Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart

Confidential

# Development of a Versioned SQL Database System for Parameter Management in the Common Powertrain Controller

*Entwicklung eines versionierten SQL-Datenbanksystems für das Parametermanagement im Common Powertrain Controller*

**Master's thesis**

In the study program
Mechanical Engineering

by
**Mahesh Kollati**
Matr.-Nr. 3635029

| | |
|---|---|
| **Examiner:** | Prof. Dr.-Ing. Hans-Christian Reuss |
| **Institutional Supervisor:** | Dan Greiner |
| **External Supervisor:** | Nikolas Schönfelder, Daimler Truck AG |

Submitted to the University of Stuttgart
Institute of Automotive Engineering Stuttgart
Chair in Automotive Mechatronics
2025

Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart

Automotive Mechatronics
Prof. Dr.-Ing. H.-C. Reuss

+49 711 685 65601
+49 711 685 65710
info@ifs.uni-stuttgart.de

# Master’s thesis

**for**

**Mr. Mahesh Kollati**

Matr.-Nr. 3635029

**Title:** Development of a Versioned SQL Database System for Parameter Management in the Common Powertrain Controller

*Entwicklung eines versionierten SQL-Datenbanksystems für das Parametermanagement im Common Powertrain Controller*

In current vehicles, the coordination of variant diversity plays an important role. For cost reasons, attempts are made to use common parts across model series and for different equipment variants within a model series. For control units, this approach also extends to the software, where possible. Different characteristics are then implemented using parameters. Parameter management is typically done using databases.

This master's thesis aims to develop a concept for an SQL database for managing parameters for a central control unit in vehicle applications. Various aspects shall be considered.
First, various versioning approaches shall be tested and evaluated. The goal is to document changes in a traceable manner.
Based on the requirements of software developers, various use cases shall be defined. Based on this, a concept for the tables within the database shall be developed.
Furthermore, it shall be described how the database can later be used by both software developers and development engineers. This also includes the development of a user interface.
Finally, the limitations and restrictions of the developed approach should be described.

The results must be documented and presented.

Universität Stuttgart
Institut für Fahrzeugtechnik Stuttgart

| | |
|---|---|
| **Confidentiality note:** | This thesis must be treated confidentially until . |
| **Institutional Supervisor:** | Dan Greiner |
| **External Supervisor:** | Nikolas Schönfelder, Daimler Truck AG |
| **Examiner:** | Prof. Dr.-Ing. Hans-Christian Reuss |
| **Start Date:** | |
| **Submission Date:** | |

Prof. Dr.-Ing. H.-C. Reuss

# Declaration

I, Mahesh Kollati, hereby confirm that I have written the present work, or the parts marked with my name, independently and that I have complied with the relevant provisions in the preparation of the work, in particular on the copyright protection of third-party contributions as well as contributions of "artificial intelligence" (AI), in particular from generative models, and have only used the sources and aids indicated.

As far as my work contains third-party contributions (e.g. images, drawings, text passages, contributions related to "artificial intelligence"), I declare that I have marked these contributions as such (e.g. quotation, reference) and that I have obtained any necessary consents from the authors to use these contributions in my work.

In the event that my work violates the rights of third parties, I declare that I will compensate the University of Stuttgart for any resulting damage and to indemnify the University of Stuttgart against any claims by third parties respectively.

The thesis has not been the subject of any other examination procedure, either in its entirety or in essential parts. Furthermore, it has not yet been published in full or in parts. The electronic copy matches the other specimens.

**Stuttgart,**

Mahesh Kollati

# Contents

# Acronyms

| | |
|---|---|
| **3NF** | Third Normal Form |
| **AUTOSAR** | AUTomotive Open System ARchitecture |
| **CPC** | Common Powertrain Controller |
| **ECU** | Electronic Control Unit |
| **ER** | Entity-Relationship |
| **NoSQL** | Not Only SQL |
| **PDD** | Parameter Definition Database |
| **PID** | Parameter ID |
| **RBAC** | Role-Based Access Control |
| **SQL** | Structured Query Language |
| **UI** | User Interface |
| **VCD** | Vehicle Configuration Database |
| **VMAP** | Variant Management and Parametrization |

## Symbols

# List of Figures

# List of Tables

# Kurzfassung

Moderne Nutzfahrzeuge basieren auf hochentwickelten elektronischen Steuergeräten (ECUs), die tausende konfigurierbare Parameter zur Steuerung von Antriebsstrangfunktionen benötigen. Der derzeitige Excel-basierte Ansatz für das Parametermanagement in der Automobilsoftwareentwicklung birgt erhebliche Risiken, darunter Dateninkonsistenzen, Versionskonflikte und eingeschränkte Nachvollziehbarkeit [38]. Diese Arbeit adressiert diese Herausforderungen durch die Entwicklung einer umfassenden Datenbankarchitektur für Variantenmanagement und Parametrisierung (VMAP) in PostgreSQL.

Die Forschung implementiert einen phasenbasierten Versionierungsansatz, der an den Entwicklungszyklen der Automobilindustrie ausgerichtet ist und gegenüber traditionellen änderungsbasierten Versionierungsmodellen erhebliche Leistungsvorteile bietet, während die Datenintegrität gewährleistet bleibt. Empirische Tests zeigen, dass dieser Ansatz Abfrageantwortzeiten unter 100ms selbst bei Datensätzen mit über 100.000 Parametern erzielt, bei akzeptabler Speichereffizienz [2]. Das hybride Rollen-Berechtigungszugriffsmodell des Systems kombiniert rollenbasierte Sicherheit mit modulspezifischen Zugriffskontrollen, was durch umfangreiche Benutzervalidierung mit realen Engineering-Workflows verifiziert wurde.

Die Integration mit Unternehmenssystemen ermöglicht synchronisierte Parameterdefinitionen und Fahrzeugkonfigurationsdaten, entsprechend den von Hohpe und Woolf [15] beschriebenen Enterprise-Integration-Patterns. Die Leistungsanalyse ergab, dass die Datenbank interaktive Antwortzeiten für die meisten Operationen auch bei Produktionsdatensätzen mit 830 Varianten und 167.990 Segmenten beibehält, wobei bestimmte komplexe Operationen wie Phasenvergleiche eine suboptimale Skalierung aufweisen, die von weiterer Optimierung profitieren würden.

Das resultierende System bietet eine 6,5- bis 21,8-fache Leistungsverbesserung gegenüber nicht-indexierten Implementierungen und löst kritische Einschränkungen des Excel-basierten Ansatzes, einschließlich gleichzeitigem Mehrbenutzer-Zugriff, automatisierter Validierung und umfassender Änderungsverfolgung. Die Leistungsanalyse zeigt, dass, obwohl Änderungshistoriendatensätze den Speicherbedarf dominieren (60,8% der Datenbankgröße), der Kompromiss zwischen Speicher und Leistung die Auditanforderungen in regulierten Automobilentwicklungsumgebungen effektiv unterstützt [34].

# Abstract

Modern commercial vehicles rely on sophisticated Electronic Control Units (ECUs) that require thousands of configurable parameters to manage powertrain functions. The current Excel-based approach to parameter management in automotive software development introduces significant risks including data inconsistency, version conflicts, and limited traceability [38]. This thesis addresses these challenges through the development of a comprehensive database architecture for variant management and parametrization (VMAP) in PostgreSQL.

