Ubiquitous, Self-tuning, Scalable Servers

Peter Spiro
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
+1 (425) 936 4523

petersp@microsoft.com

1. THESIS

Hardware developments allow wonderful reliability and essentially limitless capabilities in storage, networks, memory, and processing power. Costs have dropped dramatically. PCs are becoming ubiquitous.

The features and scalability of DBMS software have advanced to the point where most commercial systems can solve virtually all OLTP and DSS requirements.

The Internet and application software packages allow rapid deployment and facilitate a broad range of solutions.

Result: We now have the hardware and software to really solve the world's data management problems. And it *is* happening: data volumes are growing tremendously and applications are booming.

Problem: Database vendors have succeeded on the traditional measures of VDLB, parallelism, object-relational extensions, OLTP, etc, but they have failed to make their systems easy to use. Most attempts towards providing a simpler model are aimed at a small subset of the entire problem, usually tuning/administering large installations. This is fixing a symptom not the true problem. We need to provide a much broader, integrated solution to the problem of managing the data that will be produced over the next 10 years.

This paper describes the complexity within current database systems and some of the reasons for the current state. Then follows a discussion on many of the dimensions of a broad solution. Finally there's a section describing Microsoft's attempts towards an integrated solution.

1.1 Keywords

Managing data, self-tuning database

2. Complexity in current database systems

There are multiple factors that have lead to the increased complexity within database systems. The first factor is simply the drive from within the dbms software community to continually add features, capabilities, and performance enhancements.

For example, in the drive to support larger databases most systems use extensive parallelism within their products. The big parallelism push has created a tremendously complex execution model. What is seemingly a simple query ends up with an incredible flow pattern of sorts, joins, merges and other types of data flow, all needing scratch space, all needing memory, and all interacting with each other.

SIGMOD '98 Seattle, WA, USA

OLTP requirements, often driven by the TPC benchmarks rather than real systems, produce features such as partitioned buffer pools, pinned tables, strange concurrency models, etc. Again, the database vendors seem happy to continually add features and enhancements. More often that not, these capabilities are governed by some poorly documented trace flag or database parameter that the user or DBA must set.

The 'object-relational' capabilities of database systems create new access patterns, they have different locking semantics, and there are often calls out of the database system to external functions, which don't coordinate resource requirements/usage with the dbms software. Some of these systems might even manage their own buffer pool of objects thereby competing with the dbms buffer pool.

Another 'advance' causing complexity is hardware. Because of the additional hardware and operating system capabilities, customers are able to add storage online, add machines to clusters, and even reconfigure data between disks. As a result the physical configuration of the platform now presents a moving target for the dbms software. A database, which was carefully tuned at some prior time for optimal memory and disk usage, may be out of date. Certain devices may be overloaded, and other unused.

The database applications space is also booming; these apps have created unbelievably complex environments. Such products can create tens of thousands of database objects (table, index, view), they generate queries no programmer would ever devise, and they often generate mixed workloads on top of the database system (OLTP and DSS).

The Internet is also creating unplanned data access patterns. That is, Web servers are often handling a variety of data and applications: tabular data from databases, Web pages, audio/video data, etc. Furthermore the load on server machines is very often unknown ahead of time.

Beyond complexity in the feature set and application load, scalability is another major factor contributing to the explosion in database complexity. Scalability in this context refers to a product supporting both low-end and high-end requirements.

If we're building only a high-end product then it's somewhat acceptable to have a very complex product, with hundreds of knobs, that requires consultants/gurus to optimally tune for the target application. Conversely if we're building a low-end product, we need to have very few, if any, knobs; customers can live with fixed settings. But if we're aiming for a product that actually scales from the low-end to the high-end, then the

spectrum is much greater. It's this broad spectrum that exposes the limitations of current products.

For example, the low-end might not even want logging/recovery, whereas the high-end wants some sort of continuous spooling capability and integration with HSM systems. The low-end is concerned with memory requirements between the different apps that may be started on the client machine, whereas a server often has exclusive use of a machine.

