Sustainable energy-efficient data management

Goetz Graefe, Harumi A. Kuno
Hewlett-Packard Laboratories

Energy-efficient in the management of large data volumes can be approached at various levels in the hardware and software stack. For example, some people just look at the most appropriate hardware, e.g., high-performance CPUs such as Xeons versus high-efficiency CPUs such as Centrinos (and their successors). Similar thoughts apply to storage, e.g., 7200 rpm (capacity-optimized) SATA drives versus 15K rpm (performance-optimized) fiber channel drives. Others look at storage hardware, e.g., flash versus traditional disks, storage placement, e.g., RAID-1 versus RAID-5 and -6, and at storage formats, e.g., columnar storage and compression. Yet others look at caching, e.g., memcached, and in-memory databases. Hardly anyone looks at workload management mechanisms within the data management engine, e.g., deferred index maintenance and deferred view maintenance during peak load; or at 'pause and resume' functionality in database utilities such as index creation, data migration, and reformatting.

Each of these directions may reduce the power requirements for CPU, memory, or storage by a factor of 2. Their combined effect is unknown. More importantly, their interactions, be they positive or negative, are unknown. Therefore, it seems that all of these approaches must be explored and their effects measured, each approach on its own and in multiple approaches in combination. Ideally, a coordinated effort would integrate the research advances by multiple groups, integrate them into a single software and hardware instance, measure the effect of each with the others already optimized, and in particular measure and optimize their overall effect.

One particular effect is often overlooked. In order to cope with unpredictable workloads, many actual deployments are vastly over-provisioned for the actual or average load. Among all types of servers, this is particularly true for database servers, because it is as simple to submit an easy and efficient request as a hard and long-running request. For example, a query that is slow today might be fast tomorrow due to a newly added index, but also vice versa. Index creation and removal might be entirely automatic and thus seemingly unpredictable. In fact, occasional automatic index creation and the load it imposes may be a big reason for over-provisioning.

Over-provisioned, mostly idle database servers consume almost as much energy as the same servers processing a full load. What is needed, therefore, are mechanisms and policies that smooth the processing load even for uneven application loads, and that avoid or hide peak loads due to database utilities such as index creation.

A number of mechanisms immediately come to mind, many of which have been researched in recent work. For example, as database cracking does for an in-memory column store, adaptive merging creates and optimizes B-tree indexes on paged storage such as disks and flash devices. Both initial index creation and incremental optimization are side effects of query execution and thus impose no additional workload. Key ranges never queried are never optimized.

As another example, today’s automatic tools for index tuning occasionally call for creation of new indexes. Pausing and later resuming index operations is not trivial, in particular if the overall effort for index operations with 'pause and resume' functionality is to equal equivalent index operations without this capability, if pausing and resuming must not waste or repeat any work, if concurrent user transactions are to be permitted to update the table while an index operation is paused, or if paused index operations are to resume after a server reboot, a failover from one operating system instance to another, or even a crash and its recovery.
As a third example, deferred maintenance of storage structures has been employed in some column stores (using an in-memory row store) but it similarly applies to traditional stores and data structures in memory, on flash devices, and on disks. Partitioned B-trees seem to be a particularly promising approach, because a single mechanism supports deferring logical maintenance (delayed updates) of secondary structures (using partitions in the primary structure) and physical maintenance (delayed structure optimization) in both primary and secondary index structures. The same mechanisms can also support delayed maintenance of copies or replicas temporarily unavailable, e.g., to save power for a redundant device.

These three examples are not an exhaustive list of opportunities. A new join and grouping algorithm may provide superior protection against poor performance due to mistaken compile-time algorithm choices (it may also prove a superior algorithm for many map-reduce problems). Memory-intensive algorithms such as sorting can quickly and widely modify their memory allocation without losing their efficiency using recent (unpublished) techniques. Adaptive restructuring of query execution plans reduces reliance on compile-time query optimization and its error-prone cardinality estimation.

What is needed is research to tie these mechanisms together, an implementation effort for a realistic evaluation, and policies that guide their usage. This is the authors’ research agenda.

Authors’ backgrounds: Goetz Graefe has researched compile-time query optimization, runtime query execution including dynamic query execution plans, indexing with a focus on B-trees, transactions (both concurrency control and recovery), and database utilities (e.g., index creation, loading, defragmentation, and consistency checking). He has also served as a software architect of Microsoft’s SQL Server product for more than 11 years. – Harumi Kuno is currently focusing on robust query processing and database workload management. She has also researched distributed systems, IT services (e.g., managed services), and Web services.

Some publications by the authors on workload management, query processing, indexing, and transactions: