Performance & Scalability of a Spatial Database in a GIS-Web Service Environment

Mark Olthof
Arnhem, 2007

Thesis for a Ir. degree in Computer Science
Faculty of EEMCS, chair Databases, University of Twente, The Netherlands

University of Twente
Enschede - The Netherlands

Performance & Scalability of a Spatial Database in a GIS-Web Service Environment

Thesis

Project Description

Author	Mark Olthof (S0009202) Faculty of EEMCS, University of Twente Chair Databases	
Company	LogicaCMG Arnhem Meander 901, 6801 HA Arnhem The Netherlands	
Department	PSAR & GEO-ICT (Nieuwegein)	
Project Title	*SABRE: Optimization of Oracle Spatial database design*	
Period	1 February 2007 – 18 November 2007	
Gradutation Commitee	Maurice van Keulen Harold van Heerde Raynni Jourdain Olaf Lem	(1st supervisor) (2nd supervisor) (LogicaCMG, Project leader WT) (LogicaCMG, Project leader SABRE)

logicaCMG

To Lien

I Abstract

LogicaCMG developed SABRE, a spatial business rules server. Its purpose is to process queries for predefined business rules based on a given location. SABRE was rapidly developed as a project for 'master-class' employees. Although a functional demo was created, SABRE remained a simple prototype not ready for commercial purposes. LogicaCMG's goal is to redesign SABRE for commercial purposes, such as for instance tracking and tracing. This project focusses on the optimization of the database part, which will be the foundation of SABRE. If SABRE will be commercially deployed we can expect massive usage, and very high performance requirements. Therefore it's essential to develop SABRE from the ground up in such a way that maximum performance is achieved.

The SABRE architecture consists basically of a Service Provider (SP), Web service (WS) and a Database (DB). The SP requests certain information from the DB through the WS. The aim of this project is to develop a working version of the redesigned SABRE application fit for demonstration purposes. The focus will be on the DB part, therefore the WS will support only one service (e.g. AREA-event). The objective of this assignment is to study the performance and scalability of the DB. The two most important scalability aspects are: how does the DB cope when the amount of requests increases and how does the DB cope when the amount of data increases.

A scalable database design has been created of which a prototype has been implemented. The prototype was used for testing the performance and scalability of SABRE. To improve the performance and scalability three optimizations have been used, SQL Tuning, Materialized Views and Range Partitioning. The Materialized View optimization showed the best result with a 60% performance improvement. As a result of the optimization the SABRE performed well for four out of the five used datasets. The largest dataset was too large for the database to handle in terms of response times. However, since the tests have pointed out that SABRE is scalable, addressing the issue of the largest dataset should only be a matter of adding resources. When the required resources have been added the SABRE application will meet its requirements for commercial exploitation. Therefore it is expected to hear more from SABRE in the near future.

II Preface

This thesis describes my research done for an Ir. Degree in Computer Science. The graduation project took nearly nine months, of which six took place at LogicaCMG Arnhem. The other three months I worked at home, mostly busy writing this thesis. During this project I gained a better understanding of databases and performance and scalability aspects every single day. A lot of my time went into the design and realisation of a prototype. When the prototype had finished, the time came to test my prototype which also took a considerable amount of time. The last part of my project, writing this thesis, took a large amount of time as well, way more than I expected. Although sometimes the outcome of the performance and scalability tests drove me insane because of the unexpected results which caused mind boggling theorizing, I can look back at a interesting nine months.

I would like to thank my LogicaCMG supervisors Raynni Jourdain and Olaf Lem for their devoted support during my stay at LogicaCMG Arnhem. I would especially like to thank Olaf for his insights and the great discussions we have had on the subject matter. I would like to thank my UT supervisors Maurice van Keulen and Harold van Heerde for their hints and tips they gave me which resulted in a more profound research project. Especially during the writing of my thesis their support was crucial. Last but certainly not least I would like to thank my girlfriend, Caroline, for always supporting me during these nine months, especially when I needed it the most.

Mark Olthof

Nijmegen, 18 November 2007

Table of Contents

1 Introduction

This chapter is an introduction to the research done for the project. This chapter contains the motivation for the project, followed by a description of the problem statement. There is a description of the approach taken to complete this project, preceded by the project context, and finally the structure of this thesis is explained.

1.1 Motivation

LogicaCMG's Geo-ICT department at Nieuwegein, a department specialized in geographical information systems and location based services, has had the idea for quite some time to create an application capable of providing location based services. The Geo-ICT department would like to see if they are able to offer their own solution in the growing market of geography related ICT instead of being dependant on other vendors. The idea was born to create a so called spatial business rules server, SABRE in short. The application had to be able to offer location based services, more specific the application had to be able to process location related business rules. To check whether an object is within or nearby a predefined area, based on its given location, is an example of a location related business rule. To stress the relation of business rule and location the word spatial was added, hence the idea of a spatial business rules server was born.

The concept of SABRE was to offer its services through means of a web service, for the communication from and to the web service the XML[1.1] standard had to be adopted. The processing of the spatial business rules had to be done by a database, which had to be capable of processing areas and locations. Therefore the processing had to be done by a database specialized in processing areas and location, a spatial database. This lead to a conceptual division into three components, the web service component, the XML component and the spatial database component.

With the concept of SABRE came several requirements. The service had to be offered by a web service, the application needed to be very extensible to be able to support future services and last but not least the application had be deployed in a commercial environment which had some extra requirements. For commercial purposes SABRE had to be able to cope with massive usage and high performance requirements, of which a query-load of about 50 requests per second is a typical estimate. The dataset on which the query is to be performed varies from 300 records up to 3 million records, whereas the query itself consists of retrieving locations, denoted as coordinates, from the database.

The difficulty with creating the spatial business rules server is the high performance requirements of 50 requests per second in combination with the large datasets. Because of the high requirements the application has to be designed to be scalable up to at least 50 requests per second. Because of the large datasets and the processing of areas and locations, the most important aspect of the realisation of SABRE is the spatial database component.

At the end of this project LogicaCMG would like to see a prototype of a scalable spatial business rules server which can handle a query load of about 50 requests per second.

1.2 Problem Description

The research statement for the thesis states:

"Performance & Scalability of a Spatial Database in a GIS-Web Service Environment."

As stated in the motivation the most important aspect of the realisation of SABRE is the spatial database component. This leads to the main research question for this project:

"How and with what techniques can a database-design be realised that complies with the performance and scalability requirements of a GIS-Web Service environment?"

To give an answer to the main research question, the question has been split up into several other research questions, because the main research question itself contains several other questions. From answering all the other questions an answer to the main research question can be constructed. Each individual question will be addressed in this thesis, whereas an answer or explanation will be given. From the main research question the following questions are derived:

I. "What is, with respect to this project, performance and scalability?"

II. "What is a spatial database?"

III. "What is a GIS?"

IV. "What is a Web Service?"

V. "How can a database-design be realised?"

VI. "How can the scalable database-design be implemented?"

VII. "How can the performance and scalability of the database be measured?"

VIII. "How does a Spatial Database perform in a GIS-Web Service environment in terms of response and transaction times?"

IX. "How does a Spatial Database perform and scale up against increasing user loads and increasing datasets?"

X. "Are there possibilities to significantly improve the performance and scalability by means of database optimizations?"

XI. "If there are significant database optimizations, how well do they influence the performance and scalability of the database?"

XII. "If there are significant database optimizations, can they be used in conjunction or separately, and which is the best (combined) optimization with respect to this project?"

It must be stated clearly that the project is about investigating the performance and scalability of a, to be created, SABRE application. The application should be designed to be optimally scalable and performing, within the scope of this project, using optimization techniques. If the outcome of the investigation is that the new application does not meet the high performance requirements, the project can still be seen as a success. Although one may not like the outcome, and most likely LogicaCMG will not be deploying the application for commercial purposes, it is an answer to the research of performance and scalability of a spatial database in a GIS-web Service environment.

1.3 Project Context

LogicaCMG has already made an attempt to create the spatial business rules server. SABRE was rapidly developed as a project for 'master-class' employees. New employees of LogicaCMG attended a master-class, a form of study to get the new employees up to date with the latest technologies, where they were given the task to convert the SABRE idea into a working implementation of SABRE. Although the employees created a working version of SABRE, it's design did not meet all of its requirements for commercial usage. Because of the promising results of SABRE, people at Nieuwegein decided that SABRE had to be redesigned and built up from scratch again to meet the requirements for commercial usage. Not knowing if a new version of SABRE would be able to meet the necessary requirements they needed some specific research and decided to create a graduation project. This graduation project had to contain the development of a prototype and research on the performance of the prototype. The outcome of the graduation project should give the Geo-ICT department an answer to whether or not the SABRE idea could be used in a commercial environment.

The creation of a complete new SABRE application is far too much work for a single graduation project. Therefore the SABRE project was split up into two smaller projects, a Web Service part and a Database part. The latter one was chosen for this graduation project. Another graduation project is expected, which will focus on the Web Service part, to be realized after this project has successfully finished.

1.4 Approach

LogicaCMG's goal is to re-develop SABRE in a profound manner so it can be used for commercial purposes. LogicaCMG has set two different objectives. The first, to develop a working demo-version of the new SABRE application. Second, measure the performance and scalability of the newly built application. The working demo-version only contains basic functionality but should be designed in such a way that the application is scalable and is able to support future additions. The demo-version can then be used, as the word says, for demonstration purposes, trying to attract customers. Next to having a demo-version LogicaCMG also wants to know if the application can be used for commercial purposes. If the application is to be used in a commercial environment LogicaCMG must be able to say something about its performance and scalability. Therefore LogicaCMG wants to find out how well the application performs in a typical commercial environment and how the application scales up against massive usage.

If it is possible to create a new SABRE application which meets the requirements for commercial usage of about 50 requests per second and is scalable as well, we might see SABRE being used in a commercial environment in the near future.

This research project focusses on the database part of SABRE, the spatial database component to be more specific. The XML component is lightly addressed and beyond the scope of this thesis. The web service component will mainly be addressed in the redesign phase, and is further on mostly considered beyond the scope of the project, as it most likely will be thoroughly investigated in another graduation project. The research project is about the redesign of the SABRE application, whereas a simplified prototype will be implemented based on the new design. The prototype will only contain the bare necessities to accommodate the research and to be a profound realisation of SABRE as well. The new design will include optimizations to improve the performance and scalability of the application. Finally, the performance and scalability of the new design will be evaluated, including the optimizations.

The general approach for the research project is as follows:

- ✔ Study the original SABRE application, it's design and implementation, to familiarize with the functionality of the application.
- ✔ Get acquainted with related technologies and gain insight into Web Services and Spatial Databases.

- ✔ Study on Database optimization techniques, determine in what degree they influence the design of SABRE. Determine if the optimizations need to be dealt with at the design phase or if they can be realised afterwards.
- ✔ Redesign the SABRE application in a profound manner, with optimal scalability and performance in mind.
- ✔ Implement the SABRE prototype.
- ✔ Set up a test case and determine what methodology can be used to asses the performance and scalability.
- ✔ Execute the performance and scalability testing, including optimizations, and gather the results.
- ✔ Analyse the results and draw conclusions from them.
- ✔ Make recommendations based on the conclusions.

If all steps above are completed the final result is a thesis about the performance and scalability of a spatial database in a GIS-Web Service environment.

1.5 Project Goal

This research project has two distinct goals. The first, the redesign of SABRE, designed to be scalable, and the delivery of a prototype of a working demonstration version based on the new design. Second, investigate the behaviour of SABRE, test and improve, by means of optimizations, the performance and scalability.

1.6 Thesis Structure

This thesis contains a structure which can be seen as a sort of guideline throughout this thesis. The structure is based on the research questions. Each chapter answers one or more research questions. Chapter 2 discusses the first four research questions. Chapter 3, 4 and 6 discuss the fifth, sixth and seventh research question respectively. Chapter 5 discusses the tenth research question. Chapter 7 discusses the other research questions: eight, nine, eleven and twelve. In chapter 8 the conclusions are presented which describe the objective view on the outcomes of this project. The final chapter, chapter 9, contains the recommendations in which several recommendations for the future of SABRE are given.

2 Related Work

This chapter contains information on related work. It contains a description of the terminology, and related technologies. This chapter provides an answer to the first four research questions:

"What is, with respect to this project, performance and scalability?"

"What is a spatial database?"

"What is a GIS?"

"What is a Web Service?"

These terms are explained in this chapter. Amongst other definitions the terms performance and scalability are defined which are very essential for this project.

2.1 GIS

GIS is an abbreviation for Geographical Information System, an information system used for storing, retrieving and analysing geographical information. It is a computer system capable of integrating, storing, editing, analysing, sharing, and displaying geographically-referenced information. Basically, a GIS is a tool that allows users to create interactive searches, analyse the spatial information, edit data, maps, and present the results of all these operations. The application of GIS technology is very widespread, GIS can be used for:

- ✔ Scientific investigations
- ✔ Resource management
- ✔ Asset management
- ✔ Environmental impact assessment
- ✔ Urban planning
- ✔ Cartography
- ✔ Criminology
- ✔ History
- ✔ Sales
- ✔ Marketing
- ✔ Logistics

The usage of GIS application is growing rapidly, more and more GIS related applications are being used every day.

2.1.1 GIS Databases

There are several different databases which can manage spatial data. At the moment the most commonly used spatial databases are:

- ✔ Oracle Spatial
- ✔ PostgreSQL with PostGIS extension
- ✔ MySQL Spatial
- ✔ IBM DB2 with spatial extender

From the spatial databases above Oracle Spatial is by far the most dominant database, followed by PostgreSQL. Nowadays Oracle Spatial is used by many Geographical Information Systems, because with the birth of Oracle Spatial companies were able to program their own systems. Before Oracle Spatial the companies were dependant of the GIS vendors, therefore Oracle Spatial is rising in popularity. PostgreSQL is also rising in popularity but lacks the extensive amount of support that Oracle Spatial has. PostgreSQL is becoming rapidly more mature, but is no where near the maturity level of Oracle Spatial.

2.2 DB Techniques

A GIS uses digital information, the generation of the digital information can be done in several ways. The most common way is digitization, where a hard-copy plan or map is converted into a digital medium. There are two methods for storing the digital information inside the database, raster and vector. Raster data consists of rows and columns of cells where each cell contains a value. For example, a cell can contain a value representing a colour, whereas multiple cells together form a picture which can represent a map. Vector data is represented by geometries, points and lines, and or polygons, also called areas. Geometries and polygons represent objects. These objects together can be used to construct, for example a road-map. A GIS can perform spatial analysis on the spatial data, several examples of spatial analysis are:

- ✔ Data modelling
- ✔ Topological modelling
- ✔ Networks
- ✔ Cartographic modelling
- ✔ Map overlay
- ✔ Automated cartography
- ✔ Geo-statistics
- ✔ Geocoding
- ✔ Reverse Geocoding

Spatial analysis is based on mathematical calculations on raster and vector data. The mathematical calculations can be very diverse and application specific, most commonly used are graph theory and matrix algebra.

2.3 Benchmarking

Benchmarking of a database is the process of running a series of tests in order to asses the relative performance of the database. The results of a benchmark can be used to state meaningful information about the performance of a database as well as compare the performance with other benchmarks. There are several industry standard benchmarks available for benchmarking databases. These benchmarks are from the Transaction Processing Performance Council[TPC] and include the following benchmarks:

- ✔ *TPC-App*, an application server and Web Services benchmark.
- ✔ *TPC-C*, a benchmark which simulates a complete computing environment where a population of users executes transactions against a database.
- ✔ *TPC-E*, a benchmark for On-Line Transaction Processing (OLTP[2.1]).
- ✔ *TPC-H*, a benchmark for decision support systems.

These benchmarks are industry standard benchmarks, they provide a predefined simulated real-world scenario for databases and base the benchmarks on these simulated scenarios. Therefore the benchmarks do not represent the real performance of the database in its daily basis operation. For the project it is essential to benchmark the performance of the actual SABRE scenario and not a predefined simulated scenario. Therefore using an industry standard benchmark is not an option for this project.

There aren't many professional database benchmark programs available, at this moment, which can benchmark a actual database scenario. One such program is Benchmark Factory from Quest Software[BFQ], it features a high degree of customization. These customizations make it possible to create a tailored benchmark for the SABRE scenario. Therefore the logical choice of program used for benchmarking SABRE is Benchmark Factory.

2.4 Definitions

The research question for the thesis states: “Performance & Scalability of a Spatial Database in a GIS-Web Service Environment.” The sentence contains several words which can have different definitions. For the thesis it's essential to describe the definitions used, to uniformly define a terminology. The following terms need to be defined: performance, scalability, spatial database, GIS and Web Service. The three last terms are merely names for a specific technology whilst the first two are essential terms which need to be defined for the scope of the project.

- ✔ **Spatial Database**
  - *'A Spatial Database is a database that is optimized to store and query data related to objects in a defined geometric space, including points, lines and polygons. While typical databases can understand various numeric and character types of data, additional functionality needs to be added for databases to efficiently process spatial data types. These are typically called geometry or feature'.*[WKP1]
- ✔ **GIS**
  - GIS is an abbreviation for Geographical Information System, an information system used for storing, retrieving and analysing geographical information. See also chapter 2.1.
- ✔ **Web Service**
  - *'According to the World Wide Web Consortium*[W3C](*W3C) a Web Service is a software system designed to support inter operable Machine to Machine interaction over a network. The W3C Web service definition encompasses many different systems, but in common usage the term refers to clients and servers that communicate using messages (XML) that follow a standard (SOAP*[2.2]*-standard). Common in both the field and the terminology is the assumption that there is also a machine readable description of the operations supported by the server, a description (WSDL*[2.3]*)'.*[WKP2]

- ✔ **Performance**
  - ○ In perspective to the project the term performance is related to performance testing. '*In software engineering, performance testing is testing that is performed, to determine how fast some aspect of a system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability and reliability*'.[WKP3]
  - ○ In this project performance is measured as: (see chapter 6.3)
    - ■ Transactions per second.
    - ■ Response time in milliseconds.
    - ■ Transaction time in milliseconds.
- ✔ **Scalability**
  - ○ In perspective to the project, scalability is defined as the ability of a database to maintain throughput, measured in transactions per second, under an increased load in a graceful manner, whereas load is defined as query load, requests per second, or data set size, number of records inside a data set. For example if the query load increases, more requests per second, or the data set size increases, how does the database scale up against the increasing load in respect to transactions per second. If the load increases we expect to see a decrease in transactions per second. Scalability is about the relation between increase in load and decrease in transactions per second. If the decrease in transactions per second under an increased load is not in a graceful manner, the database does not scale. A database is said to be scalable if when we increase the resources in a system, it results in increased performance in a manner proportional to resources added.
  - ○ The idea behind scalability is that if a database is scalable it is possible to increase the resources to increase performance, in a proportional manner, thereby being able to cope with an increasing load. A database that does not scale has a proportionally low increase in performance while more resources are added.

2.5 Chapter Summary

This chapter has presented some terminology that is used for this project. The terms have to be described to gain a good understanding of this project. By describing these terms, an answer has been provided to the first four research questions as stated in chapter 1.2.

3 Design SABRE Application

This chapter discusses the redesign of the original SABRE application. It starts with an introduction to the original version and follows with the design and implementation of the new SABRE application. Design choices will be discussed as well as the requirements for SABRE. This chapter addresses the following research question: *"How can a database-design be realised?"*

The general idea behind SABRE is to be able to provide a service, based on a location. These services have to be customizable in a high degree. Therefore the design is centred around the idea of providing different services. Next to services, SABRE has to operate with locations. This results in a design with a relation between services and locations, each service can contain multiple locations. The idea is to map each service onto several locations, so each service queries only the locations it is mapped onto. As a result SABRE is, simply put, able to provide location based services, its only requirement to operate is the location of an object. For example, SABRE should be able to determine if an object is inside one or more areas. Which specific areas or locations to check against are determined by the service.

3.1 Requirements SABRE

With the redesign of the SABRE application came a lot of requirements. The requirements can be divided into functional and non-functional requirements. The non-functional requirements are the requirements that can be used to judge the operation of a system, rather than the specific behaviour. For SABRE the non-functional requirements consist of:

- ✔ **Performance**, SABRE has high performance requirements, it should be able to process at least 30 requests each second. This amount may likely increase to 60 request per second. The values of 30 and 60 are a rough estimate based on LogicaCMG's usage expectations.
- ✔ **Scalability**, SABRE needs to be a scalable application.
- ✔ **Security**, SABRE must be able to support authentication.
- ✔ **Extensibility**, SABRE must be designed in such a way that future Services can be easily added. The Extensibility requirement takes precedence over performance and scalability.
- ✔ **Use of Oracle Spatial** for the spatial processing functionality.
- ✔ **Use of a Web service** for the deployment of the SABRE services.

The functional requirements consist of:

- ✔ **Services**, SABRE must be able to offer location based services, referred to as Services. Given a location, denoted as a coordinate, of an object, SABRE must be able to offer Services based on the objects location. This design will only contain one type of Service, the AREA-event Service. The AREA-event Service determines whether or not a given object in inside a predefined area, based on the object's location.
- ✔ **Check Credentials**, this process is used to authenticate a user, it provides a means of security.
- ✔ **Check Coordinate**, this process checks the given coordinate, defined by X and Y, on whether it is in or out a predefined area. The process returns IN or OUT.
- ✔ **Create XML**, this process gathers the response from SABRE and encapsulates this response in a XML message to return to the Service Provider.

There are also four other functional requirements which are not part of the basic operation of SABRE.

- ✔ **Check Sensor**, this optional process can modify the Service_ID based on a sensor value. If the value of the sensor is above or below a defined threshold the Service_ID can be modified accordingly. This provides a means to change the locations to be checked, by altering the Service_id another Service is used and thus other locations are checked.
- ✔ **Process Status**, this optional process is used to determine if an object is possibly leaving or entering an area, based on the object's previous location.
- ✔ **Process History**, this optional process is able to return an object's location history.
- ✔ **Process AREA**, this optional process can insert, update or delete an area. Its purpose is to add, update or delete a location (or area) of a specific Service. Because this project only uses the AREA-event Service, so there is no need for AREA processing, this requirement is included in the SABRE architecture but intentionally left out in the SABRE detailed design as it is beyond the scope of this project.

The requirements contain al lot of terms which need some clarification, below is the terminology used shown.

- ✔ Object
  - *An object has always a location. The object itself is not used, only its location.*
  - *An Object_ID is a number used to uniquely identify an object.*
- ✔ Area
  - *An area is determined by a predefined set of coordinates.*
- ✔ Event
  - *An event determines the type of service, for example an AREA-event, see chapter 3.5 "SABRE Data Model" for more information.*
- ✔ Location
  - *Every object has a location, the location is determined by its coordinates.*
- ✔ Coordinate
  - *A coordinate is for spatial processing, a coordinate is a pair of an X and Y value.*
- ✔ Service_ID
  - *A Service_ID is a number to uniquely identify a Service, each Service has its own unique ID.*
- ✔ ACK
  - *Acknowledged, meaning accepted or admitted*
- ✔ NACK
  - *Not Acknowledged*
- ✔ Status
  - *The status of an object, whether it is entering or leaving an area.*
- ✔ History
  - *The history of an object is a list of the object's past locations. The list is linked to the object's unique Object_ID and contains coordinates representing the locations.*

3.2 Original version

SABRE in its original form was an application capable of some specific tracking and tracing. It was designed and implemented by LogicaCMG and intended as a training exercise for new LogicaCMG employees. Although a functional demo was created of the application showing promising capabilities, it did not meet the performance and scalability requirements for commercial exploitation. Because of the promising results of the prototype LogicaCMG decided that a renewed version of SABRE had to be made. The new version had to be designed in such a way it could cope with the high performance requirements next to being scalable and extensible, so it eventually could be used in a commercial environment.

The original version of SABRE consisted of a database containing locations and area's, and some lines of .NET code to perform the database transactions. It checked if an object was inside its defined area. The checking was done by spatial processing by the database, Oracle Spatial. The idea was to check if a given coordinate, the location of an object, was inside a predefined area, an Oracle Spatial geometry. Because Oracle Spatial performed very well, the redesigned SABRE application had also to be based on Oracle Spatial. Hence the requirement of usage of Oracle Spatial for the new SABRE application.

3.3 SABRE Architecture

The SABRE application can be divided into three different layers, each layer providing its own specific functionality. The idea behind this design is to see each layer as a component of SABRE. So SABRE consists of three different components. These components are the XML-component, the Web Service -component and the Database-component. Figure 3.1 shows a schematic view of SABRE.

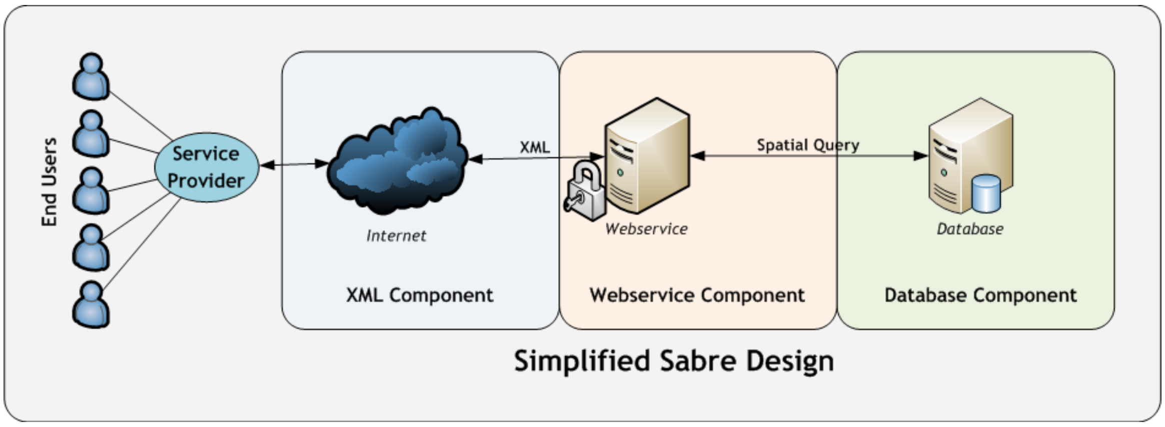


Figure 3.1: Simplified SABRE design

The diagram is a top-level view of SABRE, emphasizing the components marked by different colours. The components are described below along with their responsibilities.

XML-component

- ✔ Interface between the Internet and the Web Service
- ✔ XML file containing detailed information about available Services
- ✔ Handles all communication between the Internet and Web Service
- ✔ Provides security through means of authentication

Web Service-component

- ✔ A bridge between the Internet and the spatial database

- ✔ Delivers Services to customer and creates and modifies Services
- ✔ Responsible for all service related tasks
- ✔ Requires authentication
- ✔ Shields the database from the internet

Database-component

- ✔ Handles spatial processing requested by Web Service
- ✔ Communicates only with Web Service
- ✔ Handles primarily spatial-queries

By this design the architecture meets the requirements of security, services, use of a web service and use of Oracle Spatial. The remaining requirements of performance, scalability and extensibility are strongly related to the implementation of the data-model of the database and will be discussed in chapter 3.5. Next, the functional requirements will be discussed.

3.3.1 XML Component

The XML-component is the interface between the service provider and the Web Service. The service provider requests Services by sending a XML-encoded message. The Web Service replies through also sending a XML-encoded message, containing the reply. The are two different interfaces for the XML-component, the service provider interface and the Web Service interface.

The Service Provider interface provides the following functionality:

- ✔ Login to WS
- ✔ Life cycle:
  1. Requests available services
  2. Create service (subscribe)
  3. Modify service
  4. Delete service
  5. Start/stop service
- ✔ Operate service
- ✔ Supply object location
- ✔ Retrieve history
- ✔ Modify area

The Web Service interface provides the following functionality:

- ✔ Grant or deny access
- ✔ Supply answer based on object location
- ✔ Service modification confirmation (ACK/NACK)
- ✔ Supply object history

The XML-file should contain at least the following items. Items marked with * are mandatory.

- ✔ User name *
- ✔ Password *
- ✔ Service ID*
- ✔ X-coordinate *
- ✔ Y-coordinate *
- ✔ Object ID (optional, for history purposes)
- ✔ Sensor (optional, the sensor may contain more values, array-like)

3.3.2 Web Service Component

The Web Service component will only support one type of service, the AREA-event service. The function of this service is to determine if a given coordinate is inside a predefined area. The service can supply the following answers:

- ✔ INSIDE [YES]
  - ○ The given coordinate is inside the area
  - ○ BORDER [INSIDE]
    - ■ The given coordinate lies on the border of the area. The border of an area is part of the area itself, thus a coordinate which lies on the border is also within the area.
- ✔ OUTSIDE [NO]
  - ○ The given coordinate is not inside the area

3.3.3 Database Component

Three different databases can be distinguished for the SABRE application, each with its own specific functionality. The division into three separate databases is based on performance grounds, see chapter 3.6.1. The three databases are:

- ✔ Spatial Database (SDB)
- ✔ Non Spatial Database (NSDB)
- ✔ Web Service Database (WSDB)

The spatial database's main responsibility is to compute whether a given coordinate is within a given area. To gain maximum performance it is preferred to perform as minimal tasks as necessary at the SDB level. The interaction between the spatial database and the Web Service is shown in figure 3.2.

