FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Generic patient search mechanism for ALERT® applications

Fábio Daniel Pinto de Oliveira

Report of Project
Master in Informatics and Computing Engineering

Supervisor: Maria Teresa Galvão Dias (PhD)

2008, July

Generic patient search mechanism for ALERT® applications

Fábio Daniel Pinto de Oliveira

Report of Project
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Jorge Manuel Gomes Barbosa (PhD)

External Examiner: José Luis Oliveira (PhD)
Internal Examiner: Maria Teresa Galvão Dias (PhD)

31st July, 2008

Agreement

Resumo

Num ambiente clínico é crítico que o sistema de informação que o suporte tenha tempos de resposta curtos e ainda assim disponibilize informação relevante. No sistema ALERT®, desenvolvido pela *ALERT Life Sciences Computing, S.A.* (ALERT LSC), isto é garantido pela integração dos vários fluxos de tarefas do pessoal clínico na aplicação. No entanto, com tão grande quantidade de informação crítica guardada na base de dados é determinante a existência de uma ferramenta que permita pesquisá-la garantindo mesmo assim um tempo de resposta curto.

Num sistema com as características do ALERT®, a performance é um assunto a ter em grande conta, principalmente com a carga a que a base de dados é sujeita. A grande quantidade de dados a ser comunicada a cada momento para os vários terminais disponíveis nas instituições clínicas com este sistema, o facto de nenhum dado ser apagado e de ainda estes poderem ser apresentados em várias línguas são pontos que contribuem para uma enorme carga de trabalho no servidor, principalmente a base de dados.

O projecto apresentado neste relatório tem como objectivo melhorar a performance da ferramenta de pesquisa de pacientes do ALERT®, uma ferramenta de pesquisa com critérios disseminada pela maioria dos produtos clínicos ALERT®. Para além da performance é também desejado que a solução encontrada seja flexível o suficiente para o modelo de dados diferenciado que existe entre os vários produtos.

O sistema apresenta uma arquitectura de três camadas: Flash na interface com o utilizador, Java na camada intermédia e Oracle na base de dados. Esta arquitectura permite ter quase toda a lógica de negócio encerrada na base de dados, óptimo para um desenvolvimento mais ágil e com equipas mais multidisciplinares. É então nesta camada que estão organizadas, por pacotes *PL/SQL*, as funções que constituem a camada de acesso a dados.

O levantamento de requisitos mostrou que no que diz respeito aos requisitos funcionais, a ferramenta já os respeitava de forma completa, estando os problemas concentrados na performance e na sua manutenção. O código *SQL* Dinâmico, usado para permitir a construção dinâmica das interrogações das pesquisas, torna o código mais ilegível pois este não pode ser analisado para respeitar o estilo usado pelas melhores práticas instituídas na ALERT LSC nem os seus erros verificados no momento da compilação. Outro ponto que prejudica a manutenção desta ferramenta é a não reutilização de código entre as várias pesquisas já implementadas.

Durante o estudo e a pesquisa realizados para este projecto algumas abordagens foram encontradas e avaliadas para dar resposta ao problema. A primeira consistia em encapsular o código *SQL* usado para a pesquisa num modelo que permitia a integração da contagem do número de resultados numa só pesquisa evitando a repetição de trabalho para obter este número. No entanto o modelo era muito complexo, o que aliado ao uso de código *SQL* Dinâmico tornaria a manutenção da aplicação ainda mais difícil.

A segunda abordagem era mais exótica, recorrendo a tecnologias muito recentes como índices de texto *Oracle Text* indexando várias colunas de uma *materialized view* que representa a pesquisa pretendida. Apesar de esta solução permitir a cache de resultados e uma maior liberdade na criação de critérios graças à sintaxe do *Oracle Text*, o tempo passado a refrescar as *materialized views* e o índice de texto seria demasiado para um ambiente tão volátil como o da aplicação ALERT®.

A terceira abordagem revelou ser a mais flexível e consistente. Recorrendo a estruturas de dados *PL/SQL* para guardar os dados vindos da base de dados antes de efectuar quaisquer cálculos sobre eles, esta solução permitia que nenhum esforço de cálculo fosse desperdiçado. Com os dados em contexto *PL/SQL*, as operações realizadas sobre estes não iriam sofrer com a troca de contexto que normalmente ocorre quando se chamam funções *PL/SQL* dentro de interrogações *SQL* e que trazem sempre alguma perda de performance.

Escolhida a abordagem, foram adicionadas algumas soluções extra para aumentar a segurança e a modularidade tais como o uso de variáveis de contexto, tanto para introdução dos critérios do utilizador tanto como para passar parâmetros para dentro de vistas, criadas para conter toda a lógica da pesquisa e assim abstraindo o acesso a dados do resto do processamento. Este encapsulamento também permitiu reduzir a quantidade de código *SQL* Dinâmico usado, elevando a legibilidade do código e a facilitando a sua manutenção.

Testes foram feitos à performance e à facilidade da manutenção. Os testes de performance revelam que a solução escolhida é mais rápida do que a anterior e que o uso de materialized views aumentaria significativamente a sua escalabilidade. No que diz respeito à manutenção, a reutilização de código aliada à redução de código *SQL* Dinâmico e à separação da lógica de acesso a dados das questões de formatação dos dados permite uma melhor integração na aplicação e um acréscimo no ciclo de vida útil desta ferramenta.

A solução encontrada é rápida, mais escalável e flexível à adição de novas funcionalidades. Isto permite que se possam imaginar desde já novos melhoramentos a desenvolver para esta ferramenta. Estes podem ir desde a criação de novos tipos de entrada de dados, à criação automática de documentação e código ou mesmo integrando uma ferramenta deste tipo em todas as grelhas de pacientes da aplicação para permitir pesquisas mais rápidas e mais próximas do utilizador.

Abstract

In a clinical environment little response time and relevant information delivery are critical subjects. In ALERT®, developed by *ALERT Life Sciences Computing, S.A.* (ALERT LSC), this is guaranteed by integrating main clinical workflow tasks into the application. However, with such an amount of critical information stored it is important to have a tool that enables to search for it and still guaranteeing a short response time.

In a system like ALERT®, performance is a matter of great importance, mainly because of the load on the database. The quantity of information being traded to the several terminals, the fact that no data is deleted from the database and that all text data can be shown in several languages are very important reasons for the enormous work load on the server, mainly in the database.

This project has the following goal: increase the patients search tool performance. This tool enables criteria based search and it is already disseminated through several ALERT® products. Besides the performance improvement, it is also desired that the solution found is flexible enough for the differentiate data model that exists between each product.

ALERT® system has a three layer architecture: Flash for the user interface, Java in the intermediate layer and Oracle for the database. This architecture allows having almost all the business logic within the database, great for a more agile development with multidisciplinary teams. It is in this layer that all the data access layer functions are. They are organized by PL/SQL packages.

Requirements specification revealed that the tool had already all the functional requirements fully addressed being all the problems on the performance and maintainability. Using Dynamic SQL code made code unreadable because it couldn't be 'beautified' and its errors could only be detected at runtime. Maintainability also suffered from little code reuse through the multiple searches already developed.

During the research some approaches were designed to face the problem. The first one consisted in encapsulating query code inside a given model that allowed to integrate the results counting in a sole search preventing from repeating any effort. But the query model was too complex and would turn maintenance tasks even more difficult.

Second approach was a more exotic one, making use of very recent technologies like Oracle Text indexes to index multiple columns from a materialized view, representing the desired search. Although this approach would allow results caching and more freedom on criteria creating due to Oracle Text syntax, materialized views refresh and index rebuild times wouldn't work for a volatile environment like ALERT®.

The last approach revealed to be the most flexible and strong. Recurring to PL/SQL data structures to store data from the database after any calculation allowed to seize any calculation made. With data being in PL/SQL context any calculation over them would not suffer from context switching preventing some performance loss.

After the approach was chosen some additional solutions were added to improve security and modularity like context variables use, both for entered criteria and for passing parameters to a view hiding data access details. This encapsulation also reduces Dynamic SQL code improving code legibility and also maintainability.

Tests were runt to performance and to maintainability. Performance tests revealed that the final solution is faster and that the use of materialized views would increase its scalability. For what concerns to maintenance, reusing code, reducing Dynamic SQL code and data access details encapsulation allowed to a better tool integration and a larger life cycle.

This solution is fast, more scalable and flexible to new functionalities addition. This allows thinking of new improvements for this tool: new criteria types; automatic documentation/code; or even integrating a tool like this in all ALERT® patients grids to enable quicker searches.

Acknowledgements

I would like to thank my parents, Orlando and Carolina, and my girlfriend, Leonor, for their support during this project development and for understanding my commitment to this even through the hard times that passed.

I would also like to thank all my friends and colleagues, specially Diogo, Nelson, Maia, Bruno, Gonçalo and David, who also developed their projects in ALERT LSC and shared their thoughts, smiles or laughs, Eiras and Brito for their precious help and patience and Neves who always had the right words for me.

I must not forget to thank José Silva, my manager, who was always able to get me the help I need for my research and development and for being always by my side. I would also like to thank my supervisor, Maria Teresa Galvão Dias, for being so patient with me and for being able to make me look on the practical side of things.

Finally, I'd like to thank ALERT LSC, in the person of its human resources director Maria João Oliveira and FEUP, in the persons of Raúl Moreira Vidal and António Augusto de Sousa for having made this possible.

The author

“Nothing in life is to be feared. It is only to be understood.”

Marie Curie

Contents

List of Figures

LIST OF FIGURES

List of Tables

Glossary

DBMS	Data Base Management System
IDE	Integrated Development Environment
PL/SQL	Procedural Language/Structured Query Language
SGA	System Global Area
SQL	Structured Query Language
URL	Uniform Resource Locator
XML	eXtensible Markup Language

Chapter 1

Introduction

"Information only exists when it reaches someone who understands it."

In an information system it is of utter importance that the data is available to the person who needs it when he needs it. Today, though, we cannot speak of information systems without talking of computer information systems and information technology.

These systems make use of current technology to store, deliver and process data in much faster, more complex way and more reliable way than paper based information systems.

Computer information systems can find in databases (more commonly relational databases) a great tool to store, deliver and also process data. The developments in database related areas in the latest years allowed users **1)** to have a more comprehensive search language (or query language, SQL, which is the international standard for relational databases language [JCC05]); **2)** new semantic search capabilities and **3)** more performance tuning solutions.

So, when we think of computer information systems and their databases we need to think of the way their users can search information. Here we need to take into consideration the user needs in terms of data format and response time. Information can never come to exist if these needs are not respected. We can never call information neither to data which we don't understand nor to outdated data.

1.1 Problem Contextualization

ALERT Life Sciences Computing, S.A. (ALERT LSC) is a Portuguese company fully dedicated to the development of clinical software solutions. Created in 1999 under the name *MNI- Médicos Na Internet, Saúde na Internet, S.A.*, ALERT LSC developed its own suite of healthcare solutions, ALERT®, adopted by 8,575 institutions all around the world.

ALERT LSC's success comes from the innovative nature of its products and its user base stands over 35,000 certified trained users [ALE08], most of them clinical personnel.

Its paper-free philosophy simplifies the healthcare process but also means that it must stand over a reliable system with complete data availability. Due to its clinical nature the system must also have great responsiveness and be designed to give its users the most relevant information for its current activity. This means that all the healthcare process workflow must be present in ALERT® products.

With such a great amount of data a search tool is of great relevance allowing users to search within the database for the information they need. A tool like this has to respect the same requirements above: responsiveness, availability and reliability.