The research implements a phase-based versioning approach aligned with automotive development cycles, providing significant performance advantages over traditional change-based versioning models while maintaining data integrity. Empirical testing demonstrates that this approach yields query response times below 100ms even with datasets exceeding 100,000 parameters, while maintaining acceptable storage efficiency [2]. The system's hybrid role-permission access control model combines role-based security with module-specific access controls, verified through extensive user management validation with real-world engineering workflows.

Integration with enterprise systems enables synchronized parameter definitions and vehicle configuration data, implementing the enterprise integration patterns described by Hohpe and Woolf [15]. Performance analysis revealed that the database maintains interactive response times for most operations even with production-scale datasets containing 830 variants and 167,990 segments, though certain complex operations like phase comparison exhibit suboptimal scaling that would benefit from further optimization.

The resulting system provides a 6.5x-21.8x performance improvement over non-indexed implementations and resolves critical limitations of the Excel-based approach, including multi-user concurrent access, automated validation, and comprehensive change tracking. Performance analysis demonstrates that while change history records dominate storage requirements (60.8% of database size), the storage-performance tradeoff effectively supports the audit requirements common in regulated automotive development environments [34].

# 1 Introduction

Modern commercial vehicles represent a quintessential example of cyber-physical systems, where sophisticated software enables precise control over complex mechanical components. The software controlling these vehicles has grown exponentially in complexity over recent decades, evolving from simple engine management to comprehensive control of virtually all vehicle functions. At the core of this evolution is the ECU—a specialized computer that executes software to manage specific vehicle functions [7]. Contemporary commercial vehicles contain dozens of interconnected ECUs working in concert to ensure optimal performance, efficiency, and safety across diverse operating conditions.

## 1.1 Background and Context

The automotive industry has undergone a profound transformation over the past decades, evolving from predominantly mechanical systems to highly sophisticated mechatronic platforms [25]. This evolution has been particularly pronounced in the commercial vehicle sector, where modern trucks rely on complex networks of ECUs to manage everything from engine performance to safety systems [7]. These systems must adapt to a wide range of operational conditions, regulatory requirements, and market-specific configurations, creating a significant challenge in managing software variability.

At the heart of this variability management lies the Common Powertrain Controller (CPC)—a central ECU managing critical functions related to engine and transmission control. The CPC's operation is governed by thousands of configurable parameters that determine how the powertrain behaves under specific conditions [34]. These parameters influence everything from basic engine timing to sophisticated emission control strategies, making their precise configuration essential for vehicle performance, efficiency, and regulatory compliance.

The parameter management challenge is further complicated by the global nature of modern vehicle development. Commercial vehicles must conform to different emissions regulations, operate in diverse environmental conditions, and meet varying customer expectations across global markets. Consequently, a single vehicle model may require numerous parameter configurations, each tailored to specific combinations of market requirements, hardware configurations, and customer specifications [38].

## 1.2 Problem Statement

The current approach to parameter management in commercial vehicle development relies predominantly on distributed Excel spreadsheets, a methodology that emerged during a period when parameter counts were manageable and development teams were smaller [38]. However, as software complexity has increased exponentially, this fragmented approach has introduced significant limitations and risks to the development process.

Development teams distributed across different locations must coordinate changes to thousands of parameters, track their versions, and ensure consistency across multiple vehicle platforms. The absence of a centralized version control system makes it exceptionally difficult to track changes effectively and manage releases. This situation becomes particularly critical when dealing with safety-critical parameters that directly influence vehicle performance and regulatory compliance.

The manual nature of current processes, combined with the lack of automated validation mechanisms, introduces substantial risks of data inconsistency, version conflicts, and delayed implementation of critical parameter updates. Parameter changes are not consistently verified against established rules and constraints, potentially leading to incompatible configurations or non-compliant behavior [34].

Integration with critical enterprise systems presents another significant challenge. The current process of synchronizing data with internal database systems involves several manual steps, consuming valuable development resources and introducing potential points of failure in the configuration management workflow. The absence of automated data validation and synchronization mechanisms creates additional risks for data integrity and consistency across these interconnected systems.

Furthermore, the increasing emphasis on rapid development cycles and continuous integration in the automotive industry demands a more sophisticated approach to parameter management [7]. The existing system's limitations become particularly apparent when considering the need for simultaneous development of multiple vehicle variants, each requiring specific parameter configurations for different markets and regulatory environments.

These challenges collectively underscore the urgent need for a modern, database-driven solution that can address the complexities of contemporary automotive software development while providing a scalable foundation for future growth and adaptation.

## 1.3 Research Objectives

This thesis aims to address the fundamental challenges in automotive parameter management through the development of database architecture for Variant Management and Parametrization (VMAP), a web-based application for powertrain parameter configuration. The research objectives encompass both theoretical foundations and practical implementation considerations, focusing on creating a robust solution that meets the complex demands of modern vehicle development processes.

The primary research objective centers on developing a centralized database architecture that can effectively manage the complexity of powertrain parameters while maintaining data integrity and traceability [5]. This architecture must support sophisticated version control mechanisms that can handle parameter variations across different development stages and vehicle variants. The system should provide comprehensive audit trails and change history, enabling development teams to track modifications and understand the evolution of parameter configurations over time.

A second crucial objective focuses on the implementation of a sophisticated version control system that addresses the unique requirements of parameter management in automotive software development. This system must go beyond traditional source code version control approaches to handle the complex relationships between parameters, their variants, and their applications across different vehicle platforms [34]. The version control mechanism should support parallel development streams while maintaining consistency and preventing conflicts in parameter configurations.

The research also aims to establish a comprehensive role-based access control system that supports the diverse needs of different user groups within the development process. This includes creating specialized interfaces and permissions for Module Developers, Documentation Team members, Administrators, and Read-only Users, each with specific capabilities and restrictions aligned with their responsibilities [28]. The access control system must balance security requirements with the need for efficient collaboration among development teams.

Integration with existing enterprise systems represents another critical objective of this research. The VMAP system must establish seamless data exchange mechanisms with internal database systems, ensuring consistent information flow while minimizing manual intervention [7]. This integration should support automated validation of parameter changes and provide mechanisms for maintaining data consistency across different systems.

A final key objective involves the development of database interfaces and query optimization strategies that will support the web-based interface implementation. While the actual User Interface (UI) development falls outside the scope of the thesis, the research will focus on designing efficient database structures, stored procedures, and APIs that enable seamless integration with the planned web interface [25]. This includes developing optimized query patterns for complex operations such as parameter comparison, variant management, and release workflows, while ensuring robust data validation and business rule enforcement.

## 1.4 Significance of the Study

The significance of this research extends beyond addressing immediate technical challenges in parameter management. By developing a comprehensive database solution for variant management and parametrization, this work contributes to the broader field of automotive software engineering in several important ways.

First, the research advances the understanding of version control in parameter-centric systems, extending traditional concepts of software versioning to accommodate the unique characteristics of automotive parameter configurations. While considerable research has been conducted on code versioning, the versioning of parameter data presents distinct challenges that require specialized approaches [2]. This thesis contributes to closing this gap by developing and evaluating new methods for parameter versioning in complex automotive systems.

Second, the work addresses critical industry needs for improved quality and efficiency in vehicle development. Commercial vehicle manufacturers face increasing pressure to reduce development time while managing growing software complexity and ensuring regulatory compliance across global markets [7]. By providing a more robust and efficient parameter management solution, this research directly contributes to these industry priorities, potentially reducing development costs and improving vehicle quality through more consistent parameter configurations.