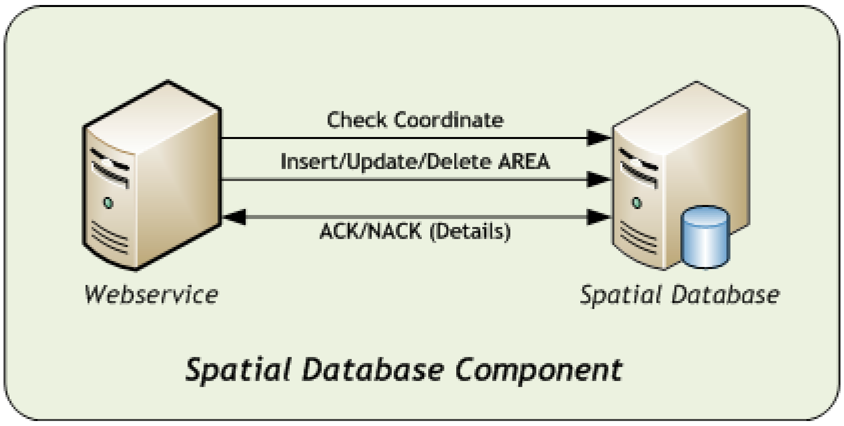


Figure 3.2: Interaction between Spatial Database and Web Service

The interaction consists of the following functionality:

- ✔ Web Service to SDB
  - ○ Check the given coordinate of an object against the location(s) corresponding to the requested service.
  - ○ Optional functionality:
    - ■ Insert/update/delete an AREA
- ✔ SDB to Web Service
  - ○ Reply [ACK/NACK] for an AREA modification or [IN/OUT] for a coordinate check.
  - ○ Optional functionality:
    - ■ Supply details on NACK, may contain a reason or description on what caused the NACK.

The requirement of the history functionality is provided by the NSDB. It is responsible for the issue of logging coordinates. The SABRE application should be able to keep track of coordinates supplied by the Service Provider. By means of storing all coordinates of a certain service by Object_ID, the application can supply the Service Provider with some sort of history. How the stored data is used to form a history isn't important right now, as it is beyond the scope of this design. The only thing important is that the coordinates will be logged and stored.

The NSDB consists of the following functionality:

- ✔ WS to NSDB
  - ○ Request history [Object_ID]
  - ○ Store coordinate [(X,Y),Object_ID]
- ✔ NSDB to WS
  - ○ Supply object history [Object_ID,(X,Y)1,...,(X,Y)n]

There is also a third Database, the Web Service Database. The WSDB has functions specific for the Web service. The WSDB has the following functionality:

- ✔ Check supplied credentials, providing a security measurement.
- ✔ Process Status, to determine if an object is entering or leaving an area.

- ✔ Check Sensor, may change the Service_ID to address another Service.

The Web service, WSDB interaction consists of the following functionality:

- ✔ WS to WSDB
  - ○ Check user name/password to allow or disallow access to the Service.
  - ○ Process Status and determine the state of the object.
  - ○ Check the sensor based on the sensor-value and determine if the Service_ID needs to be changed.
- ✔ WSDB to WS
  - ○ Reply if the login was successful [Y/N]
  - ○ Return the status of the object [Enter/Leave]
  - ○ Return the Service_ID

3.4 Detailed SABRE Design

Figure 3.3 shows the design of the SABRE application in more detail.

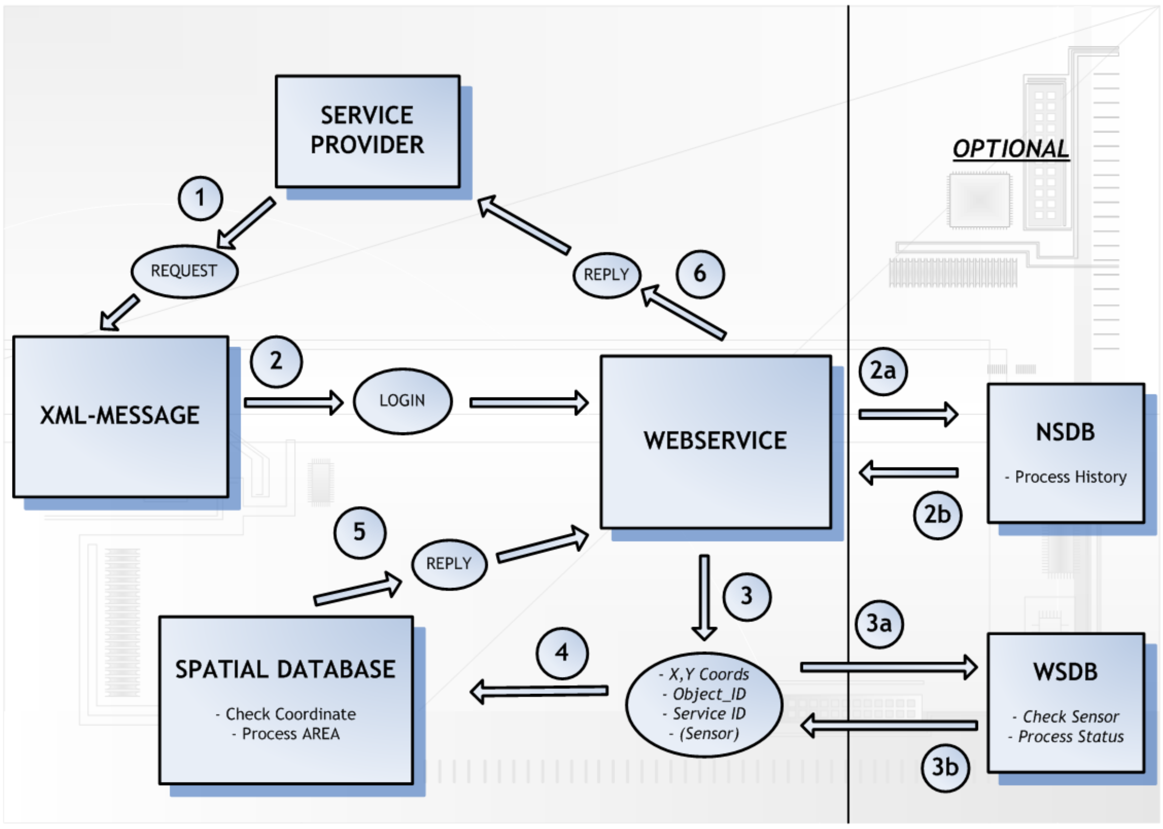


Figure 3.3: Detailed SABRE Design

The figure is zoomed in on the different components and the communication, including communication contents, between them. The optional parts are separated by the vertical line whereas the database component is split into the three different databases to accommodate the optional parts.

The figure shows the contents of the communication between the components in the ovals between the arrows. The numbering denotes the order in which the flow of data takes place. The optional routes, 2a followed by 2b and 3a followed by 3b, show the optional processes of history, sensor check and status.

Figure 3.4 shows the SABRE data flow diagram.

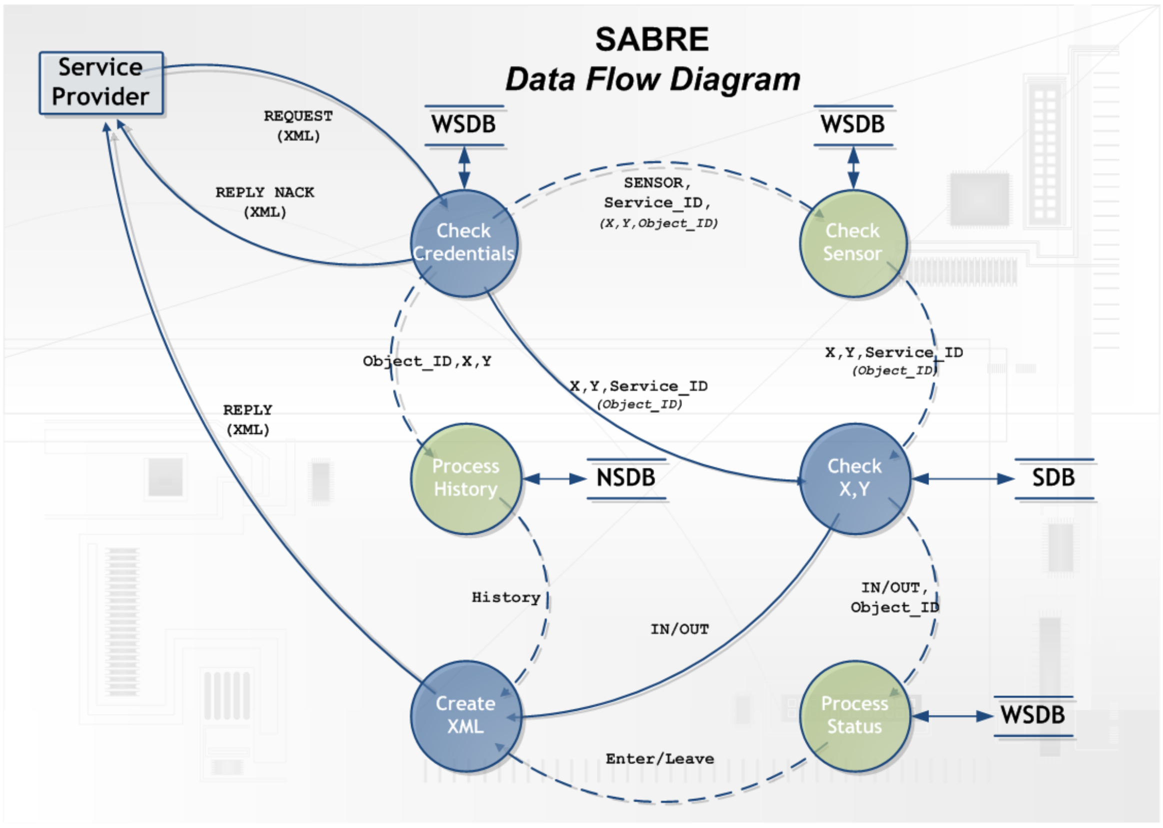


Figure 3.4: SABRE data flow diagram

This figure shows the flow of data as seen from the service provider, it shows the possible routes a request can travel. Each single request travels through the data flow diagram where it can take its desired route. The data flow diagram consists of five routes but a single route can be taken only one at a time, the flow is always forwards in the direction of the arrows. The text alongside the arrows represents the contents of the data, the text between parentheses represent the data which is inside the flow but not needed for the process the flow is heading to. The three databases, SDB, NSDB and WSDB are displayed as a data store, a location where data is held temporarily or permanently. The WSDB is addressed three times whereas the other two data stores are only used once. Each circle defines a process, the process takes the input data and creates (transformed) output data. These processes are:

- ✔ Check Credentials
- ✔ Check Sensor*
- ✔ Check X,Y (Coordinates)
- ✔ Process Status*

- ✔ Process History*
- ✔ Create XML

Note that the processes are directly derived from the functional requirements. The processes marked with * are optional processes, the data flow for these processes is marked by a dashed line in the data flow diagram, whereas the processes itself are marked green instead of blue.

The following sequence diagrams are derived from the data flow diagram. The sequence diagrams show the required components for the (optional) functional requirements of SABRE, as parallel vertical lines, the different processes or objects that live simultaneously and the messages that are exchanged between them, in order of occurrence. There are five different sequence diagrams each representing a different route in the data flow diagram.

- ✔ Basis sequence diagram, showing the basic function without any optional functionality.
- ✔ Sensor check sequence diagram, showing basic functionality including the sensor process.
- ✔ Status sequence diagram, showing basic functionality including the status process.
- ✔ Sensor check & Status sequence diagram, showing basic functionality including the sensor and status process combined.
- ✔ History sequence diagram, showing basic functionality including the history process.

The sequence diagram in figure 3.5 shows the basic operation of SABRE.

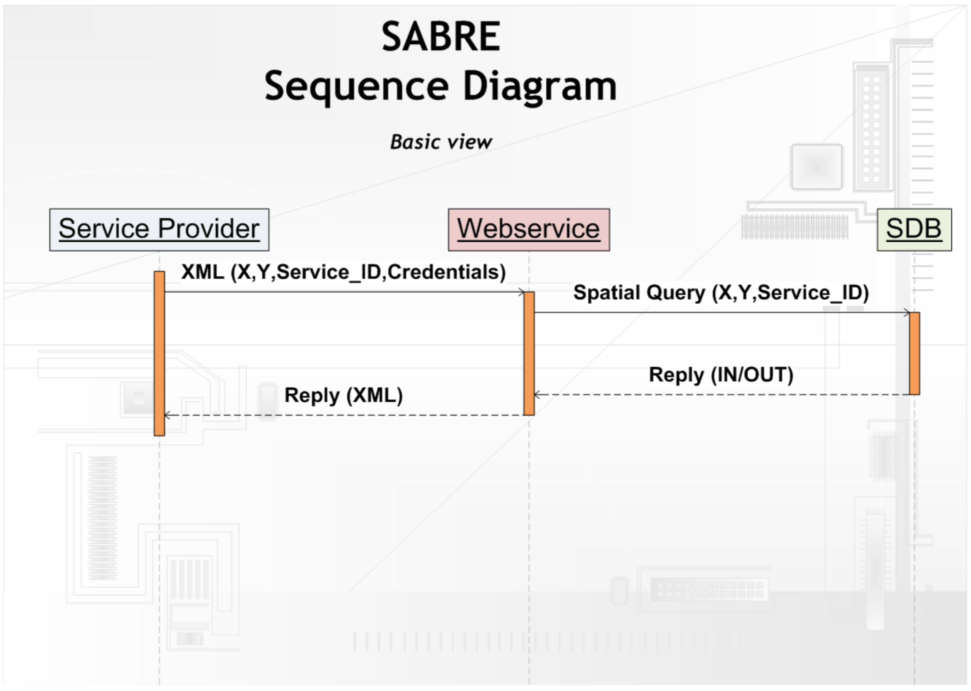


Figure 3.5: SABRE sequence diagram, Basic view

The figure shows the sequence of processing a request. The text alongside the lines represent the data of the request. The vertical orange bars denote the relative time required by the components, displayed in the text-boxes, to process the request. Figure 3.6 shows the sequence diagram including the sensor check process. There is an extra process of addressing the WSDB which may result in a change of Service_ID.

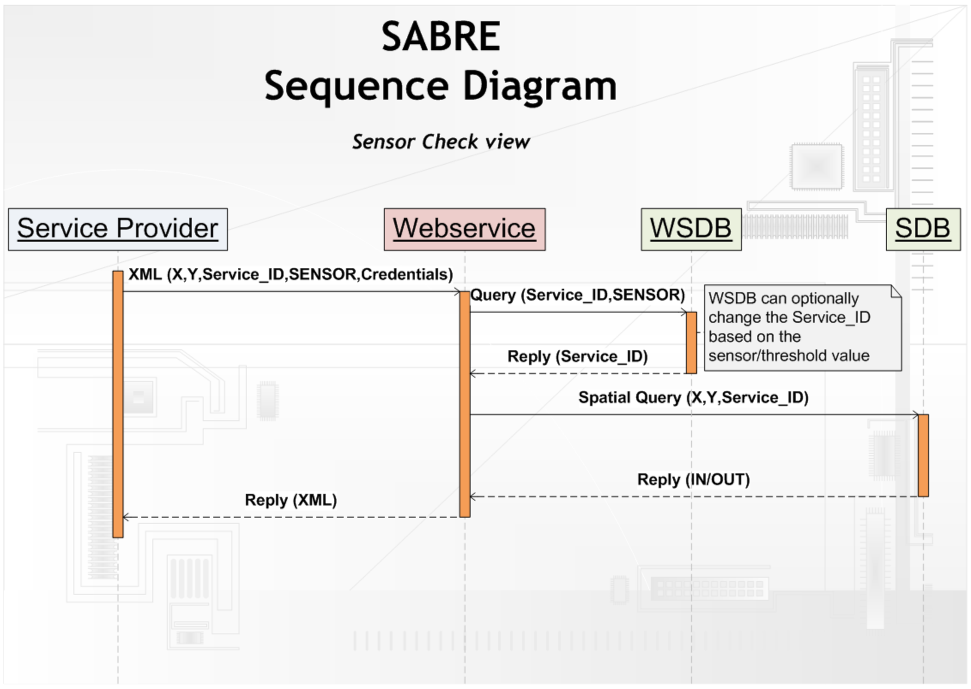

SABRE
Sequence Diagram
Sensor Check view
Service Provider
Webservice
WSDB
SDB
XML (X,Y,Service_ID,SENSOR,Credentials)
Query (Service_ID,SENSOR)
WSDB can optionally change the Service_ID based on the sensor/threshold value
Reply (Service_ID)
Spatial Query (X,Y,Service_ID)
Reply (IN/OUT)
Reply (XML)

Figure 3.6: SABRE sequence diagram, Sensor Check view

Figure 3.7 shows the sequence diagram including the status process.

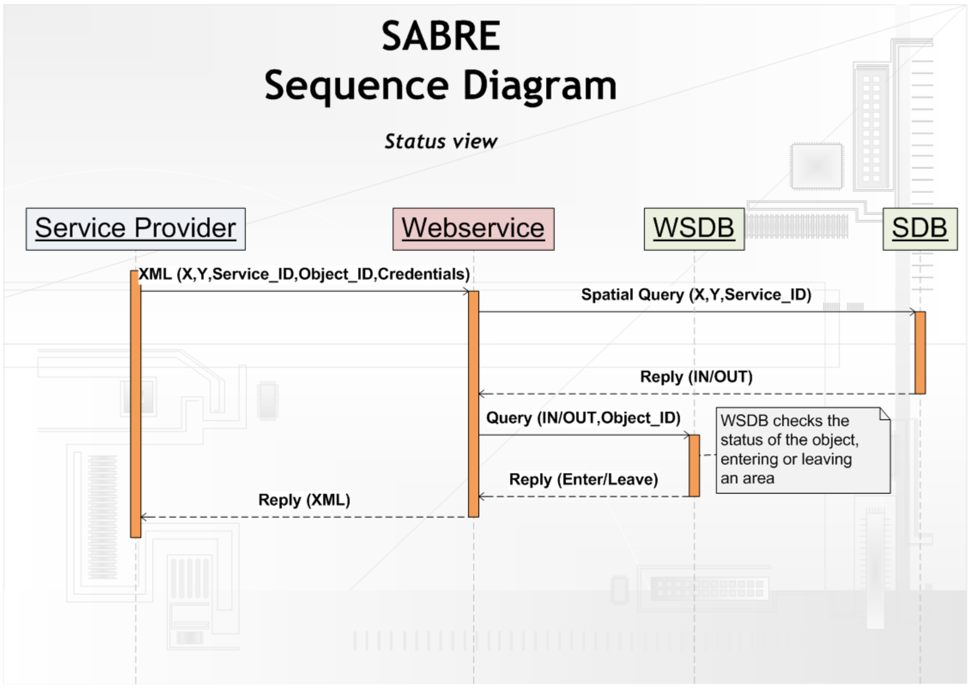

SABRE
Sequence Diagram
Status view
Service Provider
Webservice
WSDB
SDB
XML (X,Y,Service_ID,Object_ID,Credentials)
Spatial Query (X,Y,Service_ID)
Reply (IN/OUT)
Query (IN/OUT,Object_ID)
WSDB checks the status of the object, entering or leaving an area
Reply (Enter/Leave)
Reply (XML)

Figure 3.7: SABRE sequence diagram, Status view

The diagram shows an extra process of addressing the WSDB to determine if the object is entering or leaving an area. This process takes place after the SDB has been addressed to check the coordinate. Figure 3.8 shows the sequence diagram including the sensor check and status process.

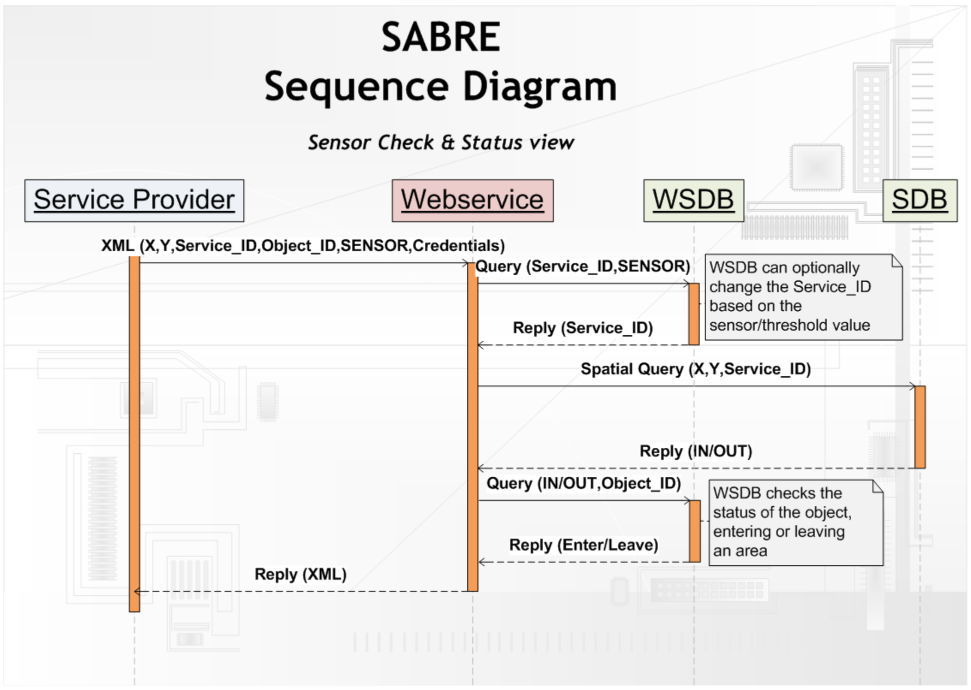

SABRE
Sequence Diagram
Sensor Check & Status view
Service Provider
Webservice
WSDB
SDB
XML (X,Y,Service_ID,Object_ID,SENSOR,Credentials)
Query (Service_ID,SENSOR)
WSDB can optionally change the Service_ID based on the sensor/threshold value
Reply (Service_ID)
Spatial Query (X,Y,Service_ID)
Reply (IN/OUT)
Query (IN/OUT,Object_ID)
WSDB checks the status of the object, entering or leaving an area
Reply (Enter/Leave)
Reply (XML)

Figure 3.8: SABRE sequence diagram, Sensor Check & Status view

The sequence diagram is a combination of the sensor check diagram and the process status diagram. The sensor check is executed first, followed by the coordinate check, followed by the status processing. Figure 3.9 shows the sequence diagram including the history process.

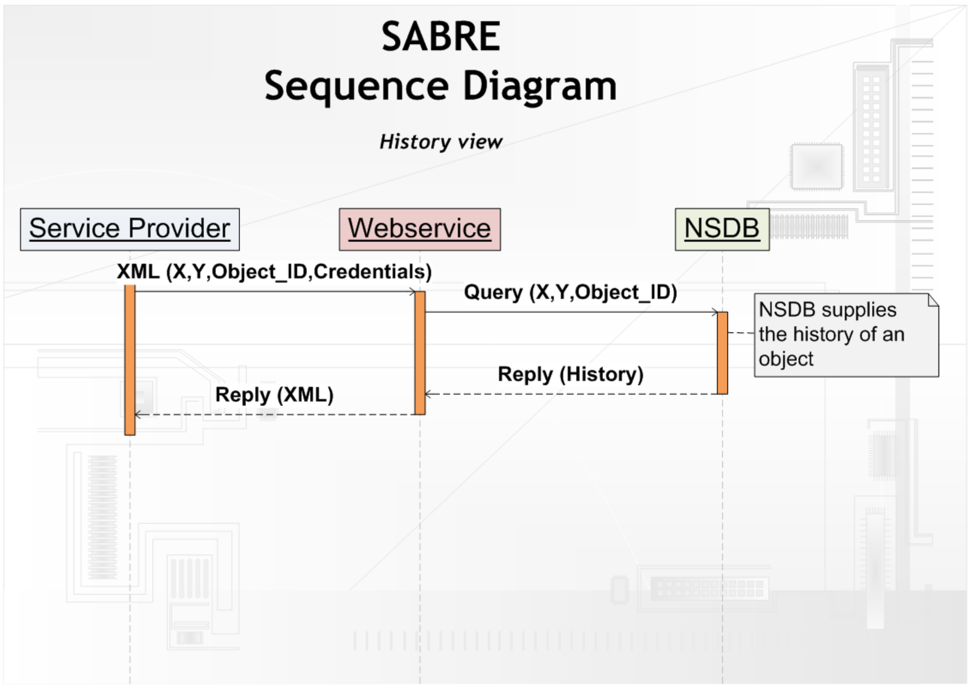


Figure 3.9: SABRE sequence diagram, History view

As the sequence diagram shows the history process requires the NSDB, the SDB is not required to retrieve the history. The history is retrieved by supplying an Object_ID to the NSDB, the NSDB will reply with the history of an object corresponding to the Object_ID.

The data flow diagram and the five sequence diagrams can be found in a larger format in appendix A.

3.5 SABRE Data Model

The SABRE data model is a fundamental aspect of the SABRE application, it represents the internal table structure of the Spatial Database (SDB). The requirements for the data model are in order of importance:

1. Extensibility, the data model must be able to support feature services, without altering the model.
2. Scalability, the data model has to be scalable, so the SABRE application can be scalable.
3. Performance, as high as possible performance must be achieved in conjunction with the other two conditions of requirements.

So the data model has to be designed for extensibility and scalability, whereas the performance has be to maximized.

The data model is centred around the idea of providing different services. Next to services, SABRE has to operate with locations. This results in a data model with a relation between services and locations. The relation is of type many-to-many, each service can contain multiple locations, whereas each location can be used by multiple services. The idea is to map each service onto several locations, so each service queries only the locations it is mapped onto.

To provide extensibility, for future additions, there is another entity called Event. An event determines the type of service, this project focuses only on the AREA-event service, whereas there may be more types of events added in the future to SABRE such as, for example, a Route-event. The three entities service,event and location are the backbone of the data model, therefore these entities can directly be used as tables, where the location entity is denoted as table Geometry. The idea is that service has a many-to-many relation with Event, each service can consist of multiple events and each event can be used by multiple services. This accommodates for a high degree of customisability, which is required for extensibility. The entity event has also a many-to-many relation with Location, thus the event entity can be seen as a bridge between service and location, providing extra customisability.

There are two other entities called EventSupl and Event_Types which are related to the event entity. EventSupl provides optional, supplemental information for the event entity. Event_Types is an extra table which determines which specific spatial processing function the database has to perform. These tables are necessary to accommodate any future additions which may require additional information and/or the use of other spatial processing functions.

There is an issue of how to establish a many-to-many relation between the entities. The issue is solved by adding another entity which sole purpose is to establish a many-to-many relation between to other entities. The data model contains four entities each which require such an extra entity, one for the relation between service and event, called R_Service_Event, one for the relation between event and location, called R_Event_Geometry, the final one, called R_Event_EventSupl is for the relation between Event and EventSupl. The entities together form the data model of SABRE, which is show in figure 3.10.

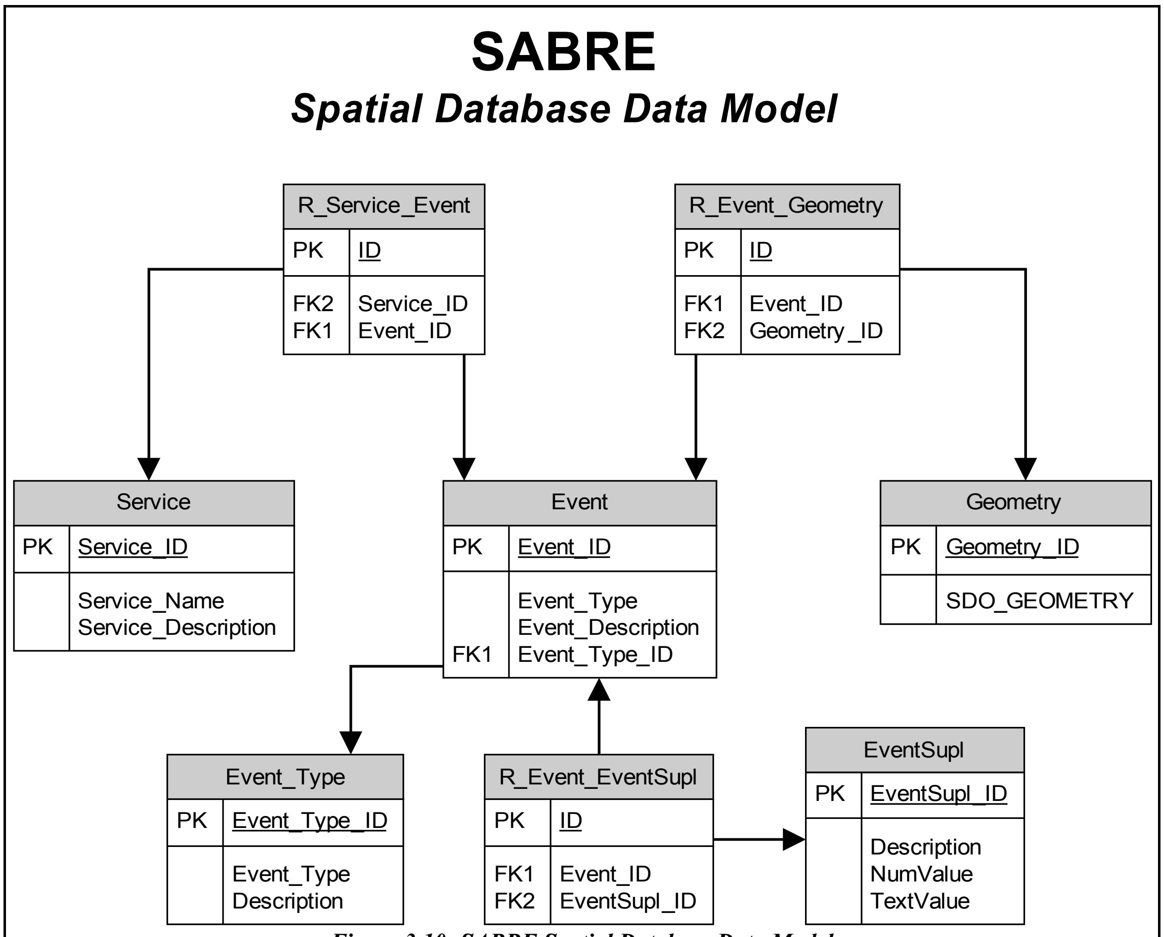


Figure 3.10: SABRE Spatial Database Data Model

As seen in the figure there are in total eight tables in the data model. In the middle of the figure are the three tables which provide the core functionality of SABRE, Service, Event and Geometry. The data types of the columns of the tables can be found in appendix B.

