be a Database Server and an Application Server. This became known as Three-tier architecture. Many organisations now use this approach. Extra scalability can come in this environment by having users connect to the application server which then manages connections to the database server, pooling connections rather than requiring the server to have a process for every user. The most recent move has been to n-tier architecture, in which many servers each cope with aspects of servicing the clients (see Fig. 9.4).

The advent of Cloud computing has now brought us the ability to scale either way (up and down) very easily, and this is seen by many as one of the great benefits of Cloud. If you had to buy and maintain a client/server system adequate to support an estimated 200 concurrent users, and then you discovered that the estimate was wildly wrong and you needed only to cope with 20 concurrent users, you would be grossly over resourced (and over spent!). Having bought hardware it is virtually impossible to downsize. However, Cloud just needs you to change your contract with your service provider. And of course virtually unlimited and very rapid upward scaling is also available on the Cloud.

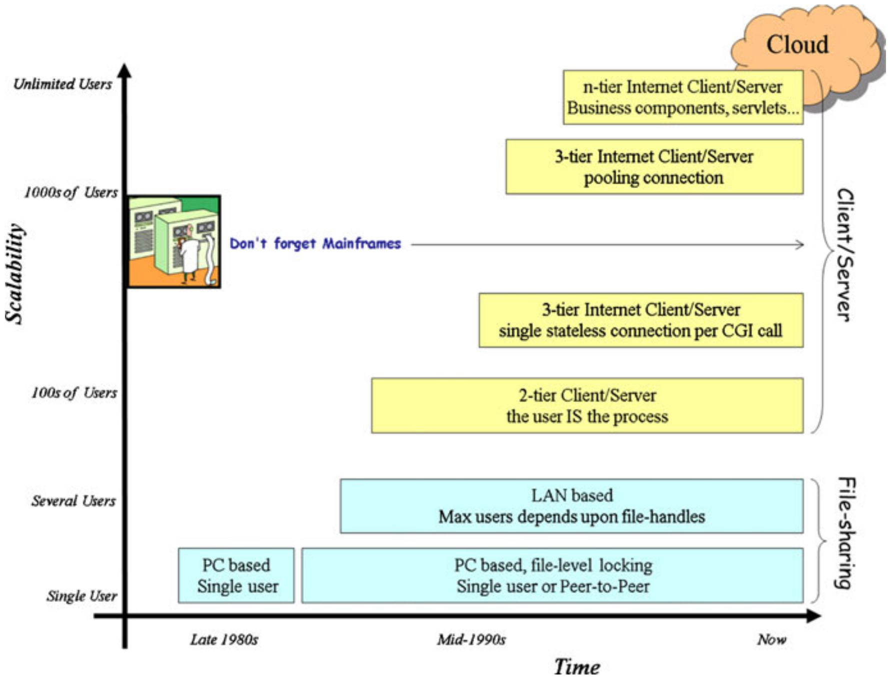

Cloud
n-tier Internet Client/Server
Business components, servlets...
3-tier Internet Client/Server
pooling connection
Don't forget Mainframes
3-tier Internet Client/Server
single stateless connection per CGI call
2-tier Client/Server
the user IS the process
LAN based
Max users depends upon file-handles
PC based
Single user
PC based, file-level locking
Single user or Peer-to-Peer
Client/Server
File-sharing
Unlimited Users
1000s of Users
100s of Users
Several Users
Single User
Scalability
Late 1980s
Mid-1990s
Now
Time

Fig. 9.3 Scalability eras

9.2.3 Scalability Eras

The different approaches to the scalability issue outline above have all taken place since the PC became a viable business tool and each approach has evolved. A summary of this evolution in shown in Fig. 9.3.

Some of the aspects discussed above also impinge heavily upon the networking domain. Having a server do all of the processing and just sending the final dataset to the client means that network traffic is minimised. But the fat client solution may mean a need for network upgrades when more users begin to use the system. As always, database and networking professionals should ideally be liaising to provide the organisation with an optimal solution.

9.2.3.1 Separating Users from Sessions

In the traditional client/server database system the user connects to the database and the database will then manage a bit of server RAM and some server-based processes on behalf of that user, called a database "session".

