Challenges in building a DBMS Resource Advisor

Dushyanth Narayanan
Microsoft Research
Cambridge, UK
dnarayan@microsoft.com

Eno Thereska
Carnegie Mellon University
Pittsburgh, PA
enothereska@cmu.edu

Anastassia Ailamaki
Carnegie Mellon University
Pittsburgh, PA
natassa@cmu.edu

Abstract

Administration increasingly dominates the total cost of ownership of database management systems. A key task, and a very difficult one for an administrator, is to justify upgrades of CPU, memory and storage resources with quantitative predictions of the expected improvement in workload performance. We present a design and prototype implementation of a Resource Advisor that is able to answer "what-if" questions about DBMS performance under hypothetical conditions. We discuss the design issues and challenges involved in building such a Resource Advisor, as well as our experiences in building a prototype Resource Advisor for SQL Server.

1 Introduction

Administering database management systems (DBMS) is a complex and increasingly expensive task. There is a pressing need to raise the level of abstraction at which database administrators (DBAs) interact with the system, by automating tasks which currently require substantial human effort and expertise [8]. In this paper we focus on the task of *resource (re)provisioning*: determining the number, type, and configuration of hardware resources most appropriate to a given workload, hardware budget, and performance goals.

Resource provisioning is typically done by human experts using experience and rules of thumb to decide whether additional resources will improve performance [3]. The cost of such experts is significant for large enterprises and prohibitive for small ones. Even experts find it difficult to *quantify* the expected benefit of a resource upgrade. The net result is over-provisioned systems with no guarantees on performance [8].

The key technical challenge in automating resource provisioning decisions is automated prediction of performance in hypothetical hardware configurations. In other words, we wish the system itself to provide accurate, quantitative answers to "what-if" questions such as "*what* would be the increase in throughput *if* the server's main memory were doubled?" In this paper we discuss the design issues and challenges in building such a predictive capability, and also our experiences in building a specific system, a *Resource Advisor* for SQL Server.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

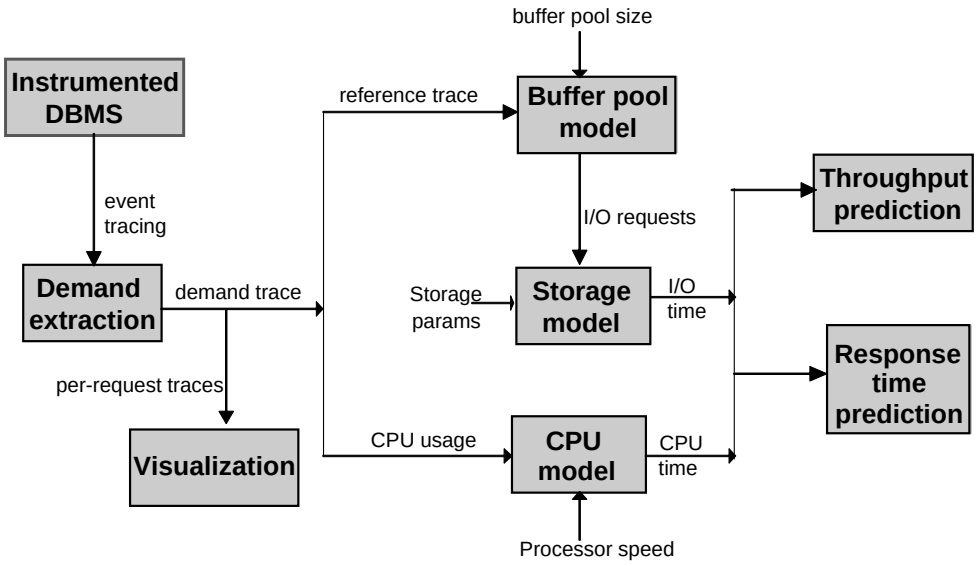


Figure 1: Resource Advisor architecture

2 Design principles

Large commercial databases are complex systems that depend on several physical resources such as the back end storage system, volatile main memory and CPUs. A database administrator (DBA) must decide on a good initial configuration of these resources, and then continuously monitor the system for new bottlenecks and changes in workload. To do this she must have an intimate understanding of the various database components, their interactions, and of the workload. Such experienced DBAs are expensive and even they do not have the tools to accurately and easily predict the performance effect of any resource provisioning decision.