The data model as described is according to the first requirement for the data model, extensibility. However, the other two requirements, performance and scalability suffer from this design. For optimal performance and scalability the number of tables used should be kept to a minimum, possibly even only one, because for each table used in a query there is a costly join needed. Even if a simple query is performed on this data model all eight tables are needed and joined together, which can have a serious impact on the performance and scalability when the number of rows in each table increases. Therefore the current form of the data model uses the minimum amount of tables while still obeying to the requirement of extensibility. If any table would be taken out of the data model the SABRE application would loose its extensibility. Other data model designs haven been taken into consideration, although none of them provides the high degree of extensibility this model does. Since LogicaCMG expects the SABRE application to be greatly extended in the near future, if it shows promising results, they required SABRE to have a high degree of extensibility. Therefore the current form of the data model had to be adopted. The performance and scalability requirements were subject to the extensibility requirement. As long as the requirement of extensibility takes precedence over the requirements for performance and scalability the current data model is under the circumstances optimal for SABRE.

3.6 Design Choices

The design of SABRE is accompanied by several design choices whereas the choices are heavily dependant on the non-functional requirements. The relevant design choices that need to be discussed are addressed below.

3.6.1 Division into three separate databases

The Database component is required for storing, retrieving and updating data. Both processing of standard and spatial data is required for SABRE. A simple widely adopted solution is to use a single database which can also handle spatial data next to standard non spatial data. Most designs contain only one single database, aspects that favour for a single database are:

- ✔ Manageability
- ✔ Cost
- ✔ Maintainability
- ✔ Compatibility

The SABRE application however uses three independent databases, each database has its own specific functionality. The motivation for usage of three instead of one database is the performance aspect. The performance of the spatial processing of the database has to be maximized in order to handle as many requests per second as possible. To maximize the performance of spatial processing all other non-spatial processing has to be minimized as they influence the performance in a negative manner. The best solution is to isolate the spatial processing and set up a database dedicated to spatial processing only. The idea behind this concept is that SABRE requires different types of time-related data processing, spatial processing is time-critical because of response time requirements, whereas other non-spatial requests are less time-critical or not time critical at all. Therefore the database component consists of the following three databases.

- ✔ DB I: Spatial Database (SDB)
  - *High performance DB*, only for spatial processing.
- ✔ DB II: Web service Database (WSDB)
  - *Medium performance DB*, supplies specific functionality for the WS.
- ✔ DB III: Non Spatial Database (NSDB)
  - *Low performance DB*, used for additional, non time critical, functionality.

Each database is dedicated to a specific purpose, whereas the databases can be divided based on their performance requirements. By this design SABRE achieves maximum performance for its spatial processing in order to handle as many requests per second as possible, which is essential to the performance requirement.

3.6.2 Data Model

The data model has as stated before three requirements of which the first and most important is extensibility. The other two requirements performance and scalability are seriously reduced by the first requirement. If we would design the data model for performance and scalability it would reduce the extensibility. Either way around there is no ideal solution to this problem. There are two possibilities to compensate:

1. Create a data model based on extensibility, try to improve the performance and scalability with other techniques in a later stadium.

2. Create a data model designed for performance and scalability, try to implement extensibility in a later stage by other means.

As stated before the requirement for extensibility takes precedence over the other two requirements therefore the data model is based on extensibility and the first possibility for compensation is used. There is also another reason for choosing to base the data model on extensibility, the first possibility, improve performance and scalability in a later stadium, seems to be very feasible and a realistic possibility. There are two ways to compensate for the high amount of tables used in the data model, these are optimizations on the database level. These optimizations are:

- ✔ SQL Tuning, improve the query as to minimize the amount of joins required.
- ✔ Materialized Views, create a precomputed view to eliminate joining completely.

The optimizations are discussed in chapter 5.

3.7 Chapter Summary

This chapter described the requirements and design of SABRE. The research question *"How can a database-design be realised?"* has been answered since this chapter presented a (database-)design for SABRE. The design is based on extensibility and requires three separate databases. The performance and scalability of this design has still got to be investigated.

4 SABRE Prototype

This chapter describes the realisation of an prototype based on the SABRE design as discussed in chapter 3. The requirements of the prototype are discussed, next the implementation and realisation are described followed by an evaluation of the prototype. This chapter addresses the following research question: *“How can the scalable database-design be implemented?”*

The SABRE prototype is an implementation of SABRE solely made for this research project. The purpose of the prototype is to provide a means to research the performance and scalability of SABRE. The prototype contains only the bare necessities to accommodate the research and to be a profound realisation of SABRE as well.

4.1 Requirements

The requirements for SABRE have been discussed in chapter 3, the SABRE prototype is based on these requirements, nonetheless the optional requirements are left out. The processes that are not implemented in the prototype are:

- ✔ Check Credentials
- ✔ Check Sensor
- ✔ Process Status
- ✔ Process History

The data model, as described in chapter 3, is essential to the functionality of SABRE, therefore the prototype implements the data model and adheres to it.

The SABRE prototype design focusses on the database component. Following the SABRE design the database component consists of three databases, the prototype however, implements only one database, the spatial database. The other two databases are outside the scope of this project.

The prototype consists of a front-end, a web service and a spatial database. The front-end constructs a request in XML and sends the request to the web service. The web service takes the request and queries the spatial database based on the request. The spatial database returns the request-responses to the web service which in turn sends the results back to the front-end.

The front-end must be able to formulate a XML-request including a coordinate, defined by an X and Y value, and a service_id. Next to creating the request the front-end is responsible for displaying the results to the user. The web service takes the XML-message from the front-end and extracts the coordinate and the service_id. The web service passes the coordinate and the service_id as arguments of a function to the database. The database processes the function and returns the results to the web service.

Figure 3.5 'SABRE sequence diagram, Basic view' shows the sequence diagram of the SABRE design which is almost the same for the prototype. The only difference is the absence of credentials and the service provider part being replaced by the front-end. The front-end simulates the part of the service provider, it creates and sends requests to the web service, which results in an environment suitable for the performance and scalability testing of this project. Since the front-end only creates and sends requests to the web service, it does not have any performance impact.

The SABRE design contains the check coordinate function, which purpose is to check whether or not a given coordinate lies within one or more areas, to support the AREA-event Service. An area consists of several coordinates which, when 'connected', define the area. The prototype, however, will only have to be be able to handle locations which consist of point-data, in other words locations defined by a single coordinate. For the prototype to still be able to check if a coordinate lies within a certain area, an area has been redefined as a point with a certain radius. The radius creates a circle around the point with a diameter of twice the radius, so the circle can be considered to be an area. The checking of the coordinate is now done by checking if the given coordinate lies within distance of the points associated with the required service. If the given coordinate lies within distance of the point, it is said to be within an area. For every point that lies within distance of the given coordinate the name of the point is returned to the web service instead of IN or OUT.

4.2 Implementation Prototype

The implementation and realisation of the prototype can be roughly divided into three parts, together these parts make up the SABRE application. The first part, the data model, focusses on the implementation of the SABRE data model into the prototype. The second part, the SABRE package, focusses on the SABRE package which provides the required functionality of the database. The third and last part, the web service, focusses on the realisation of a web service component.

4.2.1 Realisation SABRE Data Model

The core of the prototype is the SABRE data model. The data model is implemented in a spatial database, Oracle Spatial 10g. The data model is implemented in the database so that each entity in the data model represents a table in de database. The relation between the tables are based on the primary keys and foreign keys of the tables, accordingly to the data model.

Tables are created with a basic create table syntax. The lines of code to create the tables in the Oracle Spatial 10g database can be found in appendix C. There are three different data types used for the columns in the tables, NUMBER, VARCHAR2 and SDO_GEOMETRY. A Varchar2 is a data type which supports multiple different characters, in this case up to 4000, therefore the data type can support text, whereas the number data type can only contain one number. The number has in some cases a constraint applied, *'not null'*, which indicates that there must always be a value present for that column. The SDO_GEOMETRY data type is a specific data type for Oracle Spatial, this data type is required for spatial processing. The SDO_GEOMETRY data type is a complex data type and contains coordinates of a location. Appendix D provides additional information about the SDO_GEOMETRY data type. The relations between the tables are based on primary and foreign keys, the lines of code which establish the relations can be found in appendix E. A foreign key from one table references a primary key from another table, thereby establishing a relation between two tables. There are also some other constraints present, these constraints apply to the tables used to establish a many-to-many relation, the constraint enforces that the combination of two values is unique for the table the constraint applies to. This enforcement is necessary to prevent having multiple identical rows in a table, which eventually would lead to duplicate query-results, which is not desirable. The constraint only applies when records are inserted or updated, which does not occur with the prototype. Therefore there is no potentially caused performance impact by the constraints for the prototype.

Each table has as as first column a unique number, the ID. The ID is required to uniquely identify a row. For each inserted row in a specific table a new unique number should be assigned to the inserted row. One way to achieve the unique numbering is to increase the number by one for each inserted row. Some databases already have an auto-numbering mechanism which automatically increases the number and assign it to the inserted row. Oracle, however, has no such mechanism, the process of unique numbering of the inserted rows is left entirely to the user. The SABRE data model requires unique numbering for querying purposes. So, for this project it is essential to come up with some sort of unique numbering to implement the SABRE data model. It is possible to accomplish automatic unique numbering in Oracle by creating a mechanism which increases and sets a number on each row inserted. This mechanism., shown in figure 4.1, requires the use of two other database elements, triggers and sequences.

```
CREATE SEQUENCE  "SERVICE_ID_SEQUENCE"  MINVALUE 1 MAXVALUE
9999999999999999999999999999 INCREMENT BY 1 START WITH 1 CACHE 20 NOORDER  NOCYCLE ;

CREATE OR REPLACE TRIGGER "SERVICE_ID_TRIGGER"
BEFORE INSERT
ON service
REFERENCING NEW AS NEW
FOR EACH ROW
BEGIN
SELECT service_id_sequence.nextval INTO :NEW.service_ID FROM dual;
END;

ALTER TRIGGER "SERVICE_ID_TRIGGER" ENABLE;
```

Figure 4.1: The Oracle unique-numbering mechanism

The figure shows the unique numbering mechanism for the Service table, for all other tables the same procedure applies, in an analogue manner. The idea is to create for each table a corresponding sequence, initially set at zero, each call to *'sequence.nextval'* increments the specific sequence by one. The call to *'nextval'* is issued by a trigger which is triggered just before an insert statement takes place. The trigger retrieves the value of *'sequence.nextval'*, which is the current sequence number incremented by one, and sets the value for the currently being inserted row. The final effect is that on each inserted row automatically an unique number is applied, which is used by the database to uniquely identify the row. The usage of triggers and sequences comes with a small, nowadays insignificant, cost, on each row inserted there is a little extra overhead. However, the overhead occurs only when the unique-numbering mechanism is applied. Thus there is only overhead when new rows are inserted, which does not occur with the prototype. Therefore the performance of the prototype is not influenced by the unique-numbering mechanism.

4.2.2 Realisation SABRE Package

The spatial database contains a package, a package is a group of procedures, functions, variables and SQL statements created as a single unit which is used to store together related objects. The SABRE package contains stored procedures, which contain the actual SABRE SQL-code. The package is written in PL/SQL[4.1]. The PL/SQL package, containing functions, is published to the web service, the functions can be called from within the web service by addressing a specific function inside the package. The package is responsible for querying the database and returning the results to the web service. The package is called SABRE, its code can be found in appendix F.

A package consists of a package specification, or header, and the package body which contains the actual code. Figure 4.2 shows the package specification of package SABRE.

```
CREATE OR REPLACE PACKAGE "SABRE" AS
function NN (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return places;
function WD (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return places;
function SABRE (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return places;
END SABRE;
```

Figure 4.2: Package specification SABRE package

The SABRE package contains three different functions:

- ✔ SABRE (number x, number y, number service_id), see figure 4.3
- ✔ WD (number x, number y, number service_id), see figure 4.4
- ✔ NN (number x, number y, number service_id)

(The x and y in the function call represent the coordinates (x,y) of a location.)

The package's main function is function SABRE, which can be seen as the entry point of the package. Function SABRE receives as input-values the coordinates and the service id. Based on the service id the function determines which function to call next, WD or NN. The PL/SQL-code of function SABRE is displayed in figure 4.3.

```
function SABRE (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return PLACES IS
BEGIN
declare
finalresult PLACES;
service_type_id number;

Begin
select e.event_type_id INTO service_type_id from event e, service_2_event s2e
where s2e.service_id=SID
and s2e.event_id=e.event_id
and rownum=1;

IF service_type_id = 1
THEN
finalresult := NN(X,Y,SID);
ELSE
IF service_type_id = 2
THEN
finalresult := WD(X,Y,SID);
END IF;
END IF;

RETURN finalresult;
END;
END SABRE;

end SABRE;
```

Figure 4.3: PL/SQL-code containing function SABRE

By addressing the view *'service_2_event'*, which maps services onto events, the SABRE function determines the event_id that corresponds to the given service_id. Next, by looking up the event, from its id, the function can determine the service type. By the number of the service type the function knows which function to call next. This is an simple but effective mechanism to support calling of different functions. Therefore, multiple new functions can be added, which supports extensibility. This mechanism seems rather inefficient, for every request the mechanism is used before the actual request is processed, which should lead to extra overhead. However, performance tests have pointed out that there is no significant difference in performance when using this mechanism compared to directly calling the required function. Which is remarkable because of the expected negative performance impact when using this mechanism. A plausible explanation could be that the overhead is of no significance because of the processing power of nowadays computers. While older computer with less processing power could be significantly influenced, the processing power of computer these days is relatively so immense it can process the mechanism so fast it does not cause a negative performance impact any more. This results in the performance impact of the mechanism being insignificant with respect to the processing of the actual request.

The SABRE prototype supports only two spatial operators:

- ✔ SDO_WITHIN_DISTANCE (SDO_WD), see appendix G.
    - ○ Determines which objects are within a specified distance of each other.
- ✔ SDO_NN (Nearest Neighbour), see appendix H.

- Determines which object is closest to a specified object.

These spatial operators are encapsulated in the other two functions of the SABRE package. Operator SDO_WD is encapsulated in function WD, whereas operator SDO_NN is encapsulated in function NN. Function WD or NN are called by function SABRE, the required parameters, coordinates and service id, are passed along with the call to function WD or NN. When function WD or NN has finished it passes back the results to function SABRE, which in turn will return the results to the web service.

Function WD is the only function, containing a spatial operator, which is in the scope of this project. Function NN is beyond the scope of this project and is only supported to demonstrate other abilities and, or possibilities of the SABRE application. By simply adding a different spatial operator encapsulated in a new PL/SQL function, completely different results can be obtained, which is a proof-of-concept of the ease of extensibility of SABRE. For this project however, the focus is only on function WD and its spatial operator SDO_WD. The general idea is that SDO_WD compares different geometries and determines whether or not they are within a specified distance of each other. SABRE requires SDO_WD to check a given coordinate against a list of geometries determined by the service id. The PL/SQL-code of function WD is displayed in figure 4.4.

```
1  function WD (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return PLACES IS
2  BEGIN
3  declare
4  wd_rst PLACES:=PLACES();
5  j number :=1;
6  CURSOR wd_cur IS
7  select g.name from geometry g,service_2_geometry s2g
8  where g.geometry_id=s2g.geometry_id
9  and s2g.service_id=SID
10 and SDO_WITHIN_DISTANCE(g.geometry,sdo_geometry(2001,null,SDO_point_type(X,Y,null),
11 null,null),'distance=500')='TRUE';
12 BEGIN
13 OPEN wd_cur;
14 LOOP
15 EXIT WHEN wd_cur%NOTFOUND;
16 wd_rst.extend;
17 FETCH wd_cur INTO wd_rst(j);
18 j := j+1;
19 END LOOP;
20 CLOSE wd_cur;
21 RETURN wd_rst;
22 END;
23 END WD;
```

Figure 4.4: PL/SQL-code containing function WD

The function contains basic PL/SQL declarations (lines 3-6), the actual SQL-statement containing the spatial operator (lines 7-11) and a mechanism of fetching the results into the user-defined return-type places (lines 12-20). The mechanism for the fetching the results is required for returning the results to the web service, because the web service requires the results of an SQL-statement to be contained within a table because of its implementation limitations.

The SQL-statement returns the names of the geometries within the distance of 500 units of the given coordinate. The distance used is of great influence to the performance of SABRE. The prototype implements only one static value which is set at 500. The value of 500 ensures that each queried dataset returns at least one result. Varying the distance is beyond the scope of this project but would be an interesting aspect for future research. First, the service id is mapped to its corresponding geometries by the *'service_2_geometry view'*, to create a list of possible geometries for the result-set. Second, the list of geometries is passed onto the SDO_WD operator which checks which geometry is within distance of the given coordinate. To perform a within distance check the given coordinate needs to be transformed into a geometry first, after which spatial processing can take place. Finally, only the names of geometries corresponding to the service id and which are within distance of the given location, are returned. Hence the return-type 'PLACES'. which is a user-defined return-type which consist of a table containing text, the names of the geometries in the result set.

4.2.3 Realisation Web Service

The realisation of the web service for the SABRE prototype is a fairly easy straightforward process. The entire web service is created and instantiated by Oracle Containers for JAVA[OC4J] (OC4J), with the aid of Oracle's integrated development environment called JDeveloper[JDEV]. The only aspect taken into consideration is the mapping of data types, whilst the web service does not know which kind of data type to expect. JDeveloper does however, know how to map data types as long as they are contained inside a table, as return-type PLACES is, so the web service knows it can expect a table.

The aided realisation of the web service by JDeveloper can be seen as a quick-and-dirty solution, which normally is not adequate for usage in a commercial environment. However, the web service component is beyond the scope of this project, the focus is on the database component. Therefore the prototype has no requirements for the web service, the web service created by JDeveloper is sufficient enough.

4.3 Chapter Summary

This chapter describes the implementation of a prototype of SABRE which can be used to perform research on the performance and scalability of SABRE. The research question *“How can the scalable database-design be implemented?”* has been answered and explained throughout this chapter. The prototype is, although it has reduced functionality, a sound implementation of the design. The results of the research on performance and scalability should give a realistic representation of the performance and scalability of a fully implemented version of SABRE.

5 Optimizations

This chapter contains an in-depth description of the optimizations applied to the database. The goal of applying the optimizations is to improve performance and scalability. The optimization to be discussed are SQL Tuning, Materialized Views and Range Partitioning. This chapter addresses the following research question *"Are there possibilities to significantly improve the performance and scalability by means of database optimizations?"* Each optimization will be described, it will contain a technical description as well as the expectations of applying the optimization.

5.1 SQL Tuning

Implementation of decent SQL is essential for performance and scalability of a database. An inefficient implementation of a piece of SQL-code may become the main bottleneck in terms of performance for a database. Therefore the possibility of a piece of SQL-code being the main bottleneck must be ruled out. To rule out the possibility all SQL statements have to be analysed. The cost of a query needs to be calculated based on the query execution plan. When the cost is known it is essential to try to lower the cost of a SQL statement as much as possible by altering the statement or applying hints to the execution plan.

Oracle Spatial supports its own spatial index, MDSYS.SPATIAL_INDEX. The spatial index is similar to conventional indexes such as B-Trees. The spatial index is implemented as an R-Tree index, a hierarchical structure, like B-Trees, that stores rectangle approximations of geometries as key values. For each geometry the R-Tree defines the minimal bounding rectangle[5.1] (MBR) enclosing the geometry, and it creates a hierarchy of MBRs. The spatial index has several parameters. By setting the LAYER_GTYPE parameter, it can speed up the query operators thereby enhancing the performance of the index. A technical description of the spatial index can be found in appendix I.

5.1.1 Technical

SQL tuning contains two separate optimizations, tuning the execution plan and the optimization of the spatial index. The performance of the spatial index can be enhanced by including the LAYER_GTYPE parameter when the index is created. The LAYER_GTYPE parameter specifies the type of SDO_GEOMETRY that is being used. Setting LAYER_GTYPE to POINT tells the index that all geometries are points. This results in the index mechanism knowing it only has to operate on points instead of multiple different geometry types so spatial query optimizations will be invoked. Thereby the index can be sped up because it can exclude several steps needed to process different SDO_GEOMETRY types.

Tuning the SQL-code is done by altering the query execution plan as to in which order the tables should be joined and which indexes should be used. Creating an optimal query execution plan is done by adding hints to the execution plan, which should result in processing the query with minimal cost. First, the original execution plan has to be analysed to find any cost intensive operations in the query execution. Second, the cost of the found intensive operations need to be minimized. Figure 5.1 shows the original explain plan. The explain plan is based on the dataset of 300.000 geometries, the query used is from the AREA-event Service and displayed in figure 4.4 (Lines 7-10).

SABRE Original Explain Plan					
Operation	**Object**	**Order**	**Cost**	**CPU Cost**	**I/O Cost**
SELECT STATEMENT		14	365	187856560	333
HASH JOIN		13	365	187856560	333
NESTED LOOPS		11	3	222414	3
MERGE JOIN CARTESIAN		9	3	49514	3
NESTED LOOPS		5	1	9421	1
NESTED LOOPS		3	1	8371	1
INDEX UNIQUE SCAN	SERVICE_PK	1	0	1050	0
INDEX RANGE SCAN	R_SERVICE_EVENT_UK1	2	1	7321	1
INDEX UNIQUE SCAN	EVENT_PK	4	0	1050	0
BUFFER SORT		8	3	48464	3
TABLE ACCESS BY INDEX ROWID	GEOMETRY	7	3	49514	3
DOMAIN INDEX	GEOMETRY_IDX	6			
INDEX UNIQUE SCAN	GEOMETRY_PK	10	0	1900	0
TABLE ACCESS FULL	R_EVENT_GEOMETRY	12	351	124710648	330

Table 5.1: SABRE original explain plan

Figure 5.2 shows the total cost of the query execution plan. The total cost is the cost of all operations.

Total Cost of the Original Execution Plan		
Cost	**CPU Cost**	**I/O Cost**
365	187856560	333

Table 5.2: Total cost of the original execution plan

The cost of a execution plan represents units of work or resource used. The query optimizer determines the most efficient way to execute the SQL-statement. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work. So, the cost used by the query optimizer represents an estimate of the number of disk I/Os and the amount of CPU and memory used in performing an operation. The operation can be scanning a table, accessing rows from a table by using an index, joining two tables together, or sorting a row set. The cost of a query plan is the number of work units that are expected when the query is executed and its result produced.

There are two operations in the query execution plan which are cost intensive, in order of processing: the full table access and the hash join. The select statement has also a high cost, but it is not an actual operation and can therefore not be altered. It contains the final result of all other operations, it is dependant on the cost of its preceding operation. The hash join operation is heavily influenced by the preceding full table access operation. As the full table access operation passes on an entire table to its following operations. To minimize the cost of the query execution plan the operation of full table access has to be eliminated and replaced by a less costly operation. Because of the existence of a unique index on the R_EVENT_GEOMETRY table named R_EVENT_GEOMETRY_UK1, an index unique scan operation can be performed instead of a full table access operation. A full table access operation should always be avoided as much as possible, because it is a very costly last resort of finding results when no indexes are available. An SQL-profile has to be made to instruct the Oracle database to use the index instead of a full table access. Before executing a query the database checks if any profiles corresponding to the query are present. If an SQL-profile is present the query is executed by the plan supplied by the SQL-profile. If there is no SQL-profile present the database tries to execute the query as efficient as possible guided by the query optimizer-mode. Most likely the full table access and the hash join are a result of the query optimizer using the 'optimizer_mode = all_rows' setting, the optimizer chooses the best plan for fast delivery of all of the rows that queries return so the optimizer may decide to choose a full table scan over index access and hash joins instead of nested loop.

Hash join operation are, in a Oracle Database, only applied when there is no index present in the to be joined tables and when the tables contain many records. Nested loop operations are generally more efficient than hash join operations when a small number of results have to be joined. Because of usage of the R_EVENT_GEOMETRY_UK1 index, the hash join operation can be replaced by the more efficient nested loop operation. Table 5.3 shows the new query execution plan when using the index unique scan operation instead of a full table access operation, and replacement of the hash join operation by a nested loop operation.

SABRE Tuned Explain Plan					
Operation	**Object**	**Order**	**Cost**	**CPU Cost**	**I/O Cost**
SELECT STATEMENT		14	94	1614066	94
NESTED LOOPS		13	94	1614066	94
NESTED LOOPS		11	3	222414	3
MERGE JOIN CARTESIAN		9	3	49514	3
NESTED LOOPS		5	1	9421	1
NESTED LOOPS		3	1	8371	1
INDEX UNIQUE SCAN	SERVICE_PK	1	0	1050	0
INDEX RANGE SCAN	R_SERVICE_EVENT_UK1	2	1	7321	1
INDEX UNIQUE SCAN	EVENT_PK	4	0	1050	0
BUFFER SORT		8	3	48464	3
TABLE ACCESS BY INDEX ROWID	GEOMETRY	7	3	49514	3
DOMAIN INDEX	GEOMETRY_IDX	6			
INDEX UNIQUE SCAN	GEOMETRY_PK	10	0	1900	0
INDEX UNIQUE SCAN	R_EVENT_GEOMETRY_UK1	12	1	15293	1

Table 5.3: SABRE Tuned Explain Plan

Figure 5.4 shows the total cost of the query execution plan. The total cost is the cost of all operations.

Total Cost of the Tuned Execution Plan		
Cost	**CPU Cost**	**I/O Cost**
94	1614066	94

Table 5.4: Total Cost of the Tuned Execution Plan

The tuned explain plan shows no more relatively costly operations, the old costly operations have been replaced by more efficient ones, which results is a more efficient explain plan for the SABRE AREA-event Query. Comparison of the original and tuned execution plan shows a massive decrease in total cost. Table 5.5 show a comparison of the execution plans.

Comparison of Execution Plans			
	Cost	**CPU Cost**	**I/O Cost**
Original	365	187856560	333
Tuned	94	1614066	94
Difference	-271	-186242494	-239
Percentage	-74,247%	-99,140%	-71,772%
Factor	3,882979	116,3871614	3,542553

Table 5.5: Comparison of Execution Plans

The table shows a dramatic decrease of query execution plan cost, the effect of tuning the execution plan is that the total cost of the tuned execution plan is about four times lower than the total cost of the original execution plan.

5.1.2 Expectations

The total cost of the original query execution plan has been decreased, therefore the database should be able to process queries faster. Faster processing of queries means more queries per second the database can handle and a decrease in response times. However, the optimization is based on a dataset of 300.000 records, in which case the queries per second the database can handle should be about four times larger. When using other datasets, different results should be expected, although in each case the queries per second should increase, the factor of increase can be different. The cost of the execution plan is heavily dependant on the amount of records involved, larger datasets have higher costs than smaller datasets. The reduction of cost caused by tuning the execution plan will be relatively greater for the larger datasets, there is an exponential relation between amount of reduction and dataset size. Therefore when small datasets are used the influence of using a tuned execution plan is relatively small. Only a small increase of queries per second may be noticed when using small datasets. Therefore it is expected that the larger the dataset, the more significant the increase in queries per second will be when using a tuned execution plan.

5.2 Materialized Views

A materialized view is a stored summary of a precomputed result of a query. The idea behind materialized views is that because of the data is precomputed, it allows for shorter transaction times. A materialized view can be used to precompute a frequently used part of a query, or even an entire query. For example, a materialized view can contain the result of a complex query over several very large tables. The complex query can have a very small result set while the query itself may take a very long time to process. If the result of the complex query is stored within a materialized view, it allows for considerable improvement in response times, because of shorter transaction times. A materialized view can be accessed in the same manner as a normal table is accessed. The downside of using materialized views is the data becoming outdated, in which case the materialized view needs to be updated, the extra storage overhead for storing the materialized view and the updates requiring more time. This leads to a trade off between, on one hand, decrease of transaction times and, on the other hand, extra storage overhead and slower updates. When working with complex queries combined with large data sets a materialized view should decrease the transaction time considerably.

5.2.1 Technical

A query for the SABRE application can be seen consisting of two different parts, the first part the mapping of the service_id onto the corresponding geometries, second, performing spatial processing on the mapped geometries. The optimization of using materialized views focusses on the first part, the mapping of service_id's onto geometries. The general idea is to create several materialized views, for each service a materialized view so each service has its own materialized view containing its corresponding locations. The materialized view is a precomputed result of a query for mapping service_id's onto geometries, it contains records. If a materialized view is present for each service, there is no longer need for mapping of service_id's onto geometries. The spatial processing can directly take place based on the records contained inside the materialized view. By directly starting with spatial processing, a large part of the entire query-processing is omitted. This results in faster transaction processing. Figure 5.1 shows the usage of a materialized view within function WD inside the SABRE PL/SQL package. The used materialized view is SERVICE_ID2_MV, representing a materialized view for a service with service_id 2.

```
select g.name from service_id2_mv g
where
SDO_WITHIN_DISTANCE(g.geometry,sdo_geometry(2001,null,SDO_point_type(X,Y,null),
null, null),'distance=500')='TRUE';
```

Figure 5.1: Usage of a materialized view within function WD

5.2.2 Expectations

When using materialized views a part of the total query processing is omitted, therefore reducing the total time required for processing the query. If the time required is reduced the transaction time is shorter, resulting in more transactions per second the database can handle and faster response times. The improvement in performance realised by materialized views is exponentially related to the size of the used datasets, in the same manner as with the SQL-tuning optimization. Basically the SQL-tuning optimization and materialized views are not that much different, both optimizations strive to minimize the cost of performing the non-spatial processing of the query. While SQL-tuning minimizes the cost, materialized views does an even better job by totally eliminating the cost. Therefore the materialized views optimization is expected to enhance the performance of the database even better compared to SQL-tuning. The usage of materialized views is expected to enhance the performance of the database considerably, with respect to the size of the dataset used, larger datasets tend to see a relatively larger performance enhancement.