The ALERT® PAPER FREE HOSPITAL suite contains 8 products supported by the same database in each institution. This represents an enormous amount of data circulating between clients and the server at each moment. Plus, we should also consider that the business logic is as well inside the database. The combination of these factors have put the performance on a fragile situation.

1.2 Project

Every ALERT® application counts with a patients/episodes search area where users can specify multiple criteria to filter the results. This tool was differently programmed between products leading to bad maintainability and bad performance also.

The proposal was to optimize and modularize the ALERT® search infrastructure so as to allow every product to have it integrated into it.

The project here presented was proposed by ALERT LSC to Fábio Daniel Pinto de Oliveira in the context of his course final project. In order to do that, he was allocated in ALERT® EDIS team - ALERT® emergency department product - because of some particular characteristics:

- It is one of the most used products and it is also one of the most affected by the high database load since it integrates data from almost all of the other ALERT® products;
- Patients search has already got some development on it since it's a critical point in this product giving a good starting point to this project. However, performance still had a long way to go to be satisfactory.

Despite these facts, this search tool is present in almost all ALERT® clinical products.

1.3 Objectives

The project objectives are listed next:

- To allow the system to choose the best way to access information based on user criteria leading to maximum optimization;
- To achieve high performance;
- To achieve high security;
- To achieve high reliability.

This shall be done making use of the more innovative means available that guarantees data base efficiency and high performance. This includes PL/SQL code, indexes, views, materialized views and any other objects that can help in this task.

The project complexity lays on differentiate data model, sometimes between different products and on the complex algorithms behind the data base engine.

1.4 Report Structure

This report counts with 6 chapters being the first chapter this one. On chapter 2 it is possible to find the state of the art with a list of development tools/methodologies. It is possible to read through some findings and useful concepts for the global report understanding. Chapter 3 describes the analysis made for this project in search for the better solution from three different approaches. On chapter 4 a more detailed description of the chosen solution will be available as well as some development details. The solution assessment for performance and maintainability will be made on chapter 5. The report ends on chapter 6, Conclusions, giving an overview of the project, detailing on its highlights and previewing some possible future developments.

Chapter 2

State of the Art

In this chapter will be presented the state of the art and the development methodologies used throughout the research and development of this project.

There are core concepts needed to understand the rest of the report and the presented ideas.

2.1 Technological Revision - PL/SQL Developer

This section serves to show the main tool used through the development of this project. As all the project was done on the database layer of the application a single tool was used for all the development and study.

PL/SQL Developer is an IDE developed by AllRound Automations [AA08] for the development, testing, debugging and Oracle database code optimization. This tool allows creating and editing every Oracle database object i.e.: procedures, functions, packages, triggers, views, tables, indexes.

It is possible to highlight some important features:

- Breakpoint debugging;
- Code complete;
- Command line;
- Context help;
- Database objects navigation;
- Explain plan[1] window;

[1]Explain plan is going to be explained below.

- Multi-session environment;
- PL/SQL code beautifier[2];
- Plug-in architecture;
- Query constructor;
- Visual schema diagrams.

For the project performance component some specific features were used. These will be reviewed next.

PL/SQL Profiler

The PL/SQL Profiler tool is based on the DBMS_PROFILER [ORA03] package introduced in Oracle's DBMS 8i version. This package allows profiling PL/SQL code for detecting bottlenecks retrieving the number of times that each line was runt and the time statistics for each line.

This PL/SQL Developer tool is a graphical tool for the DBMS_PROFILER package. Here it is possible to see the code, line by line and watch the associated statistics. It is also possible to filter code by package. Time spent in each code is also displayed in bar style to help detecting the lines that took most time.

Explain Plan

Explain plan [Ora02c] is a SQL command that shows the plan made by the Query Optimizer to run the desired query listing the accessed tables, the join methods and a cost for each operation. This is the cost used by the optimizer to choose the best plan to run a query, choosing the plan with the lower cost.

PL/SQL Developer has also a visual interface to this where it is possible to see the various join operations in tree form and its associated costs and join methods.

2.2 Oracle Text

Formerly known as Intermedia, Oracle Text [Ora07] uses standard SQL to allow searching for keywords in a great amount of text. Oracle Text indexes text in tables and files so that it is possible to find any keyword in a text in a very fast way.

It then delivers search results in various formats e.g., raw, HTML, PDF. Oracle Text also supports multiple languages and ranks results by relevance for those languages by applying some strategies like context search, Boolean operators, pattern matching, HTML/XML

[2]Beautify is to parse the code to make it look the same by applying the same style.

section searching, etc. It already supports meta information indexing for new applications for a semantic web [W3C01] approach.

Oracle Text has several ways of indexing text [Ora05] depending on the kind of information you want to store or the way you want to access it.

2.3 Concatenated Datastore

A Datastore [Ora01b], in Oracle context, is a way of defining how the data is to be indexed by Oracle Text. This can be used to include XML tags in text or index multiple columns from a table or multiple tables.

The default datastore is the DIRECT_DATASTORE which creates a document for each row in a single table column. This is the way that text is indexed without specifying any specific datastore. But when it is needed to search in multiple columns it is better to have a single index for those columns from a performance point of view [Ora02b]. It is possible to do this using MULTI_COLUMN_DATASTORE which creates a document for each row in multiple columns in a table.

As other indexing methods are available both for indexing data in files or at a URL there is also a method to create the document to index by defining the procedure that synthesizes the data called USER_DATASTORE.

A useful Datastore created by Roger Ford from Oracle Corporation is the CONCATENATED_DATASTORE [Ora00a]. This Datastore is a USER_DATASTORE that allows indexing multiple columns in a single table in an Oracle Text index. Each column can be associated with a XML tag allowing to use the WITHIN operator; it is possible to index numeric fields and timestamps and do range search on them based on a Friedman's [BFM78] algorithm for range searching; it hides the logic to create a procedure that synthesizes the data; and allows creating a trigger to update the index whenever any of the selected columns are updated. A performance study of the CONCATENATED_DATASTORE [Ora00b] reveals the improvements against mixed queries (queries with Oracle Text indexes and regular indexes) reaching 34 times improvement on a mixed query with range search.

2.4 Oracle Optimization Techniques

This chapter has the purpose to document some of the ways to develop database applications with special attention to performance. Those readers without experience on SQL or PL/SQL language may have some difficulty understanding this particular chapter.

Note that these techniques were learned through on-line documentation, discussions with colleagues and through Oracle tools to evaluate results.

2.4.1 Pre-Aknowledgements

In order to optimize PL/SQL, it is necessary to understand some concepts related to Oracle and databases in general.

DML, DDL, DCL and SELECT [Kya06]

DML DML means data manipulation language. It includes the following SQL commands: insert, update, delete and merge and allows modifying the table's content.

DDL DDL means data definition language and its commands embrace create, alter, and drop allowing to create objects in the database.

DCL DCL means data control language and includes grant and revoke commands. A DCL command we can define in some ways user's permissions to execute DML or DDL.

Select is not included in these definitions, since it does not change database status by itself, it is just a reading command.

Context switching

Inside a typical PL/SQL block it is allowed to query using a simple select. Oracle has a very modular architecture, so the process executing PL/SQL is usually different from the one executing SQL. To step the queries SQL commands to another one (DML, DCL, DDL or select), these two processes have to communicate. This can be made in several ways, like pipes or shared memory but the creation of a communication channel between processes and the crossing information has got performance losses. So, in PL/SQL, it is advisable to minimize the number of SQL commands from a PL/SQL block or PL/SQL function calls from SQL code. This technique may bring a simpler PL/SQL code [Feu06] and avoid the accumulation of logic inside the commands SQL, which get more and more complex.

Cursor

A cursor [HP92] is a data structure which allows the process of multiple rows returned by a select. A cursor, when declared is associated to a SQL query. In order to read the cursor's contents, opening it is needed, as well as reading it row by row and then close it, so as to allow the server to clean the allocated resources, so the cursor can still be in the memory. Once it's opened, it is asked to the database management process to read the cursor's result. This process keeps in memory the ActiveSet, which is a data structure that contains query's results. An ActiveSet can be iterated row by row and has information about the type of data of the returning columns. Each fetch from cursor asks the management process to return the actual row of the ActiveSet as well as to point to the nest row. When it reaches the end of the cursor, further fetches always return NULL for all the columns and the property %NOTFOUND of the cursor's variable returns TRUE. To read exactly one row, it is used select/into which expects just one row to be returned. If not, it throws an exception. The following code gives an example of the cursor's use.

Bulk binding

As seen before, cursors can read SQL query's results row by row which brings to the application the context switching load and so, performance problems in very concurrency applications. Bulk binding lets PL/SQL and SQL to exchange a huge amount of rows. Oracle soon knew this problem and so, in 1997 decided to include in version 8 the possibility to exchange several rows between the two (PL/SQL and database) contexts. There are two situations to know: carriage of several lines to the database at the same time and the reading of some Data Base rows to PL/SQL context [Feu06]. The first situation is done through forall command and the respective DML. Forall allows to look over a PL/SQL varray or nested table through a cycle, doing a DML command in each iteration. Varrays and nested tables can be compared to arrays in other programming languages. It is possible to loop in PL/SQL using loop, but forall is different from for+loop because forall is a DML itself and is executed differently in the Data Base, so, there is no context switching in each loop iteration. For Data Base rows reading, there is bulk collect letting a query result to be in a varray or nested table simultaneously instead of iterating over a cursor and read ActiveSet row by row. But if it is desirable, it is possible to iterate over the resulting array or nested table.

SQL Optimizer

Every SQL queries before their execution are evaluated by their structure and by statistics preserved by the database about its tables. From this evaluation results an execution plan [MG02]. The used tools have functionalities such as execution plan analysis of a specific query. This execution plan parts in several algorithms to look into the tables and put together the intermediate results. Typical row search algorithms on tables are, by decreasing performance order, index, unique scan, index range scan, index fast full scan, index full scan and table access full. Depending on this algorithms, there are other ones to put together the resulting table rows like hash join, nested loops or merge join Cartesian. Depending on the amount of returning data, each one of this algorithms has different performances, The fastest, based on indexes are really efficient when it's desirable to read few lines or access by primary key. However, for a huge amount of data, these algorithms seriously degrade performance, once they use complex data structure, such as B-trees. So, in this cases, it is advisable to use table access full which has the complexity O(N), being N the number of lines in the table. The algorithms to joint intermediate results are not always controllable because they directly depend of the number of returned rows and the previous used algorithm to obtain them. These amount of intermediate data contain structure and context information that is necessary to query evaluation.

Views & Materialized Views

View A view is a type of an object which only contains a generic SQL query without variable dependencies and they can be used transparently in other SQL query like if they were a table. A view allows putting together all the code so that reutilization is easier. When a view's column is not used, the SQL optimizer knows how to ignore that column and all the necessary calculation algorithms;

Materialized View [Bur02] [Ora02a] A materialized view is used in the same way as a view but has got a different behavior. While a view is evaluated through execution time, a materialized view is pre calculated and its results are recorded in a cache, which is, internally, another table. But to do this pre-calculation, the materialized view has to be explicitly refreshed by package dbms_mview methods which is part of an Oracle database installation by default. A materialized view can be also be refreshed periodically and automatically or when a commit is executed, changing core tables. Materializes views can be normally used in databases with very similar great amount of data and whose reliability is not important. They can also be used to maintain remote data tables available DB links because there is delay between creation of data and their availability that has to be balanced with application performance. Materialized views refreshing can be complete or fast. A complete refreshing is done when a materialized view was not already calculated or when doesn't have the chance to know the changes made in the core tables. It's advisable to avoid complete refreshing so as there are huge performance repercussions because it has to reevaluate the entire query again. So, to allow quick refreshing materialized view logs are created in the core tables. Internally, they create tables where done table DML's are recorded Then it's necessary to change the view's query so that it can obey to some restrictions such as to return rowids or primary keys and so, refreshing used on materialized view logs can access to the modified rows using the index and query's structure is not very complex. Every rules about refreshing materialized views are documented in Oracle9i Data Warehousing Guide. Optionally, performance of SQL queries can be improved over the materialized view by creating indexes in their columns.