Managing sessions is an important and resource heavy task for the RDBMS. Idle sessions need to be continually checked and killed after a certain time since they are using up valuable RAM without doing anything. The activity of every session is recorded in great detail. This information is useful for performance tuning activity, allowing the DBA to ascertain which sessions used most resources. However, its primary purpose is to act to allow audits of activity. But why this activity happens need not concern us here—what we need to know is that the more sessions we have the less well the database will perform.

However, especially in a e-commerce type environment it may well be that users who are logged on to the server for as long as an hour only actually make calls to the database that take less than one minute of database process time. Consider a simplified version of the shopper who asks the site to show him all the gentlemen's trousers available:

1. The SQL is formed which says something like Select $*$ from catalog where item_type = Trouser; The SQL is sent to the server which returns the dataset asked for.
2. The shopping basket may or may not be stored on the server
3. The user then browses the dataset for 50 minutes, decided to make a cup of tea half way through the process.
4. The user completes the purchasing process.

If we were to kill this session for inactivity whilst the user made their tea we would loose our customer—not a sensible approach to take! On the other hand, tying up resources, especially when multiplied thousands of time over with the potential number of concurrent users accessing the site, could be very costly in terms of having RAM and CPU in the server "just in case" all the users need to access the database at the same time.

The solution is to have an extra layer between the client and the database server that manages the connection process. This extra layer, known as the Application Server, will own the database connection and will manage requests for data access on behalf of clients. This is often called Connection Pooling. In effect, the problem with the fifty minutes of inactivity identified above is lost since the application server will simply service other requests from other clients with the open session.

The application server can do much more than manage sessions. It can, for example, carry out the query processing, such as sorting and filtering, removing processing load from both database server and client.

9.2.3.2 Adding Tiers

As we saw above, there is a trend towards multiple tier server solutions. In our example above, we are divorcing database server from the need to worry about what happens to the data once it is served to the requesting agent. This means that physical aspects of the server (RAM and CPU) can be optimised for data-intensive processing, whilst the application server can also be optimised.

Three tier architecture is now the norm in enterprise scale architecture. The three tiers are identified as:

1. The Data tier: responsible for writing to and reading from the data store or file system
2. Business Logic: responsible for liaising between the other tiers, including sharing connections. But this layer can also contain rules, business logic and other processes that can work with any extracted data before passing it on to the client

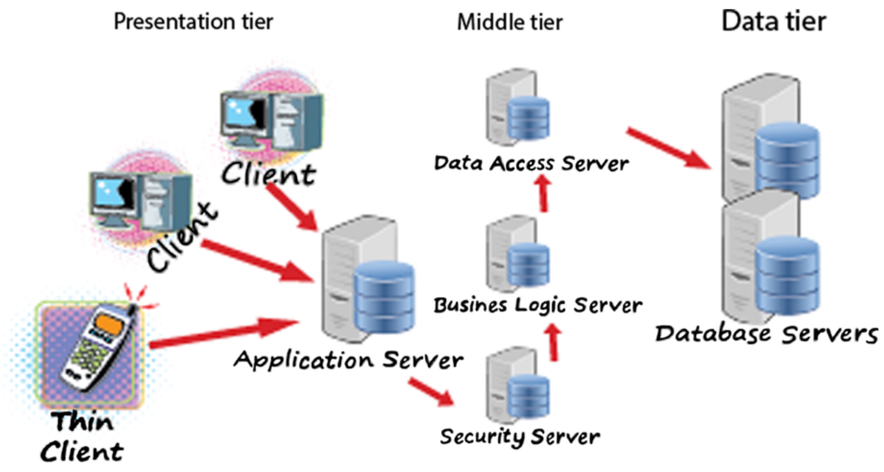


Fig. 9.4 Multi-tier systems

3. Presentation: responsible for turning the information from the second tier into something the user can read.

If the clients are thin clients then the middle tier tends to have to carry out much more work and that tier itself needs to be broken down into different layers, each sitting on a different server. A diagram of such an n-tier architecture is shown in Fig. 9.4.

9.3 Coping with Growing Volumes of Data

As we saw in Chap. 6 we are now working in an era where volumes of data are increasing at an enormous speed. Storing that data load as it grows, in a way that allows it to be accessed efficiently, is what scalability means in terms of data.