So what's the problem with the current levels of complexity?

The problem is the current state has, in general, created products that fail to deliver on the promise. The products have gotten too complex for the low-mid range scenarios. In this space, customers simply want the product to work. They don't want to set knobs, very often they don't want to even read the documentation to get started. Furthermore, although it seems rather foreign today, we should expect that the proliferation of hand-held devices will also have data management requirements. Reread this as, databases will live on the Palm Pilot and Windows CE devices. Current dbms products will fail miserably in such an environment.

For the high-end, the products are generally tunable (of course one must be an expert with the product) yet their specific failure results from it being very common for the workload and application environment to be dynamically changing, thus the tuning must also be dynamic.

So the result is current database products are too complicated for the low-mid range, they are not prepared to be a player in an integrated solution, and even on the high-end, current products require too much expert-intervention. We will not succeed in producing commodity software in this state.

3. 'Ease-of-use' in data management

Making a database product truly easy to use is a very broad and hence difficult problem. Much of the difficulty results from the fact that a database product is so widely applicable; it can be used to solve so many different problems. For example, a text-editor solves a more restricted problem, thus it's much easier to make it easy to use. In a sense, a text-editor is like a hammer, it does one thing. But a database system will be all over the board, it will be in stock exchanges, it will be managing address lists, it will be in glass houses managing crucial corporate data, and it will be in PDAs. So a database system is like the entire toolbox, it solves many different problems.

This section attempts to classify the problem space. It's important to note this section is not solely about making tuning and administration easier. It's about the wide variety of problems we need to address to make our products simpler and more cost effective in many dimensions; only then will we create ubiquitous commodity software.

3.1 Scalability

It is undesirable to create different database products for each of the market segments (low-end to high-end). Not only does that increase engineering cost, but invariably the products will get out of synch, they won't share the same capabilities and semantics, things will work differently. With differing engines it will be difficult to create applications on the low-end version and scale them to the mid-range version. For example, the SQL syntax might be different, or triggers might work differently, or the record structure might support NULLs differently (or not at all).

So the ideal solution would be for one product (actually the code base) to scale from the very low-end to the very high-end. The product should work on PDAs and be suitable for disconnected users, as well as allow 7X24 mission-critical deployment.

3.2 Design/creating new databases.

Regardless how fast the Internet and market pressure is forcing customers to deploy applications; it still makes sense for a broad range of customers to utilize some sort of visual data-modeling tool. Ideally this tool is incorporated into the base product. That is, the tools should be intimate with the core database product's capabilities; this will allow a more efficient database design. For example, if a system supported fragmented records but it was overly expensive to retrieve the records, the design tool could steer a database designer to a more efficient storage design. Or if the dbms supported varying page sizes but it complicated run-time buffer pool performance, this capability might be discouraged for casual use.

3.3 Physical data placement

Data layout usually has a significant effect for most database systems. As the systems have moved towards richer functionality such as partitioning, cluster support, pages with records from different tables, multiple blob types, a variety of access methods, etc, data placement has become more confusing and even more critical towards achieving good performance. On the other end of the spectrum, low-end systems would like very compact storage; the empty database size has to be measured in KB, not MB, the user does not want to determine where the log should reside.

The wide range of choices here dictates that this problem can't be left to the user or even the DBA. At the very least, systems should provide great tools for planning and organizing data layout. In the best case, the system adapts dynamically to storage capabilities and data access, moving records, growing/shrinking files, reorganizing pages so during scans they'll be accessed in disk order.

3.4 Importing/exporting/replicating data.

Datamarts are proliferating, again part of the success of the relational database products. Disconnected computing is a reality; users wish to access/update/manage data on their laptop and get re-synched as they connect back to the corporate server. In order for a database system to gain widespread adoption, it should allow simple and scalable data movement. This might be VLDB data loading, it might be extracting from one database and inserting into another, it might be a loosely consistent replication model for Sales Force Automation. These capabilities are very important since database products will be living in a chaotic, heterogeneous environment.