2.4.2 PL/SQL Optimization

Avoid Cursors [SF01]

Cursors, comparing to select/into seems to be more flexible because they don't throw exceptions depending on the number of returning rows and allows to group every SQL queries on the PL/SQL block declarative zone. However, each command evocation about a cursor has the cost of a context switching besides it silently fails when it was expected just one row. It is rather to choose an explicit exception in case of an error because it allows to control the application running and detect errors in a better and easier way. As seen before, a cursor has to be declared before PL/SQL blocks in the declarative zone which difficult reading of big code blocks because every time a cursor is found, to know exactly what it returns, it is necessary to go back to the declarative zone and analyze the query. At last, cursors can only be treated row by row, causing context switching problems again. So, is better to read results all at one by using bulk collect and so avoid several information requests to the database. It should be noted that bulk collect is used directly with select/into which also avoids creating ActiveSet and mitigate an extra weight in the database.

Use and abuse on explain plan [MG02]

This functionality is very important and every developer must do an effort to understand it. Optimizing query's performance requires knowing the algorithms used in some detail and explain plan allows knowing the algorithm used for each query join. Manipulating the explain plan to reach the best plan is a main task for every performance expert. Explain plan punctuation a great clue to follow the right lead as it returns punctuation to the algorithms used by the optimizer and to the amount of involved data. The lower the punctuation the higher SQL performance.

2.5 Conclusions

The available tools to performance optimization on Oracle databases are of great number being difficult to know all of them and to be able to choose the appropriate one for a given task or problem.

Oracle documentation on performance lacks detail and is by searching books, discussions and opinions from Oracle *gurus* like Steven Feuerstein and Tom Kyte and millions of Oracle users around the world that it is possible to learn about the most recent techniques available to address performance problems on Oracle databases. This require a very hands-on approach study as all experts advise not to take their conclusions for certain and test each one for each specific case.

Most of this project study was over SQL and PL/SQL optimization techniques ending in a somewhat profound knowledge about the database internals, its indexing mechanisms and the available optimization methods.

This knowledge was very useful to develop distinct approaches to the project's problem with distinct technologies and concepts and later for optimizing each specific search that was integrated in the chosen architecture.

Chapter 3

Problem Analysis

This chapter gives an overview to the problem and its technological details with the goal of showing some alternatives after describing the final solution, in the next chapter.

First we start to get an overview of the application architecture where it is possible to learn about its layers and the communication between them.

It will be possible to have a detailed view of the database layer and its organization. It will be done some digging down the tool specifics, learning about its previous state.

The requirements appear next as a result of the previous analysis to the application needs. These will be used to evaluate the former state of the tool.

Next there are presented some approaches to the problem solution which are then compared with each other by terms of requirements fulfillment.

3.1 General Architecture/Design

In an institution that have adopted ALERT® the first thing we are able to notice is a number of touch-screen enabled screens around. These computers allow clinical personnel to make track of their tasks and register all the workflow. Its patented user interface is designed to help and be intuitive even to those who are unfamiliar to computers.

ALERT® applications can be set to work with printers and/or image scanners. It is also compatible with existing billing systems.

3.1.1 Overview

The following sections will give an overview over the ALERT® architecture and design solutions.

Network

Every clinical product from ALERT LSC lies over a client-server network architecture like the following (figure 3.1).

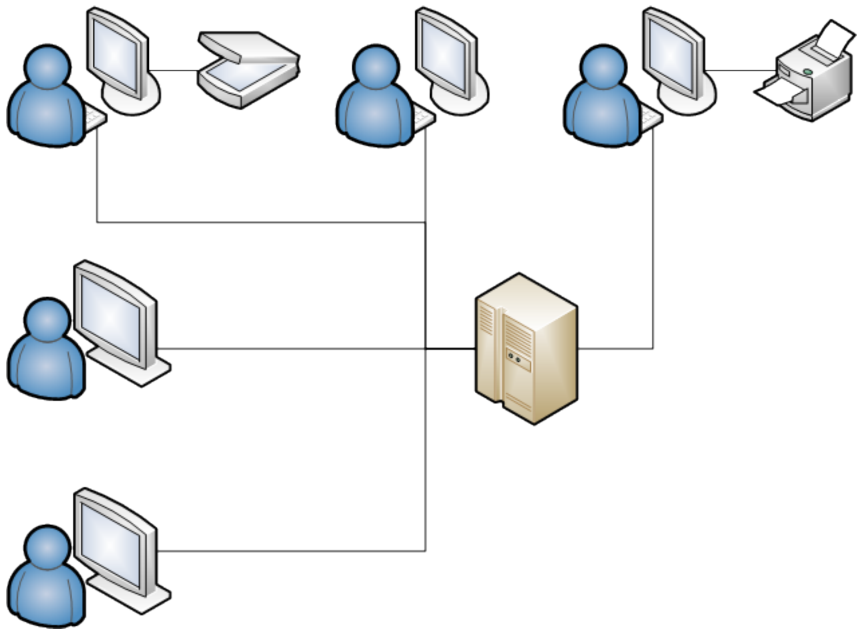


Figure 3.1: ALERT® Network Architecture

There are a number of terminals distributed in the clinical institution where every professional can log in with its fingerprint. From there he can access every information he is allowed to as every computer is connected to the same server inside the same institution.

Grids

One common design resource in ALERT® applications are grids. Grids are used through several places in the user interface as they serve as a useful and ready way to show information to the user. They are more commonly found in the entrance screen to help professionals finding the list of the patients currently in the emergency department, the current tasks list or the physician's patients (figure 3.2), just to name a few.

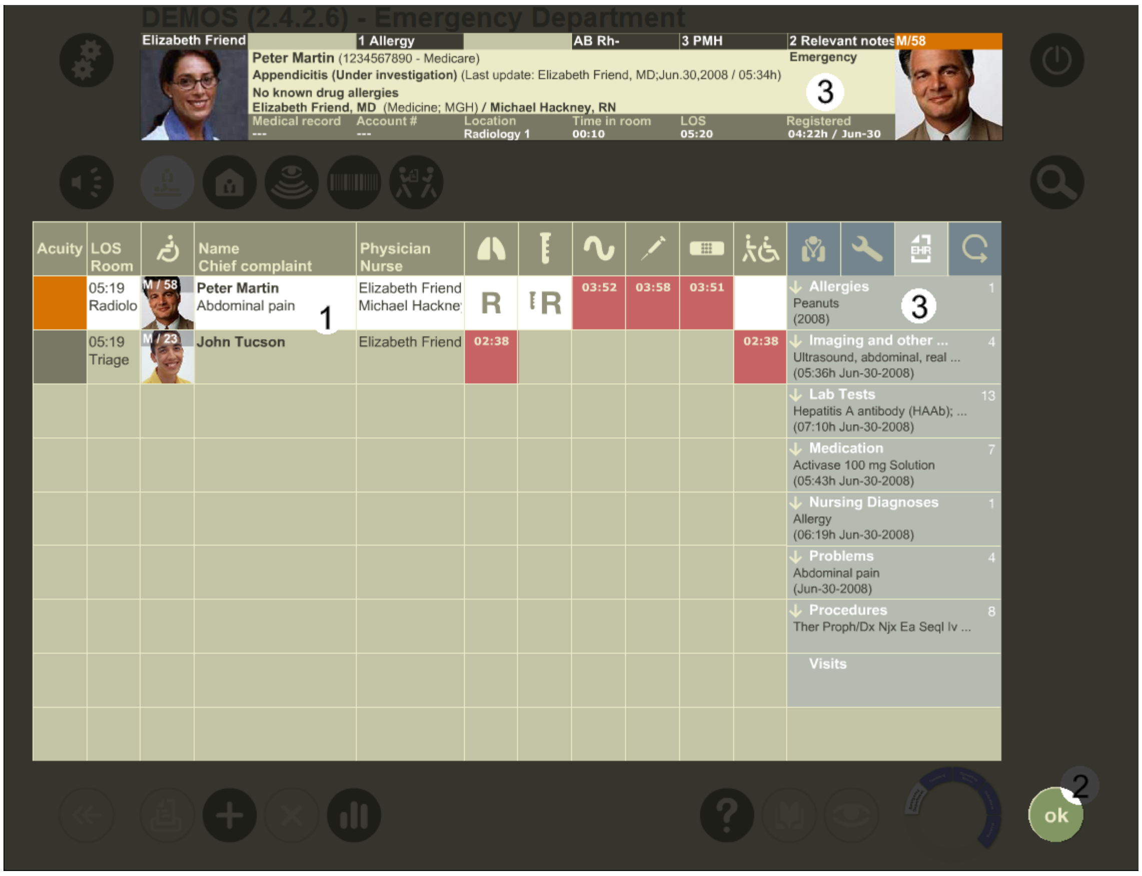
These grids have several lines, e.g., one per patient, each one with relevant information (1). They are fully navigable (2)[1] and show context information once selected (3).

Grids are also used to list the patients search results.

Layered Communication

The user interface is designed in Flash® and the communications between each client and the server are done through Flash Remoting (figure 3.3) (1).

[1]Pressing ok button at this point would take the user to patient health record screen.


DEMOS (2.4.2.6) - Emergency Department

Elizabeth Friend | 1 Allergy | AB Rh- | 3 PMH | 2 Relevant notes | M/58

Peter Martin (1234567890 - Medicare)
Appendicitis (Under investigation) (Last update: Elizabeth Friend, MD;Jun.30,2008 / 05:34h)
No known drug allergies
Elizabeth Friend, MD (Medicine; MGH) / **Michael Hackney, RN**
Emergency

Medical record	Account #	Location	Time in room	LOS	Registered
---	---	Radiology 1	00:10	05:20	04:22h / Jun-30

Acuity	LOS Room		Name Chief complaint	Physician Nurse						
	05:19 Radiolo	M / 58	**Peter Martin** Abdominal pain	Elizabeth Friend Michael Hackne	R	R	03:52	03:58	03:51	
	05:19 Triage	M / 23	**John Tucson**	Elizabeth Friend	02:38					02:38

Allergies Peanuts (2008)	1
Imaging and other ... Ultrasound, abdominal, real ... (05:36h Jun-30-2008)	4
Lab Tests Hepatitis A antibody (HAAb); ... (07:10h Jun-30-2008)	13
Medication Activase 100 mg Solution (05:43h Jun-30-2008)	7
Nursing Diagnoses Allergy (06:19h Jun-30-2008)	1
Problems Abdominal pain (Jun-30-2008)	4
Procedures Ther Proph/Dx Njx Ea Seql Iv ...	8
Visits	

Figure 3.2: Grid Example

With a Flash Remoting module in server side, all requests are translated to a service call to Java (2). By its turn, Java has that service mapped upon database functions and takes structured data from cursors (3). This data is then sent back to the client in order to fulfill its request.

3.1.2 Database

As ALERT® business logic resides mostly in its database it is important to learn about its structure. There will be no data model details observed as this is not relevant to this project development. However it will be possible to learn about its data abstraction layer organization and some technical details like data localization capabilities and performance.