Consider a DBMS running multiple application workloads with different resource demands and performance requirements. For example, DSS workloads have low concurrency and total run time is the metric of interest. OLTP workloads have high concurrency and require not only high throughput but also bounded response time. For any proposed resource provisioning the DBA must estimate the impact on the performance of each workload, taking into account the resource contention between them.

The most common approach to (re)provisioning such systems is to monitor the performance counters provided by most commercial systems. These counters measure aggregate load statistics for various resources, which is not always sufficient to find the global bottlenecks. They do not offer any insights into response time, as they do not track per-request resource usage or distinguish between critical-path and background resource usage. Finally, they place the heavy burden on the DBA of correctly interpreting 400+ performance counters.

We advocate a system architecture that addresses these problems by

1. *Tracing* per-request resource usage and control flow at fine granularity.
2. *Modelling* hardware resources and the algorithms that schedule or share them across multiple requests.
3. *Predicting* performance on hypothetical hardware by combining workload traces with hardware models.

The bottleneck in DBMS provisioning today is the human in the loop. CPU cycles are relatively abundant, allowing fine-grained yet low-overhead tracing of the live system, as well as offline trace processing using idle cycles. For example, SQL Server running an OLTP workload generate 500 events/transaction, with a CPU overhead of 1000 cycles/event and generated 68 bytes/event of trace data [5]. Extrapolating to the fastest TPC-C system as of date (3,210,540 tpmC with 64 processors at 1.9 GHz) [7], we get a CPU overhead of 5% and a trace data rate of 8 MB/s. This could be reduced further through optimisation, sampling, and runtime event filtering.

	Event Type	Arguments	Description
Control Flow	*StartRequest*		SQL transaction begins
	EndRequest		SQL transaction ends
	EnterStoredProc	*procname*	Stored procedure invocation
	ExitStoredProc	*procname*	Stored procedure completion
CPU scheduling	*SuspendTask*	*taskID*	Suspend user-level thread
	ResumeTask	*taskID*	Resume user-level thread
	Thread/CSwitchIn	*cpuID, sysTID*	Schedule kernel thread
	Thread/CSwitchOut	*cpuID, sysTID*	Deschedule kernel thread
Buffer pool activity	*BufferGet*	*pageID*	Fetch a page (blocking)
	BufferAge	*pageID*	Reduce the "heat" of a page
	BufferTouch	*pageID*	Increase the "heat" of a page
	BufferDirty	*pageID*	Mark a page as dirty
	BufferReadAhead	*startpage, numpages*	Prefetch pages (non-blocking)
	BufferEvict	*pageID*	Evict and free page
	BufferNew	*pageID*	Create a new page
	BufferSteal	*numpages*	Allocate memory from free pool
	BufferFree	*bufferID*	Release memory to free pool
Disk I/O	*DiskIO*	*startpage, numpages*	Asynchronously read/write pages
	DiskIOComplete	*startpage, numpages*	Signal read/write completion
Locking	*EnterLockAcquire*	*resourceID, mode, timeout*	Attempt to lock a resource
	ExitLockAcquire	*status*	Success/failure of lock acquisition
	LockRelease	*resourceID, mode*	Release a held lock

Table 1: Instrumentation events

```
8113984086 0 XactionStart tpcc_neworder,0
8113984086 1:0 CPU 3663
8113987749 2:1 CPU 187
...
8113990027 12:11 LOCK KEY: 5:844424932360192 (e102aa462451),S,ACQUIRE
...
8114036559 269:268 MEM ALLOC,1
...
8114152008 368:367 CPU 8544
8114160900 369:368 BUF 00000005,00000001,00000170,Fetch
...
8114160900 432:368 BUF 00000005,00000001,000001AF,Fetch
8114160900 433:368 CPU 109
...
```

Each resource demand contains a timestamp and a "demand ID", followed by a list of previous demands that must precede this one, in the transaction execution. This allows us to capture any in-transaction concurrency: e.g., demands 369–432 are asynchronous prefetch requests to the buffer manager, which are executed concurrently with demand 433, i.e. computation is overlapped with I/O here. Each demand has additional type-specific parameters, e.g. lock demands specify a resource ID, a mode, and an action (acquire or release).