Since the SABRE prototype is 'query-only', updates do not occur, the drawback of slower updates is not an issue. However, the extra storage overhead may become an issue when there are relatively high amounts of materialized views present in the database. The database could run out of storage for the materialized views, but nowadays adding extra storage is a cheap and simple process so the extra storage overhead should not pose a threat.

5.3 Range Partitioning

Partitioning is a common feature, in which a single table and its indexes are broken up into several different tables each with its own index. Oracle supports many different ways to partition data, however this project will focus on range partition. Range partitioning is currently the only form of partitioning for use with spatial indexes. Partitioning is done by using a partitioning key. A partitioning key is the criteria used by the database to determine which row goes in what partition. The partition key is based on one or more columns in a table that are associated with a value or a range of values. A row is compared to the partitioning key deciding in what partition the row has to be stored. Partitioning can enhance the performance and scalability by several means:

- ✔ Search tables or index partitions in parallel, on multiple processors
- ✔ Spread the I/O load across multiple controllers
- ✔ Store data that is likely to be processed together, closely together on the physical disk

- ✔ Eliminate partitions from consideration based on partition key

Partition elimination is the most important way of partitioning to enhance the performance and scalability of a database. Oracle automatically excludes partitions that do not take part in a query, thereby performing early elimination of data that will not be in the result set. Performance and scalability is enhanced by partition elimination by two reasons:

- ✔ By significantly reducing the amount of table data searched to return results
- ✔ By significantly reducing the amount of index information needed

The first reason enhances the performance and scalability because less table data is searched resulting is shorter transaction times. The second reason enhances the performance and scalability because of less index information is required compared to using a normal spatial index without range partitioning. When using large datasets, partitioning is supposed to enhance the performance and scalability of a database considerably.

5.3.1 Technical

Range Partitioning is implemented with a arbitrarily chosen partition key from the x or y value of the coordinate, in this case the x-value is chosen. The GEOMETRY table is decomposed into separate partitions. However, compared to the GEOMETRY table, the partitions contain one extra column, named x_value. This column contains for each SDO_GEOMETRY the value of its x-coordinate, which is necessary for filling the partitions. Without the x-value the database is unable to determine which row goes into what partition. When the GEOMETRY table has been decomposed, the partitions have been filled, the x-value is no longer necessary. For normal partition elimination the partition key is required to be in the WHERE clause, as a predicate, of the SQL-statement. Based on the value of the partition key, the database is able to quickly eliminate partitions who are not part of the result set. The SABRE query however, does not contain a partition key in the where clause. To address this issue Oracle Spatial provides a feature called Spatial Partition Pruning. Spatial partition pruning is similar to the partition elimination the optimizer does, but is based on location and therefore does not require the partition key to be present in the WHERE clause. Spatial partition pruning works only with spatial operators which specify an area of interest, such as SDO_WITHIN_DISTANCE. Normally, the minimum bounding rectangle (MBR) around the area-of-interest is compared with the MBR's of the other geometries using fast searches into the spatial index. When a partitioned spatial index is used, each partition has its own spatial index. Each partition's index meta-data includes a column called SDO_ROOT_MBR, which contains the MBR around all of the geometries in that index partition. Spatial indexing examines the SDO_ROOT_MBR column of each partition, and if the MBR of the area-of-interest does not overlap the SDO_ROOT_MBR, spatial partition pruning will occur and the index associated with that partition will never be searched. So the GEOMETRY table only needs to be decomposed based on a partition key, whereas the Spatial partition pruning will take care of eliminating partitions. There is however one drawback of using Spatial partition pruning, it is more costly than normal partition elimination. According to Oracle:

"*Currently, for every partition that Oracle Spatial needs to eliminate, the cost is just over 1 millisecond per partition. This testing was done on a Hewlett Packard Integrity RX4640 server running Red Hat Linux Advanced Server 3.0. This hardware configuration included 4 1.5 Ghz Itanium CPUs, 16 Gigabytes of memory, and an HP Storageworks Enterprise Virtual Array 5000 Running VCS 3.010. This overhead is likely to be addressed in a future release.*"[OSP]

Because of this, the amount of partitions used has to be chosen wisely, too few partitions and there is almost no elimination, too many partitions and the cost of pruning is too high. There is no guideline as to how many partitions have to be used. Examples of range partitioning have at least four partitions and the amount goes up to over one hundred partitions. Therefore it is impossible to determine the right amount of partitions which leads to a trial-and-error process of finding the right amount. For this project the amount of partitions is set to fifteen, which is assumed to give a performance improvement while being on the safe side of using too many partitions. The assumption is made that even a small amount of partitions should improve the performance, if there is no noticeable performance improvement when using fifteen partitions there is no use in increasing the amount of partitions as it will most likely only degrade the performance. If there is a performance improvement when using fifteen partitions, the trial-and-error process of finding the optimum amount of partitions is required.

Figure 5.2 shows the creation of the partition based on the GEOMETRY table.

```
CREATE TABLE geometry_partn (geometry_id NUMBER, geometry SDO_GEOMETRY,
name VARCHAR2(4000), x_value NUMBER) PARTITION BY RANGE (x_value)(
PARTITION P0_20k VALUES LESS THAN (20000),
PARTITION P20k_40k VALUES LESS THAN (40000),
PARTITION P40k_60k VALUES LESS THAN (60000),
PARTITION P60k_80k VALUES LESS THAN (80000),
PARTITION P80k_100k VALUES LESS THAN (100000),
PARTITION P100k_120k VALUES LESS THAN (120000),
PARTITION P120k_140k VALUES LESS THAN(140000),
PARTITION P140k_160k VALUES LESS THAN (160000),
PARTITION P160k_180k VALUES LESS THAN (180000),
PARTITION P180k_200k VALUES LESS THAN (200000),
PARTITION P200k_220k VALUES LESS THAN (220000),
PARTITION P220k_240k VALUES LESS THAN (240000),
PARTITION P240k_260k VALUES LESS THAN (260000),
PARTITION P260k_280k VALUES LESS THAN (280000),
PARTITION P280k_300k VALUES LESS THAN (300000));
```

Figure 5.2: Creation of the GEOMETRY table partition

Figure 5.3 shows the SQL-statement responsible for filling the partitions with the geometries and the creation of the spatial index for each partition.

```
INSERT INTO geometry_partn NOLOGGING SELECT x.geometry_id, x.geometry, x.name, t.x
FROM (select geometry_id,geometry,name from geometry) x,
table(sdo_util.getvertices(x.geometry)) t;

create index geometry_partn_idx on geometry_partn(geometry)
indextype is mdsys.spatial_index parameters ('layer_gtype=point') LOCAL(
PARTITION IP1,PARTITION IP2,PARTITION IP3,PARTITION IP4,PARTITION IP5,
PARTITION IP6,PARTITION IP7,PARTITION IP8,PARTITION IP9,PARTITION IP10,
PARTITION IP11,PARTITION IP12,PARTITION IP13,PARTITION IP14,PARTITION IP15);
```

Figure 5.3: SQL-statement for filling partitions and spatial partitioned index creation

The SQL-statement extracts the value of the x-coordinate from the geometries and uses it to put each row into its corresponding partition. If no partitioned spatial indexes are used, Spatial partition pruning cannot occur, therefore the indexes have to be explicitly created. The partitioned indexes are created by specifying the keyword LOCAL at the create index statement and by naming the individual partitions. To complete range partitioning the function WD inside the SABRE PL/SQL package needs some minor adjustments, instead of querying table GEOMETRY, the new partitioned table, GEOMETRY_PART, has to be queried. Figure 5.4 shows the adjustment to function WD to accommodate range partitioning.

```
select g.name from geometry_partn g,service_2_geometry s2g
where g.geometry_id=s2g.geometry_id
and s2g.service_id=SID
and
SDO_WITHIN_DISTANCE(g.geometry,sdo_geometry(2001,null,SDO_point_type(X,Y,null),
null,null),'distance=500')='TRUE';
```

Figure 5.4: Usage of a Range Partitioning within function WD

5.3.2 Expectations

The optimization of range partitioning is used in conjunction with the SQL-tuning optimization because both optimizations have no negative influence on each other, they are separate optimizations applied to different aspects, and therefore can supply more performance increase. The expected increase of using range partitioning is on top of the expected increase of the SQL-tuning optimization. Therefore the results of applying range partitioning need to be compared to the results of the SQL-tuning.

Oracle claims that range partitioning is useful for every database small or large, there should always be an increase of performance. However range partitioning seems only useful when using large datasets, by creating partitions not all the data needs to be searched, which implies a performance gain. However, there is a performance penalty for using Spatial partition pruning. The issue is whether the performance gain outweighs the performance penalty. When small datasets are used it is likely to perform worse, but when large datasets are used a increase in performance is expected. And with that, there is another issue, the definition of a large dataset. There is no definition of what size of dataset is considered to be large, therefore the assumption is made that from 300.000 records and more the dataset is considered to be large. Thus, the optimization is expected to bring an performance increase only for two biggest datasets, the 300K and 3M datasets. The smaller datasets are likely to suffer a performance decrease from the range partitioning optimization.

5.4 Chapter Summary

This chapter presented three optimizations for improving the performance and scalability of the SABRE application. Each optimization and its expectations have been discussed. The research question *“Are there possibilities to significantly improve the performance and scalability by means of database optimizations?”* has been answered, the three optimizations SQL Tuning, Materialized Views and Range Partitioning are all supposed to enhance the performance and scalability significantly. All optimizations have a thing in common, the increase in performance should be relatively larger when using larger datasets.

6 Test Case

This chapter contains the test case regarding the performance and scalability of the SABRE application. The following research question is addressed: *"How can the performance and scalability of the database be measured?"* This chapter contains a description of the approach, the testing environment, the metrics, the scope and the benchmark methodology.

6.1 Approach

Testing the SABRE application on performance and scalability is a difficult task, there are many factors to deal with which influence the outcome. Before the actual testing the definitions have to be described, containing definitions, metrics and the scope of the tests. After describing the definitions, metrics and scope the actual testing starts, the testing consists of an iterative process, which consists of theorizing about expectations followed by the actual test, followed by an comparison of expectations and actual result, the possible difference will be evaluated as the final step of the iterative process. With each iteration there is a different optimization applied, trying to achieve the best possible performance.

6.2 Testing environment

The spatial database is realized by using Oracle Spatial 10g. Oracle Spatial is currently the dominant spatial database available. For the benchmarking of the database, the application 'Benchmark Factory' from Quest Software is used. The tests are performed on a single processor Windows 2003 Server system with 1GB of memory. The testing environment is similar to the environment that is most likely used if SABRE is to be commercially deployed, depending on the result of the performance testing. Refer to appendix J for specific details about the testing environment.

6.3 Metrics

The process of testing the performance and scalability knows several metrics. These metrics are essential to the testing process and therefore need to be defined.

- ✔ Query-load, the amount of queries that need to be processed by a database. The query-load is defined as the amount of requests per period of time, usually one second.
- ✔ TPS, transactions per second, the amount of transactions the database can process within one second of time. The maximum amount of TPS is an indication of the performance of the database. The maximum TPS can only be determined if there is no query-load specified, the requests come in as fast as the database can handle. If a query-load is specified the TPS is always equal or less than the query-load. The higher the TPS the better the performance of the database. Since the SABRE prototype handles only queries, the TPS should be seen a queries per second.
- ✔ Response time, the amount of time elapsed between the start of a request and the arrival of the first result tuple, measured in milliseconds.
- ✔ Transaction time, the amount of time elapsed for database processing. The transaction time is the amount of time needed for the database to process a request, thus the difference between the arrival-time of a request at the database and the time the database is done processing the request, measured in milliseconds. It denotes the time period in which a database fact was stored in the database.
- ✔ Data set, a data set is a collection of data containing geographical information. The data set contains various records, each record representing a geometry, representing a coordinate.
- ✔ Threads, a means for a program to divide itself into multiple simultaneously running tasks. Threads can be executed in parallel, thereby executing multiple tasks at the same time. A database can use threads to handle multiple users simultaneously.

6.4 Scope

Because the performance and scalability testing can become very extensive, the scope needs to be defined to narrow the testing down to a manageable chunk of tests. There are several important aspects that need to be restricted, the restrictions are denoted below.

- ✔ Amount of Users
  - ○ The amount of users simultaneously doing requests is set at fifteen. The amount is deducted from a real-world scenario. The idea behind the amount of users is to have a better simulation of the usage of SABRE, it is more likely that fifteen users will use SABRE simultaneously than there is only one user. The result is that for each user a separate thread is created, which leads to fifteen threads in total. Because of thread-usage the database can handle the requests simultaneously, which results in a slight performance improvement, a performed test has pointed out that with fifteen simultaneous users the database achieves its maximum TPS while maintaining decent response and transaction times.
  - ○ As a result of having fifteen simultaneous users, the time between consecutive requests corresponding to a certain query-load has to be multiplied by fifteen. For example, a query-load of 60 requests per second results in fifteen simultaneous requests each 250 milliseconds.
- ✔ Only one service
  - ○ The prototype contains only one implemented service, the Area-event service. The testing is only focused on the Area-event service, other services can be implemented (and tested) but are

beyond the scope of the project.

- ✔ Only one spatial function
  - ○ Function SDO_WITHIN_DISTANCE (SDO_WD), used to determine which objects are within a specified distance of each other, is the only function that will be tested. Oracle spatial supports more spatial functions than SDO_WD. Nonetheless SDO_WD is more than capable to perform the tasks required, and therefore will be the only function that will actually be used. There is no need for other functions, thus no need for testing them either.
- ✔ Query-load
  - ○ The different query-loads used for testing the performance and scalability are based on a real-world scenario. They are extracted from comparable applications and should give a decent indication of query-load sizes to be expected. The constant sizes are based on the average to be expected query-load, whereas the sizes are distributed around the average value. The results of using different sizes will provide valuable information about the performance and scalability.
- ✔ Data set-size
  - ○ Analogous to the query load, the different data set-sizes are also based on a real-world scenario. The amount depicts the amount of records, the number of geometries in a set. The data sets are real data sets containing coordinates from the Netherlands. Each data set contains a number of locations (addresses) in the Netherlands, the larger the data set the more locations it contains. The locations are evenly distributed in the data sets. It is essential that the locations in the different data sets are evenly distributed to get correct results from the tests. To verify the even distribution, table 6.1 shows the different data set sizes and the corresponding number of returned results from a query with a static location. The query to check the distribution of locations has been performed two times, with each time a different radius, returning all the locations within the radius of a given location. The table shows that an increased dataset size by a factor of ten, also results in an increase by a about a factor of ten in the number of returned results. From the table it can be concluded that the locations in the different data sets are evenly distributed.

Data set Distribution		
Data set size	**Number of returned results**	
	Radius: 500m	*Radius: 10km*
300	1	12
3K	1	122
30K	10	1064
300K	112	11226
3M	1297	111916

Table 6.1: Dataset Distribution

6.5 Benchmark methodology

The benchmarking contains two different measure points. The first is at the database itself, thus the performance and the scalability of the database is tested, the second is a the Web Service whereby the performance and scalability of the database in conjunction with the Web Service is tested. The performance and scalability are measured as query load against response time and data set size against response time. Next to response times the transactions per second the database can handle and the database transaction times are measured. The query load and the data set size are the only dynamic variables, they will represent different values for each different test run. The response and transaction times and transactions per second are the results of the tests, ideally the response and transaction times should be as low as possible while the transactions per second should be as high as possible.

The problem description contains also the following two research statements:

- ✔ "How does a Spatial Database perform in a GIS-Web Service environment in terms of response times?"
- ✔ "How does a Spatial Database perform and scale up against increasing user loads and increasing datasets?"

These research statements can be translated into terms of the test procedure as follows:

- ✔ Measure the response time of the database in conjunction with the Web Service in terms of:
  1. Query load versus response time.
  2. Data set size versus response time.
- ✔ Measure the transactions per second and response and transaction times of the database in terms of:
  1. Query load versus response and transaction time.
  2. Data set size versus response and transaction time.
  3. Data set size versus transactions per second.

The third test is merely a test solely used as an indication of how many transactions per second the database can handle for each dataset in an ideal situation, it is the maximum number of transactions per second the database can handle.

Table 6.2 show a schematic view of to tests to be done.

Tests to be performed		
Versus	**Query Load**	**Data set Size**
Response Time (DB)	▨	▨
Transaction Time (DB)	▨	▨
Transactions per second (DB)		▨
Response Time (DB+WS)	▨	▨

Table 6.2: Different tests to be performed

Each cell denoted a specific test. Note that there is no TPS-test for the query-loads.

6.5.1 Test Parameters

The test contains two different parameters, the query load and the data set size. These parameters are the only two that will be changed during the testing process. For successfully testing the performance and scalability the values of the parameters have to be chosen meaningfully. As stated before, the values are derived from a real-world scenario. The lowest and highest values are values that are not likely to be expected, but in conjunction with the other values they do give a good indication of the performance and scalability. Focus is on the mid value, 60 transactions per second, because these values represent the requirements for the SABRE application for commercial exploitation.

Table 6.3 shows a schematic view of the values of the test parameters.

Test Parameter Values					
	Low	**Low/Mid**	**Mid**	**Mid/High**	**High**
Query Load (requests per second)	15	30	60	90	120
Data Set Size (amount of geometries)	300	3.000	30.000	300.000	3.000.000

Table 6.3: Test Parameter Values

6.5.2 Number of Test Results

For a complete test there are in total 55 test results. Any combination of parameter values of table 6.3 represents a run, which makes 5x5=25 results. Taken into account that these runs have to be done two times, one for the database another for the database in conjunction with the Web service, it comes to a total of 50 test results. Testing for the maximum amount of transaction per second the database can handle takes one test per data set, making it another 5 results. This makes a grand total of a minimum of 55 runs to be done for one complete test, reruns not taken into account.

6.5.3 Number of transactions and query-load

Each response and transaction time-test run needs to be running long enough to get decent results from the test. If the duration of the test is too short, incorrect result may occur. If the duration of a single run takes to long the entire testing procedure may take forever. Therefore an optimal duration for the test runs needs to be found. The optimal amount is a trade-off between running long enough for correct results and running not too long as it may take far to much time. The duration is determined by looking at the test results stabilize. For each test run the result is stabilized after a different amount of transactions. The largest amount of transactions needed for a run to stabilize is about 1800. Therefore for each test the amount of transactions is set a 1800. Considering the thread usage of 15 threads there are for each run 1800/15 = 120 iterations. The query-load is set by setting an interval for the iterations. For example, an interval of 500 milliseconds means an effective 15x2 = 30 requests per second. The largest query-load, 120 requests per second, has an interval of 1000 / (120/15) = 125 milliseconds. Theoretically, testing with 120 requests per seconds, the test should be completed in 1800/120 = 15 seconds. Although this may seem rather short, it is sufficient for the results to stabilize, because of the heavy query-load. So, each run takes 1800 transactions but the duration in seconds for a run to complete is different and dependent on the query-load. When the test has finished the average response and transaction time over all 1800 transactions is taken as the test result.

6.5.4 Influences

The outcome of a test can be influenced in many ways, therefore is essential to minimize all the possible influences as much as possible. Some influences can be ruled out, while others cannot and need to be taken into account. Influences for this project are:

- ✔ Communication overhead because of network usage. The round trip time is 10 milliseconds. This is taken into account when testing response times.
- ✔ Caching can have serious influence on the performance of a database. There is a ongoing debate on whether caching is to be turned off to gain produce correct results. For this project caching is required as it is an important performance aspect for the database when maximum performance is required for commercial exploitation. If caching is to be turned off, a final conclusion for SABRE could be that the performance requirement is not met. Therefore the application will not be used in a commercial environment while it might actually meet the performance requirement if caching was used.

6.6 Chapter Summary

This chapter described the test case for the performance and scalability testing of the SABRE application. The research question *"How can the performance and scalability of the database be measured?"* has been answered. The test case described provides a profound basis for the tests to be performed.

7 Results & Outcome

This chapter contains the results of the actual testing regarding the performance and scalability of the SABRE application. The chapter starts with the initial test run, followed by test runs where different optimizations are applied. This chapter addresses the following research questions:

- ✔ *"How does a Spatial Database perform in a GIS-Web Service environment in terms of response and transaction times?"*
- ✔ *"How does a Spatial Database perform and scale up against increasing user loads and increasing datasets?"*
- ✔ *"If there are significant database optimizations, how well do they influence the performance and scalability of the database?"*
- ✔ *"If there are significant database optimizations, can they be used in conjunction or separately, and which is the best (combined) optimization with respect to this project?"*

For each test its results, outcome and conclusion is presented. The optimization tests also include a comparison to determine the effect of applying the optimization. The last part of this chapter contains the overall conclusion for the tests.

For the tests in this chapter it is important to understand how the results from the test are measured and what the specific results represent. Figure 7.1 shows a schematic view of how the response and transaction times are measured.

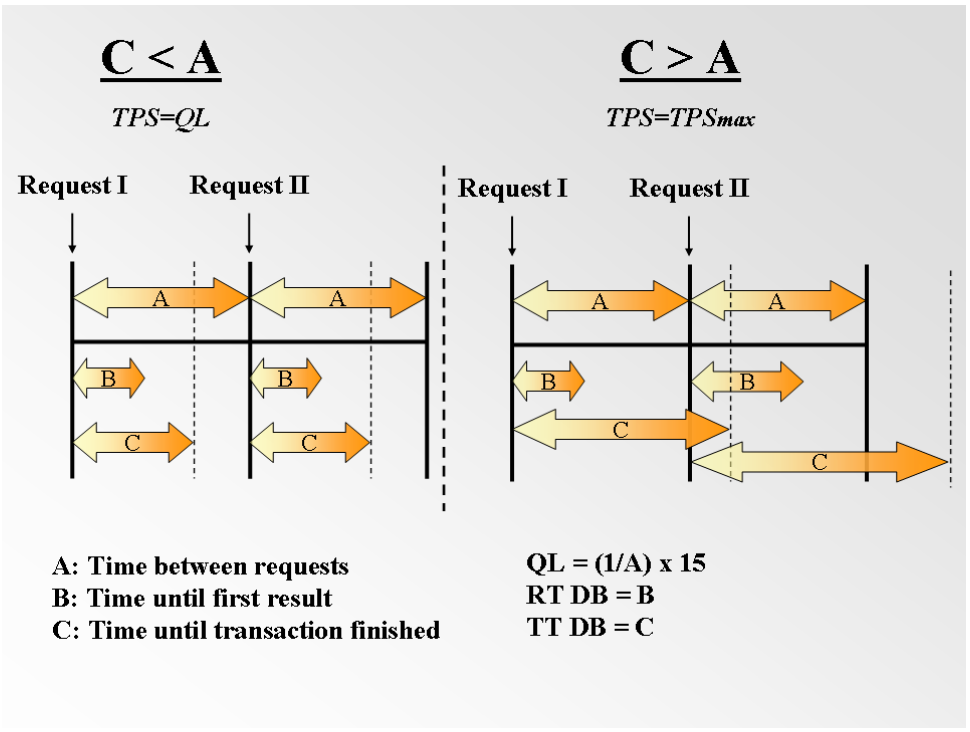


Figure 7.1: Schematic view of different timings

The figure contains several abbreviations, the abbreviations used throughout this chapter are:

- ✔ **TPS**: Transactions per second of the database
- ✔ **QL**: Query Load, defined as requests per second, sizes: 15,30,60,90 and 120
- ✔ **DSS**: Data set size, sizes: 300, 3K, 30K, 300K and 3M
- ✔ **RT DB**: Response time of the database
- ✔ **TT DB**: Transaction time of the database
- ✔ **RT DB + WS**: Response time of the database plus the web service

The figure shows two possible situations. The first situation shows that the time between consecutive requests, known as inter-arrival time, is longer than the database requires for the processing of the request. Therefore the TPS should be equal to the QL, the TPS is said to be capped by the QL. The second situation shows the time between consecutive requests to be too short for the database to complete the processing of the request. When this happens the response and transaction times are expected to increase, because the database receives more requests than it can handle. The figure also shows the QL being equal to (1/A) x 15, as stated in chapter 6.4 there are fifteen simultaneous requests. The requests are handled by the database by means of concurrency. Therefore the inter-arrival time of the QL is equal to 15/QL. For example, if the QL is 60 requests per second, the corresponding inter-arrival time is 250 milliseconds.

Unless otherwise specified all timings are displayed in milliseconds. The results of the test are marked in the tables by a grey background.

7.1 Initial Test Run

The initial test run tests the SABRE prototype for performance and scalability. There is no optimization applied at all, thus the results of this test are the foundation for the optimization tests. The results of the optimization tests will be compared to the results of this test, thereby showing if the optimizations have the desired effect.

7.1.1 Results

The first test to be performed is the transactions per second test. The test shows the maximum number of TPS the database can handle for each dataset. The purpose of this test is to determine the maximum TPS for each dataset, which is required for explaining the results of the response and transaction times test. When the QL exceeds the maximum TPS, the database is said to be 'broken', more requests are made then the database can handle each second. When this occurs the result should be a dramatic increase in response and transaction times. Table 7.1 shows the results of the transactions per second test.

Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
TPS	109,63	106,62	92,92	6,64	0,65

Table 7.1: Maximum amount of transactions per second (Duration: 60s)

The results of test show the database can maximally handle just over 100 requests per second for the two the smallest datasets. The 30K dataset drops just below 100 TPS. When the datasets increase in size even further, a dramatic decrease in TPS occurs, the 300K dataset is managing to maintain about 6 TPS while the 3M dataset needs almost two seconds to complete one single transaction. Based on these results the response and transactions time test is expected to have dramatic results for the 300K and 3M datasets, the response and transaction times are most likely going to be no where near acceptable. Table 7.2 shows the results of the response and transaction time test.

Response & Transactions times					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
Query Load: 15 requests/second					
RT DB	12	15	15	153	20126
TT DB	15	17	18	2145	22561
RT DB + WS	202	388	368	1294	22316
Query Load: 30 requests/second					
RT DB	14	15	14	153	20607
TT DB	14	17	16	2124	22137
RT DB + WS	210	469	370	1435	21873
Query Load: 60 requests/second					
RT DB	12	14	12	150	21753
TT DB	15	15	15	2169	23480
RT DB + WS	315	362	290	1733	22342
Query Load: 90 requests/second					
RT DB	19	18	22	150	20296
TT DB	25	21	27	2130	22933
RT DB + WS	367	215	244	1741	21794
Query Load: 120 requests/second					
RT DB	22	22	23	153	21008
TT DB	24	26	39	2139	23734
RT DB + WS	251	326	290	1792	22001

Table 7.2: Response & Transaction times (#Transactions: 120x15)

As expected the response and transaction times increase dramatically when the two largest datasets are used. The 300K dataset has a transaction time of over 2 seconds an a response time of about 150msecs, while the 3M dataset has a transaction time which is almost equal to its response time of over 20 seconds. Whilst the values of the 300, 3K and 30K datasets are acceptable, the database is able to handle the QL in a fair amount of time, this is definitely not the case for the 300K and 3M datasets.

7.1.2 Outcome

Table 7.3 shows the results as an average of the response and transaction times of all query loads, these values are graphically shown in figure 7.2. By using the average over all query loads, the results can be shown as dependant of the dataset sizes, therefore the influence of the different dataset size on the response and transaction times can be shown clearly.

Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
RT DB	16	17	17	152	20758
TT DB	19	19	23	2141	22969
RT DB + WS	269	352	312	1599	22065

Table 7.3: Average response and transaction times of all query loads

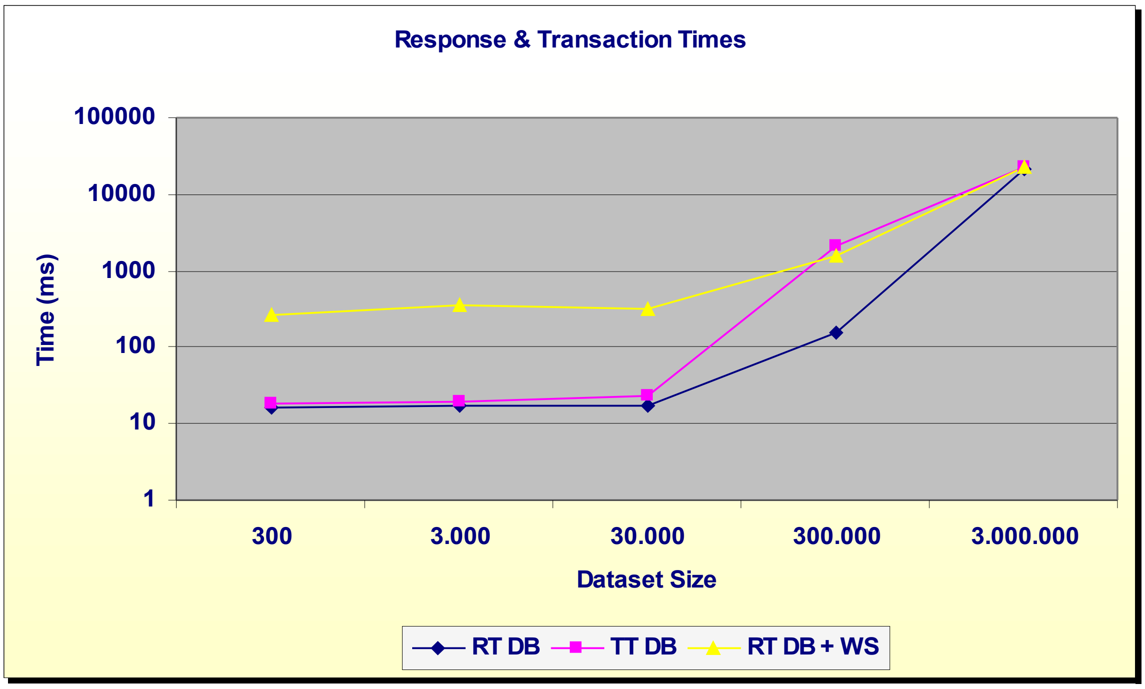


Figure 7.2: Average response and transaction times of all query loads

The lines in the logarithmic graph show that the database is indifferent of the first three dataset sizes, the line is almost straight. When the dataset size exceeds 30.000 records the response and transaction times increase dramatically. Concluding from this graph it can be stated that the database scales well when the dataset increases in size beyond 30.000 records. The database is allowed to have an increase in response and transaction times when larger datasets are used, as long as the increase in time is linear with respect to the dataset size. Another aspect is that the smaller datasets show no increase in response and transaction times while an increase is expected. It seems the database is able to maintain decent response and transaction times as long as the dataset sizes are small. The transactions per second test already showed that the database is able to maintain almost the same TPS for the 300, 3k and 30K datasets, which is an explanation for the response and transaction times being almost the same for the three datasets. With respect to the dataset size it can be concluded that the database is indifferent to different dataset sizes as long as the database is able to handle the dataset sizes. If the size goes beyond manageable the database is said to be broken, its performance drops very rapidly below acceptable.

Table 7.4 shows the results as an average of the response and transaction times of all dataset sizes, these values are graphically shown in figure 7.3.

Response & Transactions times (Average DSS)					
QL	**15**	**30**	**60**	**90**	**120**
RT DB	4064	4161	4388	4101	4246
TT DB	4951	4862	5139	5027	5192
RT DB + WS	4914	4871	5008	4872	4932

Table 7.4: Average response and transaction times of all dataset sizes

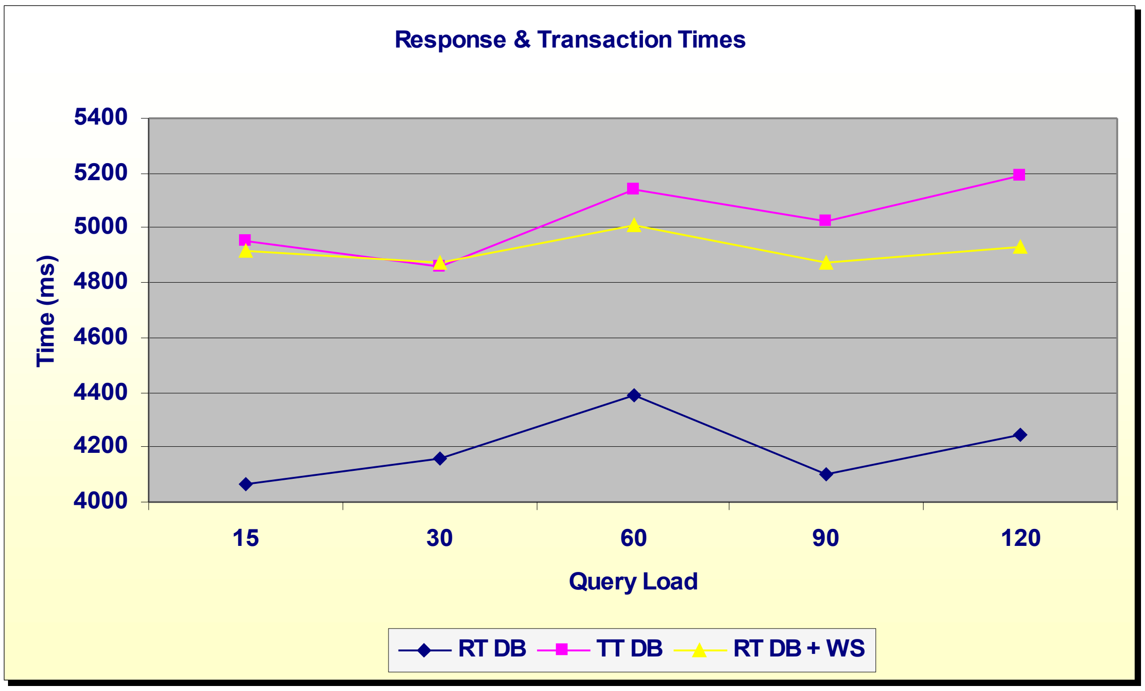


Figure 7.3: Average response and transaction times of all dataset sizes

The graph, with a linear time-axis, shows some rather unusual results. The response and transaction times are expected to increase while the QL is increased when the database is broken. A higher QL should result in a heavier load for the database thus the database needs more time to process the results. The graph however shows something entirely different, the response and transaction times are almost constant, there is only a small insignificant difference for different query loads. At QL90 and greater there is even a small drop in response and transaction times. When the database can handle the QL, it is not broken, it is only logical it is impervious to differences in query loads because as the long as the maximum TPS is higher than the QL, increasing the QL has no effect. However when the database is broken the response and transaction times are expected to increase when the QL increases, but this seems not to be the case. The results, table 7.2, show that when the database is broken, which is the case with the 300K and 3M dataset for all query loads and with the other datasets at QL120, there is an increase of transaction time. Strangely enough there is no more further increase in transaction times when the QL increases. A possible partly explanation could be that the database suffers from an external bottleneck, which could possibly be I/O bound. Concluding, it seems that the database is impervious to differences in query loads when the database is broken. Why the imperviousness occurs is beyond the scope of this project but might prove an interesting area for further research.

7.1.3 Conclusion

From the initial test two conclusions can be drawn. The first, the database seems impervious to differences in query loads after is has been broken. Second, when the database is broken when using larger datasets than the database can handle, it remains scalable with respect to dataset sizes as the increase in dataset size has a linear relation to the increase of response and transaction times.

7.2 SQL Tuning

The focus of the SQL tuning optimization is the significant decrease in response and transaction times for the larger datasets. It is expected that the 300K and 3M datasets benefit the most from this optimization. By applying this optimization the database should be better performing with respect to the dataset sizes.

7.2.1 Results

Table 7.5 shows the result from the transactions per second test with the SQL tuning optimization applied.

Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
TPS	113,73	108,73	94,12	38,68	5,97

Table 7.5: Transactions per second – SQL Tuning – Spatial Database (Duration: 60s)

The results of the transaction per second test show that with the SQL tuning optimization applied, the database is able to maintain about 100 TPS for the 300, 3K and 30K datasets. For the 300K and 3M datasets the TPS is considerably lower, almost 40 TPS and almost 6 TPS. The TPS for each dataset have increased through the applied SQL tuning optimization, therefore a decrease in response and transaction times is expected. Table 7.6 show the result of the response and transaction time test with the SQL tuning optimization applied.

Response & Transactions times					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
			Query Load: 15 requests/second		
RT DB	9	9	10	18	89
TT DB	10	11	13	43	2391
RT DB + WS	381	197	222	271	685
			Query Load: 30 requests/second		
RT DB	13	9	10	19	89
TT DB	15	11	13	38	2379
RT DB + WS	244	177	165	264	837
			Query Load: 60 requests/second		
RT DB	12	9	11	18	89
TT DB	13	10	14	358	2391
RT DB + WS	244	218	234	209	1009
			Query Load: 90 requests/second		
RT DB	9	9	10	20	91
TT DB	10	10	14	336	2377
RT DB + WS	150	232	159	273	1145
			Query Load: 120 requests/second		
RT DB	12	10	11	18	89
TT DB	17	12	23	358	2386
RT DB + WS	204	229	198	204	1119

Table 7.6: Response & Transaction times - SQL Tuning (#Transactions: 120x15)

The table shows acceptable response and transaction time for all the datasets except the largest one, the 3M dataset. The 3M dataset has a response time of about 90 milliseconds while its transaction time is almost 2,4 seconds. Although this is a considerable improvement compared to the initial test it is still not acceptable. However the 300K dataset's response and transaction times have become quite acceptable by applying the SQL tuning optimization.

7.2.2 Outcome

Table 7.7 shows the results as an average of the response and transaction times of all query loads, these values are graphically shown in figure 7.4.

Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
RT DB	11	9	10	19	89
TT DB	13	11	15	227	2385
RT DB + WS	245	211	196	244	959

Table 7.7: Average response and transaction times of all query loads – SQL Tuning

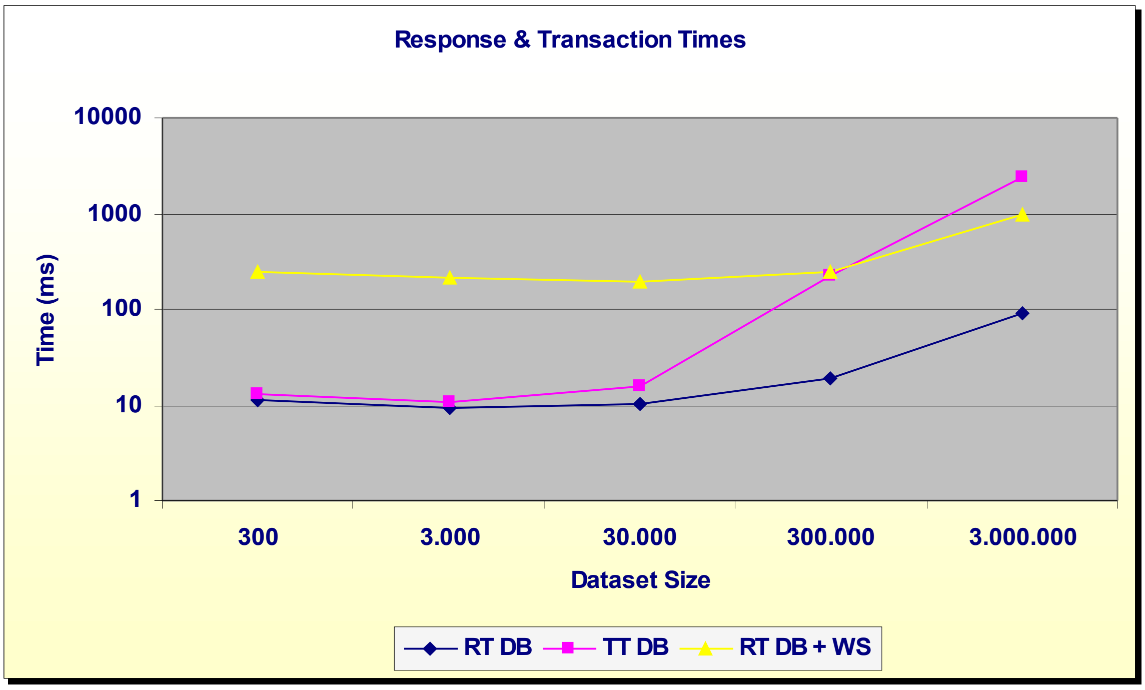


Figure 7.4: Average response and transaction times of all query loads – SQL Tuning

The graph shows an increase in response and transaction times when the dataset size increases beyond 30.000 records. While the graph seems similar to the graph of the initial run, each point in the graph has a lower value, especially for the larger datasets. Therefore the SQL tuning optimization has had the desired effect of lowering response and transaction times for the larger datasets.

Table 7.8 shows the results as an average of the response and transaction times of all dataset sizes, these values are graphically shown in figure 7.5.

Response & Transactions time (Average DSS)					
QL	**15**	**30**	**60**	**90**	**120**
RT DB	27	28	28	28	28
TT DB	494	491	557	549	559
RT DB + WS	351	337	383	392	391

Table 7.8: Average response and transaction times of all dataset sizes – SQL Tuning

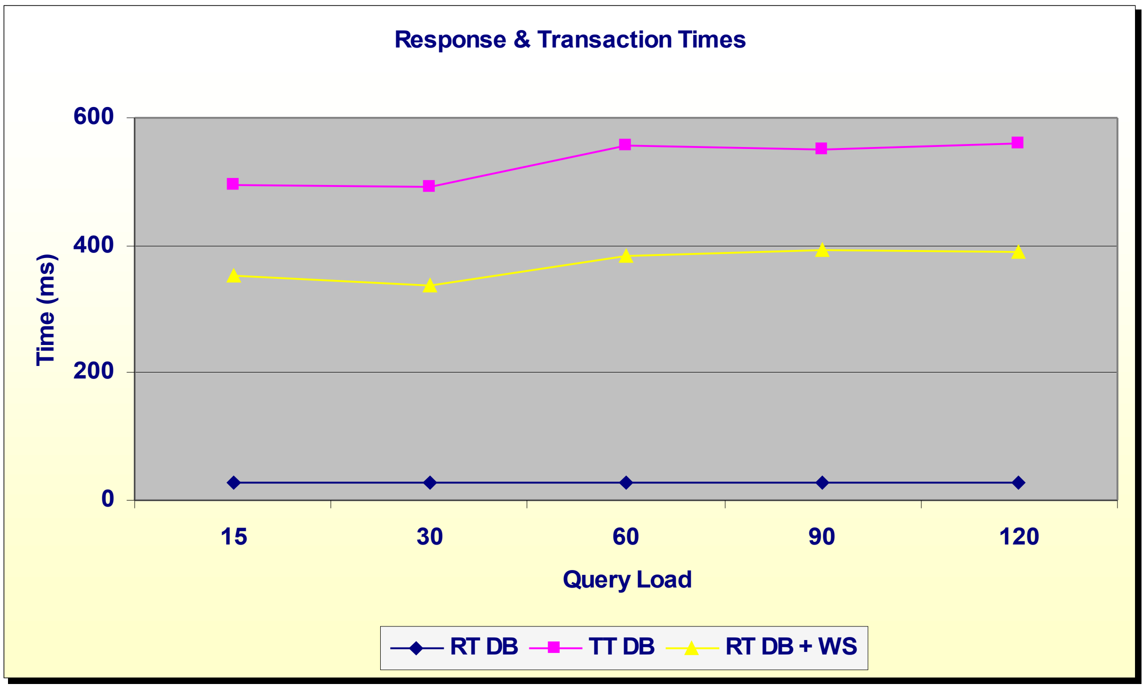


Figure 7.5: Average response and transaction times of all dataset sizes – SQL Tuning

The graph shows once again that the database is impervious to differences in query loads when it is broken. The RT DB is constant over all query loads. There is only a small increase in the TT DB and RT DB + WS when the QL increases from 30 to 60. The increase in TT DB between QL30 and QL60 can be explained by looking at the results of the 300K dataset. For the 300K dataset the database is broken when the QL is higher than the maximum TPS of 38,68 which has occurred when the QL is at 60. When the database is broken the result is an increase in TT DB. After the increase the graph shows again that the database is impervious to differences in query loads when is has been broken.

7.2.3 Comparison

To measure the influence of the SQL tuning optimization the result of the tests have to be compared to the initial tests. Table 7.9 shows a comparison of results. The difference between the results is displayed as absolute difference, absolute relative difference and as a factor value.

Comparison of Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
Initial TPS	109,63	106,62	92,92	6,64	0,65
Optimized TPS	113,73	108,73	94,12	38,68	5,97
Absolute Difference	4,1	2,11	1,2	32,04	5,32
Absolute Relative Difference	3,74%	1,98%	1,29%	482,53%	818,46%
Factor Value	1,0374	1,0198	1,0129	5,8253	9,1846

Table 7.9: Comparison of TPS - Initial versus optimized

For all the datasets there is a increase in maximum amount of TPS. The 300, 3K and 30K datasets show only a small percentage of increase, whilst the largest two datasets show a huge percentage of increase in maximum TPS. The factor values of increase show that for the 300K dataset the maximum TPS has increased almost 6 times, while for the 3M dataset the increase is more than 9 times. The increase in maximum TPS in in accordance with the expected results of the SQL tuning optimization, the largest datasets benefit the most. The results of the comparison can be used as an indication for the comparison of response and transaction times tests. The comparison of response and transaction times should also show an increase of performance which is comparable to the increase of the TPS. The results of the initial test showed that the database is impervious to differences in query loads when the database has been broken. Therefore the comparison based on the average DSS is not useful, whereas the comparison based on the average QL is. Table 7.10 shows the comparison of results based on the average QL. It must be noted that the comparison shows a decrease in response and transaction times, therefore the percentage denotes a decrease, of a maximum of 100%, based on the initial value, whereas the factor value denotes the new optimized value divided by the initial value.

Comparison of Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
Initial Run					
RT DB	16	17	17	152	20758
TT DB	19	19	23	2141	22969
RT DB + WS	269	352	312	1599	22065
SQL Tuning Optimization					
RT DB	11	9	10	19	89
TT DB	13	11	15	227	2385
RT DB + WS	245	211	196	244	959
Absolute Difference					
RT DB	5	8	7	133	20669
TT DB	6	8	8	1915	20584
RT DB + WS	24	141	117	1355	21106
Absolute Relative Difference					
RT DB	30,38%	45,24%	39,53%	87,75%	99,57%
TT DB	30,11%	43,75%	33,04%	89,42%	89,62%
RT DB + WS	9,07%	40,17%	37,39%	84,73%	95,65%
Factor Value					
RT DB	0,6962	0,5476	0,6047	0,1225	0,0043
TT DB	0,6989	0,5625	0,6696	0,1058	0,1038
RT DB + WS	0,9093	0,5983	0,6261	0,1527	0,0435

Table 7.10: Comparison of Response & Transactions times of all query loads – Initial versus optimized

The percentages show for each dataset size considerable improvements, as expected the larger the datasets are, the relatively more they are influenced by the SQL tuning optimization. The 3M dataset has an almost 100% improvement in RT DB, which is a staggering result. Table 7.11 shows the average difference for the RT DB, TT DB and RT DB + WS based on all dataset sizes. The last row of the table contains an average over the averages, the values can be used to rate the optimization, the rating is used to compare different optimizations.

Average Differences – SQL Tuning Optimization			
	Average Difference	**Average Relative Difference**	**Factor Value**
RT DB	-4164	-60,49%	0,3951
TT DB	-4504	-57,19%	0,4281
RT DB + WS	-4549	-53,40%	0,4660
Average RT & TT	-4334	-58,84%	0,4116

Table 7.11: Average differences in response and transaction times - SQL Tuning

The table shows a average decrease of 4.3 seconds, next to a 59% decrease and factor value of 0,4116. Therefore the SQL tuning optimization can be rated at an average decrease of 59% of RT DB and TT DB.

7.2.4 Conclusion

The aim of the SQL tuning optimization is to lower the response and transaction times considerably. The effect of applying the optimization was expected to be more significant for larger datasets. The results of the test showed an considerable decrease in database response and transaction times of an average of 59%. Especially the larger datasets were influenced by the optimization. Based on the results the optimization of SQL tuning can be considered as a useful optimization with promising results.

7.3 Materialized Views

The focus of the Materialized views optimization is, similar to the SQL tuning optimization, the significant decrease in response and transaction times for the larger datasets. It is expected that the 300K and 3M datasets benefit the most from this optimization. By applying this optimization the database should be better performing and better scalable with respect to the dataset sizes.

7.3.1 Results

Table 7.12 shows the results of the transaction per second test with the Materialized view optimization applied.

Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
TPS	112,89	112,79	97,57	45,63	7,32

Table 7.12: Transactions per second – Materialized Views - Spatial Database (Duration: 60s)

The results show that the database is able to maintain about 100 TPS for the 300, 3K and 30K datasets. For the 300K dataset this amount is lowered to less then a half of the other datasets, while the 3M dataset has an amount of just over 7 TPS. The results are far better than the results from the initial run, therefore this optimization is expected to decrease the response and transaction times by a large amount. Table 7.13 show the results of the response and transaction time test with the Materialized view optimization applied.

Response & Transactions times					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
			Query Load: 15 requests/second		
RT DB	9	10	10	21	73
TT DB	10	12	14	46	1830
RT DB + WS	274	331	242	334	814
			Query Load: 30 requests/second		
RT DB	9	9	13	17	63
TT DB	10	10	16	42	2078
RT DB + WS	241	211	262	250	1036
			Query Load: 60 requests/second		
RT DB	10	9	10	17	90
TT DB	11	11	13	343	1883
RT DB + WS	181	317	379	247	1146
			Query Load: 90 requests/second		
RT DB	9	10	11	21	80
TT DB	11	12	14	306	1839
RT DB + WS	182	279	298	290	1133
			Query Load: 120 requests/second		
RT DB	10	10	10	17	85
TT DB	12	12	24	341	1936
RT DB + WS	224	240	355	372	1070

Table 7.13: Response & Transactions times - Materialized Views (#Transactions: 120x15)

Similar to the SQL tuning optimization the results show acceptable response and transaction times for all the datasets except the 3M dataset. The 3M dataset has a RT DB of about 80 milliseconds with a TT DB of about 1900 milliseconds, almost 2 seconds. While it is an huge improvement for the 3M dataset, even better than the improvement of applying the SQL tuning optimization, it is still unacceptable.

7.3.2 Outcome

Table 7.14 shows the results as an average of the response and transaction times of all query loads, these values are graphically shown in figure 7.6.

Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
RT DB	9	10	11	19	78
TT DB	11	11	16	216	1913
RT DB + WS	220	276	307	299	1040

Table 7.14: Average response and transaction times of all query loads - Materialized Views

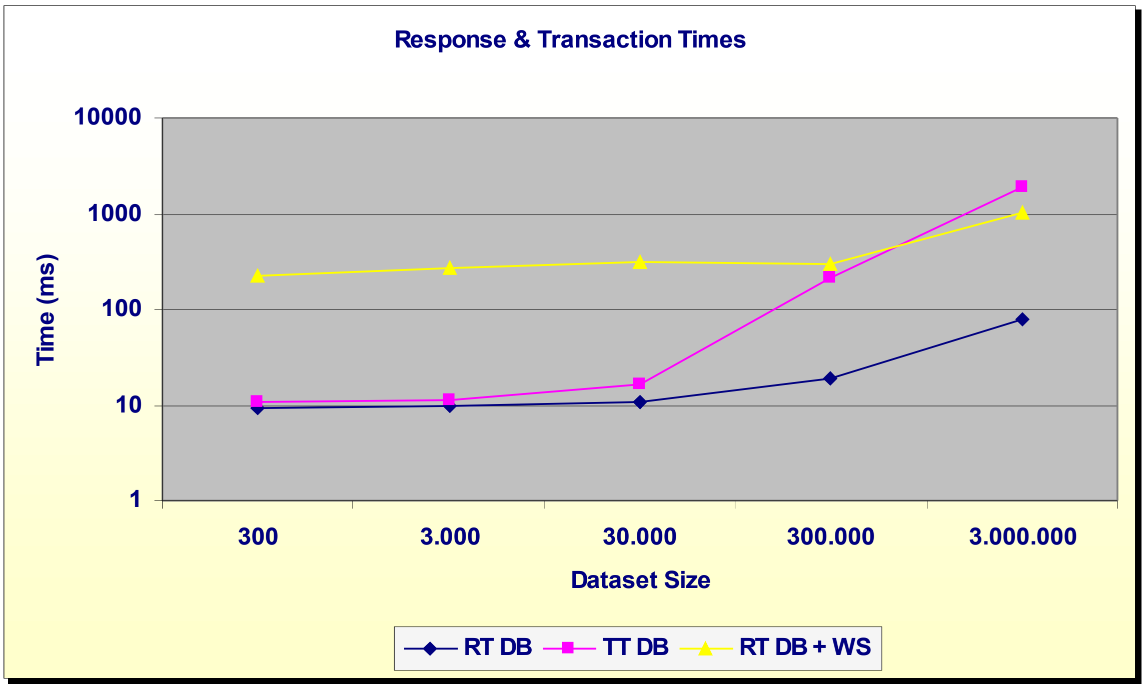


Figure 7.6: Average response and transaction times of all query loads - Materialized Views

The graph shows a increase in RT DB and TT DB when the dataset size increases beyond 30.000. The RT DB + WS increases considerably after a dataset size of over 300.000 records. The values of the graph are significantly lower compared to the values of the graph of the initial run, therefore it can be stated that the Materialized view optimization has had the desired effect of lowering response and transaction times. The larger the dataset size the more significant the influence of applying the Materialized view optimization becomes.

Table 7.15 shows the results as an average of the response and transaction times of all dataset sizes, these values are graphically shown in figure 7.7.

Response & Transactions time (Average DSS)					
QL	**15**	**30**	**60**	**90**	**120**
RT DB	25	22	27	26	26
TT DB	382	431	452	436	465
RT DB + WS	399	400	454	436	452

Table 7.15: Average response and transaction times of all dataset sizes - Materialized Views

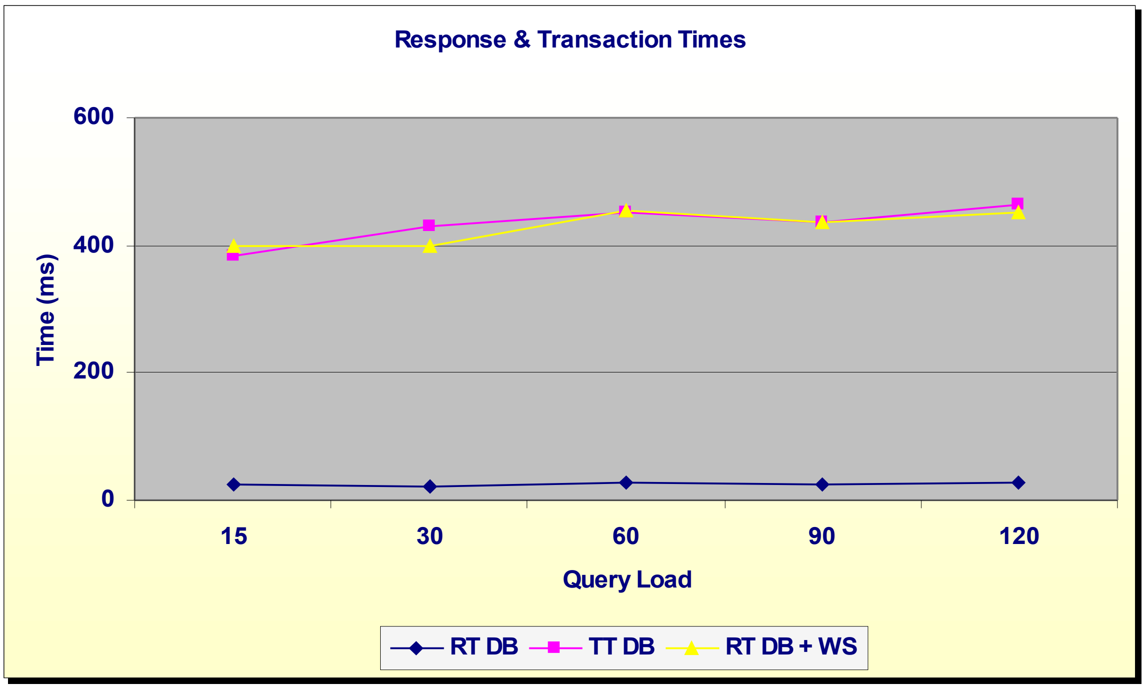


Figure 7.7: Average response and transaction times of all dataset sizes - Materialized Views

Once again the graph shows the database being impervious to differences in query loads when the database has been broken. The RT DB is constant for all query loads there is, once again, an increase in TT DB when the QL increases from 30 to 60. After the increase the TT DB and RT DB + WS are constant again.

7.3.3 Comparison

To measure the influence of applying the Materialized view optimization the results of the tests have to be compared to the results of the initial tests. Table 7.16 shows a comparison of results. The difference between the results is displayed as absolute difference, absolute relative difference and as a factor value.

Comparison of Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
Initial TPS	109,63	106,62	92,92	6,64	0,65
Optimized TPS	112,89	112,79	97,57	45,63	7,32
Absolute Difference	3,26	6,17	4,65	38,99	6,67
Absolute Relative Difference	2,97%	5,79%	5,00%	587,20%	1026,15%
Factor Value	1,0297	1,0579	1,0500	6,8720	11,2615

Table 7.16: Comparison of TPS - Initial versus optimized

For all the datasets there is a increase in maximum amount of TPS. The 300, 3K and 30K datasets show only a small percentage of increase, whilst the largest two datasets show a huge percentage of increase in maximum TPS, the TPS of the 3M dataset has increased over 1000%. The factor values of increase show that for the 300K dataset the maximum TPS has increased almost 7 times, while for the 3M dataset the increase is more than 11 times. The increase in maximum TPS in in accordance with the expected results of the

Materialized view optimization, the largest datasets benefit the most.

Table 7.17 shows the comparison of results based on the average QL.