Data Abstraction Layer

Every service referenced by Java calls a database function to perform some action on the database. The response can be received both from the output parameters and the function return. Receiving a complex data structure is only possible through a cursor due to the library used to communicate with the database.

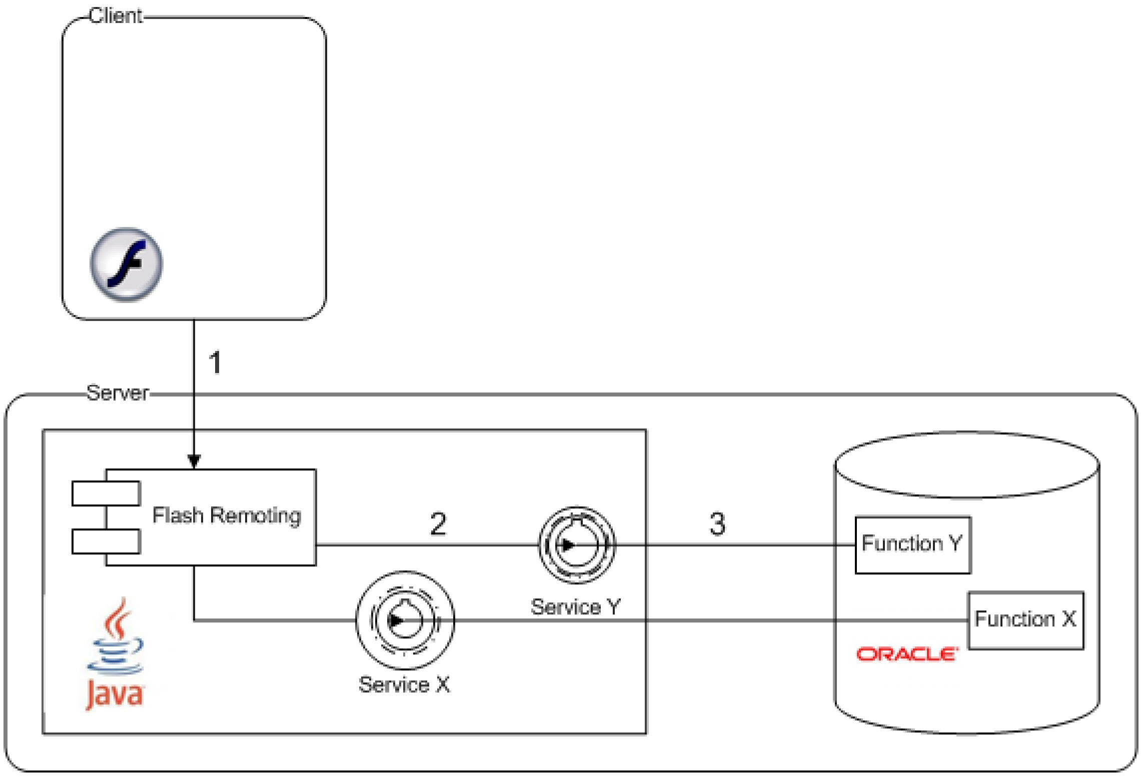


Figure 3.3: Layered Architecture

Database functions are all grouped inside specific packages. The grouping is made by context to optimize the package loading into cache whenever a function is called.

Almost all the data abstraction is based on PL/SQL functions. This is useful for developing in big, multidisciplinary teams but makes no use of the set evaluation of the SQL Engine and the SQL Optimizer leading to less performance.

Data Localization

All the data inside the database that shall appear on the user interface can be localized[2]. This is done for text strings and timestamps:

- Each text string that is to be shown to the user can be localized as a unique reference is stored in a table for translations. This table stores several translations for each reference, one per supported language, so that each user can be free to choose his language in the application;
- Each stored date or timestamp is automatically converted in comparisons between dates or timestamps. There are also specific packages for dates and timestamps conversions. The timezone is defined per institution.

[2]Translated to a given language or different timezone.

Performance

Performance in ALERT® is a critical issue for it is a real-time application and due to its decision support role.

Because no data is deleted some databases have thousands of episodes stored. This represents a heavy load in the database in most of the grid loads and other data intensive queries.

Some techniques used to improve database performance are:

- Materialized views;
- Updated index statistics.

Despite these techniques some functionalities remain slow and performance is still an issue to deal.

3.1.3 Patients Search Tool Former State

At the project beginning the patients search tool was already implemented in every clinical product.

This section aims to capture the initial state of the tool at the time the project began covering every relevant aspect and addressing them to the requirements listed above.

Overview

The patients search tool is one of the main use cases available to users (figure 3.5)[3]. It can be accessed directly from the main screen (figure 3.4).

It allows any professional to find relevant patient information by simply entering some criteria to filter the information. Then, after he gets the results, he can change the criteria entered and execute other search or navigate from the results to get more patients details.

The information is then presented to them in the form of a grid that allows navigating to a patient's health record by simply selecting him.

In figure 3.6 there's an activity diagram that illustrates this.

This diagram starts with the user entrance in the patients search screen and ends whenever the user exits the search screen either by choosing other option or navigating to user's health record.

Showing Results

Each search has a product-specific grid showing only relevant information for its context. This grid is filled with information retrieved from the database and lets users order columns by its value.

[3]This fade out effect serves to highlight the use case to be studied and show that there are other use cases also.

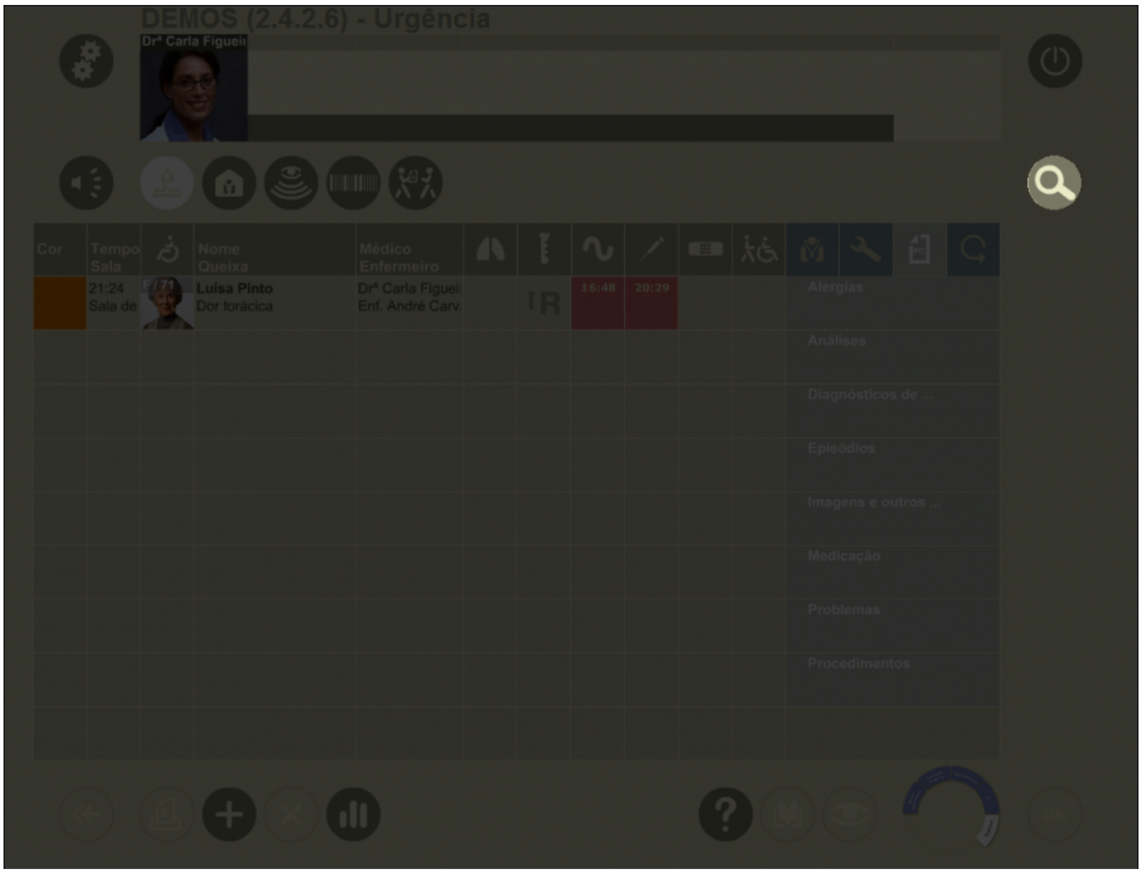


Figure 3.4: Search Button Localization

Usually, in each patient's search the user is presented with a grid containing multiple rows each per patient's episode in the institution. These grid's characteristics are most of the times reused from the main product grids.

For collecting the results there is a specific function for each search (although they can be reused through products) with a common structure:

1. Based on the parameters chosen by the user, construct a string that can be appended to the WHERE list of a query;
2. Run a SELECT against the database to count the number of results. This query can be simpler than the main query because it doesn't need almost none of the outer joins (those that don't affect the final number of results). This query must append the parameters string;
3. Test the number of results and set any necessary message;
4. Run the main query. This query must append the parameters string too;
5. Return the results cursor.

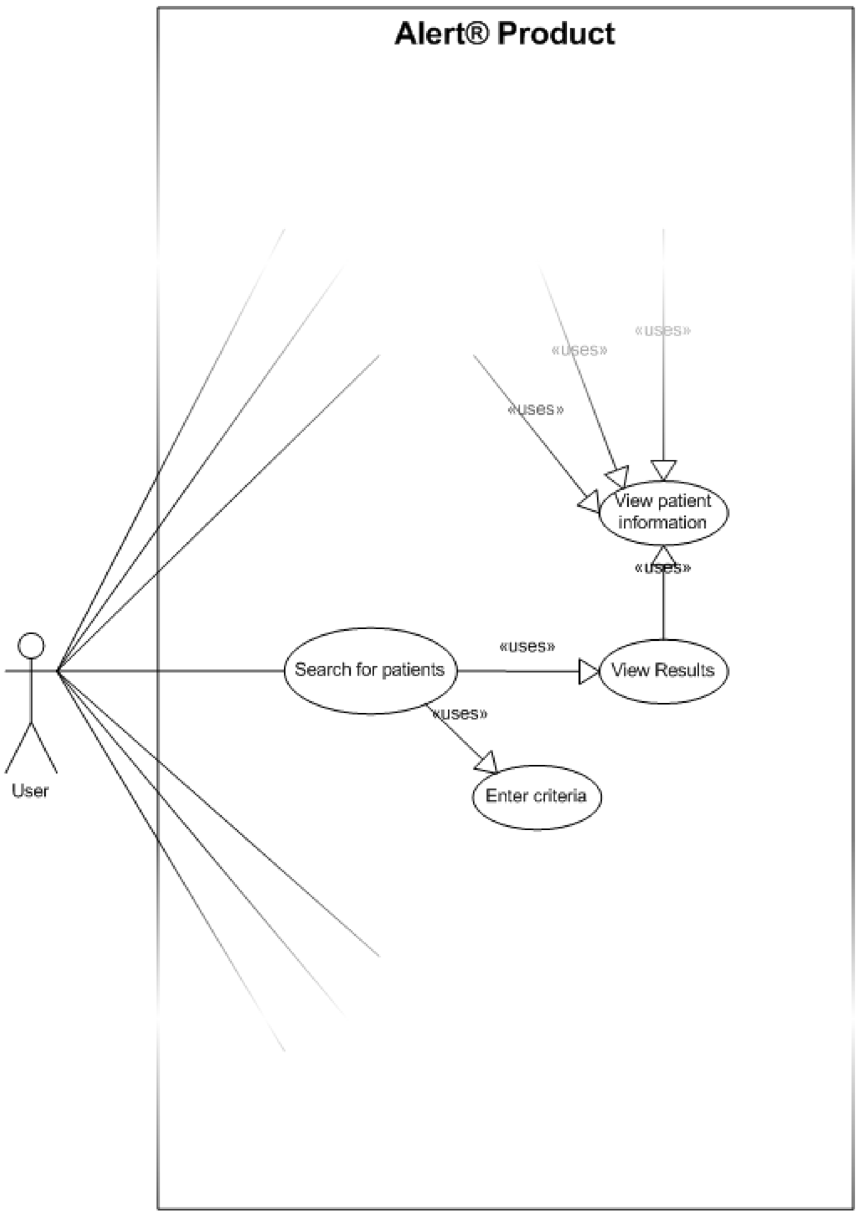


Figure 3.5: Patients Search Use Case

This structure allowed the user to choose whether results should be filtered by patient's name, by date of birth or/and by episode number (just to name a few). This also respected the main architecture which requires that any structured data from a database function should be passed inside a cursor to the Java layer.