Figure 2: Simplified snippet of demand trace

3 Experience

Based on the principles and high-level design described above, we have designed and implemented a *Resource Advisor* for SQL Server, which predicts the performance of a live workload under hypothetical hardware upgrades. Here we briefly describe our experiences with an early prototype based on analytic models, which was described in detail in an earlier paper [6]. We then describe our current simulator-based approach.

3.1 Analytic modelling

Figure 1 shows the high-level design of the Resource Advisor. It relies on fine-grained, low-overhead event tracing from an instrumented DBMS. The instrumentation points are chosen to enable *end-to-end tracing* [2] of each request from the moment it enters the system to its completion. We record each use of system resources — CPU, memory, I/O — as well as virtual resources such as locks. Table 1 shows the set of events traced by our instrumentation. These events allow the Resource Advisor to reconstruct exactly the sequence of resource demands issued by the workload. Since this sequence is an aggregate of many concurrently executing requests, the Resource Advisor first separates it out into *per-request* demand traces. This requires instrumentation of all context switches: points where a resource such as CPU stops working on one request and starts work on another.

The raw event trace is transformed into a per-request demand trace, where each request is represented as a partially ordered set of resource demands, each for a specific resource. Figure 2 shows a simplified snippet of a demand trace for an OLTP transaction. The aggregate demand on the system is then the effect of concurrently executing these per-request demands.

Subsequent steps in processing are parametrised by the characteristics of the hypothetical "what-if" hardware: the buffer cache memory size, the CPU clock speed, and disk parameters such as rotational speed. The buffer references are processed by a cache simulator to generate an I/O trace, and the I/O and CPU traces are fed to analytic models that predict the throughput and mean response time of each transaction type.

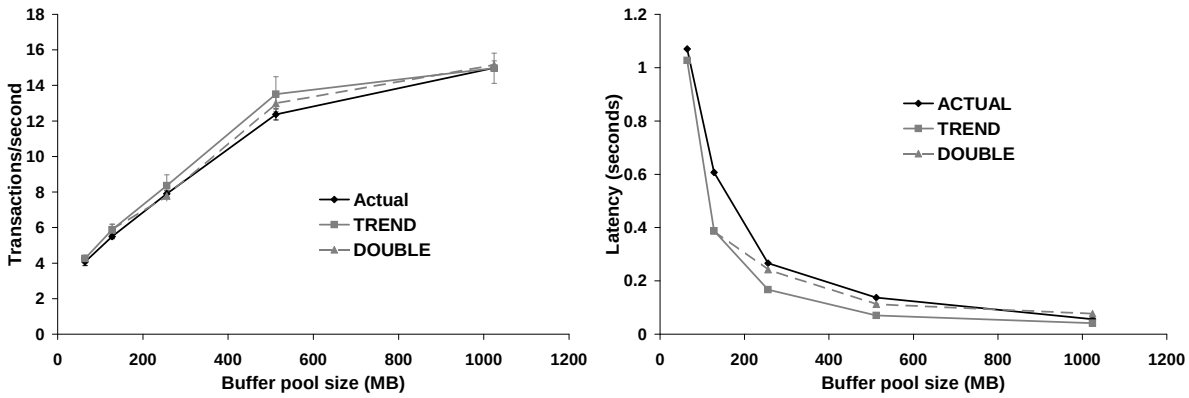
Our analytical models are able to accurately predict the effect of changing the buffer cache memory on the throughput and response time of an OLTP workload. Figure 3 shows the prediction accuracy for two different types of "what-if" questions. DOUBLE predicts the effect on performance of doubling the memory of the current configuration (e.g. from 128 MB to 256 MB). TREND predicts performance over the entire range of memory sizes, bases on observing the system with 64 MB.

