

Prof. Hasso Plattner



A Course in In-Memory Data Management

The Inner Mechanics
of In-Memory Databases

September 4, 2015

This learning material is part of the reading material for Prof. Plattner's online lecture "In-Memory Data Management" taking place at www.openHPI.de. If you have any questions or remarks regarding the online lecture or the reading material, please give us a note at openhpi-imdb@hpi.uni-potsdam.de. We are glad to further improve the material.

Research Group "Enterprise Platform and Integration Concepts",
<http://epic.hpi.uni-potsdam.de>

Implications on Application Development

In the previous chapters, we introduced the ideas behind our new database architecture and their technical details. In addition, we showed that the in-memory approach can significantly improve the performance of existing database applications.

In this chapter, we discuss how the existing applications should be re-designed and how new applications should be designed to take full advantage of the new database technology. Our research and the prototypes we built show that in-memory technology greatly influences the design and development of enterprise applications. The main driver for these changes is the drastically reduced response time for database queries. Now, even more complex analytical queries can be executed directly on the transactional data in less than one second. With this performance, we are able to develop new applications and enhance currently existing applications in a way that was not possible before. Modern applications can especially benefit from the database performance when it comes to better granularity and actuality of the processed data.

The most important approach to achieve this performance is to move application logic closer to the database. While traditional approaches try to encapsulate complex logic in the application server, with the advent of in-memory computing it becomes crucial to move data intensive logic as close as possible to the database. An additional advantage of moving data-intensive application logic closer to the database is that the amount of data that has to be transferred between the application server and the database system is significantly reduced when most of the data intensive operations are executed directly in the database system.

33.1 Optimizing Application Development for In-Memory Databases

A typical enterprise application contains three main architectural layers as shown in Figure 33.1.

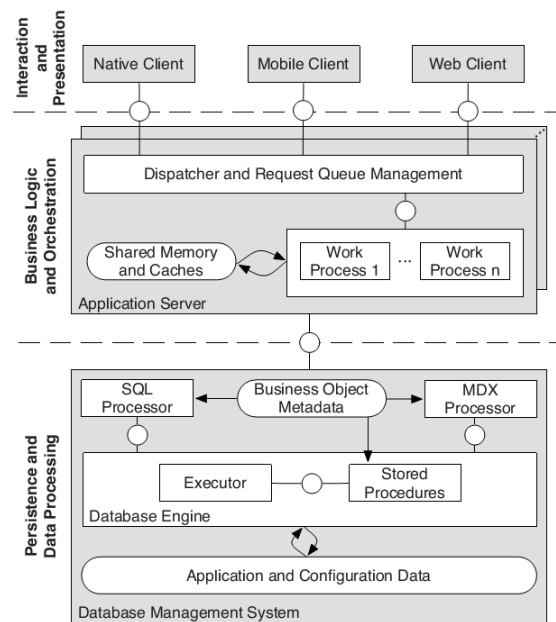


Fig. 33.1: Three Layer Enterprise Application

These three main layers are usually distributed over three independent physical systems, which leads to a three tier setup. To ensure a common understanding of the terms layer and tier, the words are shortly explained: A *layer* separates program code and its responsibility on the logical level, but it does not state how the deployment of the code looks like. The word *tier* describes the physical architecture of a system, so it gives details about the hardware setup used to run the program code.

The *interaction and presentation layer* is responsible for providing a user interface. Moreover, the presentation layer gets user information requests and forwards them to the underlying layers. The complete user interface

may consist of many different independent parts for different devices or platforms.

The business logic and orchestration layer acts as a mediator between the presentation and the persistence layer. It handles user requests obtained from the presentation layer. This can either be the direct execution of data operations by using the application's cache or delegation of calls to the persistence layer.

Data persistence and processing provides interfaces for requesting data with the help of declarative query languages, such as SQL or Multidimensional Expressions (MDX), and prepares data for further processing in the upper layers.