Comparison of Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
Initial Run					
RT DB	16	17	17	152	20758
TT DB	19	19	23	2141	22969
RT DB + WS	269	352	312	1599	22065
Materialized Views Optimization					
RT DB	9	10	11	19	78
TT DB	11	11	16	216	1913
RT DB + WS	220	276	307	299	1040
Absolute Difference					
RT DB	6	7	6	133	20680
TT DB	8	8	7	1926	21056
RT DB + WS	49	76	5	1300	21025
Absolute Relative Difference					
RT DB	40,51%	42,86%	37,21%	87,75%	99,62%
TT DB	41,94%	40,63%	29,57%	89,93%	91,67%
RT DB + WS	18,07%	21,70%	1,66%	81,33%	95,29%
Factor Value					
RT DB	0,5949	0,5714	0,6279	0,1225	0,0038
TT DB	0,5806	0,5938	0,7043	0,1007	0,0833
RT DB + WS	0,8193	0,7830	0,9834	0,1867	0,0471

Table 7.17: Comparison of Response & Transactions times of all query loads – Initial versus optimized

The percentages show for each dataset size considerable improvements, as expected the larger the datasets are the relatively more they are influenced by the Materialized views optimization, the 3M dataset, for example, has an almost 100% improvement in RT DB.

Table 7.18 shows the average difference for the RT DB, TT DB and RT DB + WS based on all dataset sizes.

Average Differences – Materialized Views Optimization			
	Average Difference	**Average Relative Difference**	**Factor Value**
RT DB	-4167	-61,59%	0,3841
TT DB	-4601	-58,75%	0,4125
RT DB + WS	-4491	-43,61%	0,5639
Average RT & TT	-4384	-60,17%	0,3983

Table 7.18: Average differences in response and transaction times - Materialized Views

The table shows a average decrease of 4.4 seconds, next to a 60% decrease and factor value of 0,3983. Therefore the Materialized views optimization can be rated at an average decrease of 60% for RT DB and TT DB.

7.3.4 Conclusion

The aim of the Materialized views optimization is to lower the response and transaction times of the database considerably. The effect of applying the optimization was expected to be more significant for larger datasets. The results of the test showed an considerable decrease in database response and transaction times of an average of 60%. Especially the larger datasets were influenced by the optimization. Based on the results the Materialized views optimization can be considered as a useful optimization with good results.

7.4 Range Partitioning

The focus of the Range partitioning optimization is the significant decrease in response and transaction times for the larger datasets. It is expected that the 300K and 3M datasets benefit the most from this optimization. By applying this optimization the database should be better performing and better scalable with respect to the dataset sizes. The optimization is based on the SQL tuning optimization, the expected performance improvement is on top of the improvement caused by SQL tuning. Therefore the results of applying Range partitioning are not compared to the initial tests, but to the results of the SQL tuning optimization tests.

7.4.1 Results

Table 7.19 shows the result from the transactions per second test with the Range partitioning optimization applied.

Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
TPS	92,64	93,43	81,65	35,50	5,62

Table 7.19: Transactions per second – Range Partitioning – Spatial Database (Duration: 60s)

The results of the transaction per second test show that with the Range partitioning optimization applied, the database is able to maintain about 90 TPS for the 300, 3K and 30K datasets. For the 300K and 3M datasets the TPS is considerably lower, almost 36 TPS and almost 6 TPS. Table 7.20 show the result of the response and transaction time test with the SQL tuning optimization applied.

Response & Transactions times					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
Query Load: 15 requests/second					
RT DB	20	21	27	42	328
TT DB	21	23	34	71	2163
RT DB + WS	260	248	268	290	991
Query Load: 30 requests/second					
RT DB	19	18	19	35	215
TT DB	21	19	22	90	2038
RT DB + WS	250	294	243	223	1201
Query Load: 60 requests/second					
RT DB	19	19	21	47	326
TT DB	21	21	26	256	2035
RT DB + WS	233	263	238	364	1370
Query Load: 90 requests/second					
RT DB	23	21	24	30	251
TT DB	27	25	35	302	2108
RT DB + WS	232	213	243	306	1377
Query Load: 120 requests/second					
RT DB	15	15	12	21	316
TT DB	19	20	36	391	2136
RT DB + WS	212	217	252	364	1329

Table 7.20: Response & Transaction times - Range Partitioning (#Transactions: 120x15)

The table shows acceptable response and transaction time for all the datasets except the largest one, the 3M dataset. The 3M dataset has a response time of about 300 milliseconds while its transaction time is about 2,1 seconds.

7.4.2 Outcome

Table 7.21 shows the results as an average of the response and transaction times of all query loads, these values are graphically shown in figure 7.8.

Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
RT DB	19	19	21	35	287
TT DB	22	22	31	222	2096
RT DB + WS	237	247	249	309	1254

Table 7.21: Average response and transaction times of all query loads – Range Partitioning

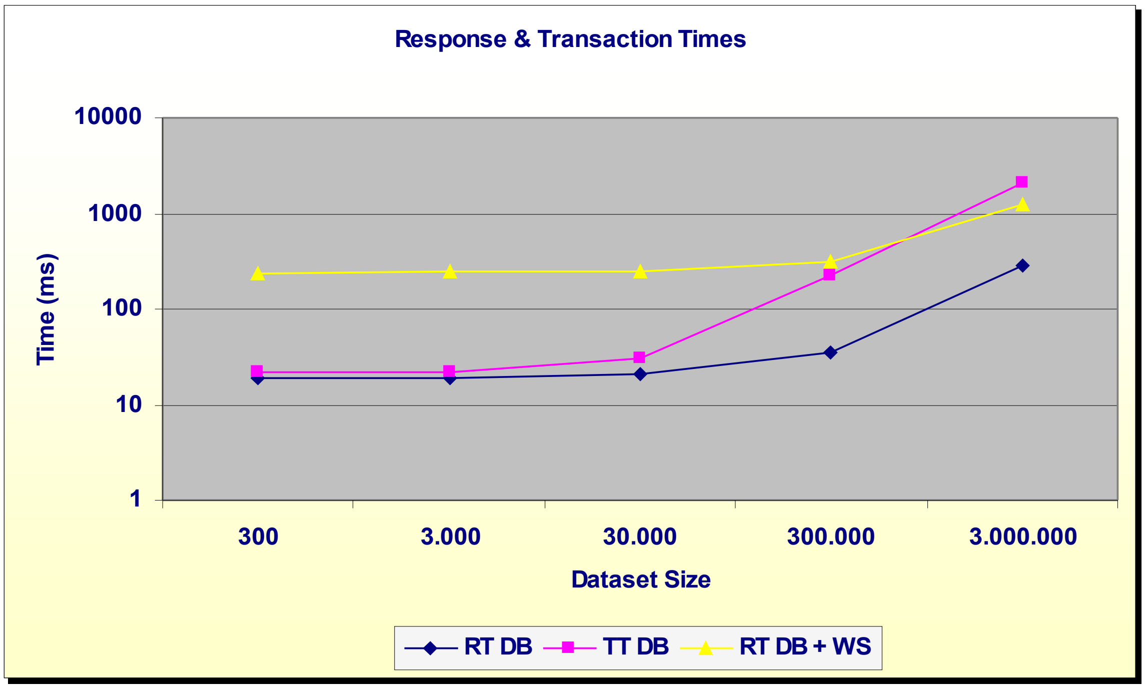


Figure 7.8: Average response and transaction times of all query loads – Range Partitioning

The graph shows an increase in response and transaction times when the dataset size increases beyond 30.000 records. The graph shows a similar result as the graph of the SQL tuning optimization, although most points in this graph have a higher value. Although the graph shows better results than the graph of the initial test, the improvement is caused by the SQL tuning optimization on which the Range partitioning optimization is based. Therefore it can already be concluded that the Range partitioning optimization has not had the desired effect of lowering response and transaction times as expected.

Table 7.22 shows the results as an average of the response and transaction times of all dataset sizes, these values are graphically shown in figure 7.9.

Response & Transactions time (Average DSS)					
QL	**15**	**30**	**60**	**90**	**120**
RT DB	88	61	86	70	76
TT DB	462	438	472	499	520
RT DB + WS	411	442	494	474	475

Table 7.22: Average response and transaction times of all dataset sizes – Range Partitioning

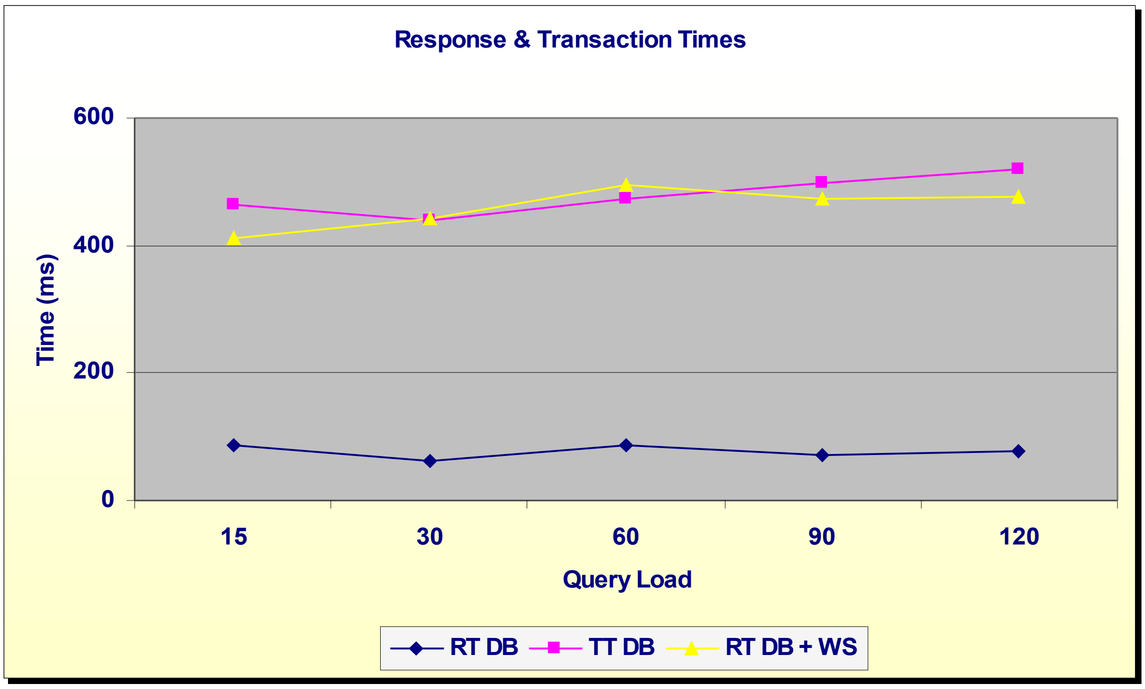


Figure 7.9: Average response and transaction times of all dataset sizes – Range Partitioning

The graph shows once more that the database is impervious to differences in query loads after the database has been broken. Whereas the TT DB shows, from QL30 to QL60, the known increase. This time however, the graph shows that the TT DB increases even further when the QL increases, nonetheless the increase is not significant.

7.4.3 Comparison

To measure the influence of the applied Range partitioning optimization the results should not be compared to the results of the initial test, but to the results of the SQL tuning optimization test. Table 7.16 shows a comparison of results. The difference between the results is displayed as absolute difference, absolute relative difference and as a factor value.

Comparison of Transactions per second – Spatial Database					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
SQL Tuning TPS	113,73	108,73	94,12	38,68	5,97
Optimized TPS	92,64	93,43	81,65	35,50	5,62
Absolute Difference	21,09	15,30	12,47	3,18	0,35
Absolute Relative Difference	18,54%	14,07%	13,25%	8,22%	5,86%
Factor Value	0,8146	0,8593	0,8675	0,9178	0,9414

Table 7.23: Comparison of TPS - SQL Tuning versus Range Partitioning

The table shows for all datasets a decrease in maximum TPS, therefore it can be concluded that, based on the maximum TPS, Range partitioning has a negative influence on the performance of the database. However the optimization is expected to have a performance penalty which should be more significant the smaller the dataset is. Therefore it can be concluded that the Range partitioning extra overhead has a larger performance penalty than can be compensated for, in this specific test case. The table clearly shows that the performance penalty is more significant the smaller the datasets are, as expected.

The results of the TPS comparison can be used as an indication for the comparison of response and transaction times tests. The comparison of response and transaction times should also show the negative performance impact of applying the Range partitioning optimization. Table 7.24 shows the comparison of results based on the average QL. It must be noted that the comparison shows both increase and decrease in response and transaction times, whereas the factor determines whether the differences denote an increase or decrease.

Comparison of Response & Transactions times (Average QL)					
DSS	**300**	**3.000**	**30.000**	**300.000**	**3.000.000**
SQL Tuning Optimization					
RT DB	11	9	10	19	89
TT DB	13	11	15	227	2385
RT DB + WS	245	211	196	244	959
SQL Tuning & Range Partitioning Optimization					
RT DB	19	19	21	35	287
TT DB	22	22	31	222	2096
RT DB + WS	237	247	249	309	1254
Absolute Difference					
RT DB	8	10	10	16	198
TT DB	9	11	15	5	289
RT DB + WS	7	36	53	65	295
Absolute Relative Difference					
RT DB	74,55%	104,35%	98,08%	88,17%	221,25%
TT DB	67,69%	100,00%	98,70%	2,03%	12,11%
RT DB + WS	2,94%	17,28%	27,20%	26,70%	30,72%
Factor Value					
RT DB	1,7455	2,0435	1,9808	1,8817	3,2125
TT DB	1,6769	2,0000	1,9870	0,9797	0,8789
RT DB + WS	0,9706	1,1728	1,2720	1,2670	1,3072

Table 7.24: Comparison of Response & Transactions times of all query loads

Almost all individual results show an increase in response and transaction times, only the results corresponding to TT DB – DSS:300K, TT DB – DSS:3M and RT DB + WS – DSS:300 show a decrease of the timings. The decrease of RT DB + WS – DSS:300 is very insignificant and has an unknown cause at the web service. It will be neglected because the RT DB has increased so the RT DB + WS is expected to increase as well, the actual reason for the decrease is beyond the scope of this optimization. The other two increases are within the scope and are a direct result of the Range partitioning optimization. A side-effect of using Range partitioning is the increase of response times in conjunction with a decrease of transaction times for large datasets. The table clearly shows the dramatic increase of RT DB for the 300K and 3M dataset in conjunction with a small decrease of RT TT. However, this shift from DB TT to DB RT still has as a final result an increase of average response and transaction times compared to the SQL tuning optimization.

Table 7.25 shows the average difference for the RT DB, TT DB and RT DB + WS based on all dataset sizes.

Average Differences – Range Partitioning Optimization			
	Average Difference	**Average Relative Difference**	**Factor Value**
RT DB	48	117,28%	2,1728
TT DB	-52	50,45%	1,5045
RT DB + WS	88	19,79%	1,1979
Average RT & TT	-2	83,86%	1,8386

Table 7.25: Average differences in response and transaction times – Range Partitioning

The table shows a average decrease of 2 milliseconds, next to a 84% increase and factor value of 1,8386. Therefore the Range partitioning optimization can be rated at an average increase of 84% of RT DB and TT DB.

7.4.4 Conclusion

The aim of the Range partitioning optimization is to lower the response and transaction times considerably for the larger datasets. The results of the test showed an considerable increase in database response and transaction times of an average of 84%. Based on these results two conclusions can be drawn. The first, the optimization of Range partitioning is not recommended. Second, the performance penalty induced by Range partitioning is more significant then the expected performance gain, which eventually leads to a decrease of database performance.

7.5 Overall Conclusion

This chapter has discussed and rated three different database optimizations, SQL tuning, Materialized views and Range partitioning. As table 7.26 shows the SQL tuning and Materialized views optimizations have a positive performance impact while Range partitioning has a negative impact on performance.

Comparison of Optimizations		
	Performance Impact	**Factor Value**
SQL Tuning	58,84%	0,4116
Materialized Views	60,17%	0,3983
Range Partitioning	-83,86%	1,8386

Table 7.26: Comparison of optimizations

The factor value denotes the change in performance, the SQL tuning optimization has the effect of improving the response and transaction times by making them almost (1/0,4116=2,43) 2,5 times as fast. The Materialized views optimization makes the response and transaction times (1/0,3983=2,51) more than 2,5 times as fast. The Range partitioning optimization has the effect of making the response and transaction times almost twice (1,8386) as slow, so it has the effect of almost doubling the response an transaction times. It is easy to see that the Range partitioning optimization is not the best optimization for this project as it lowers the performance. The other two optimizations do not differ much in performance improvement. The Materialized view optimizations has a slightly better performance improvement compared to the SQL tuning optimization. Therefore based on performance and scalability grounds, Materialized Views is with a 60% performance improvement the best optimization for this project.

Next to determining the best optimization, the tests also revealed two important aspects of the database. These aspects are:

- ✔ The database seems to be impervious to differences in query loads when the database has been broken.
- ✔ The database still being scalable after it has been broken.

All the tests showed the same results in perspective of query loads after the database had been broken. No matter what QL was used, the database returned the same values for response and transaction times. This leads to the statement that the Oracle Spatial 10g database is, within the scope of this project, impervious to differences in query loads after it has been broken. There is no explanation for this rather unexpected phenomenon, normally an increased QL results in increased response and transaction times when the database has been broken. Theoretically speaking is query load imperviousness impossible. Most likely a condition has been created by external factors in which it seems that the database is impervious to differences in query loads.

The second aspect is about the scalability of the database. As stated before, when the datasets reach a size too large to handle for the database, the database responds with unacceptable response and transaction times. Although the response and transaction times increase, their increase has a linear relation to the increase of the datasets. Therefore it can be concluded that the database shows decent scalability. The massive increase in response and transaction times is most likely related to the datasets being too large to fit in the database memory, which results in a large amount of extra I/O operations. It might be interesting for future research to investigate what happens when the database memory is increased to accommodate even the largest dataset.

The Range partitioning optimization resulted into a performance degradation, which was unexpected. The performance degradation is caused by the performance penalty of Range partitioning which is not compensated for by the expected performance gain. Most likely the optimization is not applicable for this project, the used environment is not very suitable. Although Oracle claims that every database will benefit from Range partitioning, it was not the case for this project. Most likely this optimization is well suited in areas with multiple connected computers with extremely large datasets, such as grid-computing or data warehousing.

7.6 Chapter Summary

This chapter described the performance and scalability testing of the SABRE application. It tested and evaluated three optimizations, SQL Tuning, Materialized Views and Range Partitioning which all had the expectations of improving the performance. The best improvement was seen by applying the Materialized Views optimization. This chapter presented also two other aspects, the database being impervious to differences in query loads when it has been broken, and the database showing decent scalability. The following research question have been addressed in this chapter:

- ✔ *“How does a Spatial Database perform in a GIS-Web Service environment in terms of response and transaction times?”*
- ✔ *“How does a Spatial Database perform and scale up against increasing user loads and increasing datasets?”*
- ✔ *“If there are significant database optimizations, how well do they influence the performance and scalability of the database?”*
- ✔ *“If there are significant database optimizations, can they be used in conjunction or separately, and which is the best (combined) optimization with respect to this project?”*

All the above research questions have been answered during this chapter.

8 Conclusion

This chapter presents the conclusions of the research to the performance and scalability of a spatial database in a GIS-web service environment. This chapter provides a overall conclusion for the research questions and a conclusion for the issue of commercial exploitation of SABRE.

8.1 Research

The main research question stated: “How and with what techniques can a database-design be realised that complies with the performance and scalability requirements of a GIS-Web Service environment?” The research presented in this thesis has provided several results which altogether provide an answer to the main research question.

- ✔ A design of a scalable and extensible SABRE application has been realised, which adhered to its requirements. The design provided a profound basis for the implementation of SABRE. The design was focussed on the spatial database-component of SABRE.
- ✔ A prototype has been created based on the design of SABRE. The prototype implemented only one service, the AREA-event service. The idea behind the prototype was to have a means of testing the performance and scalability of SABRE. The prototype consisted of a PL/SQL package inside the Oracle Spatial 10g database which was published to a rapidly generated web service. The generating of results was done by spatial processing of the database. The final result was a prototype which supported the AREA-event service.
- ✔ A test case has been defined to support the testing of the performance and scalability of SABRE. The test case described the scope, metrics, approach and methodology of the to be performed tests.
- ✔ The results of the initial test showed how SABRE performed in terms of response and transaction times. Only for the smaller datasets, the 300, 3K and 30K datasets, it showed acceptable performance in terms of response and transaction times. The two largest datasets, 300K and 3M, showed unacceptable response and transaction times, the 3M dataset showed response and transaction times of over twenty seconds.
- ✔ The result of the tests showed that SABRE seemed to be impervious to differences in query loads after it had been broken, but remained scalable as the increase of response and transaction times had a linear relation to the increase of the datasets.
- ✔ There have been three database optimizations described which all had the intention of improving the performance and scalability of SABRE. These optimizations where SQL Tuning, Materialized Views and Range Partitioning.
- ✔ The optimizations showed different results, both the SQL Tuning and Materialized Views optimizations had a positive effect on the performance and scalability of the database whereas the Range Partitioning optimization had an negative influence on the database as it degraded the performance. The SQL Tuning optimization improved the performance and scalability on average by 59%, the Materialized Views optimization caused an improvement of 60% whereas the Range Partitioning degraded the performance and scalability by 85%.
- ✔ The optimization Materialized Views could not be used in conjunction with other optimizations. The Range Partitioning optimization could be used in conjunction with SQL Tuning, but proved not useful as Range Partitioning degraded performance. The SQL Tuning optimization showed better results when used solely. However, the best optimization for SABRE is the Materialized Views optimization.

8.2 Commercial Exploitation

The motivation for this project was that LogicaCMG would like to see SABRE being used in a commercial environment. To be used in a commercial environment SABRE has to be able to cope with high performance requirements which stated that SABRE had to be able to handle 50 requests each second.

The results of the research to the performance and scalability of SABRE showed that SABRE, when the SQL Tuning or Materialized Views optimization is applied, is able to handle a query load of 50 requests per second with acceptable response and transaction times for all datasets except one. The largest dataset, containing 3 million records, showed unacceptable response and transaction times. Therefore SABRE, in its current form, does not meet its performance requirements for commercial exploitation. However, the research to the performance and scalability of SABRE showed that SABRE is scalable. Because SABRE is scalable it is doable to increase the performance of SABRE, by adding resources to SABRE, so it can cope with the performance requirements. The scalability aspect defines that adding resources should lead to a performance increase in a manner proportional to the added resources. The only thing refraining LogicaCMG from commercially exploiting the SABRE idea is addition of the required resources to SABRE, adding more database memory for example.

9 Recommendations

This chapter describes the recommendations and future work for SABRE. During this project several issues have come up which are beyond the scope of the project but might prove interesting for future research. The recommendations and future work are addressed below.

- ✔ Adding resources to SABRE
  - For SABRE to be used in a commercial environment its performance has to be increased by adding resources. If sufficient resources are added SABRE might be deployed in a commercial environment in the near future. Assuming the bottleneck of SABRE is I/O bound, it might prove useful to increase the amount of memory for the database. If the database has more memory, the I/O might be seriously reduced. A research to the impact on performance with increasing database memory sizes is advised.
- ✔ Research and development on the web service and XML component
  - This project focussed on the database component of SABRE, the web service and XML component were beyond the scope of this project. The web service and XML component need to be investigated as well, to create a profound application.
- ✔ Addition of services
  - SABRE supports the addition of services. Because of the high degree of extensibility of SABRE, adding different types of services should be easy. By adding different types of services the functionality of SABRE can be largely enhanced.
- ✔ Oracle 11g
  - Oracle has just released a new version of its database, version 11g. Oracle claims significant performance improvement with the new version. It might be worth the effort to investigate this claim as to whether version 11g can significantly improve the performance of SABRE.
- ✔ Range Partitioning
  - Oracle claimed that Range Partitioning improves the performance of a database, small or large. However, this project pointed out that this is not the case. The performance of the database is degraded when Range Partitioning is applied. The cause of the degradation is unknown, most likely the extra Range Partitioning overhead is more than can be compensated for. It might be interesting to investigate what caused the degradation, to be certain of the cause.
- ✔ Horizontal Scaling
  - If in the future the performance requirements of SABRE increase to a level where SABRE is no longer scalable, it might be interesting to investigate performance enhancements by means of horizontal scaling. By using a “divide and conquer” approach of horizontal scaling in conjunction with partitioning, the number of transactions per second is only constrained by the processing power of the servers. Oracle claims that this concept should provide almost linear scalability. However, the cost of horizontal scaling can be enormous. Nonetheless it might be a solution to increasing performance requirements.
- ✔ Radius of SDO_WITHIN_DISTANCE
  - The radius for the spatial function SDO_WD is determined by the value of the SDO_WD function-call argument. For SABRE the radius is static and hard coded as 500 units as it is beyond the scope of this project. Future versions of SABRE might require a dynamic radius for

certain services. The value of the radius can have a dramatic impact on the performance of SABRE. Increasing the radius is certain to decrease the performance of SABRE. If future versions of SABRE require a dynamic radius and the radius needs to be larger than 500 units, a investigation on the impact of a larger radius is strongly advised. This project provides no indication as to what might happen to the performance of SABRE if the radius in increased. Most likely the increase of radius will not have a linear relation to the decrease of performance, which might result in SABRE being non-scalable.

- ✔ Query Load Imperviousness
  - The results of the performance and scalability tests showed that after the database had been broken, the database seemed to be impervious to differences in query loads. A database being impervious to differences in query loads is most remarkable, as it is theoretically speaking impossible. A investigation of what causes the database to seem impervious to differences in query loads might prove useful.

10 Bibliography

[1] *Pro Oracle Spatial*, E.Beinat & A.Godfrind & R.Kothuri
Apress 2004, ISBN-13: 978-1590593837

[2] *Oracle High Performance SQL Tuning*, D.K.Burleson
McGraw-Hill Osborne Media, 2001, ISBN-13: 978-0072190588

[3] *First Course in Database Systems*, J.D.Ullman & J.Widom
Prentice Hall, 1997, ISBN-13: 978-0138613372

[4] *Database Systems: An Application Oriented Approach*, M.Kifer & A.Bernstein & P.M.Lewis
Addison Wesley, 2004, ISBN-13: 978-0321228383

[5] *Oracle Spatial Best Practices: An Oracle Technical White Paper*, 2003
Retrieved from: www.oracle.com/technology/products/spatial/pdf/spatial_best_practices.pdf

[6] *Oracle Spatial Partitioning Best Practices: An Oracle White Paper*, 2004
Retrieved from: www.expobadge.com/dldev/dc/download.cfm?pdfFILE=spatial_twp_partitioningbp_10gr2_0513.pdf

[7] *Oracle® Database Performance Tuning Guide 10g Release 2*, 2005
Retrieved from: http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14211.pdf

11 References

[TPC]	-	Transaction Processing Performance Council http://www.tpc.org/default.asp
[BFQ]	-	Benchmark Factory http://www.quest.com/benchmark-factory/
[WKP1]	**-**	Spatial Database http://en.wikipedia.org/wiki/Spatial_Database
[W3C]	-	World Wide Web Consortium http://www.w3.org/
[WKP2]	**-**	Web Service http://en.wikipedia.org/wiki/Web_service
[WKP3]	**-**	Software Performance Testing http://en.wikipedia.org/wiki/Software_performance_testing
[OC4J]	-	Oracle Containers for Java http://www.oracle.com/technology/products/oc4j
[JDEV]	-	Oracle JDeveloper http://www.oracle.com/technology/products/jdev
[OSP]	-	Oracle Spatial Partitioning Best Practices: An Oracle White Paper http://www.expobadge.com/dldev/dc/spatial_twp_partitioningbp_10gr2_0513.pdf

12 Terminology

[1.1]	**XML**	-	eXtensible Markup Language http://www.w3.org/XML/
[2.1]	**OLTP**	-	Online Transaction Processing http://en.wikipedia.org/wiki/Online_transaction_processing
[2.2]	**SOAP**	-	Simple Object Access Protocol http://www.w3.org/TR/soap/
[2.3]	***WSDL***	-	Web Services Description Language http://www.w3.org/TR/wsdl
[4.1]	**PL/SQL**	-	Procedural Language/ Structured Query Language http://en.wikipedia.org/wiki/PL/SQL
[5.1]	**MBR**	-	Minimum Bounding Rectangle http://en.wikipedia.org/wiki/Minimum_bounding_rectangle