One of the reasons to use normalisation was to reduce the amount of data being stored. And one of the key reasons for that was that storage media were very expensive until quite recently. They were also very unwieldy (see Fig. 9.5). The continuing reduction in relative cost and enhanced functionality of computing technologies are key in driving new expectations and requirements. The fact that we can now store 1 terabyte of data for around $100 (under £70) means that the business that wants to store that amount of data can easily do so. Only twenty years ago the cost would have been hundreds of times higher and would therefore have been far more likely to prevent the storage from happening.

The physical performance of data stored on disk, too, has increased significantly. In terms of data transfer rates, today's 6 gigabits per second SATA drives are 4× faster than when SATA drives started to become available in 2003. There are some

Fig. 9.5 What HDD used to look like! Courtesy of International Business Machines Corporation, ©International Business Machines Corporation

who argue that disk speeds are beginning reach the best they can possibly be as a result of the constraints set by the physical elements that make up the HDD (spinning platters on a spindle and a moving arm).

SSD technology on the other hand has no moving parts and can therefore be much faster. This may well be the way that data storage moves in the future, but, though the cost/Gb of SSD is dropping, it may be a few years before the cost and reliability matches HDD technology.

9.3.1 E-Commerce and Cloud Applications Need Scalable Solutions

The rise of companies whose core business is selling to consumers by using the internet has brought a whole new requirement in terms of scalability. When starting new businesses in this area the entrepreneur does not know how many customers they will have using its site. They can only guess. And guesses can be considerably out (either way!).

But the hope is usually that customers will come flocking in to your new e-commerce site. And once you have your reputation built, even more will come, month on month increasing the number of hits your site receives.

But the worry is that your website is only as good as its ability to cope with peak traffic. As the hits increase you need to be able to be sure that your website, and its underlying database will be able to scale upwards. Failure to scale will result in disgruntled customers who could even ruin the business if it gets that bad!

If your solution is cloud-based then you can afford to worry a little less since you can quickly buy your way out of scaling problems by renting more CPU and disk from your provider.

9.3.2 Vertical and Horizontal Scaling

These two terms have come to the fore recently to describe two completely different approaches to allowing database applications to scale. Vertical is also referred to as Scaling UP and Horizontal as Scaling OUT.

Unfortunately these terms are amongst many in computing that suffer from definition creep. For ease we will here suggest that Vertical Scaling is the act of adding resource, such as CPU or RAM to a database server, and/or using database architectural techniques to assist in allow more data to be added to the system without degrading performance. Horizontal is the act of connecting several cheap (usually referred to as "commodity servers") computers to share out the storage and processing load.

Both of these scaling techniques have their strengths and weaknesses. There can be little doubt that the complexity involved in managing multiple nodes in an Hadoop cluster might be enough to put even technical experts off from using that approach, especially if their expertise lies in the relational arena. But if the business need is to store and analyse masses of data that can be spread across multiple nodes, it may be that the complexity is worth it. Google, for example, would not have gone to the trouble of creating their own file system if they had thought scaling up an standard RDBMS could have coped.

9.3.3 Database Scaling Issues

So why are there problems in adding more and more data to a database? Surely a table can simply keep growing? Well, yes it can keep growing—provided the underpinning disk system can keep expanding. There are physical limits to the amount of storage that can be addressed, but the main limit is still likely to be cost. However, even if you have a disk array that will house all of your data, other problems occur as a consequence of constantly adding data.

The most significant problem is the time it takes to access the data in response to SQL queries. There is a pre-retrieval overhead for all queries that will not get slower because there is more data, but the actual retrieval phase could take twice as long if you are reading twice as many rows.

One of the most frequent processes carried out by an SQL query execution plan is a full table scan. In Oracle, for example, if the optimiser calculates that your query will return more than around 15 % of the total rows, the execution plan will not include the use of indexes (even if they exist), but will read every row in the table, discarding unwanted ones in memory.

This is quicker because the use of indexes will frequently lead to several read operations per row fetched, and each read may require the disk read head to have to jump to different physical addresses. A full table scan only requires one move—to the address of the start of the table—and then to keep reading until it gets to the end of the table (identified by a High Water Mark). It can also use much larger I/O calls, biting more off with each read. Making fewer large I/O calls is less costly than making many small calls.

Any query that uses a full table scan, then, will take longer the more rows you add to the table concerned. Queries that use index retrieval will also probably take longer as rows are added for a couple of reasons. Firstly, more rows means more leaf blocks to the index and possibly more branches (each requiring a read) to get to those nodes. Secondly, having retrieved the rowid (which contains the hex-address of the row on the disk) it is likely that the disk head will need to move further to access the data concerned.