33.1.1 Moving Business Logic into the Database

As mentioned before, in traditional applications the application logic is mainly stored in the orchestration layer to allow easier scaling of the complete application and to reduce the load of the database. To leverage the full performance, we have to identify which application logic should be moved closer to the persistence layer. The ultimate goal is to leave only such logic in the orchestration layer that provides functionality that is orthogonal to what can be handled within the user interaction request. This reduced layer would then mostly translate user requests into SQL and MDX queries, or calls to stored procedures on the database system.

To illustrate the impact, we will explain the changes and the effects using an example that performs an analytical operation directly on the transactional data. In the following, two different implementations of the same user request will be compared. The request identifies all due invoices per customer and aggregates their amount. This is usually referred to as dunning and is one of the most important applications for consumer companies. It is typically a very time-consuming task, because it involves read operations on large amounts of transactional data.

Listing 33.1 implements business logic directly in the application layer. It depends on given object structures and encodes the algorithms in terms of the used programming language.

Using this approach, all customer data is required to be loaded from the database and an object instance for each customer will be created. To create the object, all attributes will be loaded although only one attribute is needed. After that, for all invoices of each customer it will be determined whether it is considered paid or not. For that, it is checked whether the due date at which the invoice should be paid has already passed. Finally, the total unpaid amount for each customer is aggregated.

For each iteration of the inner loop, a query is executed in the database to retrieve all attributes of the customer invoice. This causes bad runtime performance.

Listing 33.1: Imperative Implementation (Pseudo Code) of a Dunning Run

```
for customer in allCustomers() do
  for invoice in customer.unpaidInvoices() do
    if invoice.dueDate < Date.today()
      dueInvoiceVolume[customer.id] +=
        invoice.totalAmount
    end
  end
end
```

The second approach, presented in Listing 33.2, uses a single SQL query to retrieve the same result set. All calculations, filtering and aggregations are handled close to the data. Therefore, the efficient and parallelized operator implementation introduced in previous chapters can be used. The other advantage is that only the required result set is returned to the application layer. Consequently, network traffic is reduced.

Listing 33.2: Declarative Implementation of a Dunning Run in SQL

```
SELECT invoices.customerId ,
       SUM(invoices.totalAmount) AS dueInvoiceVolume
FROM invoices
WHERE invoices.isPaid IS FALSE AND
       invoices.dueDate < CURDATE()
GROUP BY invoices.customerId
```

When using small amounts of data, the performance differences are barely noticeable. However, once the system is in production mode and filled with realistic amounts of data, using the imperative approach results in much slower response times. Accordingly, it is very important to test performance of different algorithms with realistic customer data sets that represent realistic sizing settings and value distributions.

The ability to express application logic using SQL can be a huge advantage because expensive calculations are done inside the database. That way, calculations as well as comparisons can work directly on the compressed data. Only as the last step, when returning the results, the compressed values are converted to the original values to present them in human readable format.

33.1.2 *Stored Procedures*

An additional possibility to move application logic into the database are *stored procedures* which allow to reuse data-intensive application logic. The main benefits of using stored procedures are:

- Business logic centralization and reuse,
- Reduction of application code and simplification of change management,
- Reduction of network traffic, and
- Pre-compilation of queries increasing the performance for repeated execution.

Stored procedures are typically written in a special mixed imperative-declarative programming language as shown in Listing 33.3. Such programming languages support both declarative database queries, e.g. SQL, and imperative controlling sequences, such as loops or conditions, and concepts, e.g. variables or parameters. Once a stored procedure is defined, it can be used and reused by several applications. Applicability across different applications is usually established via individual invocation parameters. Our tiny example does not contain such parameters, but we could alter it so that we pass a country to the procedure which is used as a selection criterion and only customers of this country would be part of the dunning run.