Although every search function share this structure the code is repeated for each one of them making it hard to maintain or modify.

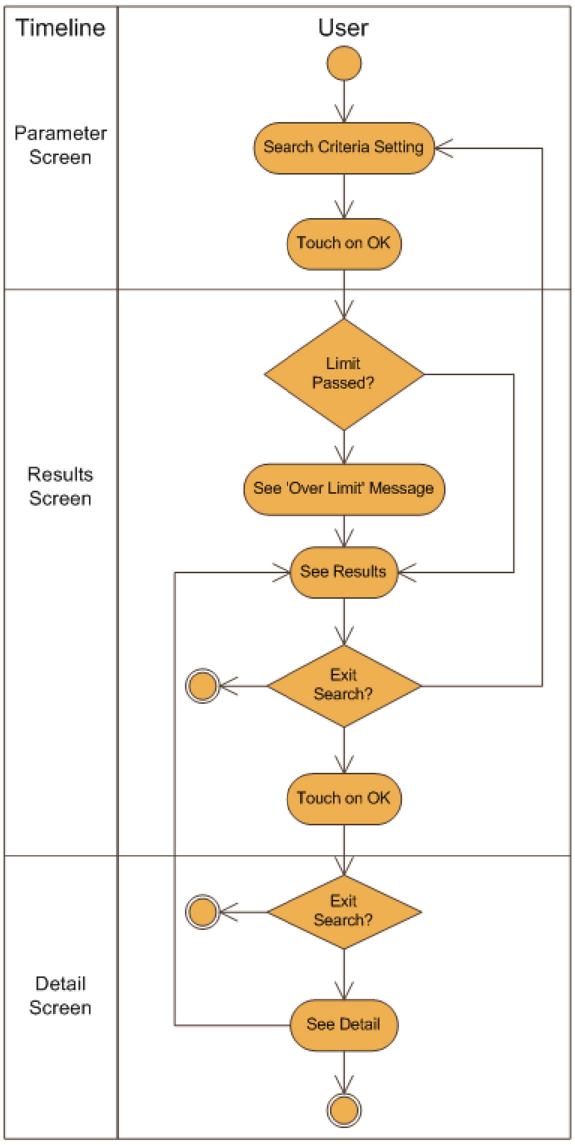


Figure 3.6: Activity Diagram

Entering Criteria

The first screen a user sees when he enters a search screen is a list of criteria to be filled. This screen is constructed in run-time with the criteria set information he gets from the database and calls the screen responsible to show the results after the user presses the 'OK' button.

The filled criteria list is passed to the following screen and posteriorly passed to the function that gets the searching result where these criteria are checked.

Each criterion can be of a type:

- Text;
- Numeric;

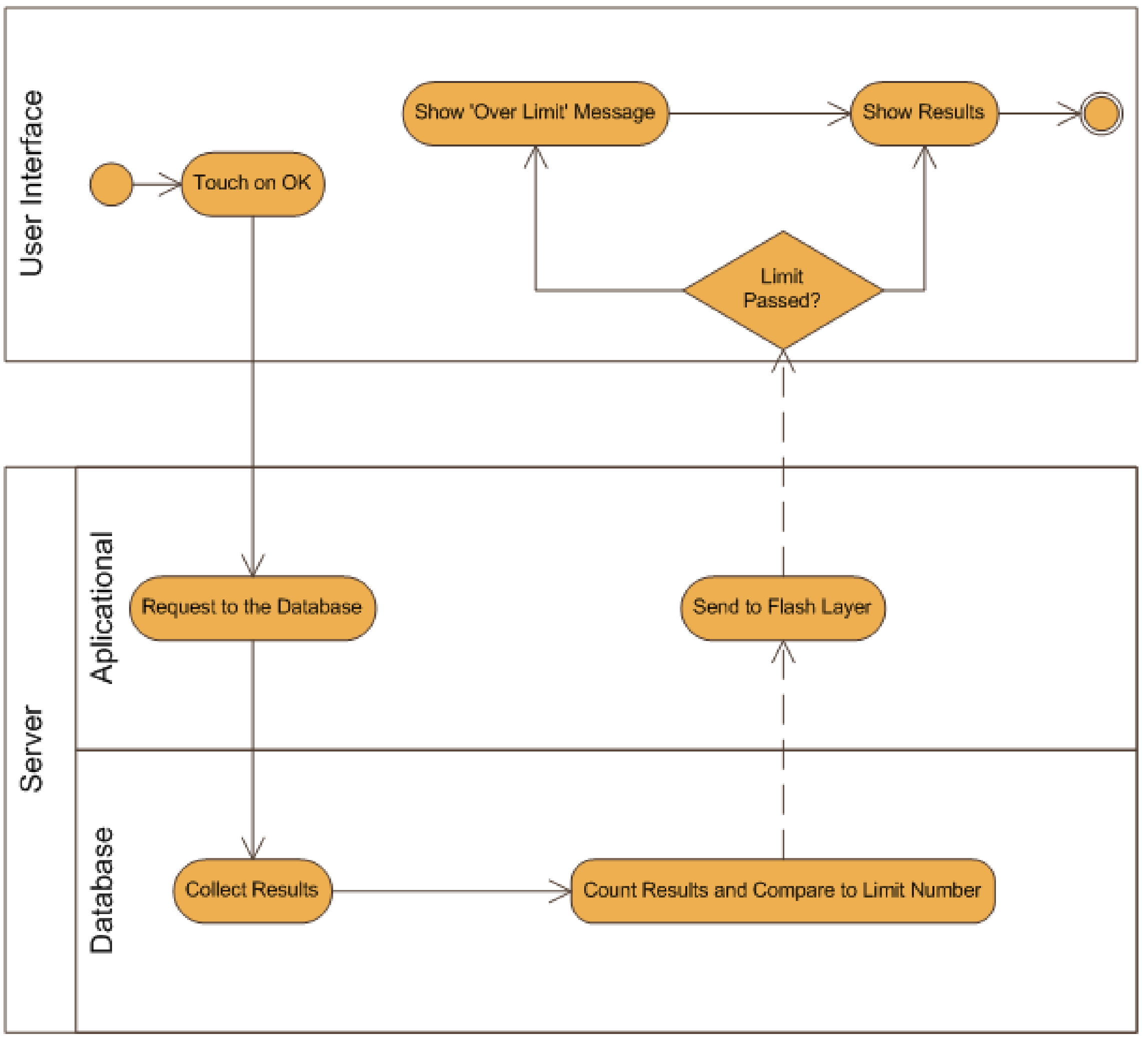


Figure 3.7: Communication Between Components

- Date;
- Multichoice.

This will enable a different data entering method to be shown to the user varying on the type. The method used to gather the data that appears in the multichoice is stored inside the database for each criterion. The data model used to handle the criteria is showed next in figure 3.8.

Table BUTTON_CRITERIA works to define the criteria set that appears for each search (identified by the ID_BUTTON of the button associated to it in the user interface). Each criterion has some properties associated to it as its visible or mandatory flag that are specific for each criteria set.

The text to be appended to the query is stored in the field TEXT. It uses special tokens that are replaced by user text when the search is executed.

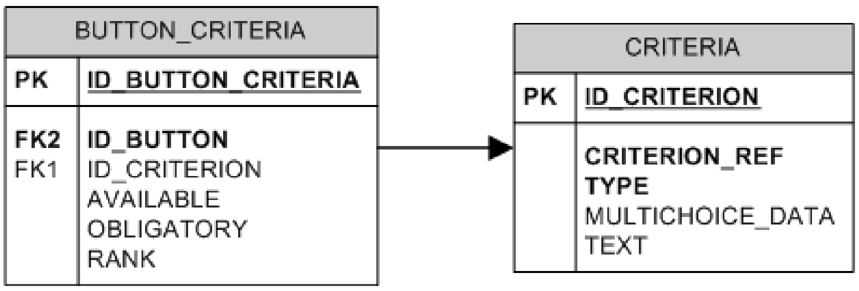


Figure 3.8: Former Data Model

Issues

The main issue in the search tool was its non-functional requirements, mainly the performance, and its difficulty to implement new searches or to maintain previous searches whenever the business rules changed.

Next it'll be presented, in detail, what were the issues with the search tool.

Security In order to allow user to choose the active criteria set it was needed that the SQL query was built in runtime. This was accomplished with Dynamic SQL.

So, whenever a search was run, each criterion text was appended into the query text. This could lead to security issues through a technique called 'SQL Injection' [Int04] as the user inserted text that gets replaced inside the criterion text could contain malicious code.

This was overcome by truncating any quotation mark in the text the user has inputted translating any code to a simple text string.

Performance Patients search tool have some performance issues. This is because most of the accessed tables are core tables like the patient's personal data and each episode information. Most of the time, though, is spent doing calculation by PL/SQL function calls (about 75%).

Because of these performance problems it was decided that the search should return no results if the results set size was to pass 150[4] results. This would avoid some extra effort and oblige user to choose a tighter criteria set. Every approach presented will take this into account for its development.

Maintainability Another approach to achieve some of the results desired to this project was to give the developers tools to build their own searches and one of the problems found was that none of the code was being reused between searches leading to a great effort reproducing the required structure resulting in a more error prone task too.

[4]Although this value may be configured it'll be used throughout this report to refer to the results number limit for the sake of convenience and readability.

A recurring maintainability problem is whenever any search query is updated. This required that the corresponding query counting the number of results was also modified.

3.2 Requirements Specification for a new Patients Search Approach

In this chapter it will be presented a requirements analysis. This requirements were obtained from the projects objectives, the existing product state and internal company objectives.

They are to be defined in a unambiguous and consistent way. This is so that it is possible to compare a set of possible solutions and evaluate them in terms of requirements fulfillment.

These requirements will be used throughout this section as a term of comparison between different approaches.

3.2.1 Functional Requirements

A search tool must have a way to gather the information needs from the user and return relevant information based on that understanding.

Also, the information relevancy should be measured in terms of information needs satisfaction and results precision[5] (FR#2). Said so, the criteria set available must capture the user needs in the most complete manner (FR#1). This then enables users to define his search strategy and reiterate over it (FR#4).

A search screen should not be the end of the way for its user. Navigation should be possible from each result (FR#3). This allows to get more detail about a specific patient and even allow some action over it.

As some results sets can be quite extensive it should also be possible to order each column by its value (FR#5) making possible to scroll through the results more objectively.

