Use of a Component Architecture in Integrating Relational and Non-Relational Storage Systems

Robert Atkinson, Microsoft Corporation, BobAtk@Microsoft.Com

A fundamental service provided by the products offered by various software vendors is that of managing the persistent storage of application data The characteristics and usage patterns of the data stored by various different applications varies widely: a storage service designed and tuned for one class of application may not serve another application well To a first approximation, two extremes in this spectrum of storage services required by applications are today provided by relational databases and file systems.[1] Consider for a moment some of the design and performance characteristics of these two extremes

Relational systems historically have had as their design focus the goal of supporting vast quantities of data of mostly homogeneous schema The data records are relatively fixed in size, and, on the whole, relatively small compared to the quantity of such records. Records are logically independent of each other, each record being semantically distinct from its predecessor and successor in the table in which it is found. Relational systems' historical focus has been on supporting high-performance transactional access by large numbers of concurrent users who make relatively short-term and small updates.

File systems, on the other hand, have historically had quite a different design focus They understand quite little of the schema of the data that they store Data transfers made during updates are relatively large and of varying size the performance of the system in supporting streaming I/O is crucial to overall application performance. Any internal relationship between semantic elements within a file are unknown to the system, thus, relationships must conservatively be assumed to exist Traditional file systems lack transaction support, relegating that functionality instead to the application The design point for file systems has been that of a group of users working for relatively lengthy periods on independent pieces of data.

In the past, most applications have needed either one type of storage service or the other. An application was for example either a on-line credit card transaction system or a word processor, the former having little need for file systems, the latter having little need for databases.

Recent years, however, have seen this clear distinction becoming blurry and indistinct Applications such as email or imaging services find that the data they manipulate and the usage pattern under which they do so has some characteristics that lend themselves to the use of a relational storage system yet other characteristics for which historically a file system is better suited The header of a mail message along with its read-unread status might be an example of the former, while the body of the message would be an example of the latter

How, then, should the storage systems for such applications be built? The common trend in the industry seems to be to begin with a relational database implementation and attempt to internally add to it enough features to provide adequate performance on file-system-like data In other approaches, storage engines are being built that from the outset try to address in one design all the extremes of required storage performance characteristics

We believe that these approaches are misguided This belief stems from our observation that the underlying supporting data structures and algorithms necessary to provide no-compromise performance to each kind of data are in fundamental design tension By the time the database is enhanced enough to acquire state-of-the-art file system performance, one will have in effect implemented a file system inside the database and have reused very little if any of the existing database infrastructure Further, in the give and take of engineering tradeoffs, with this approach it is easy to fall short of this goal, leaving the performance of the storage engine for file system data less than optimal.

Rather than attempting to build one monolithic storage engine which addresses the needs of all applications, why not use several engines together, each tuned for a particular kind of data? For example, one could begin with both a high-performance relational database implementation and a separate high-performance file system implementation, then enhance each with just enough additional infrastructure and functionality so that they can be cooperatively used together by one application

To accomplish this, an example of one of the important new feature necessary for the file system is that of transaction support. However, once this is done the most important addition, necessary in both systems, is the design and specification of a component architecture by which a transaction in each system can be coordinated by a third party, a transaction manager component

In the past several years, we have gained much experience in designing, building, and deploying successful architectures involving similar component design and integration problems Based on that experience, we believe the approach articulated here to the storage integration problem is of relatively minor impact to the existing internal architectures and practice of database and file system design As such, we feel that it holds out a better chance than the alternatives of actually achieving an overall storage infrastructure capable of storing both relational and non-relational data with state-of-the-art performance characteristics for each

SIGMOD ' 95,San Jose , CA USA

[1] For illustrative clarity we omit here a discussion of object databases