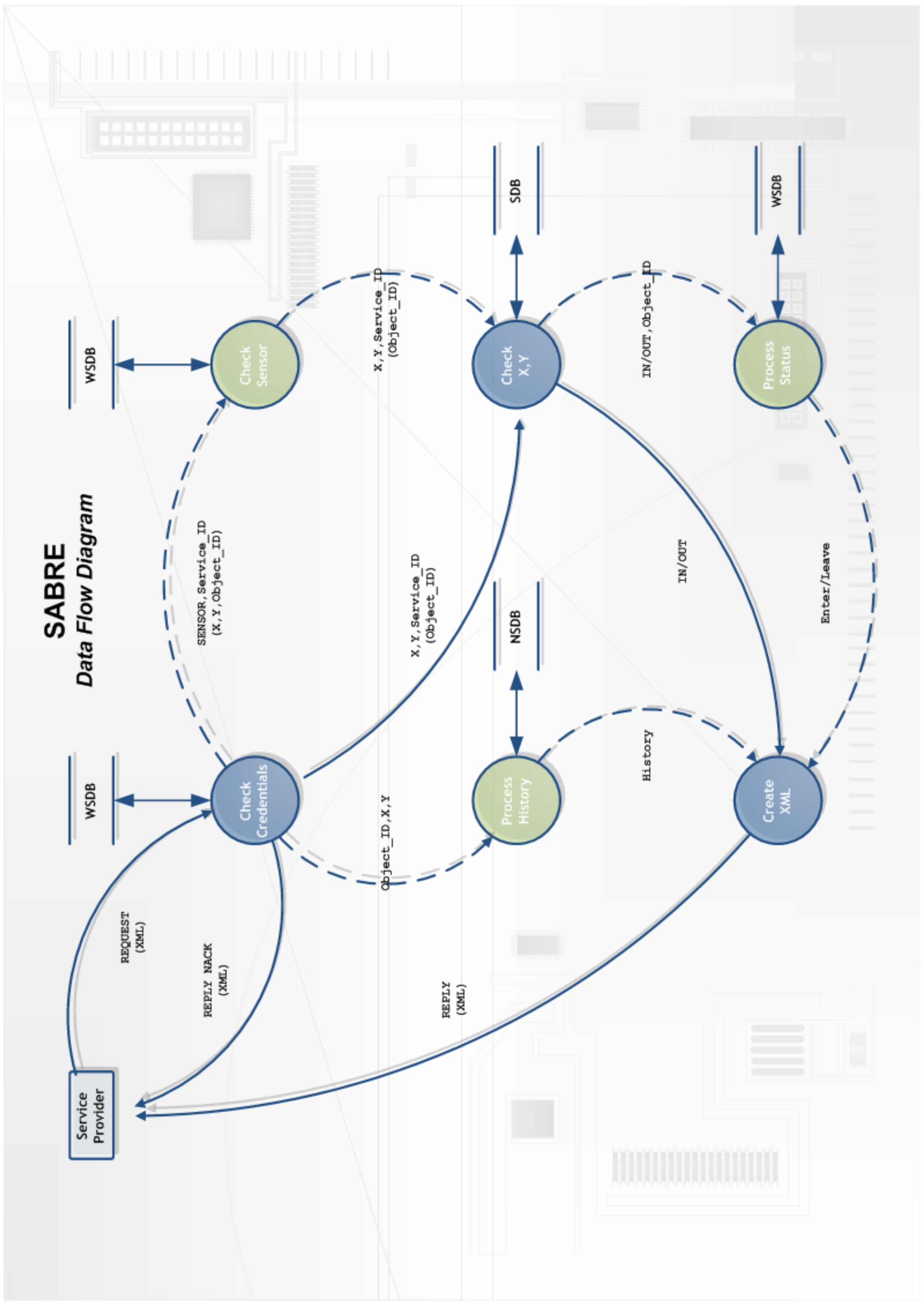
13 Appendix

A SABRE Diagrams

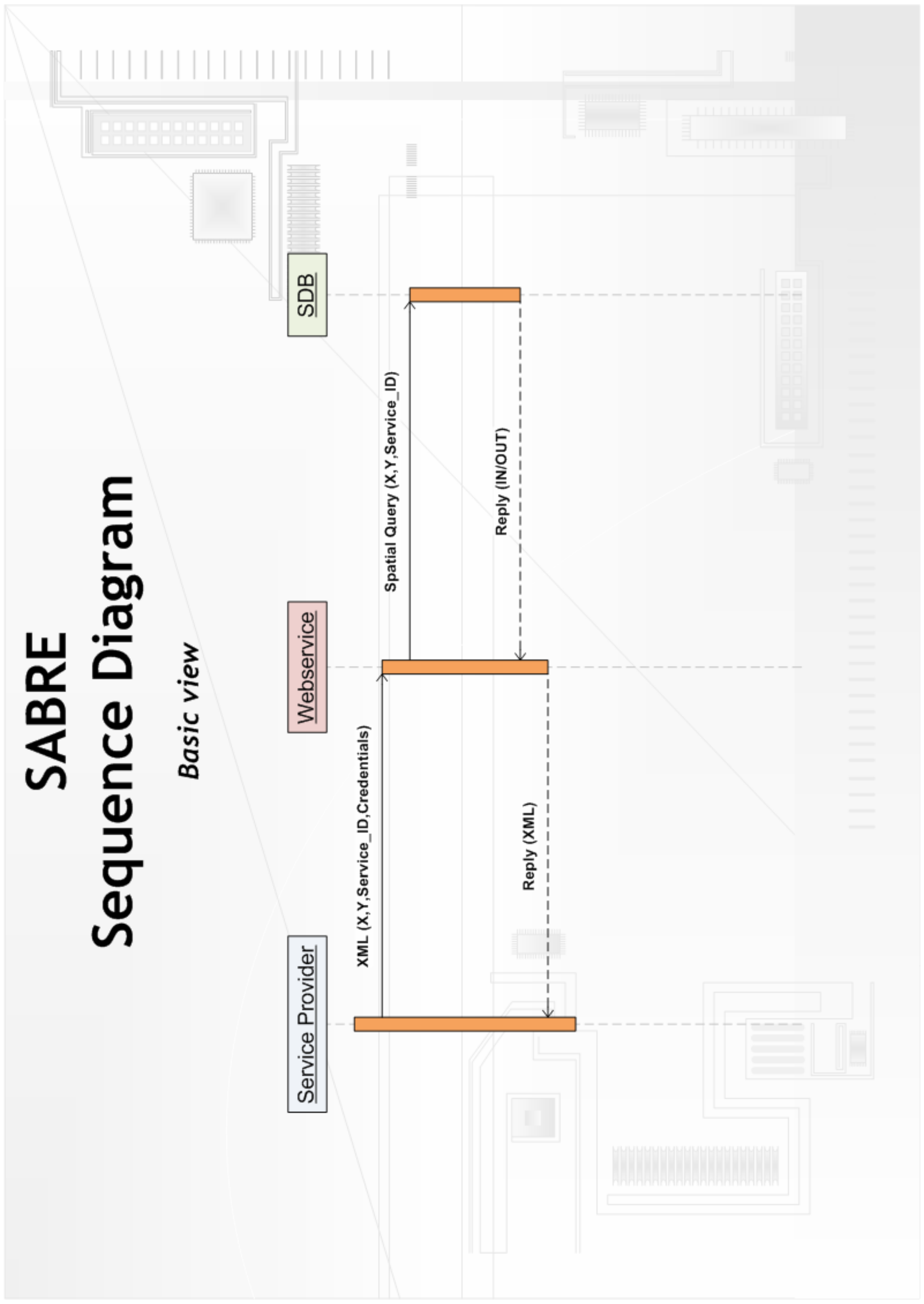
The following pages contain:

1. Data Flow Diagram
2. Basic Sequence Diagram
3. Sensor Sequence Diagram
4. Status Sequence Diagram
5. Sensor & Status Sequence Diagram
6. History Sequence Diagram

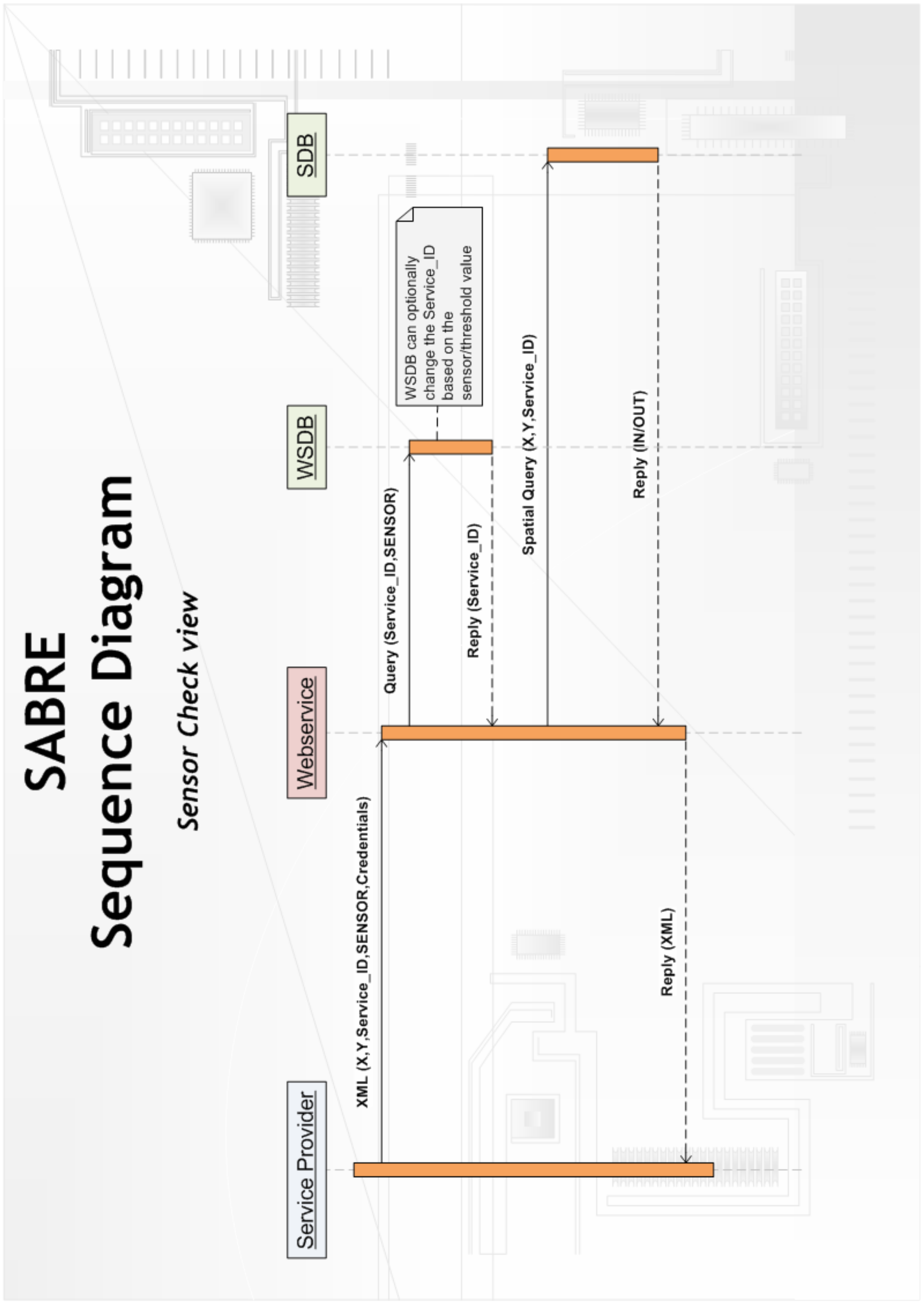
A.I Data Flow Diagram



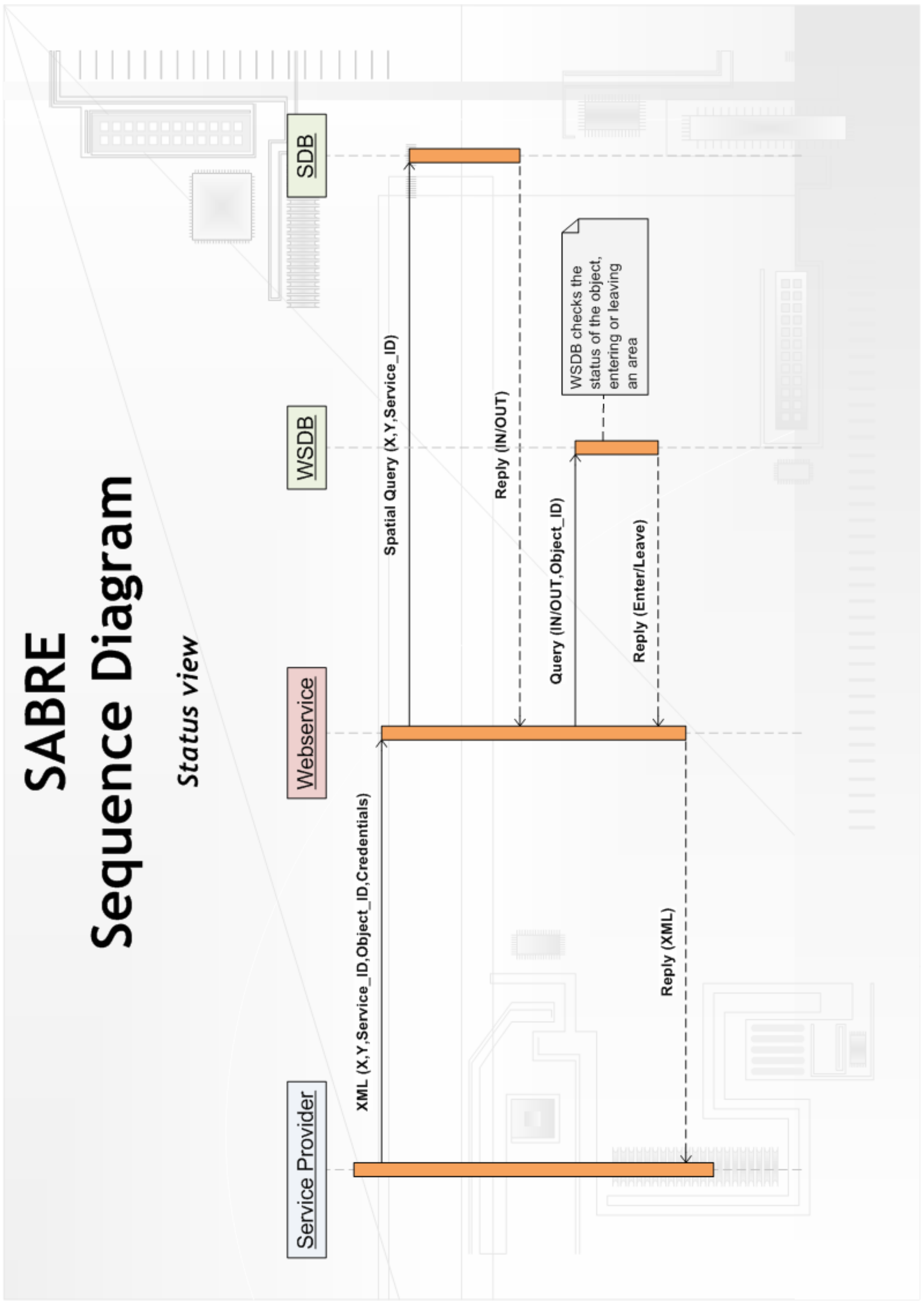
A.II Basic Sequence Diagram



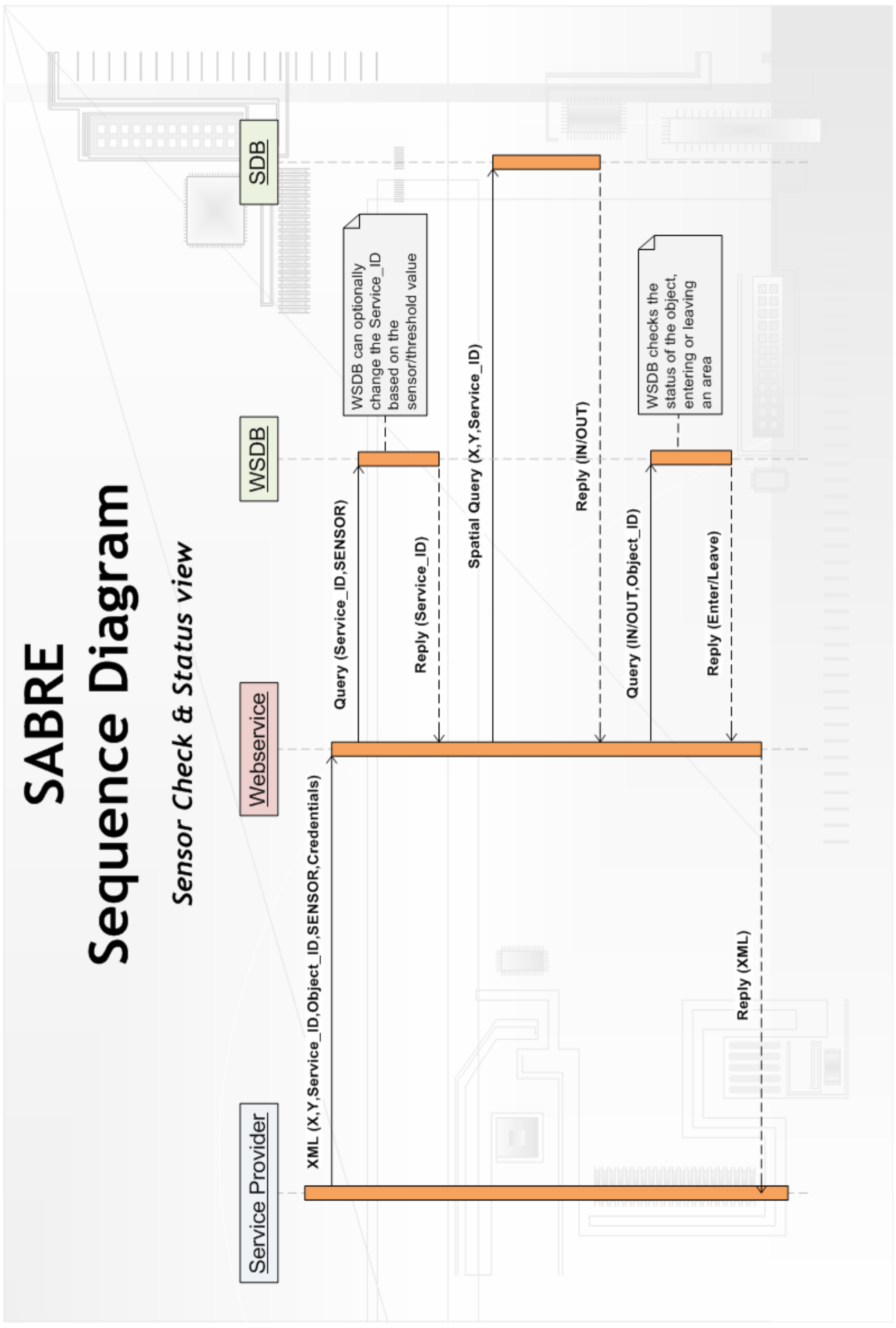
A.III Sensor Sequence Diagram


SABRE
Sequence Diagram
Sensor Check view
Service Provider
Webservice
WSDB
SDB
XML (X,Y,Service_ID,SENSOR,Credentials)
Query (Service_ID,SENSOR)
WSDB can optionally change the Service_ID based on the sensor/threshold value
Reply (Service_ID)
Spatial Query (X,Y,Service_ID)
Reply (IN/OUT)
Reply (XML)

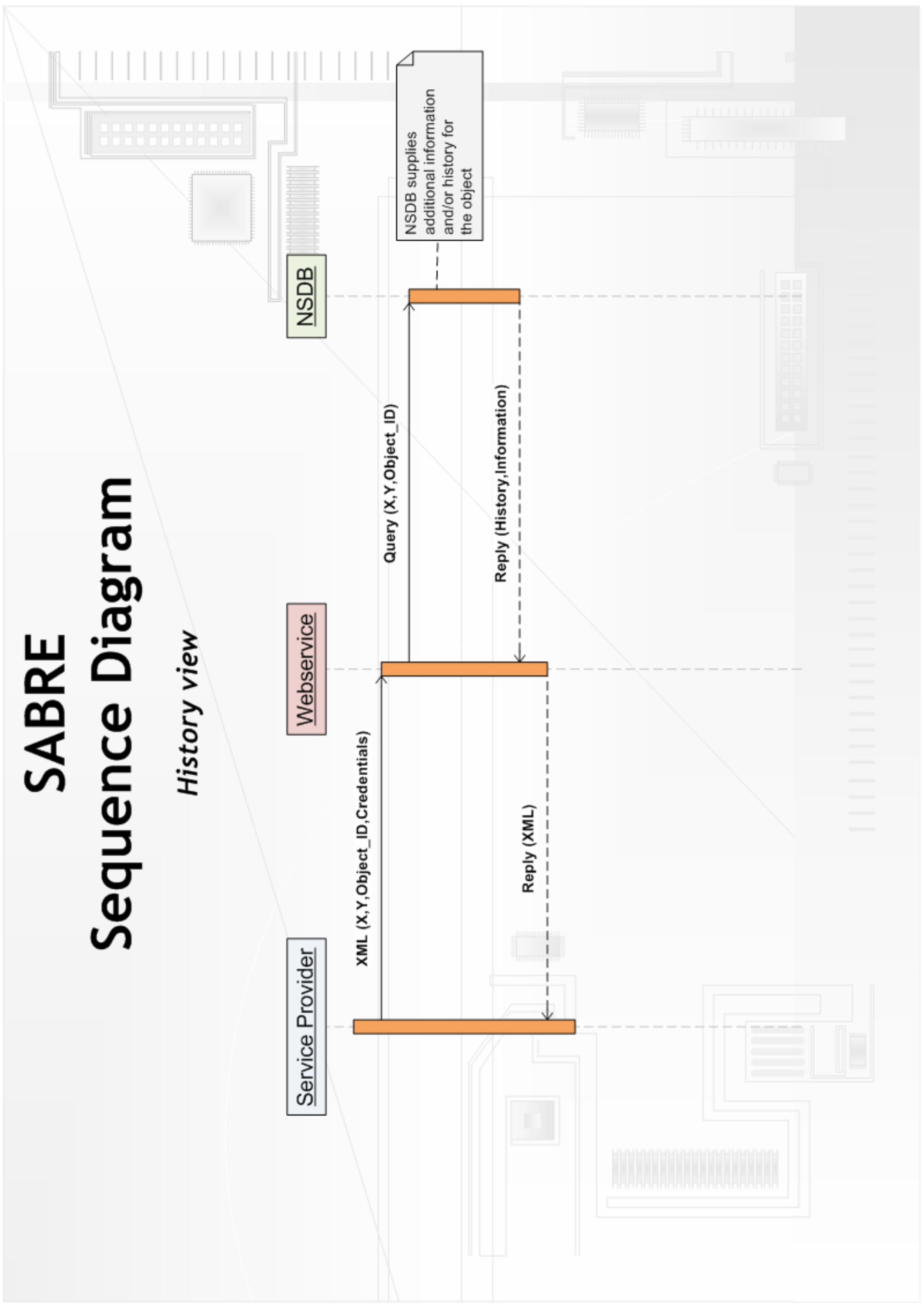
A.IV Status Sequence Diagram


SABRE
Sequence Diagram
Status view
Service Provider
Webservice
WSDB
SDB
XML (X,Y,Service_ID,Object_ID,Credentials)
Spatial Query (X,Y,Service_ID)
Reply (IN/OUT)
Query (IN/OUT,Object_ID)
WSDB checks the status of the object, entering or leaving an area
Reply (Enter/Leave)
Reply (XML)

A.V Sensor & Status Sequence Diagram


SABRE
Sequence Diagram
Sensor Check & Status view
Service Provider
Webservice
WSDB
SDB
XML (X,Y,Service_ID,Object_ID,SENSOR,Credentials)
Query (Service_ID,SENSOR)
Reply (Service_ID)
WSDB can optionally change the Service_ID based on the sensor/threshold value
Spatial Query (X,Y,Service_ID)
Reply (IN/OUT)
Query (IN/OUT,Object_ID)
Reply (Enter/Leave)
WSDB checks the status of the object, entering or leaving an area
Reply (XML)

A.VI History Sequence Diagram


SABRE
Sequence Diagram
History view
Service Provider
Webservice
NSDB
XML (X,Y,Object_ID,Credentials)
Query (X,Y,Object_ID)
NSDB supplies additional information and/or history for the object
Reply (History,Information)
Reply (XML)

B Data model - Table descriptions

Core Tables – Spatial Database			
Table	**Attribute**	**Type**	**Primary Key**
Service	Service_ID	Number	X
	Service_Name	Text	
	Service_Description	Text	
Event	Event_ID	Number	X
	Event_Type	Text	
	Event_Description	Text	
EventSupl	EventSupl_ID	Number	X
	Description	Text	
	NumValue	Number	
	TextValue	Text	
Event_Type	Event_Type_ID	Number	X
	Event_Type	Text	
	Description	Text	
Geometry	Geometry_ID	Number	X
	SDO_GEOMETRY	SDO_GEOMETRY	

Supporting Tables – Spatial Database				
Table	**Attribute**	**Type**	**Primary Key**	**Foreign Key**
R_Service_Event	ID	Number	X	
	Service_ID	Number		Service
	Event_ID	Number		Event
R_Event_Geometry	ID	Number	X	
	Event_ID	Number		Event
	Geometry_ID	Number		Geometry
R_Event_EventSupl	ID	Number	X	
	Event_ID	Number		Event
	EventSupl_ID	Number		EventSupl

C Create Table Script

```
CREATE TABLE "EVENT"
 (      "EVENT_ID" NUMBER NOT NULL ENABLE,
"EVENT_TYPE" VARCHAR2(4000),
"EVENT_DESCRIPTION" VARCHAR2(4000),
"EVENT_TYPE_ID" NUMBER DEFAULT NULL
 ) ;

CREATE TABLE "EVENTSUPL"
 (      "EVENTSUPL_ID" NUMBER NOT NULL ENABLE,
"DESCRIPTION" VARCHAR2(4000),
"NUMVALUE" NUMBER,
"TEXTVALUE" VARCHAR2(4000)
 ) ;

CREATE TABLE "EVENT_TYPE"
 (      "EVENT_TYPE_ID" NUMBER NOT NULL ENABLE,
"EVENT_TYPE" VARCHAR2(4000) NOT NULL ENABLE,
"DESCRIPTION" VARCHAR2(4000)
 ) ;

CREATE TABLE "GEOMETRY"
 (      "GEOMETRY_ID" NUMBER NOT NULL ENABLE,
"GEOMETRY" "SDO_GEOMETRY"
 ) ;

CREATE TABLE "R_EVENT_EVENTSUPL"
 (      "ID" NUMBER NOT NULL ENABLE,
"EVENT_ID" NUMBER NOT NULL ENABLE,
"EVENTSUPL_ID" NUMBER NOT NULL ENABLE
 ) ;

CREATE TABLE "R_EVENT_GEOMETRY"
 (      "ID" NUMBER NOT NULL ENABLE,
"EVENT_ID" NUMBER NOT NULL ENABLE,
"GEOMETRY_ID" NUMBER NOT NULL ENABLE
 ) ;

CREATE TABLE "R_SERVICE_EVENT"
 (      "ID" NUMBER NOT NULL ENABLE,
"SERVICE_ID" NUMBER NOT NULL ENABLE,
"EVENT_ID" NUMBER NOT NULL ENABLE
 ) ;

CREATE TABLE "SERVICE"
 (      "SERVICE_ID" NUMBER NOT NULL ENABLE,
"SERVICE_NAME" VARCHAR2(4000) NOT NULL ENABLE,
"SERVICE_DESCRIPTION" VARCHAR2(4000) NOT NULL ENABLE
 ) ;
```

D SDO_GEOMETRY

From: http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14255/sdo_objrelschema.htm#i1004087

2.2 SDO_GEOMETRY Object Type

With Spatial, the geometric description of a spatial object is stored in a single row, in a single column of object type SDO_GEOMETRY in a user-defined table. Any table that has a column of type SDO_GEOMETRY must have another column, or set of columns, that defines a unique primary key for that table. Tables of this sort are sometimes referred to as spatial tables or spatial geometry tables.

Oracle Spatial defines the object type SDO_GEOMETRY as:

```
CREATE TYPE sdo_geometry AS OBJECT (
 SDO_GTYPE NUMBER,
 SDO_SRID NUMBER,
 SDO_POINT SDO_POINT_TYPE,
 SDO_ELEM_INFO SDO_ELEM_INFO_ARRAY,
 SDO_ORDINATES SDO_ORDINATE_ARRAY);
```

Oracle Spatial also defines the SDO_POINT_TYPE, SDO_ELEM_INFO_ARRAY, and SDO_ORDINATE_ARRAY types, which are used in the SDO_GEOMETRY type definition, as follows:

```
CREATE TYPE sdo_point_type AS OBJECT (
   X NUMBER,
   Y NUMBER,
   Z NUMBER);
CREATE TYPE sdo_elem_info_array AS VARRAY (1048576) of NUMBER;
CREATE TYPE sdo_ordinate_array AS VARRAY (1048576) of NUMBER;
```

Because the maximum SDO_ORDINATE_ARRAY size is 1,048,576 numbers, the maximum number of vertices in an SDO_GEOMETRY object depends on the number of dimensions per vertex: 524,288 for two dimensions, 349,525 for three dimensions, and 262,144 for four dimensions.

The sections that follow describe the semantics of each SDO_GEOMETRY attribute, and then describe some usage considerations (Section 2.2.6).

The SDO_GEOMETRY object type has methods that provide convenient access to some of the attributes. These methods are described in Section 2.3.

Some Spatial data types are described in locations other than this section:

- Section 5.2 describes data types for geocoding.
- Oracle Spatial GeoRaster describes data types for Oracle Spatial GeoRaster.
- Oracle Spatial Topology and Network Data Models describes data types for the Oracle Spatial topology data model.

2.2.1 SDO_GTYPE

The SDO_GTYPE attribute indicates the type of the geometry. Valid geometry types correspond to those specified in the Geometry Object Model for the OGIS Simple Features for SQL specification (with the exception of Surfaces). The numeric values differ from those given in the OGIS specification, but there is a direct correspondence between the names and semantics where applicable.

The SDO_GTYPE value is 4 digits in the format dltt, where:

- d identifies the number of dimensions (2, 3, or 4)
- l identifies the linear referencing measure dimension for a three-dimensional linear referencing system (LRS) geometry, that is, which dimension (3 or 4) contains the measure value. For a non-LRS geometry, or to accept the Spatial default of the last dimension as the measure for an LRS geometry, specify 0. For information about the linear referencing system (LRS), see Chapter 7.
- tt identifies the geometry type (00 through 07, with 08 through 99 reserved for future use).

Table 2-1 shows the valid SDO_GTYPE values. The Geometry Type and Description values reflect the OGIS specification.