Bigger datasets also probably mean more data maintenance—things like updates to, or deletions of, rows. Both updates and deletions have effects on any indexes concerned too. Inserts to tables with Foreign Key constraints will take longer if the referenced table is growing significantly.

All this extra work as a result of the volume of data stored will slow the system down. The need to protect against database failure means that Redo Logs activity may well increase significantly with increased usage and may result in the need for more disk space to be devoted to Redo. More data changes also means increased Undo activity as the need to be able to Rollback and provide read consistency expands. Again this means more disk space and slower access to the contents as it increases in size.

If you have a Disaster Recover policy that relies on a mirrored server in another physical location you will have double the issues of disk space to worry about, as both servers need to be in step. Moreover the network traffic will increase as more information needs to be passed to the mirror.

Back-up will take longer as you have more data. There may be only slight increases in any incremental backing up carried out, but full, cold back ups will take longer. More worrying, the fact that there is more data involved means that the time to recovery will be much slower should you need to use those back ups.

All of these are just natural bi-products of a growth in data. Just as it is true that it takes longer to read War and Peace than Of Mice and Men, so it is true that reading lots of data takes longer than reading small amounts.

Even if it the data involved made it appropriate, the database architect may well not be able to make sweeping changes to the infrastructure in response to problems caused by growing data. If your organisation has invested heavily in large, powerful servers running Oracle or SQLServer, it would be a very brave DBA indeed that suggested throwing that all on the waste heap and replacing it with many commodity servers running Hadoop. As always, the constraints on the DBA will not always be technical ones.

For this reason, in the next few sections we will examine possible scalability-related solutions for different environments, recognising as we do, that the current environment is often the given that has to be worked around.

9.3.4 Single Server Solutions

In this section we consider the example of a single server running a Oracle or SQLServer database which is growing and which, as a consequence, beginning to suffer from poor performance. It is taken for granted that the extra data is being handled by extending the storage disk-farm.

It is true that there may be an immediate cures in terms of simply upgrading the CPU or providing more RAM, but they can be costly, and are not always the right solution. There is also the possibility that improvements can be found by changing the underlying operating system, though this latter may have far too many organisation-wide repercussions to be considered realistic.

Extra RAM—lots of it—might seem like the obvious solution to our problem. After all we will discover in the performance chapter later, reading a row from a HDD is far slower than reading from memory. So more RAM means less HDD reading, which means we can scale our database up and ensure our users do not notice any change in performance.

There is some truth in this. As we saw in the In-Memory chapter, there are examples (Oracle Times-Ten, VoltDB) of databases running entirely in memory. However, just having picked at random a cheap 1 Tb HDD and 1 Gb RAM on Amazon, the cost/Gb difference is very significant; it is around 300 times more expensive/Gb for RAM than it is for HDD. Except for some very time critical applications it is probably that we will need to continue working with HDD technology as the means of making our data permanent.

The other important factor is just what you are doing with the data. If this is a very volatile many user OLTP system, it may well be the case that much of the data that is cached into memory as a result of one users SQL statement will simply time-out and disappear as it isn't needed again before the RAM is needed for other data. In this circumstance performance will not improve at all by adding RAM. As usual we need to understand what the application needs from the database before we make our decisions about how to tackle scalability.

Memory management is a very important part of any RDBMS system. Newer databases will often manage the available RAM automatically, but if the RDBMS is not the only software running on the server, some manual intervention and decision making will need to be carried out.

What we do know is that there will come a time when the benefit accrued from adding RAM, or CPUs, will not seem worth the expense. So are there other, software related, ways of improving performance when our data increases?

9.3.4.1 Partitioning

Partitioning data, also known as vertical scaling, is one approach to extending data storage within the existing schema. At its core the concept is relatively simple: if you are struggling to access large volumes of data, simply break the storage object (usually a table) into smaller objects.

There are several ways of doing this. One is to take a table and attempt to break it into two or more tables, with data that is regularly accessed in one table, and the other tables holding the less active columns. It is sometimes referred to as row-splitting.

The original, unsplit table can be recreated with a view which joins the tables this split. Care needs to be taken however since this activity obviously increases (resource intensive) join activity so, as usual, it will depend upon what activity your database is supporting as to whether the split is worthwhile.