Third, the research advances the integration of database technology with domain-specific engineering processes. By developing specialized database structures and functions tailored to the unique requirements of automotive parameter management, this work demonstrates how database technology can be adapted to support complex engineering workflows [29]. This integration perspective is valuable not only for automotive applications but also for other engineering domains facing similar challenges in managing complex, highly variable system configurations.

Finally, the research contributes to the growing field of model-based systems engineering by providing a structured approach to managing the parametric aspects of system models. As the automotive industry continues to adopt model-based approaches for system development, the management of parameter configurations becomes increasingly critical for maintaining model integrity and traceability [34]. This thesis provides insights and solutions that support this evolution toward more systematic model-based development practices.

## 1.5 Thesis Structure

The thesis is organized into six chapters that systematically address the research objectives and present a comprehensive solution for automotive parameter management. The structure follows a logical progression from theoretical foundations through practical implementation, ensuring thorough coverage of both academic and industry perspectives.

Following this introduction, Chapter 2 presents a comprehensive review of the theoretical background relevant to automotive electronic control systems and database management approaches. This chapter examines the hierarchical organization of automotive electronic systems, explores database management systems with a focus on relational databases, investigates database design methodologies including Entity-Relationship (ER) modeling and normalization, and evaluates access control models and version control concepts applicable to parameter management.

Chapter 3 presents a comprehensive review of the state of the art in database version control systems and automotive parameter management. This chapter examines existing approaches to software configuration management in the automotive industry, analyzes current database versioning techniques, and evaluates their applicability to parameter management systems. The review encompasses both academic research and industry practices, providing a solid foundation for the proposed solution.

Chapter 4 details the methodology and concept development, beginning with a thorough requirements analysis based on industry needs and academic best practices. This chapter explores the system architecture design, consisting of the database schema, version control mechanisms, and user management frameworks. Particular attention is given to the integration requirements with existing systems and the development of robust validation mechanisms for parameter management.

The implementation strategy and technical design are presented in Chapter 5, which outlines the practical realization of the VMAP system. This chapter describes the development of the database structure, the implementation of version control mechanisms, and the creation of the database interfaces. The chapter also details the integration approaches with internal database systems, highlighting the technical challenges and solutions developed during the implementation phase.

Chapter 6 focuses on system evaluation and validation, presenting a comprehensive assessment of the VMAP system against the defined research objectives. This chapter includes detailed performance analyses, user acceptance testing results, and comparative evaluations against existing parameter management solutions. The evaluation framework incorporates both quantitative metrics and qualitative assessments to provide a thorough understanding of the system's effectiveness.

The thesis concludes with Chapter 7, which summarizes the research findings and presents recommendations for future development. This chapter reflects on the contributions of the research to both academic knowledge and industry practice, discussing the implications for automotive software development and configuration management. Additionally, it outlines potential areas for future research and system enhancement based on the insights gained during the project.

Throughout these chapters, the research methodology combines theoretical analysis with practical implementation, ensuring that the resulting system meets both academic standards and industry requirements. Special attention is given to database versioning approaches, user role management, and integration strategies with existing systems, addressing the unique challenges of automotive software configuration management.

# 2 Theoretical Background

This chapter establishes the theoretical foundation necessary for understanding the design and implementation of the VMAP database system. It begins with an overview of automotive electronic control systems and parameter management, explaining the fundamental concepts that drive the requirements for the VMAP system. The chapter then explores database management systems and design methodologies relevant to the implementation. Finally, it discusses foundational concepts in access control and version management that underpin the system architecture.

## 2.1 Automotive Electronic Control Systems

Modern commercial vehicles represent sophisticated cyber-physical systems where software controls virtually every aspect of vehicle operation. The complexity of these systems has grown exponentially over recent decades, with contemporary trucks containing dozens of interconnected ECUs working in concert to ensure optimal performance, efficiency, and safety [34]. Understanding the structure and organization of these systems is essential for designing an effective parameter management solution.

### 2.1.1 ECU Hierarchy and Parameter Organization

Automotive electronic systems follow a hierarchical organization that structures parameters into logical groupings reflecting the actual architecture of vehicle electronic systems. This hierarchy is not merely organizational but represents the fundamental structure of automotive software development, where components are modularized for maintainability, reusability, and functional separation [25].

At the highest level, ECUs represent distinct hardware components controlling specific vehicle functions such as engine management, transmission control, or brake systems. Each ECU contains specialized software designed to manage particular aspects of vehicle operation, with the CPC serving as a central control unit for critical powertrain functions [19]. Within each ECU, modules represent functional software units that implement specific capabilities such as cruise control, emission control, or diagnostic functions. This modular organization enables independent development and testing of different vehicle functions while maintaining clear interfaces between components.

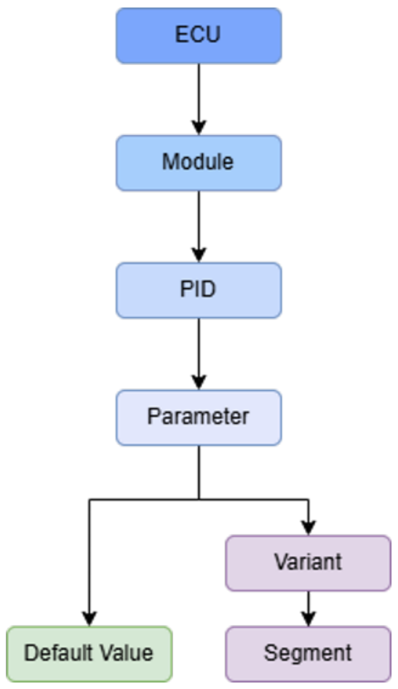


Figure 2.1: Hierarchical Organization of Automotive Electronic Systems

Each module contains Parameter IDs (PIDs) that group related parameters according to their functional purpose. The PID level provides a logical organization that reflects engineering workflows, where parameters are typically managed in functionally related groups rather than individually. Finally, individual parameters define specific configuration values that directly affect system behavior, ranging from simple scalar values to complex multi-dimensional lookup tables that define sophisticated control strategies [35].

Parameters themselves exhibit significant complexity beyond simple configuration values. Automotive control systems frequently employ parameters representing lookup tables, characteristic curves, and multi-dimensional maps that define control strategies for various operating conditions. For instance, an engine timing parameter might be represented as a three-dimensional array where engine speed, load, and temperature serve as independent variables determining the optimal ignition timing angle. This complexity in parameter structure creates specific requirements for database systems designed to manage automotive parameter data.

The hierarchical structure shown in Figure 2.1 illustrates how parameters can follow two distinct paths for value determination. Parameters may use their default values as defined in the baseline configuration, or they may be customized through variants that specify modified values for particular vehicle configurations or operating conditions. This dual-path approach enables efficient parameter management by storing only modifications rather than complete parameter sets for each vehicle variant.

### 2.1.2 Parameter Variants and Customization

The automotive industry faces the challenge of supporting multiple parameter configurations for different vehicle variants, regional requirements, and operating conditions. Rather than maintaining separate complete parameter sets for each configuration, which would lead to significant redundancy and maintenance complexity, automotive systems implement a variant mechanism that allows selective overriding of parameter values based on specific conditions [7].