Listing 33.3: Creation of a Stored Procedure

```
// Definition
CREATE PROCEDURE dueInvoiceVolumePerCustomer ()
BEGIN
    SELECT invoices.customerId ,
           SUM(invoices.totalAmount) AS dueInvoiceVolume
    FROM invoices
    WHERE invoices.isPaid IS FALSE
           AND invoices.dueDate < CURDATE()
    GROUP BY invoices.customerId
END

// Invocation
CALL dueInvoiceVolumePerCustomer ()
```

33.1.3 *Example Application*

One prominent example, where we were able to achieve an astonishing performance increase over a traditional implementation was in the area of

Original Version needed about 20 minutes

→ Factor 800x acceleration achieved

#	Operation	HANA2 Version	Variant 2	Variant 3
1	Select Open Items	0.63s	1.01s (incl. T047 & KNB5 Join)	0.6s (incl. T047 & KNB5 Join)
2	Due date, dunning level	27s	deferred to aggregation	0.5s
3	Filter 1 (Verify Dunning levels)	≈ 19s	1.1s	0.5s
4	Filter 2 (Check Last Dunning)	≈ 15s	0.8s	0.4s
5	Generate MHNK (Aggregate)	done in #1	1.2s	done in #1
6	Generate MHND (Execute Filters)	done in #1	140ms	done in #1
	Total	≈ 1 Minute	≈ 3.0s (#3, #4 exec. in parallel)	≈ 1.5s (#3, #4 exec. in parallel)

Fig. 33.2: Comparison of Different Dunning Implementations

financial applications. Here, we analyzed the dunning run, meaning extraction of all overdue accounting entries from the accounting tables. The traditional picture of the dunning run showed that the original application was implemented as follows: First, all accounts that have to be dunned are selected and transferred as a list to the application server. Second, for each account all open account items are selected and the due date for each of them calculated. Third, for all items to be dunned, additional configuration logic is loaded and the materialized result set written to a dedicated dunning table. From the discussion in the previous sections we see that this implementation is clearly disadvantageous since it executes a lot of individual SQL statements and transfers intermediate results from the database system to the application server and back. In addition, the implementation looks like a manual join implementation connecting accounts with account items.

In several iterations on the dunning implementation, we were able to reduce the overall runtime of the dunning implementation from initially 1200s to 1.5s. Figure 33.2 shows the summary comparison of these implementations. The main difference between the versions is that the fastest implementation tries to push as much selection already down to the first filter predicates and executes as much as possible in parallel. Thus, we were able to achieve a speedup of factor 800.

To summarize, in our new implementation of the dunning run we followed the principles that have been presented earlier in this section. The most important of these principles is to move data intensive application logic as close as possible to the database.

33.2 Best Practices

In the following section, the discussion of chapter Chapter 33 will be summarized by outlining the most important rules which should be followed when developing enterprise applications.

- *The right place for data processing:* This is an important decision, which developers have to make during implementation. The more data is processed during a single operation, the closer it should be executed to the database. Aggregations should be executed in the database while single record operations should be part of the application layer.
- *Avoid SELECT *:* Only really required attributes for the application should be loaded. Developers often tend to load more data than is actually needed, because this apparently allows easier adoption to unforeseen use cases. The downside is that this leads to intensive data transfer between the application and database servers which causes significant performance penalties. Furthermore, tuple reconstruction in a column-oriented data format is slightly more complex than in a row-oriented data format as shown in Chapter 13.
- *Use real data for application development:* Only real data can show possible bottlenecks of the application architecture and identify patterns that may have a negative impact on the application performance. Another benefit is that user feedback during development tends to be much more productive and precise if real data is used.
- *Work in inter-disciplinary teams:* We believe that only joint, multidisciplinary efforts of user interface designers, application programmers, database specialists, and domain experts will lead to the creation of new, innovative applications. Each of them has its own point of view and is able to optimize one aspect of a possible solution, but only if they jointly try to solve problems, the others will benefit from their knowledge.