3.5 Appropriate algorithms for scalability.

This is the aspect that has generated literally hundreds of knobs in certain database products. The scenario is as follows: a bright software engineer thinks up some clever optimization, they convince themselves their feature is wonderful and important, it may give a 7% performance improvement on TPC-C, or scans can go 5% faster with their new data layout. So traditionally the feature is added but since it might not have wide applicability,

or worse it might conflict with another feature, it would be added as a non-default parameter only to be enabled via some setting/knob. Examples are buffer pool settings, memory settings for sort or hashing, various locking characteristics (escalation, carry-over locks), query optimizer directives, checkpoint intervals, etc.

This is a disaster. 95% of the customers don't need 95% of the knobs, especially given the advances in hardware. For the other fairly useful ones, the system itself should dynamically recognize and adjust behavior. This is a very important point, in fact, it's the crux of the statement around fixing the symptom rather than the problem. Most vendors are leaving in the knobs/features and creating tuning/admin tools to mask the problem. That's the wrong approach. We need to make the systems inherently simpler.

3.6 Queries

Regardless of its power and flexibility, SQL is not the complete answer. If databases are going to become truly ubiquitous, end-users and developers need to find another way to store/find their data. Again similar to the need for a simple and intuitive database designer tool, we need to provide a simpler programming and query model. Perhaps it's an integrated forms package, perhaps it's Web based queries, maybe it's a voice recognition product coupled with natural language processing capabilities.

3.7 Shared resources

Another problem that should be addressed relates to resource contention. For example, the query processor needs memory for compiling and maintaining query plans, temporary operations such as sorting and hashing use lots of memory, the buffer pool uses memory for storing database pages, logging/recovery subsystems can be a large user of memory. It's usually the case these subsystems do not coordinate with each other well. To make matters worse some systems run multiple instances on a node, each requiring fixed settings. This leads to components that compete for memory or may even not use available memory. Another complication is there might be additional server applications running on the system such as mail or Web server products.

In an ideal, simple and cost-efficient system, the applications would coordinate internally and with each other. System administrators would not need to set knobs to control memory usage on either the high-end or the low-end. Again this broader goal comes from the fact database software will be interacting with other end-user and system software.

3.8 Tuning

Most database systems vendors are pursuing some sort of automatic tuning mechanisms. Because of the tuning knobs, which proliferated for various optimizations, current systems do not reach their maximum performance with default settings. A sophisticated user or a DBA can generally tune systems to some degree but vendors are realizing the problem has gotten too complex.

The solutions in this space rely on some sort of data capture mechanisms which indicate how the system is performing, and then some suite of tools which recommend changes such as partitioning tables, changing buffer pool sizes, etc. While this is a good first step, it's not nearly broad enough to encompass the entire range of the problem space.

3.9 Administration

Along with the auto-tuning systems, most vendors also seem to be uniformly pursuing a graphical tool for administrating databases, often times allowing remote administration. This allows DBAs to setup backup schedules, for example. Or the tools may setup replication schedules. This type of facility is usually very valuable to DBAs, certainly it's better than nothing; but like the Tuning facility, this capability is treating the symptom rather than the problem. In other words, they help govern complex systems, but they don't make database systems intrinsically simpler.

3.10 Integrated Applications and Storage

Database vendors are very proud of their products. As a result, it's very common for them to view all data management problems as database problems; the truth is it's not always the case. Many other software programs are much better at solving some target problem. Web/Document management systems, spreadsheets, word processors, mail systems, etc, all solve different problems. The key point is these other programs produce data in some other native format. The data won't reside in the database system.

Hence the database system needs to become a cooperating program in an integrated storage system. System services such as backup, replication, transactions, event mechanisms need to coordinated across all the data formats and applications. Allowing database records to have a 'link' to another non-database file is a good example of progress in this space.