This variant approach operates on the principle that most parameters retain their default values across different vehicle configurations, with only a subset requiring modification for specific applications. Each parameter maintains a default value defined in the baseline configuration, which serves as the fallback value when no variant-specific override exists. Variants are created to represent specific vehicle configurations, market requirements, or operating conditions, with segments defining the actual modified parameter values within these variants. When no segment exists for a particular parameter within an applicable variant, the system automatically uses the default value, ensuring complete parameter coverage while minimizing storage requirements.

The variant selection process relies on code rules, which are boolean expressions that determine when a variant applies based on vehicle configuration codes. These expressions can represent simple conditions such as engine type selection or complex combinations involving multiple vehicle attributes such as transmission type, emission standard, and market destination. The code rule evaluation process occurs during parameter file generation, where the system determines which variants apply to a specific vehicle configuration and resolves the effective parameter values accordingly.

This approach creates specific requirements for database systems managing automotive parameters. The system must efficiently store and retrieve variant definitions while maintaining the complex relationships between parameters, variants, and segments. Additionally, the parameter resolution process must correctly apply variants based on vehicle configuration codes, ensuring that appropriate parameter values are selected for each specific vehicle configuration while maintaining performance for large parameter sets.

### 2.1.3 Release and Phase Management

Automotive software development follows a structured release process with well-defined phases representing increasing levels of maturity and stability. This process reflects the engineering rigor required for safety-critical automotive systems, where parameter configurations must undergo systematic validation before implementation in production vehicles [7].

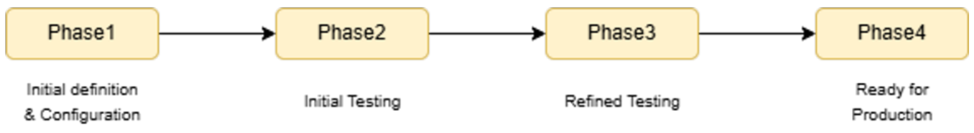


Figure 2.2: Automotive Parameter Release Cycle

The typical release cycle consists of bi-annual releases that align with automotive development schedules and market introduction timelines. Each release progresses through four sequential phases that represent distinct stages in the parameter development lifecycle. The Phase1 involves initial parameter definition and configuration, where new parameters are introduced and baseline configurations are established. Phase2 focuses on initial testing and refinement, where parameters undergo preliminary validation and adjustment based on early testing feedback. Phase3 continues the refinement process with more extensive testing, while Phase4 represents the completed configuration ready for production release.

Figure 2.2 illustrates the linear progression through these phases, with each phase building upon the work completed in the previous phase. This sequential structure ensures that parameter configurations mature systematically, with increasing levels of validation at each stage. The process accommodates the reality that different ECUs may progress through phases at different rates based on their development schedules, testing requirements, and dependency relationships with other vehicle systems.

Phase transitions represent critical points in the development process where parameter configurations are copied forward to establish a new baseline for continued development. This copying mechanism preserves the integrity of completed phases while enabling continued development in subsequent phases. Changes made in earlier phases should typically propagate to later phases unless explicitly overridden, creating requirements for change management and propagation mechanisms within the parameter management system.

The phase-based development process also includes mechanisms for freezing phases at specific development milestones. When a phase is frozen, its parameter configuration becomes read-only, creating a stable reference point for documentation, testing, and compliance activities. This freezing capability is essential for maintaining configuration control in regulated automotive development environments, where stable reference configurations must be preserved for audit and traceability purposes [34].

## 2.2 Database Management Systems

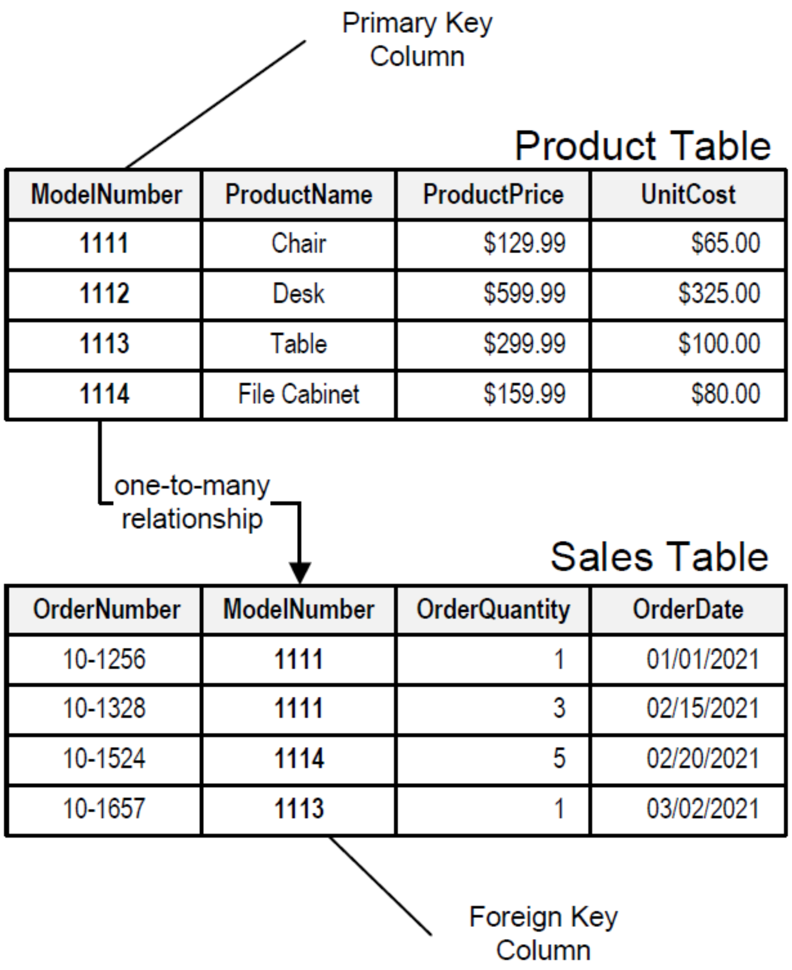
Database management systems provide the technological foundation for structured information storage, organization, and retrieval. They implement sophisticated mechanisms for ensuring data integrity, security, and concurrent access while supporting the complex queries and transactions required by modern applications [29]. For automotive parameter management, the selection and implementation of an appropriate database approach is critical for meeting the performance, reliability, and scalability requirements of engineering workflows.

### 2.2.1 Relational Database Fundamentals

Relational database management systems organize data into structured tables composed of rows and columns, implementing the relational model proposed by E.F. Codd in 1970 [10]. The relational model establishes a mathematical foundation for representing data as relations with well-defined operations for data manipulation, providing a theoretically sound basis for database system implementation.

The relational model's strength lies in its systematic approach to data organization and integrity enforcement. Tables adhere to predefined schemas that specify structure, data types, and constraints applicable to the stored data. Each table includes a primary key that uniquely identifies each row, while foreign keys establish relationships between tables, implementing referential integrity that ensures consistency across related data. This structured approach enables complex queries across multiple related entities while maintaining data consistency through constraint enforcement.