We can observe in table 3.1 the functional requirements or, in other words, what users may be able to do in each search screen.

Table 3.1: Functional Requirements Table

Identifier	Description
FR#1	Input some criteria for the search
FR#2	Limit each result set to 150 results
FR#3	Navigate from each result to get more detail
FR#4	Change the search criteria text
FR#5	Order each column by its value

[5]It is of no use delivering an unlimited number of results to the user because he'll never navigate through all of them.

3.2.2 Non-Functional Requirements

Each search is a critical access in many ways. First, from the performance perspective, each search must deliver the results as fast as possible (NFR#6) as part of its decision support role. Second, from the security perspective, each search is a personified access to the database where lays much confidential information also in patient health record and company business rules (NFR#4). Third and last, from the reliability perspective, the results must be as accurate as possible (NFR#5). A physician will be making life or death decisions based on the information he gets.

Relevant info should be delivered to user to help him understand the results he gets. This includes any error/information messages but also localized data.

In table 3.2 it is possible to find the non-functional requirements, searches musts.

Table 3.2: Non-Functional Requirements Table

Identifier	Description
NFR#1	Deliver search results in accordance to user locale and language preferences
NFR#2	Show message to the user if there are no results
NFR#3	Show message to the user if there are no results
NFR#4	Be secure
NFR#5	Be reliable
NFR#6	Be fast

3.2.3 Design Requirements

The major problem with maintaining each search is that with a changing business logic each search must be modified. One way to avoid this difficulty is to modularize the search, enabling code reuse (DR#1) and individual module maintenance (DR#2).

One should not write the same code many times as this can be more error prone. Being able to reuse code leads to less errors. Less code is also important in PL/SQL context as this code stays in the SGA memory when pre-compiled.

Maintaining code is also easier in these conditions. One would not need to modify the same code in different places. In more specific maintenance tasks it is important to have readable code and to separate business logic from presentation matters.

These needs can be translated to a grid of design requirements, table 3.3.

Table 3.3: Design Requirements Table

Identifier	Description
DR#1	Module reuse
DR#2	Business logic maintenance

3.2.4 Former State Requirements Review

In its former state, all the functional requirements were fully incorporated leaving nothing to do in the scope of this project.

We can see a listing of the functional requirements review in table 3.4.

Table 3.4: Functional Requirements Review for Patients Search Tool Former State

Identifier	Comment	Evaluation
FR#1	Data Model allows criteria set creation	Great
FR#2	Function verifies number of results	Great
FR#3	Each result in grid is navigable	Great
FR#4	Entered criteria text is recorded for one time	Great
FR#5	Grid allows ordering columns	Great

For what it concerns on non-function requirements, performance is the only requirement to be far from being satisfied. From now on requirement NFR#5 will refer to performance improvements because “be fast” is too sparse to evaluate. Check table 3.5 for non-functional requirements review this tool former state.

Table 3.5: Non-Functional Requirements Review for Patients Search Tool Former State

Identifier	Comment	Evaluation
NFR#1	Data Model allows translations and timezone differences	Great
NFR#2	*Count* query returns number of results	Great
NFR#3	*Count* query returns number of results	Great
NFR#4	SQL Injection proof is guaranteed by programming techniques	Good
NFR#5	Search is always available	Great
NFR#6	The patient’s search tool is reported to be very slow	Poor

As reliability is guaranteed by the system architecture and it is not addressed to any data model question it won’t appear in any posterior requirements review. This will happen to NFR#1 too.

Every search is handled by a single function. The main query is written in Dynamic SQL which could not be ‘beautified’ and it is only parsed at run-time preventing Oracle to check its syntax. This is summarized in table 3.6.

Table 3.6: Design Requirements Review for Patients Search Tool Former State

Identifier	Comment	Evaluation
DR#1	Each search is represented by a unique function	Poor
DR#2	Great amount of Dynamic SQL	Poor

3.3 Patients Search Approaches

Through the study there were developed some approaches to the solution. These were original ideas that had to be tested so a series of proofs of concept were developed for that purpose.

The next section serves to describe each approach concept, having their strengths and weaknesses identified and finally compared.

There won't be no review for functional requirements as the former state already fulfill the tool's needs.

3.3.1 1st Approach

"No previous counting"

In an early analysis to the search functions structure it was easy to identify one possible improvement, which was trying to eliminate the need to query almost the same tables twice. It would be possible to avoid this embedding the results count into the main query in a dummy row. This way it would be possible to fetch the row, test the value and return the rest of the cursor. This had the desired effect of not having to run the same query twice and hence improving the search performance.

One setback was that the query model that made it possible to create that dummy row was too big and complex. As all the search queries were made in Dynamic SQL (to allow appending the parameters string) this would led to a complex string, much harder to code and much, much harder to maintain. But the improvement in performance was too significant to drop this solution. It was clear that the path was, somehow, to allow count the number of the results without having to run a second query.

Requirements Review

Table 3.7: Non-Functional Requirements Review for 1st approach

Identifier	Comment	Evaluation
NFR#2	Previous fetch allows testing the number of results	Great
NFR#3	Previous fetch allows testing the number of results	Great
NFR#4	SQL Injection proof is guaranteed by programming techniques	Good
NFR#6	One query only but cannot avoid some calculations	Better

Table 3.8: Design Requirements Review for 1st approach

Identifier	Comment	Evaluation
DR#1	Each search is represented by a unique function	Poor
DR#2	Great amount of Dynamic SQL	Poor

3.3.2 2^{nd} Approach

"Concatenated Datastore"

The second approach was to previously cache the search results making use of materialized views. This would also allow to build a text index on the materialized view over most of its columns[6].

The materialized view would cache the search data for faster response times and the text index would allow more comprehensive searches. Originally, the text index would be on the patient's name column but as there is more columns to be searched, creating one index over the most used columns would be faster. This led to the use of a datastore, more specifically the concatenated datastore, in order to do range search over dates or integers.

Having criteria text constructed with a less complex syntax (Oracle Text context indexes) and without table aliases would make it easy to construct a criteria set for the user to use it. Also, the separation of business logic from presentation matters with the creation of a materialized view for each search would help with maintenance tasks.

But materialized views refresh times plus the indexes rebuild in a high volatile data environment like ALERT® would be bad for performance. And although there would be no need to use Dynamic SQL the query model suggested in the previous approach would have to be applied to get the results set size. Also it would not be possible to prevent some effort if the results were not to be shown.

Requirements Review

Table 3.9: Non-Functional Requirements Review for 2^{nd} approach

Identifier	Comment	Evaluation
NFR#2	Previous fetch allows testing the number of results	Great
NFR#3	Previous fetch allows testing the number of results	Great
NFR#4	Criteria are passed through a parameter. No SQL Injection	Great
NFR#6	One query only but cannot avoid some calculations. Materialized Views and Oracle Text refresh times	Better

3.3.3 3^{rd} Approach

"Table function"

[6] See 2.3 Concatenated Datastore.

Table 3.10: Design Requirements Review for 2nd approach

Identifier	Comment	Evaluation
DR#1	Each search is represented by a materialized view	Good
DR#2	No Dynamic SQL but rather complex query model	Good

Other approach to take was to minimize the quantity of context switching because of the high quantity of PL/SQL function calls per row for formatting purposes. One possibility was to gather all the data from database to a PL/SQL collection and do all the formatting over the collection data but this was not possible due to the necessity to send an open cursor to the Java layer to pass the search results. The solution was to create a table function and do all the work there and selecting it into a cursor after.

This would automatically lead to a new module creation for each search, the table function, where all the specific work for search would be made freeing the previously existing function to do all the common work. One possibility arise with this that was to create a unique function that would know what table function to call for each specific search.

This would enable other interesting developments:

- One could check the results set size after doing any calculations by just checking the collection size;
- As all the function calls were to be made in PL/SQL context a PL/SQL cache could be created for functions with less variable results.

Reducing context switching for 1,500 function calls (about 10 function calls per row) brought a little performance improvement but it was the ability to query the data only once and avoid unnecessary work that brought the most improvement.

The table function needed that its return type was defined previously. This requires creating a SQL Type that describes the type of each column returned. Although this means some extra work and some maintenance effort it can eventually be done automatically and besides, this would only be needed while sending a PL/SQL collection into Java is not supported.

Requirements Review

3.3.4 Comparison

Regarding the project objectives, performance was the most meaningful result obtained. This highlights the Table Function approach as the right one to choose. It should be taken into account that this is also the most flexible solution allowing a big step on modularity with a unique function for common tasks on searches.

Table 3.11: Non-Functional Requirements Review for 3rd approach

Identifier	Comment	Evaluation
NFR#2	Test number of results before calculations	Great
NFR#3	Test number of results before calculations	Great
NFR#4	SQL Injection proof is guaranteed by programming techniques	Good
NFR#6	Counting takes part before all the calculations. PL/SQL calls are made in PL/SQL context	Even Better

Table 3.12: Design Requirements Review for 3rd approach

Identifier	Comment	Evaluation
DR#1	Each search is represented by a table function. Unique function to handle common search tasks	Good
DR#2	Less Dynamic SQL	Good

Table 3.13: Non-Functional Requirements Comparison

Identifier	1st approach	2nd approach	3rd approach	Final Comment
NFR#2	Great	Great	Great	All approaches guarantee this
NFR#3	Great	Great	Great	All approaches guarantee this
NFR#4	Good	Great	Good	With no Dynamic SQL no SQL Injection is possible through here on 2nd approach
NFR#6	Better	Better	Even Better	3rd approach can avoid much more effort than the other approaches. Other approaches always added extra work one way or the other

Table 3.14: Design Requirements Comparison

Identifier	1st approach	2nd approach	3rd approach	Final Comment
DR#1	Poor	Good	Good	Having a unique function to handle all common tasks is great for code reuse on 3rd approach
DR#2	Poor	Good	Good	Separating presentation purpose function calls from data gathering helps with maintainability either on 2nd and 3rd approaches

3.4 Conclusions

In this section it was possible to learn about the application architecture and how the database plays its role revealing details about its functional organization. The patients search tool was analyzed as a main use case and as a very important tool to help users finding relevant data for their tasks. The requirements for this project were listed in order to understand the amount of work left to do. These requirements were used to compare three different approaches and help choosing one.

The next chapter will develop the chosen approach in more detail.

Chapter 4

Search Screens

In the previous chapters the various approaches created have been described and it was possible to compare them. It was also possible to understand why the Table Function approach was chosen to be developed.

In this chapter the designed solution will be described expanding it and adding new improvements. Development details like integration details will be described in section 4.2.

4.1 Search Screen Architecture

The solution's description will be resumed to architecture's details, with no implementation details at all. So, it will be available a general vision on the architecture, seizing the opportunity to resume what's behind and then to show some improvements made in the final solution as well as their finalities. Among these improvements there are introduction of parameterized views, modifications made on the criteria list and the generic function's structure.

4.1.1 Overview

This solution is based on the approach presented on the previous chapter, Table Function. The idea is to load every necessary data from the database into a PL/SQL collection and make any additional calculation on that collection, in PL/SQL context. By doing so, many context switching can be avoided. To achieve it, it was necessary to use Table Functions to let SQL engine create a cursor with the results and send it to the Java layer.

Before, it was said that this approach is quite versatile allowing to get more control over the calculations. By doing so, it is possible to avoid some testing repeats and to cache function's results. So, the greatest advantage of this solution is the possibility of testing the number of returned results before making any previous calculations over their data.