Another way of subdividing the data is by breaking the table into logical portions based upon a key value. For example, in an OLTP system you could partition the data such that data for the current month is stored on one tablespace, whilst data for previous months which can't be updated, is stored on faster to access read-only disks.

As partitioning requires the DBA to identify criteria for breaking up the structures it is not useful in situations where the data distribution changes often. Predictability and some regularity in the distribution of data between the new structures makes partitioning more likely to succeed.

9.3.5 Distributed RDBMS: Shared Nothing vs. Shared Disk

One of the problems with scaling up data storage on a traditional RDBMS sitting on a single server is that the server itself is expensive.

RAID technologies (see Chap. 3) began the process of utilising cheap technology (in that case HDD technology) in enterprise systems that had hitherto feared that reliability would suffer if cheaper hardware was used.

In 2006 Oracle released Version 10g of its database. The "g" stands for grid. Grids are just pools of computers (cheap, off-the-shelf) that can share the processing required to run the database between themselves. Scalability is achieved by permitting extra computers to be added to (or removed from) the grid. Naturally, in order for this to work, there has to be some sort of Grid Control mechanism which manages the flow of tasks and data.

Whilst the term "grid" was the cause of some disagreement in the industry because grid computing was a term already in use by Computer science, the concept is still seen as a sensible approach to the problem of scalability and cost-effective performance.

In the Oracle grid solution, whilst additional nodes can be added to the grid, what they actually provide is extra only places for processing to take place. The data is not owned by any individual node. Rather it accesses a single shared database. This may, in actual fact, reside on many different physical disks in a SAN, but, as far as

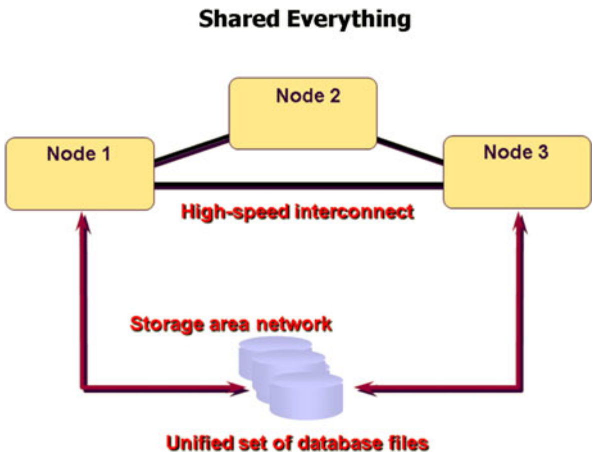


Fig. 9.6 Shared Everything (or Shared Disk)

the RDBMS is concerned, it is treated as a single database which is attached to by the nodes when they need data. All the nodes share the database, hence one of the names for this approach: Shared Disk (see Fig. 9.6).

Confusingly, different manufacturers use different terms to describe concepts. The shared disk approach is also referred to as Clustering. Oracle calls this approach Real Application Clustering (RAC). Its real strength is in enhancing availability since, should one of the nodes fail, the other nodes are available for users to attach to, and thence gain access to the shared data.

Query performance can also improved since each node can, if appropriate, use its computing resources to return parts of an overall query at the same time; in effect allowing for parallel processing.

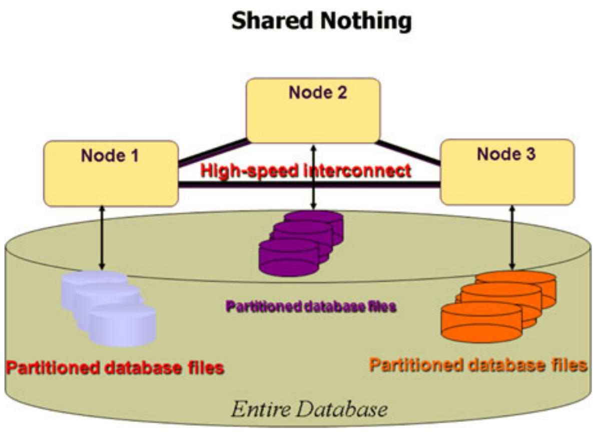
Unfortunately, although there are some performance advantages with this approach the fact that each node is accessing the same data means that there will be many more locks in play than on a single server, and those locks need synchronising across all the node. It can be argued that this makes this approach less than truly scalable as a result of this.