Thus the models have good accuracy but restricted applicability. They make two major assumptions about the workload, which are valid for OLTP but not for other workloads such as DSS:

- that buffer cache misses cause a random-access I/O pattern;
- that the throughput bottleneck remains the same throughout the workload execution, i.e. the workload does not have different phases with different bottlenecks.

With analytic models based on operational analysis, it is easy to predict aggregate throughput, assuming sufficient concurrency that the bottleneck resource is always busy. However, if there are multiple concurrent users, each with a different workload (for example a different transaction mix), then it is difficult to predict the throughput of each user individually.

Analytic models also make it difficult to predict response time. Our models predict mean response time per transaction type but are specific to OLTP. They also assume that a request's response time is dominated by its resource demands rather than queueing and scheduling delays caused by concurrently executing requests. To correctly model queueing and scheduling delays, and to compute second-order metrics such as variance in response time, we need queueing models. However, analytic queueing models rely on assumptions about request arrival time distributions that are often unrealistic.



(a) Throughput (b) "New order" response time

Figure 3: Predicting OLTP performance for hypothetical memory changes

3.2 Simulation-based modelling

The current version of the Resource Advisor is based on event-driven simulation rather than analytical throughput and response time models. Live workload traces are decomposed into per-request demand traces as before, and the concurrent execution of the requests on hypothetical hardware is modelled by the simulator. The result of the simulation is a execution trace with the predicted timing information (including scheduling delays) of each resource demand within each request. This allows us to compute the predicted throughput, response time, or any other performance metric of interest. Unlike the analytic models, the simulation approach is workload-agnostic and also enables a wider range and a finer granularity of performance metrics.

Each request's demands are executed by the simulator according to the partial order specified in the demand trace. Demands are executed by passing them to the appropriate resource model, which determines their completion time by adding any queueing/scheduling delay as well as the predicted service time:

- The *CPU model* computes scheduling delay by simulating a non-preemptive FIFO scheduler. Service time is computed by scaling, i.e. the speed of CPU execution is assumed to be proportional to the clock speed.
- The *buffer cache model* simulates an LFU eviction policy. Cache misses generate disk demands which are handled by the disk model. Note that disk demands are not directly present in the demand trace: we capture the workload's reference trace above the buffer cache, so that we can model the effect of changes in buffer cache memory.
- The *disk model* is a simple approximation of a single-spindle storage system without on-disk caching. Track and sector positions are inferred from LBNs (logical block numbers) based on the known disk geometry, and seek and rotational times are inferred from these. Based on this, the disk model is able to simulate the SSTF (shortest seek time first) scheduling policy used by most disks.
- The *lock model* handles requests for locks at various granularities — page, record, etc. — and in different modes — shared, exclusive, shared-intention, etc. It uses the same default policies as the DBMS to make decisions on competing lock grant/upgrade requests.

We are confident that these models, although simple, can provide good accuracy for a wider range of workloads than the analytic models. Our philosophy is to start with these simple models and refine them only if necessary for improved accuracy.

Simulation has a higher overhead than analysis but is still typically much faster than real time: we simulate a CPU computation of arbitrary length in constant time, and a disk access with a few cycles of computation. For in-memory, lock-bound workloads simulation is slower than real time, as the simulator's buffer and lock management are no faster than that of the DBMS.

4 Ongoing challenges

There are many open questions on designing, building, and deploying systems such as the Resource Advisor. Here we present some of these questions and our thoughts on answering them.

Granularity. What is the best granularity to represent resource demands? For example, we represent a CPU "demand" as a single number: the number of cycles of computation. Including information such as L2 cache misses and the integer/floating point instruction mix could allow "what-if" questions about different processor architectures rather than just different clock speeds. However, this finer granularity comes at the cost of increased complexity in instrumentation and modelling.

We envision a need for models at multiple levels of complexity, with the DBA using a "drill-down" approach to increase complexity where needed. For example, crude CPU and disk models might suffice to indicate that a faster CPU would be more valuable than a faster disk. The DBA could then use a more refined CPU model to exactly quantify the performance benefits of different processor upgrade options.

Scope. How much of the system should we model? The key insight that makes performance prediction feasible is that we only need to model those aspects of the system that affect performance and are affected by resource availability. Aspects which are essential to the correct functioning of the system but independent of resource availability can be ignored. For example, when simulating a disk access we need to predict its timing but not the contents of the accessed block.