The scheme presented in figure 4.1 represents the functional organization with parameters being passed among the several components. So, in this scheme is possible to see that the business logic in each search can be found in the results screen grid and in the table function, the only components that could not be generalized. For this version, the table function's name to be called is not yet saved in the database, being instead kept in the results screen's grid.

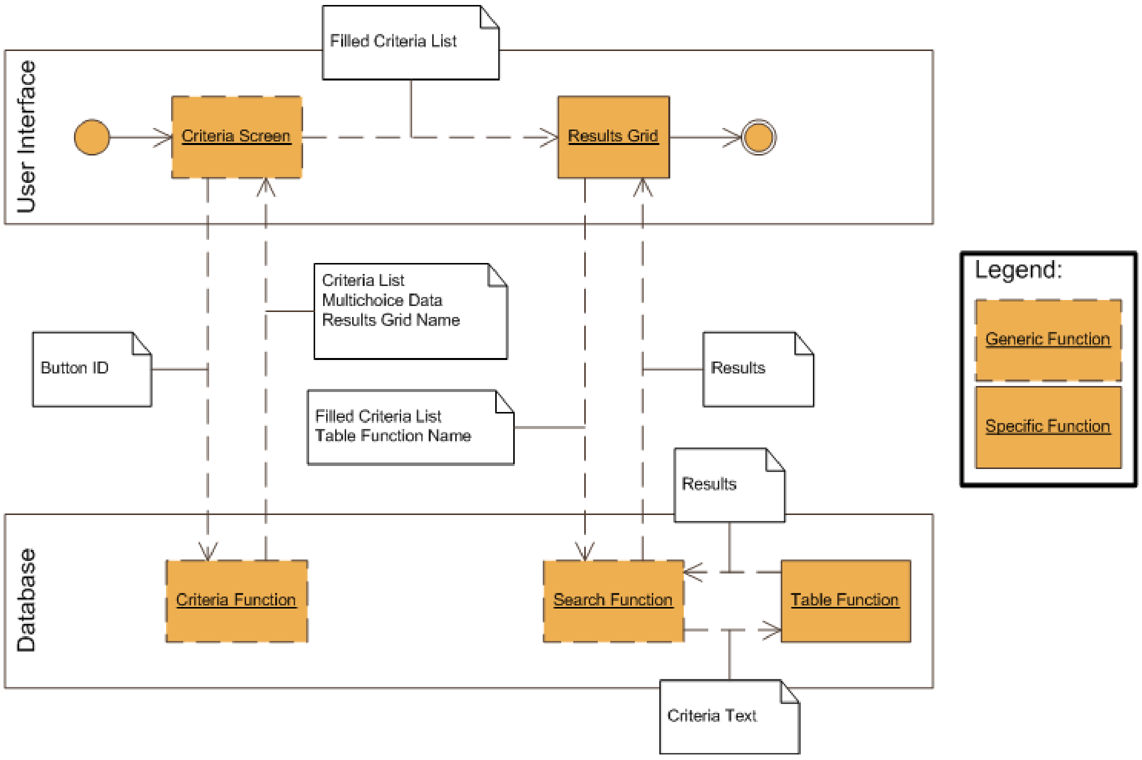


Figure 4.1: Function Structure

Additional modifications are described next.

4.1.2 Data Model

One basic[1] modification was to create the Search Screen entity. This is what names the solution as it reveals the identity of each Search in the database and is used to gather some unique information for each screen like the name of the grid used to display the results. This is useful for any future developments concerning modularity.

The classes diagram for the search it is presented in figure 4.2.

In this diagram it is important to notice a few things:

- Each Search Screen is associated to a button in the user interface;

[1]Basic as in basis and easy also.

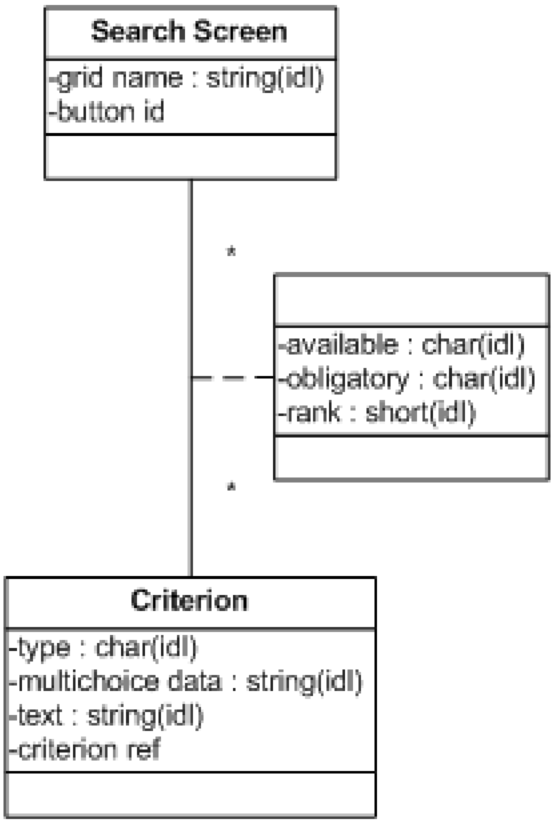


Figure 4.2: Classes Diagram for the Search

- The connection class is used to define some specific properties for each criteria in a search screen like if it is visible, mandatory and its relative position in the screen;
- Each criteria has a text associated to it.

The previous class diagram can be translated to the data model in figure 4.3.

4.1.3 General Function

In section 3 was said that the third approach included the creation of a function which reunited every common tasks in the search tool.

In fact, this development was already possible in other approaches but in this case the impact was bigger as the creation of a table function per search doubles the number of functions.

So, this generic function becomes responsible for the following tasks:

- Create the text to be used to filter the results based on introduced criteria by the user;
- Open a cursor calling a table function;
- Check if it any result was found or if these result's set surpassed the bounds and define any necessary message;

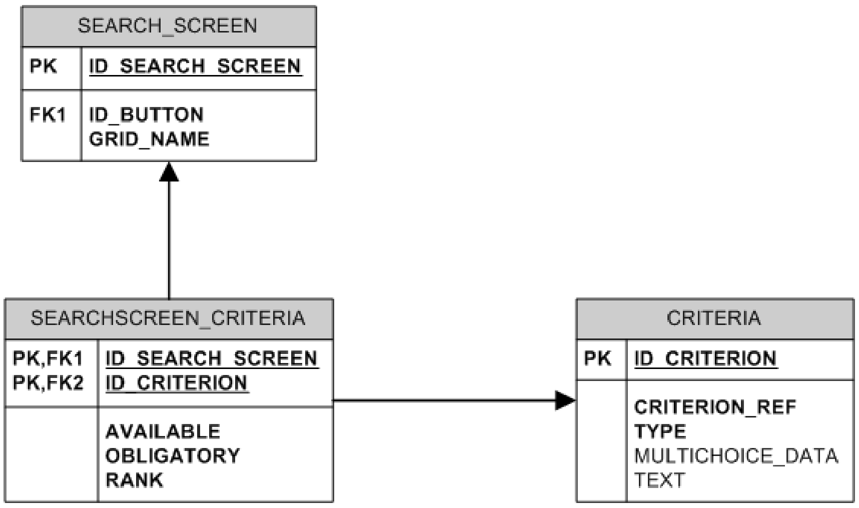


Figure 4.3: Relational Diagram for the Search

- Return the cursor to Java's layer.

The resulting communication sequence diagram between layers is shown in figure 4.4.

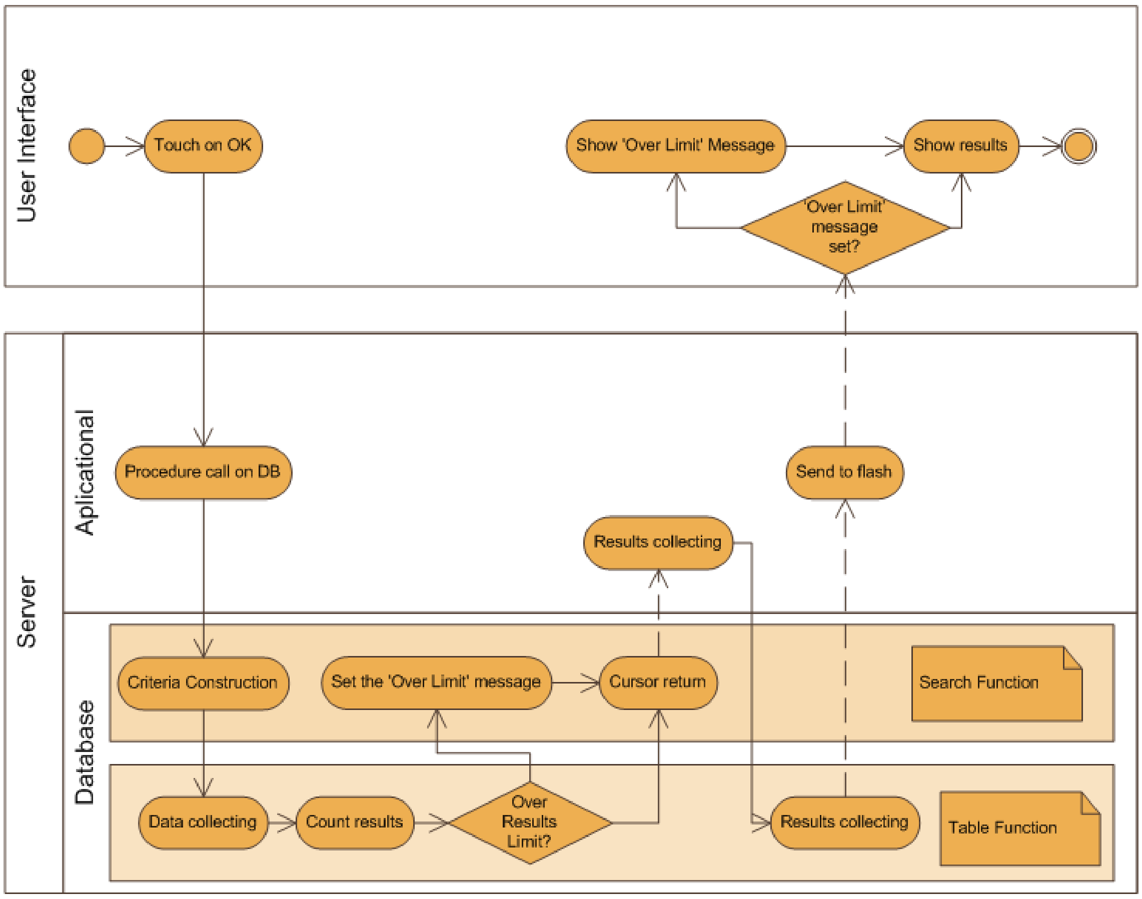


Figure 4.4: Communication Diagram

4.1.4 Parameterized Views

Reducing the amount of Dynamic SQL is one of the techniques that can be used to achieve better maintainability. But so that the criteria text can be appended to filter results Dynamic SQL must be used.

A solution for this is saving the query in a view. It's not easy to do this though, because it is impossible to input parameters into a view directly. To solve this there were used context variables [Ora01a] that can be set for the user session and accessed from inside the SQL query.

This made possible to create a view for each search containing all the SQL code that is used to get the necessary data for the search. This is a step in modularity and maintainability both abstracting data model details from presentation details and reducing the amount of Dynamic SQL.

4.1.5 Criteria

A problem in reusing criteria text is that the defined names for tables' aliases vary from query to query. An advantage in the use of parameterized views is to be possible to define only one alias to use in every criteria, increasing their reuse among the different searching queries.