Shared disk is not the only approach to distribution. A shared nothing approach means that each node also looks after its own data and it has its own memory and disk resource. Examples of this approach are IBM's DB2 and Teradata.

Here the whole database is actually a logical construct made up of the individual databases on each node. This can be its performance weakness since joins across multiple nodes can be very slow. Moreover, if a node fails you loose far more than you do in the shared disk approach—you loose part of your whole database, so availability is an issue which needs to be thought carefully about. However, it is very much easier to scale. You just add another commodity server when you need more resource. This approach is popular for e-commerce sites for which the degree of scaling is unsure, or certain to be high.

There are, as usual, pluses and misuses to each approach. The business needs will need to be the driver in the decision making.

Fig. 9.7 Shared Nothing


Shared Nothing
Node 2
Node 1
Node 3
High-speed interconnect
Partitioned database files
Partitioned database files
Partitioned database files
Entire Database

We are now moving into the grey area between shared nothing and sharding. Indeed, it could be argued that Sharding is the same as Shared Nothing is the same as horizontal scaling (see Fig. 9.7).

9.3.6 Horizontal Scaling Solutions

As we have seen, there are various names for the basic idea of splitting your data across two or more databases (often referred to as nodes) such that all nodes have a subset of the data. This works well in cases when a query only needs to connect to one node. It can be more cumbersome if many nodes need to be accessed to return data from a query. This is sometimes referred to as "scaling out" as opposed to the traditional approach of "scaling up".

As the data stored for web usage expanded dramatically early in 2000s Google realised that the traditional RDBMS solutions would not cope with work load being placed on them by the various applications like Google Search, Web-indexing, Google Earth, and others. They set about inventing their own file system (GFS) and sat their own database (BigTable) on this platform. The need to scale easily was paramount. They achieved this in a cost-effective way by using cheap, commodity servers which can be easily slotted in and out of the data centre infrastructure.

Google started a trend. GFS and BigTable are proprietary products, but very similar tools began to appear in the open source environment. At the time of writing the industry standard horizontal scaling tool seems to be becoming Apache Hadoop. Other NoSQL databases (see Chap. 5) also adopt this approach. The examples we looked at earlier, Cassandra and MongoDB both shard their data to allow scaling. But it is not just the open source arena that is turning to sharding. Microsoft's Azure allows sharding too.

One of the first problems when designing sharded systems is in identifying a key on which to split the data. Let review a simple example:

An application starts by looking up information about a client based upon their surname.

You have two servers available. Perhaps it may be sensible to have two shards, one with all the names beginning with A to L, and the other M to Z.

And whilst the only query asked of the data is about one individual, then this split will reduce the searching that needs to be carried out. But if you were to want to discover all clients who lived in a particular town, this query may well result in joins needing to be carried out and the management of that process may take more time than if all the data were in one single table. It is clear that selection of the sharding key is key to the success, or otherwise of this approach.

The management of the process includes the need to maintain an centrally controlled index of the keys used in the sharding in order to make sure that the request for data is made to the correct database. All this means there is an extra overhead involved in managing the database connections.

There can be little doubt that the use of cheap servers to allow easy scalability is very attractive and this is a significant factor in the recent rise in the use of horizontal scaling. As always in IT, however, we need to be careful not to just jump on the current band-wagon. We should assess every situation individually since Sharding is not a silver bullet.

It may be that Sharding the whole system is overkill. Perhaps only the part that comes under most load—the web front-end, for example—needs sharding whilst the traditional back-office and transactional systems remain on single servers.

As we have seen above, sharding is not the only way of coping with the need to scale. It could be that when or if the price of RAM falls in the future we will look back at this fundamentally disjointed approach to database architecture and laugh. However, for the foreseeable future the scaling out approach is likely to be the *de facto* standard for large datasets.

9.3.6.1 Cloud Computing

Many texts about Cloud site flexibility and scalability as inherent benefits from adopting a cloud-based strategy. Hill et al. (2013) say:

It is probably true that a need for scalability is a significant driver towards adopting cloud. If an organisation understands its business well and it is relatively stable, it can plan what capacity is required and purchase as and when required. Many organisations, however, go through unexpected sharp up- and downturns in their OLTP traffic in step with the business performance. Not having to purchase extra capacity 'just in case' in such circumstances can make public cloud more appealing.

