Letter from the Special Issue Editor

When Compilers Meet Databases

Although the worlds of databases and compilers have a long history of interaction, in recent years that have come together in interesting new ways.

First, there is a resurgence of interest in compiling queries to machine code. This idea actually dates back to the original System R prototype, but was abandoned early on since it was found to be too cumbersome and slow. New technologies, such as LLVM, which allow efficient just-in-time generation of optimized machine code have greatly improved the speed of compilation. Further, CPU is often the bottleneck, not only for main-memory databases (where it is to be expected, and which are becoming increasingly important), but also for many queries on disk-based databases. Compilation of queries to machine code can lead to significant performance benefits in such scenarios. The first two papers in this issue (by Neumann and Leis from TU Munich, and Viglas et al. from Edinburgh/Microsoft Research) describe key techniques for compiling queries to machine code. The next two papers (by Freedman et al. from Microsoft and by Wanderman-Milne and Li from Cloudera) describe how the Hekaton main-memory database engine, and Cloudera's Impala system compile queries to machine code, and the very impressive real-world benefits obtained from such compilation.

Second, static analysis and dynamic profiling of programs has been used very successfully, in recent years, to find bugs and security vulnerabilities and to detect performance problems. The fifth paper in this issue, by Chaudhuri et al. from Microsoft, describes tools using these techniques to assist database developers.

Third, there has been a lot of interest of late in optimizing database applications by analysis/rewriting of application programs. Poor performance of database applications is a significant real-world problem even today. Such problems are often due to programs accessing databases in inefficient ways, for example invoking a large number of database queries, each returning a small result, where the round-trip delays for each query quickly add up across multiple queries. Database query optimization cannot solve this problem. Programmers can be asked to rewrite their code, but that can be cumbersome and error prone. Object-relational mapping systems such as Hibernate, as well as language-integrated querying systems such as LINQ exacerbate this problem since they insulate programmers from the plumbing details of database access, making it easier to write applications that access databases in an inefficient manner.

The key to improving performance in such systems is the automated rewriting of application programs to improve performance. Such rewrite systems must first analyze the database interactions of the program using static analysis, and when possible rewrite the program m to improve the database access patterns. Two of the papers in this issue, by Cheung et al. from MIT/Cornell, and by Ramachandra and Guravannavar from IIT Bombay, describe several different ways to rewrite application programs to optimize database access. Techniques described in these papers can also potentially be used to optimize accesses to Web services.

Finally, the issue is rounded off by an eloquently written paper by Christoph Koch from EPFL, who argues that the time has come for (i) building database systems using high-level code, without worrying about efficiency, and (ii) building optimizing compilers that can generate efficient implementation from such high-level code. A similar paradigm has been very successful at the level of queries, leading to the dominance of SQL, but Christoph's article makes a strong case for applying it even to the task of building database systems.

I enjoyed reading all these articles, and I'm sure so will you.

S. Sudarshan
Indian Institute of Technology, Bombay