It was said previously that code input by the user should have their quotation marks truncated in order to avoid the introduction of malicious code (SQL Injection). Plus, there is another problem. When a query is being built with hard coded parameters, the variation of one of these values will force an hard parse from the query each time one of these parameters is modified.

The solution to this problem, one more time, consisted by using context variables. As context variables are seen as bind variables the change of one value in a parameter doesn't force to hard-parse the whole query again. Besides, for security purposes any text passed to a context variable can not be interpreted as code.

4.2 Development

In this chapter the first stage of the implementation of this solution will be described. This first stage was integrated in the development of the last product's version ALERT® and aimed the integration of all EDIS' searches, 4 searches total.

These searches differed from their information requirements and deep knowledge in SQL query optimization was required in order to guarantee the best execution plan for each one of them. This also shows how the patients search can have all kinds of queries and how they can be used in any product regardless of their data model.

With data model change it was necessary to update every function that made use of that part of the data model. This obliged the author to search in the database code for every functions recurring to some of these tables and inform related teams that these needed reformulation.

This change had much bigger consequences in the flash layer. To support the verification of filling in the mandatory fields, the numbers that screens used to identify them were from BUTTON_CRITERIA table. With the change of this check to the interface, the used identifiers become those of the criteria table. This forced the modification of all the screens that used this check and the recompilation of all screens that depended of the first ones.

Obviously, one function that had to be modified was the one responsible for compiling the criteria's test to be used in the search queries. The tasks of this function are now:

- Obtain the code of each filled criterion by the user;
- Create a context variable for each criterion with the filled text by the user;
- Return a text string with every compiled criteria.

An introduced modification to allow some performance gains was the creation of an index in the patient's birth date. As this column is many times used to filter the search, it makes sense to have an index that allows, from the beginning, greatly bound the set of results.

4.3 Conclusions

The designed solution revealed to be very flexible and strong. Its modularity and performance details guarantees that the tool's life cycle is now extended with less maintenance effort.

The current state of integration guarantees that every search that was upgraded keeps full functionality as well as those who weren't upgraded yet. This planning did have to consider that all of present information requirements were to be kept. This required great attention to the integration, specially to query optimization.

Chapter 5

Solution Assessment

5.1 Performance

Performance revealed to be a very relative term to consider. Every performance expert advises that tests should be runt in each specific database before implementing any new performance hint they came to discover. The test part of an application's performance must be done in a controlled environment to prevent any wrong conclusions to be taken.

5.1.1 Methodology

To correctly test performance improvements all tests that will have their results compared shall run in similar environments in terms of work load.

A variety of tests will be chosen and these will be runt in different environments. First, all the tests will be compared with each other on the same environment. Finally these results will be compared to reach some final conclusions.

Tests will be runt in a Test Window in PL/SQL Developer and times will be gathered from the execution times that appear at the status bar.

An arithmetic mean of 4 consecutive measures will be calculated to reach the final measures used.

5.1.2 Tests

Three different tests were chosen to run:

1. Original solution;

2. Table function as described in the previous chapter;

3. Table function making use a materialized view.

These tests will be runt in two different environments:

1. Server with more than 10,000 episodes and high concurrency;
2. Server with more than 70,000 episodes and low concurrency.

5.1.3 Results

The times measured appear on table 5.1 and they reveal that the Table Function solution has 33% improvement in relation to the original solution (table 5.2). The improvement is noticeable in both environments. Using a Materialized View is also a good choice as it gives more performance improvements.

Table 5.1: Tests Time Measurements

	Test 1	Test 2	Test 3
Environment 1	1,588 ms	1,065 ms	1,041 ms
Environment 2	2,080 ms	1,555 ms	1,342 ms

Table 5.2: Performance Improvements Comparisons

	1 - Test 2/Test 1	1 - Test 3/Test 1
Environment 1	32.9%	34.4%
Environment 2	25.2%	35.6%

As it is possible to see the performance improvement is of about 30% in a little database and about 25% for a bigger database. This is because the results counting query is actually more scalable than the the main query, so the improvement of not running it comes to be less for bigger databases.

An improvement for this scalability matter is using a materialized view leading to improvements of 10% in bigger databases.

5.2 Maintainability

One of the problems in the way the tool was implemented was that the architecture was not documented. This led to the implementation of new searches by copying the code and adapting it, making the architecture's maintainability very painful for who had to make it. It was necessary to spread any modification for all the search implementation.

Due to the fact that there were no documented architecture, every search was adapted according to necessities. There was common to find searches where there were introduced some extra steps to the defined structure in section 3.1.3, pushing them away from the possibility to be integrated in a structure with a common architecture.

The achieved solution allows to get a more objective maintainability because it separates the details of the data access. This allows that, in the future, more general views are built and can be used in some more specific cases, depending of the passed parameters, so that can be reused between searches.

Chapter 6

Conclusions and Future Work

6.1 Results

The solution here presented gives a common structure to all the patients search throughout different products. This project's goal was to give each team a strong tool to develop new ways to search in its products.

This new architecture also brings performance improvements to the patients search tool but, unfortunately, this project scope did not allowed more improvements as it would require data model reformulations and some specific functions to be changed. However, much of the project was spent researching performance techniques giving the author knowledge on query optimization.

This knowledge was useful as each reformulated search query was extensively optimized but this research suffered from the lack of official documentation on Oracle optimization.

Maintainability was also made easier and more objective. Now developers don't have to face mountains of Dynamic SQL code and they can also check for errors at compilation time. This is less error prone and leads to better code legibility.

Maintainability also profited from the increased modularity. Now a search screen can have their presentation details reviewed without having to care about data abstraction details and vice versa. Creating a new search is also easier as more information is stored inside the database and not scattered through the various layers requiring less work for Flash developers.

6.2 Project Overview and Highlights

Oracle database is very flexible and it presents multiple solutions for developers problems. This project showed three different approaches to a single problem, all of them making use of different Oracle tools like Oracle Text, datastores, materialized views, table functions and context variables.

This study approached different aspects like security, performance, maintainability and usability and the resulting project brought new improvements in all of these aspects:

- This project can take advantage of the cost based optimizer and makes extensive use of bind variables, avoiding query hard-parses;
- It does not compromise application security preventing the use of SQL Injection;
- Avoids a great amount of context switching despite the fact that most of the data abstraction layer relies on PL/SQL functions;
- Allows better performance as it avoids repeated access to tables;
- Reduces the amount of Dynamic SQL to a minimum increasing maintainability;
- Increases code reuse from an increased modularity.

6.3 Future Work

There is still some work to do on the current solution integration as there are still some searches to be adapted to the new architecture. This integration phase is to end when all the maintenance work is passed back to each product's team. Until then it is necessary to develop some tools to help with the maintenance tasks and to do some automatic code generation.

With a common structure behind ALERT® Patients Search Tool is easier to think of new functionalities to add.

6.3.1 Extending Criteria Types

The available types of criteria should be extended to support choosing multiple values in a multichoice criterion e.g., selecting multiple values in a multichoice, or giving user the ability to view some available values in the database for that field e.g., in searching for a patient's name.

6.3.2 Improving Modularity

Other possible improvement in the structure is the creation of a criteria set entity. This would enable multiple search screens to reuse the same criteria set. At the moment every search screen requires the creation of a criteria set of its own many times repeating the criteria set used by other screens.

6.3.3 Self Documenting Searches

This structure also enables each search to reuse grids requirements information requirements. The ability to document these requirements may enable some grids (both main grids and search screen grids) to be automatically generated given its columns.

6.3.4 New Scopes For Patients Search

One interesting development that would revamp the search screens role is to enable users to filter the results grid. Nowadays users may use the patients search when they don't want to scroll through an enormous list looking for a specific patient. Creating this filter functionality on grids would also make possible to users to input their criteria while they scroll through the search results, all in the same screen, improving usability.

References

[AA08] AllRound Automations. AllRound Automations, 2008. http://www.allroundautomations.nl/, 19 June 2008.

[ALE08] ALERT Life Sciences Computing, S.A. alert.pt Company Overview, 2008. http://www.alert.pt, 18 June 2008.

[BFM78] Jon Louis Bentley, Jerome H. Friedman, and H. A. Maurer. Two Papers on Range Searching: A Survey of Algorithms and Data Structures for Range Searching. Efficient Worst-Case Data Structures for Range Searching., 1978. http://stinet.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA060584, 19 June 2008.

[Bur02] Burleson Consulting. SQL tuning performance optimization with Oracle materialized views, April 2002. http://www.dba-oracle.com/art_mv.htm, 19 June 2008.

[Feu06] Steven Feuerstein. Everything you need to know about collections, but were afraid to ask - best practices for pl/sql, 2006.

[HP92] Hewlett Packard. ALLBASESQL FORTRAN Application Programming Guide, 1992. http://docs.hp.com/en/36216-90079/36216-90079.pdf, 20 June.

[Int04] Integrigy Corporation. An Introduction to SQL Injection Attacks for Oracle Developers, January 2004. http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf, 19 June 2008.

[JCC05] JCC Consulting, Inc. JCC's SQL Standards Page, 2005. http://www.jcc.com/sql.htm, 18 June 2008.

[Kya06] KyaPoocha.com. What are the difference between ddl, dml and dcl commands?, August 2006. http://kyapoocha.com/oracle-interview-questions/what-are-the-difference-between-ddl-dml-and-dcl-commands/, 20 June 2008.

[MG02] Mark Gurry. Oracle SQL Tuning Pocket Reference, January 2002.

[Ora00a] Oracle Corporation. The Concatenated Datastore - A Utility for Oracle Text, 2000. http://www.oracle.com/technology/sample_code/products/text/htdocs/concatenated_text_datastore/cdstore_readme.html, 19 June 2008.

[Ora00b] Oracle Corporation. Concatenated Datastore - Performance Improvements, 2000. http://www.oracle.com/technology/sample_code/products/text/htdocs/concatenated_text_datastore/cdstoreperf.html, 19 June 2008.

[Ora01a] Oracle Corporation. Ask Tom "Parameterized View & Truncation of Table", 2001. http://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:1448404423206, 19 June 2008.

[Ora01b] Oracle Corporation. Datastore Types, 2001. http://www.peacetech.com/flipper/oracle9i/901_doc/text.901/a90121/cdatadi3.htm#34439, 19 June 2008.

[Ora02a] Oracle Corporation. Materialized View Concepts and Architecture, 2002. http://download-uk.oracle.com/docs/cd/B10501_01/server.920/a96567/repmview.htm, 19 June 2008.

[Ora02b] Oracle Corporation. Selecting an Index Strategy, 2002. http://www.stanford.edu/dept/itss/docs/oracle/9i/appdev.920/a96590/adg06idx.htm, 19 June 2008.

[Ora02c] Oracle Corporation. Using EXPLAIN PLAN, 2002. http://download-uk.oracle.com/docs/cd/B10501_01/server.920/a96533/ex_plan.htm, 19 June 2008.

[ORA03] ORACLE-BASE.com. Oracle9i DBMS_PROFILER, 2003. http://www.oracle-base.com/articles/9i/DBMS_PROFILER.php, 19 June 2008.

[Ora05] Oracle Corporation. Indexing with Oracle Text, 2005. http://youngcow.net/doc/oracle10g/text.102/b14217/ind.htm, 19 June 2008.

[Ora07] Oracle Corporation. Oracle Text, 2007. http://www.oracle.com/technology/products/text/index.html, 19 June 2008.

[SF01] Steven Feuerstein. Oracle PLSQL Best Practices, April 2001.

[W3C01] W3C Semantic Web. W3C Semantic Web Activity, 2001. http://www.w3.org/2001/sw/, 19 June 2008.