Relational databases adhere to ACID properties that ensure reliable transaction processing in multi-user environments. Atomicity guarantees that transactions are treated as indivisible units that either complete entirely or have no effect, preventing partial updates that could leave the database in an inconsistent state. Consistency ensures that transactions transform the database from one valid state to another, maintaining all defined integrity constraints. Isolation prevents interference between concurrent transactions, enabling multiple users to work with the database simultaneously without



| ModelNumber | ProductName | ProductPrice | UnitCost |
|---|---|---|---|
| **1111** | Chair | $129.99 | $65.00 |
| **1112** | Desk | $599.99 | $325.00 |
| **1113** | Table | $299.99 | $100.00 |
| **1114** | File Cabinet | $159.99 | $80.00 |

| OrderNumber | ModelNumber | OrderQuantity | OrderDate |
|---|---|---|---|
| 10-1256 | **1111** | 1 | 01/01/2021 |
| 10-1328 | **1111** | 3 | 02/15/2021 |
| 10-1524 | **1114** | 5 | 02/20/2021 |
| 10-1657 | **1113** | 1 | 03/02/2021 |

Figure 2.3: Example of a Relational Schema [36]

conflicts. Durability ensures that committed transactions persist permanently, even in the event of system failures [29].

For automotive parameter management, ACID compliance provides essential guarantees for data integrity and consistency. Parameter configurations directly affect vehicle behavior and safety, making the strong consistency guarantees of relational databases crucial for maintaining reliable parameter data. The hierarchical structure of automotive parameter systems, with well-defined relationships between ECUs, modules, PIDs, and parameters, aligns naturally with the relational model's representation of structured data and entity relationships.

### 2.2.2 Alternative Database Approaches

Non-relational database systems, commonly referred to as Not Only SQL (NoSQL) databases, have emerged as alternatives to the relational model for specific application domains [23]. These systems typically prioritize scalability, flexibility, or performance characteristics over the strong consistency guarantees provided by relational databases [6].

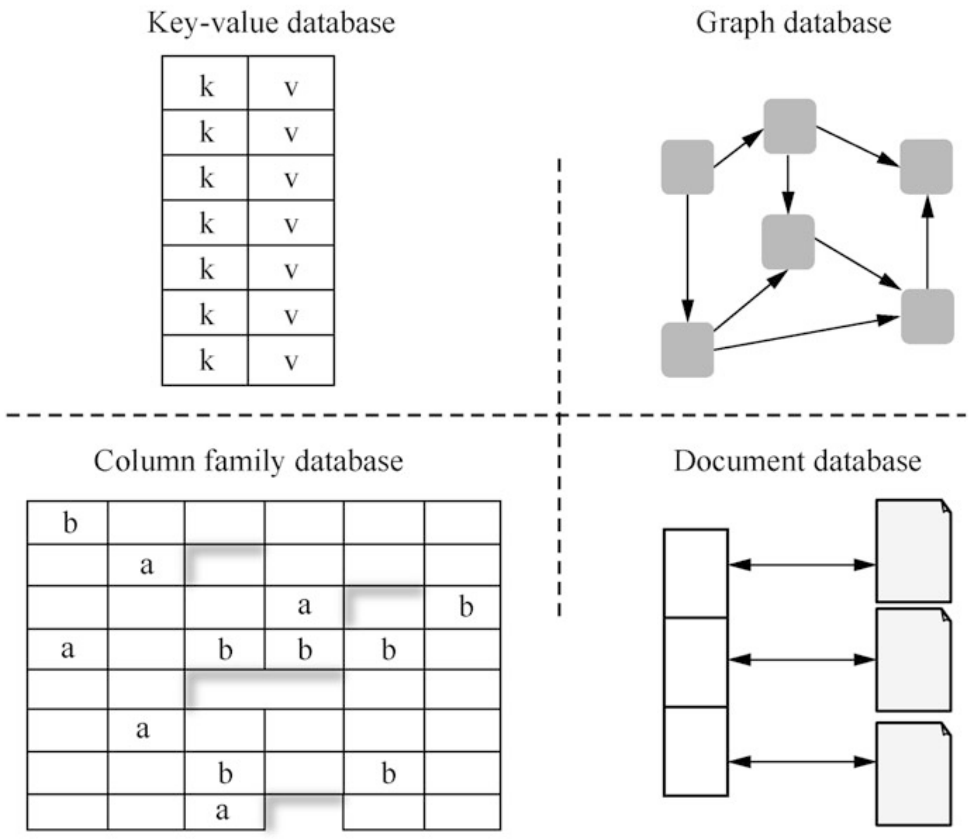


Figure 2.4: Major Types of NoSQL Databases [14]

NoSQL databases can be categorized into several distinct types based on their data models and storage approaches [23]. Document databases store semi-structured data as documents, typically in JSON or XML formats, providing schema flexibility while maintaining query capabilities. Key-value stores offer simple key-based data access with high performance for specific access patterns. Column-family databases organize data in column-oriented structures that can efficiently handle sparse data and analytical workloads. Graph databases represent data as networks of interconnected nodes and relationships, excelling at representing and querying complex relationship structures.

Many NoSQL systems implement the BASE principle (Basically Available, Soft state, Eventually consistent) rather than ACID properties, prioritizing availability and partition tolerance over immediate consistency. This approach enables horizontal scaling across distributed systems but introduces complexity in managing data consistency and transaction semantics [20].

While NoSQL databases excel in specific domains such as high-volume web applications, real-time analytics, and content management systems, they present challenges for applications requiring complex transactions, strict data integrity, or sophisticated queries across related entities [23]. For automotive parameter management, these limitations generally make NoSQL systems less suitable than relational databases, particularly given the critical importance of data consistency and the complex relational structure of automotive parameter data.

## 2.3 Database Design Methodologies

Database design methodologies provide systematic approaches for translating real-world information requirements into efficient, reliable database implementations. These methodologies ensure that database systems effectively support user needs while maintaining data integrity, performance, and maintainability [8]. This section explores the fundamental design approaches that form the theoretical foundation for database system development.

### 2.3.1 Conceptual Modeling with Entity-Relationship Diagrams

Entity-relationship modeling provides a conceptual framework for representing the data structure needed to support identified system requirements. Introduced by Peter Chen in 1976, ER modeling has become the predominant approach for conceptual database design due to its intuitive representation of real-world entities and their relationships [8].

The ER model identifies three fundamental components that describe the structure of information systems. Entities represent the objects or concepts about which information needs to be stored, such as customers, products, or in an automotive context, vehicles, ECUs, or parameters. Attributes describe the specific properties or characteristics of each entity, such as names, identifiers, values, or descriptive information. Relationships describe the associations between entities, expressing how different objects in the system interact or relate to one another.

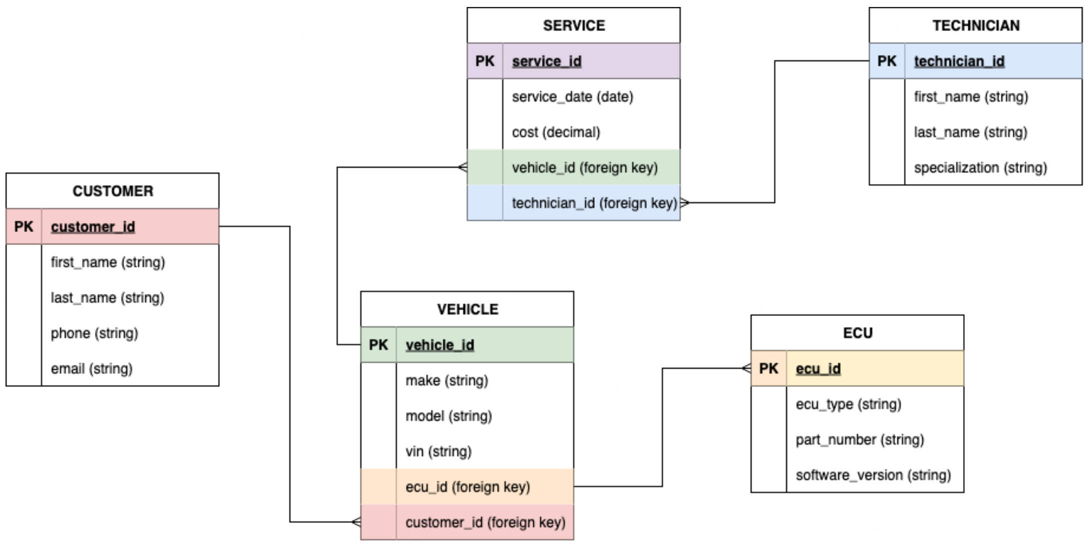


Figure 2.5: Entity-Relationship Diagram for an Automotive Service Center Database

Figure 2.5 illustrates these concepts in a practical context, showing how entities, attributes, and relationships combine to represent a comprehensive information structure. The diagram demonstrates the use of primary keys for unique entity identification, foreign keys for relationship implementation, and the various types of relationships that can exist between entities in a complex system.