Another aspect of this concept is more targeted uses of subsystems of database technology. The classic example is allowing access to just the 'storage engine' component of a database product. Many operating systems provide such a record management system; also there are various individual software packages offering such a lightweight store. Perhaps the handheld devices will be able to use this type of lightweight record store. It's better to have the storage system be the native subsystem that resides in the relational database rather than a new store. In fact, proliferation of new record stores for each different application is a classic problem contributing towards chaotic and non-uniform data management.

Query processing technology provides another good example. In a truly integrated system the query technology from the relational database would be used for scanning spreadsheet data or properties in many different documents.

4. Solutions from Microsoft

At Microsoft we're aiming for a broad solution towards simplified data management as described above. Our solutions are divided into four main categories each of which address some portion of the larger problem.

4.1 Scalability, Components, and Integrated Storage

With the Microsoft product set we are building core systems which will scale to the largest possible range of database solutions. That is, we're building common components (query processing technology and record management services) which are part of Microsoft's relational database product, SQL Server,

yet these components are intrinsically designed to scale from the very low-end to the very high-end. As a result we will subsequently use these components as subsystems within other types of products such as low-end database offerings, PDA data management services, mail products, any products which store and retrieve data. In many instances these products will coordinate access through OLE/DB, which is a set of interfaces that provide data access to/from data providers.

Furthermore because we will have the same subsystems within a variety of products, we are designing/building common services to coordinate among these different products. Examples are transaction services, common logging/backup mechanisms, global memory management facilities, common replication mechanisms and semantics, etc.

This will greatly facilitate sharing of data and simplified management in the future.

4.2 Server Internals

Within these common components we are striving to eliminate settings. So rather than propagate settings and create tools to manage them, we are rethinking algorithms to make them simpler and self-adjusting. For example, the storage engine will grow and shrink its buffer pool depending on system pressure. Internal to the database system we're coordinating memory usage among the major client subsystems (buffer pool, sort, hashing, query processor). Other examples include files growing/shrinking automatically, statistics gathered in the background, we've eliminated settings defining locking structures, open objects, max users, etc, etc. With the 7.0 release of SQL server, we've actually decreased the number of settable parameters.

4.3 SQL Server Enterprise Manager

In addition to the ubiquitous, self-managing components, we're also putting great energy into Microsoft's versions of the classic administration/tuning tool, SQL Server Enterprise Manager. Enterprise Manager was first introduced in SQL Server 6.0, it defined 'ease-of-use' as a database feature. Subsequently we've continued to expand its role and make administration and tuning even easier.

Enterprise Manager scales from the desktop to large organizations. For the low-end it strives to make systems self-configuring and self-managing. For the high-end, its goal is to reduce total cost of ownership by providing multi-server operations. It provides a central tool for managing all server and database settings in a graphical format.

Within the Enterprise Manager there are a number subsystems or facilities which aim to provide a solution for different problems.

- Visual Database Tools to design databases and create SQL queries/views.
- Event and Agent mechanisms which are tied into the database server. This provides for job scheduling, notifications, alerting/paging, monitoring when system events occur, etc.
- Data Transformation Services, which is a graphical tool for importing/exporting data from heterogeneous sources.
- Wizards, which aid many database activities such as scheduling backups or setting up replication scenarios.
- Profiling and Tuning tools, which can capture workloads, graphically display query execution plans. As an example, we have an 'index tuning' wizard which can recommend optimal index configurations based on real workloads.

4.4 One Product Set

Another important advantage, which should go a long way to simplifying data management, is that we're generally trying to integrate various components into one product set. So for example, we've built a natural language mechanism, English Query, into SQL Server. It will be part of the base system. We're pushing OLAP capabilities into the server. Replication is an intrinsic part of SQL Server. We're adding content-indexing capabilities. Having an integrated product helps move the sub-systems to similar management tools, similar forms packages, similar documentation and help facilities; the look and feel become the same thereby benefiting the customer.