It is true that the cost of the physical resources needed to cope with the maximum processing load a database might face can be expensive. Being able to temporarily gain extra resource and pay for it "as used" can be a good way to keep capital expenditure down on a large scale database project. The costs, however, will not disappear, but rather will appear as rental fees in revue costs.

That caveat aside, it is clear that Cloud computing does offer virtually unlimited scalability, provided the costs do not become prohibitive. Some vendors make relational databases available in the cloud, allowing the single (albeit virtualised) server solution that can, almost instantly, be turned into a distributed solution. The NoSQL tools are also used heavily in the cloud. Having a system with multiple data nodes across many geographic regions becomes a relatively simple task for the system designer, if such horizontal scaling is required.

9.3.7 Scaling Down for Mobile

A recent phenomenon that has begin to impact upon the world of database design is that of mobile computing in general, and more specifically the need to integrate mobile devices into enterprise-wide systems. The term being used is BOYD (Bring Your Own Device). It recognises that employees are often willing to work in places other than at their desk and that they want to access data from all sorts of place, at all sorts of times, from all sorts of devices.

The problem with this move is that, despite the continuing evolution and improvement of mobile devices, they suffer from resource poverty (RAM and Storage) compared to desktop PCs. This can be a major obstacle for many applications and means that computation on mobile devices will always involve some form of compromise.

Perhaps the easiest solution to this problem is to treat mobile devices just the same as a very thin client and allow a server to carry the processing and data storage load. Unfortunately, however, this solution requires ubiquitous broadband to be of any use, and even in some developed nations there are places where communications are unavailable, meaning the system is, in effect, unusable for the mobile client.

At the other end of the spectrum, another solution is to have the system run entirely on the mobile device and have a database there to store the data locally. In these systems data responsibility for data persistence is handed to the device in question. There are databases designed to operate with a very small footprint, such as the public domain *SQLite*. But this also has it's own problem; enterprise systems called for sharing data, not lots of independent data stores.

Synchronising with a Cloud-based server whilst communications allow it is one solution used. This can also be a means for ensure vital corporate data is regularly backed-up. Management of availability when there any many different client system types is far more complicated, but if all devices are connecting to the Cloud this can be made easier and automated.

9.4 Summary

This chapter has reviewed scalability in terms of growing data and growing user-bases. We have seen that small, PC based databases have their strengths, but they do not allow scalable systems to be created. We have seen that there are many possible approaches to database design which may help with scalability. We saw that the systems designer needs to consider where the processing should happen (thin/thick client) in a client-Server design. If more scalability is required there is the option to distribute the processing and/or the data and that many of today's Big Data focused databases using Sharding to allow them to easily scale.

9.5 Review Questions

The answers to these questions can be found in the text of this chapter.

- What is the maximum number of concurrent users that can log on to Access databases?
- Why is the separation of a database session from a user logging-on to an application an important factor in scalability?
- What does Sharding mean?
- Describe the elements of a three-tier architecture
- How does partitioning help when data in a table grows to extent that performance worsens?

9.6 Group Work Research Activities

*These activities require you to research beyond the contents of the book and can be tackle*d *individually or as a discussion group*.

Discussion Topic 1 You are asked to discuss appropriate database designs for an e-commerce site your organisation is wanting to create. As the application becomes available they expect only a few users, but they believe that, once the word gets round about how good the application is, that the number of concurrent users will dramatically increase in the coming months. Your aim is to architect a scalable back-end to the application being created. What alternative design options can you suggest, and what are their respective strengths and weaknesses?

Discussion Topic 2 What are the arguments for and against thin client solutions for an office-based environment? Review some of the sales material from market leaders in the area, such as Citrix and see what they say.

References

Dunmall J, Clarke K (2003) Real-world load testing tips to avoid bottlenecks when your web app goes live. http://msdn.microsoft.com/en-us/magazine/cc188783.aspx

Hill R, Hirsch L, Lake P, Moshiri S (2013) Guide to cloud computing: principles and practice. Springer, London

Hoskins J, Frank B (2002) Exploring IBM EServer ZSeries and S/390 servers: see why IBM's redesigned mainframe computer family has become more popular than ever! Maximum Press, 464 pages