ER modeling is particularly valuable for complex domains like automotive systems because it provides a visual representation that stakeholders can understand while being precise enough to guide database implementation. Chen explains that the ER model adopts a natural view of the world consisting of entities and relationships, making it an intuitive approach for modeling real-world systems while maintaining the precision required for technical implementation [8].

### 2.3.2 Data Normalization Principles

Database normalization provides a systematic process for organizing data to minimize redundancy and prevent update anomalies. Developed by E.F. Codd as part of the relational model, normalization proceeds through several normal forms, each addressing specific types of data inconsistencies and structural problems [10].

The normalization process begins with First Normal Form, which requires that each cell in a table contains only atomic values rather than lists or repeating groups. This ensures that data elements are indivisible and can be manipulated consistently using relational operations. Second Normal Form builds upon this foundation by requiring that all non-key attributes depend functionally on the entire primary key, preventing partial dependencies that can lead to update anomalies. Third Normal Form further refines the structure by eliminating transitive dependencies, ensuring that non-key attributes depend only on the primary key rather than on other non-key attributes [29].

For most practical applications, achieving Third Normal Form provides an appropriate balance between data integrity and system performance. As explained by Simon, Third Normal Form is considered adequate for most practical purposes, with further normalization typically performed only when specific data integrity requirements demand additional refinement [32].

In automotive parameter management, normalization helps organize complex data about ECUs, modules, and parameters into a structure that maintains consistency while supporting efficient access. Proper normalization ensures that parameter definition changes need to be recorded in only one location, eliminating the risk of inconsistent values across the database while supporting the complex relationships inherent in automotive electronic systems.

## 2.4 Access Control and Security Fundamentals

Database systems frequently contain sensitive information that requires controlled access based on user roles and responsibilities. Access control mechanisms provide systematic approaches to managing permissions while maintaining security and supporting organizational workflows [28]. This section examines the fundamental concepts that underpin access control implementation in database systems.

### 2.4.1 Role-Based Access Control Principles

Role-Based Access Control represents a systematic approach to managing user permissions by associating permissions with roles rather than individual users. Introduced by David Ferraiolo and Richard Kuhn, Role-Based Access Control (RBAC) has become the predominant model for access control in enterprise systems due to its balance of security effectiveness and administrative simplicity [28].

The fundamental concept of RBAC involves organizing permissions around roles that correspond to job functions within an organization. Rather than granting permissions directly to individual users, the system defines roles such as administrator, engineer, or analyst, with each role receiving a specific set of permissions appropriate to its responsibilities. Users are then assigned to appropriate roles, inheriting the permissions associated with those roles.

This approach provides significant advantages over direct permission assignment. Administrative complexity is reduced by managing permissions at the role level rather than for individual users, simplifying the process of granting appropriate access to new users or modifying access when job responsibilities change. Security is enhanced through implementation of the principle of least privilege, ensuring that users receive only the permissions necessary for their specific responsibilities. The role abstraction also provides a clear mapping between organizational responsibilities and system permissions, making access control policies easier to understand and audit [12].

The theoretical foundation of RBAC includes several key components that work together to provide flexible access control. Users represent individuals who require access to system resources. Roles represent collections of permissions that correspond to specific job functions or responsibilities within the organization. Permissions define specific operations that can be performed on particular resources or data elements. Sessions provide temporary bindings between users and their assigned roles, enabling dynamic activation of different permission sets based on current activities [27].

### 2.4.2 Database Security Implementation

Modern database management systems provide various mechanisms for implementing access control policies, ranging from basic user authentication to sophisticated role-based permission systems. These mechanisms operate at multiple levels within the database system, from table-level access control to row-level security policies that can restrict access based on data content [31].

Database-level security typically begins with user authentication, where the system verifies user identity before granting access to database resources. Once authenticated, authorization mechanisms determine which operations the user can perform on specific database objects such as tables, views, or stored procedures. Most enterprise database systems support role-based approaches where users can be assigned to roles that define their permitted operations, implementing RBAC principles at the database level.

However, database-level RBAC implementations typically focus on controlling access to database objects rather than providing application-level access control that considers domain-specific entities and operations. For complex applications like automotive parameter management, database-level security must be complemented with application-level access control logic that maps domain-specific concepts to database operations while maintaining the security guarantees provided by the underlying database system [29].

## 2.5 Version Control and Temporal Data Concepts

Many applications require tracking how data changes over time, maintaining historical states, and supporting queries based on temporal relationships. This section examines the fundamental concepts that underpin version control and temporal data management in database systems.

### 2.5.1 Version Control Fundamentals

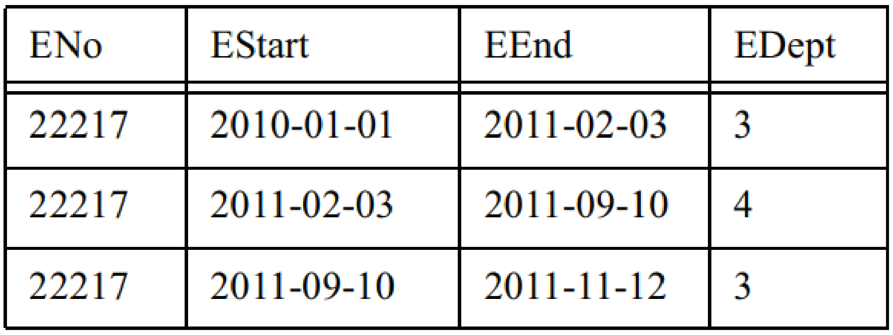
Version control for database content addresses the challenge of tracking changes to structured data with complex relationships and constraints. Unlike traditional source code version control systems that manage text files, database version control must maintain referential integrity while preserving historical states and supporting evolution through distinct development stages [37].

Several fundamental approaches exist for implementing version control in database systems. The snapshot approach captures complete database states at specific points in time, providing straightforward retrieval of historical states but potentially consuming significant storage resources. The delta-based approach records only modifications to database content, reducing storage requirements but requiring reconstruction of historical states through the application of change records. The temporal approach extends traditional database structures with time dimensions, enabling direct querying of historical states through time-based predicates [2].

The selection of an appropriate version control approach depends on specific application requirements, particularly the frequency of historical data access, the complexity of change patterns, and the performance requirements for both current and historical data operations. Bhattacherjee et al. note that domain-specific versioning approaches often provide better performance and usability than generic temporal database techniques when tailored to specific application requirements [2].

