How to Program the Next Generation of Business Applications

Invited Talk, Extended Abstract

Christian Mathis
SAP AG
christian.mathis@sap.com

Cafer Tosun
SAP Innovation Center
cafer.tosun@sap.com

Vishal Sikka
SAP Labs Palo Alto LLC
vishal.sikka@sap.com

Categories and Subject Descriptors H.2.0 [*Database Management*]: General

General Terms Design

Keywords Modularity, In-Memory Data Management

In-memory database management systems, that exploit the abundance of main memory and available CPU cores provided by current hardware architectures, allow to push data-intensive operations into the database server. In the classical three-tier architecture, which has been implemented by systems like for example SAP R/3 in the 1990s, the relational back-end was considered a bottleneck that had to be protected from compute-intensive operations. The reason was that scaling the system at the back-end tier was only considered possible by scaling up (i. e., by buying more powerful and expensive back-end hardware), and not by scaling out (i. e., by buying more back-end hardware with the same capacity). Because of the natural physical and economical limits of up-scaling, application scalability in R/3 was achieved by computing data-intensive operations at the application server layer.

Nowadays, the separation between application code running on the application server and data being stored on the relational back-end blurs, and that is a positive development. Data-intensive operations can be computed in the back-end. The reason is the ever increasing compute power and main memory size of back-end systems. Systems with up to 2560 CPU cores and 16 TB of main memory distributed across 16 blades, but appearing as one instance, are available.[1]

[1] See `http://www.sgi.com/products/servers/uv/index.html`

MISS'12, March 27, 2012, Potsdam, Germany.
ACM 978-1-4503-1217-2/12/03.

Pushing down application logic has various advantages. For example, it avoids expensive data transfers from the back-end system to the application server. It allows code simplification, because data access executed by the database system internally and no data buffering has to be implemented at the application server side. Furthermore, data-specific code and runtime optimizations can be exploited much more easily, because (at least a part of the) application code is now managed and run by the database system.

However, pushing application logic into the database system leads to challenges:

1. Which part of the application logic shall be pushed down? Can we identify design patterns?
2. How to organize the interplay between back-end and application server code?
3. How to do software life-cycle management?
4. How to analyze, profile and debug back-end coding?
5. How to do exception handling?
6. How to structure our database application logic to allow software re-use?
7. How to allow application programmers to write scalable code that makes use of the CPU cores available?

Let us consider the last question: In the past, database research and development has put a large effort into the development of scalable SQL engines, e. g., through automatic SQL optimization and parallel execution. However, SQL is in some cases not the right choice to express application semantics. If necessary, we would like to switch to procedural programming. For the procedural part of our application, we also want to achieve good scalability and resource usage. How to achieve this? An idea would be to generalize and modularize existing database-internal algorithms and data structures for easy re-use and re-combination in application programs.

When thinking about it, software modularity is entangled with *all* of the questions listed above. The solution to these questions are fundamental to us, because they characterize our programming model, i. e., the way how we design and develop efficient business applications running on our in-memory database system.