Thus we must trace the system at a level *above* that of the resource manager but *below* that of any resource-agnostic components, to avoid the complexity of modelling them. For example, in the Resource Advisor, we trace page accesses above the buffer cache rather than below, since the latter will change with the size of the buffer cache. In contrast, we trace the physical execution of query plans, i.e. below the query optimiser. This frees us from the task of modelling the query optimiser and tracing all its run-time inputs. However, it limits us to modelling resource-agnostic query optimisers that are not adaptive to changes in resource availability but make decisions solely based on data statistics and cardinality estimates.

Evolution. When building a Resource Advisor for a legacy DBMS, we chose to insert only passive instrumentation, while maintaining the simulation/analytic models separately. However, this introduces the additional burden of keeping the models consistent with the DBMS components as they evolve. For example, if the lock scheduling algorithm changes in the DBMS, a corresponding change must be made to the lock model. If the code itself is restructured, then the tracing instrumentation points may need to be changed; if this results in a change in the semantics of the traced events, this will cause a change in the models as well. We surmise that tighter integration of predictive models with DBMS components, i.e. making each component truly self-predictive, will help to alleviate this problem. However, we currently lack the programming tools and techniques for developers to maintain a performance model for each component in tandem with its functionality.

Hierarchical models. The drill-down approach also requires us to ask and answer "what-if" questions at different component granularities. For example, the storage component could be a file server with a network RAID back end. For the initial phase of resource planning, the DBA might simply ask "What if the entire storage subsystem were twice as fast?" If the predicted benefits of this look promising, she might investigate different ways to achieve this speedup, for example "What if I made the file server 4-way SMP" or "What if I moved from mirroring to RAID-5?" This hierarchical approach would avoid the need for asking "what-if" questions about all possible hardware configurations.

Administrative boundaries. In a typical 2- or 3-tier architecture, there are multiple components — application servers, database servers, networked storage — typically from different vendors and possibly with different administrators. We could hope that in the future each of these would be self-predicting, but it is likely that they will provide this prediction as a "black-box" functionality that does not expose model internals. Thus the tight integration of different predictive components that we use in the Resource Advisor may not be feasible. Rather than predict the performance of individual resource demands, we might have to process the entire workload trace with the DBMS to create a "storage access trace" and pass that to the storage model to get the timings of the I/Os generated. Since the I/O timings would affect the timings of the entire workload, we would have to iterate this process to converge on a solution.

Distributed modelling. End-to-end performance prediction for large distributed systems is a significant challenge. Individual hosts can efficiently generate local trace information; however, a request in a multi-tiered or clustered configuration might trigger activity on multiple hosts. Backhauling all event traces to a centralised location is a simple but non-scalable solution, and hence we need distributed modelling and prediction algorithms.

5 Related work

Our work on end-to-end tracing in SQL Server was directly inspired by the Magpie project [2], which used end-to-end tracing in 2-tier web services to model workload resource demand and control flow. Our broad aim — automated resource provisioning — is one of many self-tuning scenarios suggested by Weikum et al [8]. Other researchers have investigated self-tuning for other aspects of the DBMS: for example, the DB2 Advisor [4] and the Database Tuning Advisor [1] suggest the most appropriate set of indexes and materialised views as well as the best physical layout of tables.

References

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *Proc. 30th VLDB conference*, Aug. 2004.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[3] J. Gray. *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann, 1992.

[4] G. Lohman, G. Valentin, D. Zilio, M. Zuliani, and A. Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, Feb. 2000.

[5] D. Narayanan. End-to-end tracing considered essential. In *Proceedings of High Performance Transaction Systems – Eleventh Biennial Workshop (HPTS '05)*, Sept. 2005.

[6] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *Proceedings of IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005)*, Sept. 2005.

[7] Transaction Processing Performance Council. Top ten TPC-C by performance. `http://www.tpc.org/tpcc/results/tpcc_perf_results.asp`, Mar. 2005.

[8] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *Proc. 28th VLDB conference*, Aug. 2002.