### 2.5.2 Temporal Database Concepts

Temporal database concepts provide theoretical foundations for managing time-varying data in database systems. Unlike traditional databases that store only current states, temporal databases maintain historical information and support queries based on time relationships [33].



| ENo | EStart | EEnd | EDept |
|---|---|---|---|
| 22217 | 2010-01-01 | 2011-02-03 | 3 |
| 22217 | 2011-02-03 | 2011-09-10 | 4 |
| 22217 | 2011-09-10 | 2011-11-12 | 3 |

Figure 2.6: Temporal Table Tracking Employee Department History [22]

Temporal databases typically support two fundamental time dimensions. Valid time represents when facts are true in the modeled reality, independent of when they are recorded in the database. Transaction time represents when facts are recorded in the database system, independent of when they become true in reality. Databases that support both dimensions simultaneously are known as bi-temporal databases, providing comprehensive temporal functionality for applications requiring both historical accuracy and change auditability [22].

Figure 2.6 illustrates a practical implementation of temporal concepts, showing how historical information can be maintained through time-based table structures. The example demonstrates how temporal tables can track the complete history of entity state changes without overwriting previous records, enabling historical queries and analysis.

Temporal database implementations often employ specialized table structures that extend traditional schemas with timestamp columns defining validity periods for each record. This approach enables systematic tracking and querying of historical data without requiring application-level version management, providing a foundation for maintaining comprehensive audit trails and supporting historical analysis requirements [26].

# 3 State of the Art

This chapter examines the current state of the art in database version control systems and automotive parameter management. It begins by analyzing existing approaches to software configuration management in the automotive industry, followed by an evaluation of database versioning techniques and their applicability to parameter management systems. The chapter also explores role-based access control models and integration strategies for enterprise systems, establishing the theoretical foundation for the VMAP system design.

## 3.1 Parameter Management in Automotive Software Development

The complexity of automotive software has grown exponentially in recent decades, with modern vehicles containing up to 100 million lines of code distributed across dozens of ECUs [25]. This growth has significantly increased the importance and complexity of parameter management in automotive development.

### 3.1.1 Evolution of Automotive Parameter Management

Parameter management in automotive systems has evolved from simple calibration tables to sophisticated configuration frameworks managing thousands of parameters across multiple vehicle variants. Broy [7] describes the fundamental challenges in automotive software engineering, highlighting that software complexity is driven by the need to address multiple variants, market requirements, and technical functions. The parameter configuration problem is specifically identified as one of the key challenges in this domain.

Early approaches to parameter management relied on specialized tools provided by ECU suppliers, which typically stored parameters in proprietary formats with limited version control capabilities. Pretschner et al. [25] note that these tools evolved from simple memory editors to more sophisticated calibration environments, but remained focused on individual ECUs rather than system-wide parameter management.

Staron [34] describes how AUTomotive Open System ARchitecture (AUTOSAR) has contributed to more structured parameter management by defining standard interfaces and component models that separate parameters from implementation. However, the practical implementation of these standards varies across organizations and ECU suppliers, creating integration challenges for comprehensive parameter management.

### 3.1.2 Challenges in Automotive Parameter Management

The management of parameters in automotive software development presents specific challenges that distinguish it from general software configuration management. Pretschner et al. [25] identify several key challenges related to variability management in automotive software, including the need to maintain multiple parameter configurations for different vehicle variants, markets, and operating conditions.

Broy [7] emphasizes the challenge of managing interdependencies between parameters, noting that changes to one parameter often require coordinated changes to related parameters to maintain system consistency. This creates a need for sophisticated dependency tracking mechanisms that go beyond traditional version control systems.

Another significant challenge relates to validation requirements for parameter changes. Unlike source code, which can be validated through compilation and static analysis, parameters require functional testing to verify their correctness. Pretschner et al. [25] describe how this validation often involves specialized hardware-in-the-loop or vehicle-level testing, creating a significant gap between parameter modification and validation.

Kiencke and Nielsen [19] discuss the specific challenges related to powertrain control parameters, noting the complex interactions between engine control parameters and their effects on vehicle performance, emissions, and fuel economy. These interactions create a need for sophisticated parameter testing and validation processes beyond simple version control.

### 3.1.3 Current Approaches and Tools

Current parameter management solutions in the automotive industry span a spectrum from general-purpose tools to specialized automotive calibration systems.

AUTOSAR is a global partnership of automotive manufacturers, suppliers, and technology companies that defines a standardized software architecture for automotive electronic control units (ECUs). Established in 2003, AUTOSAR aims to create an open industry standard for automotive E/E (electrical/electronic) architecture to manage the growing complexity of automotive software development. Staron [34] discusses how AUTOSAR tools provide standardized interfaces for parameter management in modern automotive systems, but notes that these tools focus primarily on the technical aspects of parameter definition rather than the organizational processes of parameter development and validation through multiple development phases.

Broy [7] identifies the challenges of integrating parameter management into broader software development processes, noting that many organizations maintain separate workflows for software development and parameter calibration. This separation creates coordination challenges, particularly when parameter changes affect multiple software components or require software modifications.

Pretschner et al. [25] discuss how model-based development approaches are increasingly used in automotive development, with parameters linked to model elements to provide traceability and support automated validation. However, they note that the integration between parameter management tools and modeling environments remains incomplete in many organizations.

For database-oriented approaches to parameter management, Bhattacherjee et al. [2] provide a theoretical foundation by examining the principles of dataset versioning. They describe the fundamental trade-offs between storage efficiency and reconstruction performance, which are particularly relevant for systems that must maintain multiple parameter configurations across different development phases.

## 3.2 Database Version Control Systems

Version control for database content presents distinct challenges compared to traditional source code version control. While source code version control focuses on tracking changes to text files, database version control must address structured data with complex relationships and constraints [2]. This section examines current approaches to database version control and their applicability to automotive parameter management.

### 3.2.1 Traditional Database Versioning Approaches

Traditional approaches to database versioning fall into several categories, each addressing different aspects of the versioning challenge. Schema evolution tools focus on tracking and managing changes to database structure through migration scripts or schema manipulation languages. Curino et al. [11] describe an approach for automating database schema evolution in information system upgrades, focusing on maintaining data integrity during schema transitions.

Bhattacherjee et al. [2] provide a comprehensive analysis of dataset versioning approaches, identifying a fundamental trade-off between storage and recreation costs. They categorize versioning strategies into several approaches:

1. Version-first approaches maintain complete snapshots of datasets at specific version points, providing simple retrieval of historical states but requiring substantial storage space.

2. Delta-based approaches store only the changes between versions, reducing storage requirements but increasing the computational cost of reconstructing historical states.

3. Hybrid approaches combine elements of both strategies, typically storing periodic full snapshots with incremental deltas between snapshots.

The authors note that the optimal strategy depends on specific usage patterns, particularly the ratio between storage costs and the frequency and complexity of historical data access operations.

Mueller and Müller [24] describe a practical implementation of database versioning between research institutes, highlighting the challenges of maintaining consistency across systems with different update cycles. Their approach uses a combination of schema versioning and data synchronization mechanisms to maintain consistency while supporting independent evolution.

### 3.2.2 Temporal Database Approaches

Temporal database approaches provide a theoretical foundation for managing time-varying data in database systems. Kulkarni and Michels [22] describe the temporal features introduced in SQL:2011, which formalized support for period data types and temporal tables in the Structured Query Language (SQL) standard. These features enable tracking of both valid time (business time) and transaction time (system time) dimensions, supporting bi-temporal data management.