Table 2-1 Valid SDO_GTYPE Values

Value	**Geometry Type**	**Description**
dl00	UNKNOWN_GEOMETRY	Spatial ignores this geometry.
dl01	POINT	Geometry contains one point.
dl02	LINE or CURVE	Geometry contains one line string that can contain straight or circular arc segments, or both. (LINE and CURVE are synonymous in this context.)
dl03	POLYGON	Geometry contains one polygon with or without holes.Foot 1
dl04	COLLECTION	Geometry is a heterogeneous collection of elements. COLLECTION is a superset that includes all other types.
dl05	MULTIPOINT	Geometry has one or more points. (MULTIPOINT is a superset of POINT.)
dl06	MULTILINE or MULTICURVE	Geometry has one or more line strings. (MULTILINE and MULTICURVE are synonymous in this context, and each is a superset of both LINE and CURVE.)
dl07	MULTIPOLYGON	Geometry can have multiple, disjoint polygons (more than one exterior boundary). (MULTIPOLYGON is a superset of POLYGON.)

Footnote 1 For a polygon with holes, enter the exterior boundary first, followed by any interior boundaries.

The d in the Value column of Table 2-1 is the number of dimensions: 2, 3, or 4. For example, an SDO_GTYPE value of 2003 indicates a two-dimensional polygon.

Note:

The 1-digit SDO_GTYPE values from before release 8.1.6 value are still supported. If a 1-digit value is used, however, Oracle Spatial determines the number of dimensions from the DIMINFO column of the metadata views, described in Section 2.6.3.

Also, if 1-digit SDO_GTYPE values are converted to 4-digit values, any SDO_ETYPE values that end in 3 or 5 in the SDO_ELEM_INFO array (described in Section 2.2.4) must also be converted.

The number of dimensions reflects the number of ordinates used to represent each vertex (for example, X,Y for two-dimensional objects). Points and lines are considered two-dimensional objects. (However, see Section 7.2 for dimension information about LRS points.)

In any given layer (column), all geometries must have the same number of dimensions. For example, you cannot mix two-dimensional and three-dimensional data in the same layer.

The following methods are available for returning the individual dltt components of the SDO_GTYPE for a geometry object: Get_Dims, Get_LRS_Dim, and Get_Gtype. These methods are described in Section 2.3.

2.2.2 SDO_SRID

The SDO_SRID attribute can be used to identify a coordinate system (spatial reference system) to be associated with the geometry. If SDO_SRID is null, no coordinate system is associated with the geometry. If SDO_SRID is not null, it must contain a value from the SRID column of the SDO_COORD_REF_SYS table (described in Section 6.6.9), and this value must be inserted into the SRID column of the USER_SDO_GEOM_METADATA view (described in Section 2.6).

All geometries in a geometry column must have the same SDO_SRID value.

For information about coordinate systems, see Chapter 6.

2.2.3 SDO_POINT

The SDO_POINT attribute is defined using the SDO_POINT_TYPE object type, which has the attributes X, Y, and Z, all of type NUMBER. (The SDO_POINT_TYPE definition is shown in Section 2.2.) If the SDO_ELEM_INFO and SDO_ORDINATES arrays are both null, and the SDO_POINT attribute is non-null, then the X and Y values are considered to be the coordinates for a point geometry. Otherwise, the SDO_POINT attribute is ignored by Spatial. You should store point geometries in the SDO_POINT attribute for optimal storage; and if you have only point geometries in a layer, it is strongly recommended that you store the point geometries in the SDO_POINT attribute.

Section 2.5.5 illustrates a point geometry and provides examples of inserting and querying point geometries.

Note:

Do not use the SDO_POINT attribute in defining a linear referencing system (LRS) point or an oriented point. For information about LRS, see Chapter 7. For information about oriented points, see Section 2.5.6.

2.2.4 SDO_ELEM_INFO

The SDO_ELEM_INFO attribute is defined using a varying length array of numbers. This attribute lets you know how to interpret the ordinates stored in the SDO_ORDINATES attribute (described in Section 2.2.5).

Each triplet set of numbers is interpreted as follows:

- SDO_STARTING_OFFSET -- Indicates the offset within the SDO_ORDINATES array where the first ordinate for this element is stored. Offset values start at 1 and not at 0. Thus, the first ordinate for the first element will be at SDO_GEOMETRY.SDO_ORDINATES(1). If there is a second element, its first ordinate will be at SDO_GEOMETRY.SDO_ORDINATES(n), where n reflects the position within the SDO_ORDINATE_ARRAY definition (for example, 19 for the 19th number, as in Figure 2-3 in Section 2.5.2).
- SDO_ETYPE -- Indicates the type of the element. Valid values are shown in Table 2-2.

  SDO_ETYPE values 1, 2, 1003, and 2003 are considered simple elements. They are defined by a single triplet entry in the SDO_ELEM_INFO array. For SDO_ETYPE values 1003 and 2003, the first digit indicates exterior (1) or interior (2):

1003: exterior polygon ring (must be specified in counterclockwise order)

2003: interior polygon ring (must be specified in clockwise order)

Note:

The use of 3 as an SDO_ETYPE value for polygon ring elements in a single geometry is discouraged. You should specify 3 only if you do not know if the simple polygon is exterior or interior, and you should then upgrade the table or layer to the current format using the SDO_MIGRATE.TO_CURRENT procedure, described in Chapter 17.

You cannot mix 1-digit and 4-digit SDO_ETYPE values in a single geometry. If you use 4-digit SDO_ETYPE values, you must use 4-digit SDO_GTYPE values.

SDO_ETYPE values 4, 1005, and 2005 are considered compound elements. They contain at least one header triplet with a series of triplet values that belong to the compound element. For SDO_ETYPE values 1005 and 2005, the first digit indicates exterior (1) or interior (2):

1005: exterior polygon ring (must be specified in counterclockwise order)

2005: interior polygon ring (must be specified in clockwise order)

Note:

The use of 5 as an SDO_ETYPE value for polygon ring elements in a single geometry is discouraged. You should specify 5 only if you do not know if the compound polygon is exterior or interior, and you should then upgrade the table or layer to the current format using the SDO_MIGRATE.TO_CURRENT procedure, described in Chapter 17.

You cannot mix 1-digit and 4-digit SDO_ETYPE values in a single geometry. If you use 4-digit SDO_ETYPE values, you must use 4-digit SDO_GTYPE values.

The elements of a compound element are contiguous. The last point of a subelement in a compound element is the first point of the next subelement. The point is not repeated.

- SDO_INTERPRETATION -- Means one of two things, depending on whether or not SDO_ETYPE is a compound element.

  If SDO_ETYPE is a compound element (4, 1005, or 2005), this field specifies how many subsequent triplet values are part of the element.

  If the SDO_ETYPE is not a compound element (1, 2, 1003, or 2003), the interpretation attribute determines how the sequence of ordinates for this element is interpreted. For example, a line string or polygon boundary may be made up of a sequence of connected straight line segments or circular arcs.

  Descriptions of valid SDO_ETYPE and SDO_INTERPRETATION value pairs are given in Table 2-2.

If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the SDO_ORDINATES varying length array.

For compound elements (SDO_ETYPE values 4, 1005, or 2005), a set of n triplets (one for each subelement) is used to describe the element. It is important to remember that subelements of a compound element are contiguous. The last point of a subelement is the first point of the next subelement. For subelements 1 through n-1, the end point of one subelement is the same as the starting point of the next subelement. The starting point for subelements 2...n-2 is the same as the end point of subelement 1...n-1. The last ordinate of subelement n is either the starting offset minus 1 of the next element in the geometry, or the last ordinate in the SDO_ORDINATES varying length array.

The current size of a varying length array can be determined by using the function varray_variable.Count in PL/SQL or OCICollSize in the Oracle Call Interface (OCI).

The semantics of each SDO_ETYPE element and the relationship between the SDO_ELEM_INFO and SDO_ORDINATES varying length arrays for each of these SDO_ETYPE elements are given in Table 2-2.

Table 2-2 Values and Semantics in SDO_ELEM_INFO

SDO_ETYPE	**SDO_INTERPRETATION**	**Meaning**
0	(any numeric value)	Type 0 (zero) element. Used to model geometry types not supported by Oracle Spatial. For more information, see Section 2.5.7.
1	1	Point type.
1	0	Orientation for an oriented point. For more information, see Section 2.5.6.
1	n > 1	Point cluster with n points.
2	1	Line string whose vertices are connected by straight line segments.
2	2	Line string made up of a connected sequence of circular arcs. Each circular arc is described using three coordinates: the start point of the arc, any point on the arc, and the end point of the arc. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a line string made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc, where point 3 is only stored once.
1003 or 2003	1	Simple polygon whose vertices are connected by straight line segments. You must specify a point for each vertex, and the last point specified must be exactly the same point as the first (to close the polygon), regardless of the tolerance value. For example, for a 4-sided polygon, specify 5 points, with point 5 the same as point 1.
1003 or 2003	2	Polygon made up of a connected sequence of circular arcs that closes on itself. The end point of the last arc is the same as the start point of the first arc. Each circular arc is described using three coordinates: the start point of the arc, any point on the arc, and the end point of the arc. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a polygon made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc. The coordinates for points 1 and 5 must be the same (tolerance is not considered), and point 3 is not repeated.

SDO_ETYPE	SDO_INTERPRETATION	Meaning
1003 or 2003	3	Rectangle type (sometimes called optimized rectangle). A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it. The rectangle type can be used with geodetic or non-geodetic data. However, with geodetic data, use this type only to create a query window (not for storing objects in the database). For detailed information about using this type with geodetic data, including examples, see Section 6.2.3.
1003 or 2003	4	Circle type. Described by three distinct non-colinear points, all on the circumference of the circle.
4	n > 1	Compound line string with some vertices connected by straight line segments and some by circular arcs. The value n in the Interpretation column specifies the number of contiguous subelements that make up the line string. The next n triplets in the SDO_ELEM_INFO array describe each of these subelements. The subelements can only be of SDO_ETYPE 2. The last point of a subelement is the first point of the next subelement, and must not be repeated. See Section 2.5.3 and Figure 2-4 for an example of a compound line string geometry.
1005 or 2005	n > 1	Compound polygon with some vertices connected by straight line segments and some by circular arcs. The value n in the Interpretation column specifies the number of contiguous subelements that make up the polygon. The next n triplets in the SDO_ELEM_INFO array describe each of these subelements. The subelements can only be of SDO_ETYPE 2. The end point of a subelement is the start point of the next subelement, and it must not be repeated. The start and end points of the polygon must be exactly the same point (tolerance is ignored). See Section 2.5.4 and Figure 2-5 for an example of a compound polygon geometry.

2.2.5 SDO_ORDINATES

The SDO_ORDINATES attribute is defined using a varying length array (1048576) of NUMBER type that stores the coordinate values that make up the boundary of a spatial object. This array must always be used in conjunction with the SDO_ELEM_INFO varying length array. The values in the array are ordered by dimension. For example, a polygon whose boundary has four two-dimensional points is stored as {X1, Y1, X2, Y2, X3, Y3, X4, Y4, X1, Y1}. If the points are three-dimensional, then they are stored as {X1, Y1, Z1, X2, Y2, Z2, X3, Y3, Z3, X4, Y4, Z4, X1, Y1, Z1}. Spatial index creation, operators, and functions ignore the Z values because this release of the product supports only two-dimensional spatial objects. The number of dimensions associated with each point is stored as metadata in the xxx_SDO_GEOM_METADATA views, described in Section 2.6.

The values in the SDO_ORDINATES array must all be valid and non-null. There are no special values used to delimit elements in a multielement geometry. The start and end points for the sequence describing a specific element are determined by the STARTING_OFFSET values for that element and the next element in the SDO_ELEM_INFO array, as explained in Section 2.2.4. The offset values start at 1. SDO_ORDINATES(1) is the first ordinate of the first point of the first element.

E Relations by Keys

```
ALTER TABLE "SERVICE" ADD CONSTRAINT "SERVICE_PK" PRIMARY KEY ("SERVICE_ID") ENABLE;

ALTER TABLE "EVENT" ADD CONSTRAINT "EVENT_PK" PRIMARY KEY ("EVENT_ID") ENABLE;

ALTER TABLE "GEOMETRY" ADD CONSTRAINT "GEOMETRY_PK" PRIMARY KEY ("GEOMETRY_ID")
ENABLE;

ALTER TABLE "EVENTSUPL" ADD CONSTRAINT "EVENTSUPL_PK" PRIMARY KEY ("EVENTSUPL_ID")
ENABLE;

ALTER TABLE "EVENT_TYPE" ADD CONSTRAINT "EVENT_TYPE_PK" PRIMARY KEY
("EVENT_TYPE_ID") ENABLE;

ALTER TABLE "R_SERVICE_EVENT" ADD CONSTRAINT "R_SERVICE_EVENT_PK" PRIMARY KEY ("ID")
ENABLE;

ALTER TABLE "R_EVENT_GEOMETRY" ADD CONSTRAINT "R_EVENT_GEOMETRY_PK" PRIMARY KEY
("ID") ENABLE;

ALTER TABLE "R_EVENT_EVENTSUPL" ADD CONSTRAINT "R_EVENT_EVENTSUPL_PK" PRIMARY KEY
("ID") ENABLE;


ALTER TABLE "R_SERVICE_EVENT" ADD CONSTRAINT "R_SERVICE_EVENT_SERVICE_FK1" FOREIGN
KEY ("SERVICE_ID") REFERENCES "SERVICE" ("SERVICE_ID") ENABLE;

ALTER TABLE "R_SERVICE_EVENT" ADD CONSTRAINT "R_SERVICE_EVENT_EVENT_FK1" FOREIGN
KEY ("EVENT_ID") REFERENCES "EVENT" ("EVENT_ID") ENABLE;

ALTER TABLE "R_EVENT_GEOMETRY" ADD CONSTRAINT "R_EVENT_GEOMETRY_EVENT_FK1" FOREIGN
KEY ("EVENT_ID") REFERENCES "EVENT" ("EVENT_ID") ENABLE;

ALTER TABLE "R_EVENT_GEOMETRY" ADD CONSTRAINT "R_EVENT_GEOMETRY_GEOMETRY_FK1"
FOREIGN KEY ("GEOMETRY_ID") REFERENCES "GEOMETRY" ("GEOMETRY_ID") ENABLE;

ALTER TABLE "R_EVENT_EVENTSUPL" ADD CONSTRAINT "R_EVENT_EVENTSUPL_EVENT_FK1"
FOREIGN KEY ("EVENT_ID") REFERENCES "EVENT" ("EVENT_ID") ENABLE;

ALTER TABLE "R_EVENT_EVENTSUPL" ADD CONSTRAINT "R_EVENT_EVENTSUPL_EVENTSU_FK1"
FOREIGN KEY ("EVENTSUPL_ID") REFERENCES "EVENTSUPL" ("EVENTSUPL_ID") ENABLE;


ALTER TABLE "R_SERVICE_EVENT" ADD CONSTRAINT "R_SERVICE_EVENT_UK1" UNIQUE
("SERVICE_ID", "EVENT_ID") ENABLE;

ALTER TABLE "R_EVENT_GEOMETRY" ADD CONSTRAINT "R_EVENT_GEOMETRY_UK1" UNIQUE
("EVENT_ID", "GEOMETRY_ID") ENABLE;

ALTER TABLE "R_EVENT_EVENTSUPL" ADD CONSTRAINT "R_EVENT_EVENTSUPL_UK1" UNIQUE
("EVENT_ID", "EVENTSUPL_ID") ENABLE;
```

F SABRE PACKAGE

```
create or replace PACKAGE BODY              "SABRE" AS

function NN (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return PLACES IS
BEGIN
declare
nn_rst PLACES:=PLACES();
j number :=1;
CURSOR nn_cur IS
select g.name from geometry g,service_2_geometry s2g
where g.geometry_id=s2g.geometry_id
and s2g.service_id=SID
and
sdo_nn(g.geometry,sdo_geometry(2001,8307,SDO_point_type(X,Y,null),null,null),'SDO_NU
M_RES=1',1)='TRUE';
BEGIN
OPEN nn_cur;
LOOP
EXIT WHEN nn_cur%NOTFOUND;
nn_rst.extend;
FETCH nn_cur INTO nn_rst(j);
j := j+1;
END LOOP;
CLOSE NN_cur;
RETURN nn_rst;
END;
END NN;

function WD (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return PLACES IS
BEGIN
declare
wd_rst PLACES:=PLACES();
j number :=1;
CURSOR wd_cur IS
select g.name from geometry g,service_2_geometry s2g
where g.geometry_id=s2g.geometry_id
and s2g.service_id=SID
and
SDO_WITHIN_DISTANCE(g.geometry,sdo_geometry(2001,8307,SDO_point_type(X,Y,null),null,
null),'distance=500')='TRUE';
BEGIN
OPEN wd_cur;
LOOP
EXIT WHEN wd_cur%NOTFOUND;
wd_rst.extend;
FETCH wd_cur INTO wd_rst(j);
j := j+1;
END LOOP;
CLOSE wd_cur;
RETURN wd_rst;
END;
END WD;

function SABRE (X IN NUMBER, Y IN NUMBER, SID IN NUMBER) return PLACES IS
BEGIN
declare
finalresult PLACES;
service_type_id number;

```

```
Begin
select e.event_type_id INTO service_type_id from event e, service_2_event s2e
where s2e.service_id=SID
and s2e.event_id=e.event_id
and rownum=1;

IF service_type_id = 1
THEN
finalresult := NN(X,Y,SID);
ELSE
IF service_type_id = 2
THEN
finalresult := WD(X,Y,SID);
END IF;
END IF;

RETURN finalresult;
END;
END SABRE;

end SABRE;
```

G SDO_WITHIN DISTANCE

From: http://download-uk.oracle.com/otn_hosted_doc/intermedia/inter.816/a77132/sdo_oper.htm#77655

Purpose

Uses the spatial index to identify the set of spatial objects that are within some specified Euclidean distance of a given object (such as an area-of-interest or point-of-interest).

Syntax

SDO_WITHIN_DISTANCE(T.column, aGeom, params) ;

Keywords and Parameters

T.column	Specifies a geometry column in a table. The column has the set of geometry objects that will be operated on to determine if they are within the specified distance of the given object (*aGeom*). The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.
aGeom	Specifies the object to be checked for distance against the geometry objects in *T.column*. Specify either a geometry from a table (using a bind variable) or a transient instance of a geometry (using the SDO_GEOMETRY constructor). Data type is MDSYS.SDO_GEOMETRY.
PARAMS	Determines the behavior of the operator. Data type is VARCHAR2.

Keyword	Description
distance	Specifies the Euclidean distance value. This is a required parameter. Data type is NUMBER.
idxtab1	Not supported in this release. Specifies the name of the index if there are multiple spatial index tables for *T.column*.
querytype	Set 'querytype=FILTER' to perform only a primary filter operation. If *querytype* is not specified, both primary and secondary filter operations are performed (default). Data type is VARCHAR2.
layer_gtype	Allows special processing for point data. If the objects in *T.column* have only point data, set this parameter to 'POINT' for optimal performance. Do not set this parameter to 'POINT' if *T.column* contains any *n* on-point objects. Data type is VARCHAR2. Default is 'NOTPOINT'.

Returns

The expression SDO_WITHIN_DISTANCE(arg1, arg2, arg3) = `TRUE' evaluates to TRUE for object pairs that are within the specified distance, and FALSE otherwise.

Usage Notes

Distance between two extended objects (nonpoint objects such as lines and polygons) is defined as the minimum distance between these two objects. The distance between two adjacent polygons is zero.

The operator must always be used in a WHERE clause and the condition that includes the operator should be an expression of the form:

```
SDO_WITHIN_DISTANCE(arg1, arg2, 'distance = <some_dist_val>') = 'TRUE'
```

H SDO_NN

From: http://download-uk.oracle.com/otn_hosted_doc/intermedia/inter.816/a77132/sdo_oper.htm#78069

SDO_NN

Purpose

Uses the spatial index to identify the nearest neighbors for a geometry.

Syntax

SDO_NN(geometry1, geometry2, param) ;

Keywords and Parameters

geometry1	Specifies a geometry column in a table. The column must be spatially indexed. Data type is MDSYS.SDO_GEOMETRY.
geometry2	Specifies either a geometry from a table or a transient instance of a geometry. The nearest neighbor or neighbors to *geometry2* will be returned from *geometry1*. (*geometry2* is specified using a bind variable or SDO_GEOMETRY constructor.) Data type is MDSYS.SDO_GEOMETRY.
PARAM	Determines the behavior of the operator. Data type is VARCHAR2.

Keyword	Description
sdo_num_res	Specifies the number of results (nearest neighbors). If not specified, the default is 1. For example: 'sdo_num_res=10'

Returns

This operator returns the *sdo_num_res* number of objects from *geometry1* that are closest to *geometry2* in the query. In determining how close two geometry objects are, the shortest possible distance between any two points on the surface of each object is used.

Usage Notes

The operator must always be used in a WHERE clause, and the condition that includes the operator should be an expression of the form SDO_NN(arg1, arg2, `sdo_num_res=<some_val>') = `TRUE'.

You should not make any assumptions about the order of the returned results. For example, the first of several returned objects is not guaranteed to be the one closest to *geometry2*.

If two or more objects from *geometry1* are an equal distance from *geometry2*, any of the objects can be returned on any call to the function. For example, if *item_a*, *item_b*, and *item_c* are closest to and equally distant from *geometry2,* and if 'SDO_NUM_RES=2', two of those three objects are returned, but they can be any two of the three.

SDO_NN is not supported for spatial joins.

I MDSYS.SPATIAL_INDEX

From: http://download-uk.oracle.com/docs/cd/B14117_01/appdev.101/b10826/sdo_objindex.htm#i78196

CREATE INDEX

Syntax

```
CREATE INDEX [schema.]<index_name> ON [schema.]<tableName> (column)
   INDEXTYPE IS MDSYS.SPATIAL_INDEX
   [PARAMETERS ('index_params [physical_storage_params]' )]
   [{ NOPARALLEL | PARALLEL [ integer ] }];
```

Purpose

Creates a spatial index on a column of type SDO_GEOMETRY.

Keywords and Parameters

Value	Description
INDEX_PARAMS	Determines the characteristics of the spatial index.
geodetic	`'geodetic=FALSE'` allows a non-geodetic index to be built on geodetic data, but with restrictions. (FALSE is the only acceptable value for this keyword.) See the Usage Notes for more information. Data type is VARCHAR2.
layer_gtype	Checks to ensure that all geometries are of a specified geometry type. The value must be from the Geometry Type column of Table 2-1 in Section 2.2.1 (except that UNKNOWN_GEOMETRY is not allowed). In addition, specifying POINT allows for optimized processing of point data. Data type is VARCHAR2.
sdo_indx_dims	Specifies the number of dimensions to be indexed. For example, a value of 2 causes the first two dimensions to be indexed. Must be less than or equal to the number of actual dimensions (number of SDO_DIM_ELEMENT instances in the dimensional array that describes the geometry objects in the column). If the value is 3 or higher, the only Spatial operator that can be used on the indexed geometries is SDO_FILTER; the other operators described in Chapter 12 cannot be used. Data type is NUMBER. Default = 2.
sdo_non_leaf_tbl	`'sdo_non_leaf_tbl=TRUE'` creates a separate index table (with a name in the form MDNT_...$) for nonleaf nodes of the index, in addition to creating an index table (with a name in the form MDRT_...$) for leaf nodes. `'sdo_non_leaf_tbl=FALSE'` creates a single table (with a name in the form MDRT_...$) for both leaf nodes and nonleaf nodes of the index. See the Usage Notes for more information. Data type is VARCHAR2. Default = FALSE
sdo_rtr_pctfree	Specifies the minimum percentage of slots in each index tree node to be left empty when the index is created. Slots that are left empty can be filled later when new data is inserted into the table. The value can range from 0 to 50. Data type is NUMBER. Default = 10.
PHYSICAL_STORAGE_PARAMS	Determines the storage parameters used for creating the spatial index data table. A spatial index data table is a regular Oracle table with a prescribed format. Not all physical_storage_params that are allowed in the STORAGE clause of a CREATE TABLE statement are supported. The following is a list of the supported subset.
tablespace	Specifies the tablespace in which the index data table is created. Same as TABLESPACE in the STORAGE clause of a CREATE TABLE statement.

Value	Description
initial	Is the same as INITIAL in the STORAGE clause of a CREATE TABLE statement.
next	Is the same as NEXT in the STORAGE clause of a CREATE TABLE statement.
minextents	Is the same as MINEXTENTS in the STORAGE clause of a CREATE TABLE statement.
maxextents	Is the same as MAXEXTENTS in the STORAGE clause of a CREATE TABLE statement.
pctincrease	Is the same as PCTINCREASE in the STORAGE clause of a CREATE TABLE statement.
work_tablespace	Specifies the tablespace for temporary tables used in creating the index. (Applies only to creating spatial R-tree indexes, and not to other types of indexes.)
{ NOPARALLEL | PARALLEL [integer] }	Controls whether serial execution (NOPARALLEL) or parallel (PARALLEL) execution is used for the creation of the index and for subsequent queries and DML operations that use the index. For parallel execution you can specify an integer value of degree of parallelism. See the Usage Notes for more information about parallel index creation. Default = NOPARALLEL. (If PARALLEL is specified without an integer value, the Oracle database calculates the optimum degree of parallelism.)

Prerequisites

- All current SQL CREATE INDEX prerequisites apply.
- You must have EXECUTE privilege on the index type and its implementation type.
- The USER_SDO_GEOM_METADATA view must contain an entry with the dimensions and coordinate boundary information for the table column to be spatially indexed.

Usage Notes

For information about spatial indexes, see Section 1.7.

Before you create a spatial index, be sure that the rollback segment size and the SORT_AREA_SIZE parameter value are adequate, as described in Section 4.1.1.

If an R-tree index is used on linear referencing system (LRS) data and if the LRS data has four dimensions (three plus the M dimension), the `sdo_indx_dims` parameter must be used and must specify 3 (the number of dimensions minus one), to avoid the default `sdo_indx_dims` value of 2, which would index only the X and Y dimensions. For example, if the dimensions are X, Y, Z, and M, specify `sdo_indx_dims=3` to index the X, Y, and Z dimensions, but not the measure (M) dimension. (The LRS data model, including the measure dimension, is explained in Section 7.2.)

A partitioned spatial index can be created on a partitioned table. See Section 4.1.6 for more information about partitioned spatial indexes, including benefits and restrictions.

A spatial index cannot be created on an index-organized table.

You can specify the PARALLEL keyword to cause the index creation to be parallelized. For example:

```
CREATE INDEX cola_spatial_idx ON cola_markets(shape)
   INDEXTYPE IS MDSYS.SPATIAL_INDEX PARALLEL;
```

For information about using the PARALLEL keyword, see the description of the `parallel_clause` in the section on the CREATE INDEX statement in *Oracle Database SQL Reference*. In addition, the following notes apply to the use of the PARALLEL keyword for creating or rebuilding (using the ALTER INDEX REBUILD statement) spatial indexes:

- The PARALLEL clause is not supported for adding an index table with the ALTER INDEX statement; however, it is supported for rebuilding such an index table with the ALTER INDEX REBUILD statement. One useful scenario is to add a small second index table, and later rebuild the index table specifying the desired parameters and using parallel execution. See the parallel execution example for the ALTER INDEX REBUILD statement.
- The performance cost and benefits from parallel execution for creating or rebuilding an index depend on a system's resources and load. If the system's CPUs or disk controllers are already heavily loaded, you should not specify the PARALLEL keyword.
- Specifying PARALLEL for creating or rebuilding an index on tables with simple geometries, such as point data, usually results in less performance improvement than on tables with complex geometries.

Other options available for regular indexes (such as ASC and DESC) are not applicable for spatial indexes.

Spatial index creation involves creating and inserting index data, for each row in the underlying table column being spatially indexed, into a table with a prescribed format. All rows in the underlying table are processed before the insertion of index data is committed, and this requires adequate rollback segment space.

If a tablespace name is provided in the parameters clause, the user (underlying table owner) must have appropriate privileges for that tablespace.

For more information about using the `layer_gtype` keyword to constrain data in a layer to a geometry type, see Section 4.1.4.

The `'geodetic=FALSE'` parameter is not recommended, because much of the Oracle Spatial geodetic support will be disabled. This parameter should only be used if you cannot yet reindex the data. (For more information about geodetic and non-geodetic indexes, see Section 4.1.2.)

Moreover, if you specify `'geodetic=FALSE'`, ensure that the tolerance value stored in the USER_SDO_GEOM_METADATA view is what would be used for Cartesian data. That is, do not use meters for the units of the tolerance value, but instead use the number of decimal places in the data followed by a 5 (for example, 0.00005). This tolerance value will be used for spatial operators. When you use spatial functions that require a tolerance value with this data, use the function format that allows you to specify a tolerance value, and specify the tolerance value in meters.

J Testing Environment

Host Name:	NL-GEO-PS6
OS Name:	Microsoft(R) Windows(R) Server 2003, Standard Edition
OS Version:	5.2.3790 Service Pack 1 Build 3790
OS Manufacturer:	Microsoft Corporation
OS Configuration:	Standalone Server
OS Build Type:	Uniprocessor Free
System Manufacturer:	IBM
System Model:	8183VAX
System Type:	X86-based PC
Processor(s):	1 Processor(s) Installed.
	[01]: x86 Family 15 Model 4 Stepping 1 GenuineIntel ~2792 Mhz
Total Physical Memory:	1.015 MB
Available Physical Memory:	188 MB
Page File: Max Size:	2.457 MB
Page File: Available:	1.752 MB
Page File: In Use:	705 MB

SGA 276

PGA 91

Current PGA Allocated (KB) 35832

Maximum PGA Allocated (KB) 315656
(since startup)

Automatic Shared Memory Management Enabled

Total SGA Size (MB)

SGA Component Current Allocation (MB)

Shared Pool 84

Buffer Cache 176

Large Pool 4

Java Pool 4

Other 8

List of figures

List of Tables