The valid time dimension represents when facts are true in the modeled reality, independent of when they are recorded in the database. This dimension supports business-oriented temporal queries such as "What was the value of this parameter in a specific phase?" or "When did this parameter change from value A to value B?" [4]. The transaction time dimension represents when facts are recorded in the database, supporting auditability through questions like "Who changed this parameter, and when did they change it?" [22].

Bi-temporal databases combine both dimensions, providing a comprehensive framework for tracking both when changes occurred in the system and when they became effective in the real world [4]. This approach is particularly valuable for regulated industries like automotive development, where both historical accuracy and change auditability are essential for compliance and quality assurance.

Snodgrass [33] provides a comprehensive guide to developing time-oriented database applications in SQL, describing practical techniques for implementing temporal functionality in relational database systems. The author presents various approaches to tracking historical data, including transaction-time tables, valid-time tables, and bi-temporal tables, with practical implementation guidance for each approach.

Biriukov [3] examines practical implementation aspects of bi-temporal databases, highlighting the challenges of schema design, query formulation, and performance optimization. The author notes that domain-specific temporal approaches often provide more practical solutions than generic bi-temporal frameworks, particularly for applications with specialized temporal requirements.

### 3.2.3 Version Control for Parameter Management

Version control for automotive parameter management presents specific requirements that differ from general database versioning needs. Drawing from the literature, several key requirements can be identified:

Broy [7] discusses the need for version control approaches that align with automotive development processes, which typically follow a structured progression through predefined development phases. Unlike source code versioning, which often follows continuous development with arbitrary version points, parameter versioning must support specific phase-based workflows.

Bhattacherjee et al. [2] examine the trade-offs between different versioning strategies, which are particularly relevant for parameter management systems that must maintain multiple configurations across different development phases. The authors' analysis of storage versus reconstruction costs provides a theoretical foundation for designing efficient parameter versioning systems.

Snodgrass [33] describes techniques for tracking valid-time information in database systems, which aligns with the need to maintain parameter configurations that are valid for specific development phases or vehicle configurations. However, the author's focus on general temporal database approaches does not address the specific requirements of phase-based development.

Bhattacherjee et al. [2] note that domain-specific versioning systems often provide more effective solutions than generic versioning frameworks, particularly for domains with structured development processes and complex entity relationships. This observation supports the development of specialized versioning approaches tailored to automotive parameter management rather than adopting generic temporal database techniques.

## 3.3 Role-Based Access Control in Enterprise Systems

RBAC has become a dominant paradigm for managing access rights in enterprise systems, providing a structured approach to security management that aligns with organizational responsibilities [28]. For automotive parameter management, where different user roles have distinct responsibilities and access requirements, RBAC provides a foundation for implementing appropriate security controls.

### 3.3.1 RBAC Model and Extensions

The core RBAC model, as defined by Sandhu et al. [28], consists of users, roles, permissions, and sessions. Users are assigned to roles that correspond to job functions, and roles are granted permissions that authorize specific operations on protected resources. This indirect association between users and permissions through roles simplifies security administration while maintaining the principle of least privilege.

Several extensions to the basic RBAC model have been developed to address more complex security requirements. Sandhu et al. [28] describe hierarchical RBAC, which introduces role hierarchies that enable permission inheritance between roles, supporting organizational structures with senior roles inheriting permissions from junior roles.

Sandhu and Bhamidipati [27] present administrative RBAC (ARBAC), which addresses the management of the RBAC system itself, defining who can assign users to roles and modify role permissions. This extension is particularly relevant for enterprise systems where role and permission management is distributed across different administrative domains.

Ferraiolo et al. [12] describe policy-enhanced RBAC, which combines role-based permissions with attribute-based policies to provide context-sensitive access control.

This hybrid approach is particularly valuable for systems where access decisions depend on both user roles and context-specific factors such as time, location, or resource attributes.

### 3.3.2 RBAC in Database Systems

Modern database management systems provide varying levels of support for RBAC principles. Sciore [29] describe the evolution of database security mechanisms from simple user-based privileges to more sophisticated role-based models. Most enterprise database systems now include native support for roles, user-role assignments, and permission management through SQL statements like GRANT and REVOKE.

Shaik [31] detail PostgreSQL's implementation of RBAC concepts, including role hierarchies through role inheritance, permission management through fine-grained privileges, and row-level security policies for content-based access control. These capabilities provide a foundation for implementing domain-specific access control models on top of the database system's native security features.

However, database-level RBAC implementations typically focus on controlling access to database objects like tables, views, and functions, rather than providing application-level access control that considers domain-specific entities and operations. For complex applications like automotive parameter management, database-level RBAC must be complemented with application-level access control logic that maps domain-specific concepts to database operations [12].

### 3.3.3 Access Control for Parameter Management

Access control for automotive parameter management presents specific requirements that extend beyond basic RBAC models. Drawing from the literature, several key access control requirements can be identified:

Sandhu et al. [28] provide the theoretical foundation for role-based access control, which aligns with the organizational structure of automotive development teams. Different roles such as parameter engineers, module developers, and system integrators require different access rights to parameter data.

Ferraiolo et al. [12] describe policy-enhanced RBAC, which combines role-based permissions with attribute-based policies. This hybrid approach is particularly relevant for parameter management, where access rights may depend on both user roles and attributes of the parameters being accessed, such as their development phase or module assignment.

Hu et al. [16] discuss practical aspects of implementing and managing policy rules in attribute-based access control, providing insights into the challenges of combining role-based and attribute-based approaches. Their work highlights the importance of balancing security requirements with usability considerations, which is particularly relevant for parameter management systems used by diverse stakeholder groups.

Sandhu and Bhamidipati [27] address the administrative aspects of RBAC, which are important for parameter management systems where access control administration may be distributed across different organizational units. Their ARBAC97 model provides a framework for delegating administrative responsibilities while maintaining central governance.

## 3.4 Database Integration with Enterprise Systems

Integration between database systems and enterprise applications presents significant challenges in automotive development environments, where parameter management must interact with numerous other systems across the development lifecycle. Effective integration strategies must address both technical interoperability and semantic consistency while maintaining performance and security [15].

### 3.4.1 Enterprise Integration Patterns

Enterprise integration patterns, as documented by Hohpe and Woolf [15], provide a catalog of solutions for common integration challenges. These patterns address various aspects of system integration, including messaging styles, messaging channels, message construction, and message transformation.

For database-centric applications like parameter management systems, several integration patterns are particularly relevant. Fowler [13] describes the Repository pattern, which provides a structured approach to data access, abstracting the database implementation details behind a domain-focused interface. This abstraction simplifies integration by providing a stable API for other systems to interact with the parameter repository.

Fowler [13] also documents the Data Transfer Object (DTO) pattern, which addresses the challenge of transferring data between systems with different data models. By defining specialized objects for inter-system communication, this pattern enables consistent data exchange while isolating each system's internal representation.

The Canonical Data Model pattern, as described by Hohpe and Woolf [15], establishes a common data representation across multiple systems, simplifying data transformation and ensuring consistent interpretation. This pattern is particularly valuable for parameter management, where the same parameter concepts may be represented differently in various systems across the development lifecycle.

### 3.4.2 Database Synchronization Approaches

Database synchronization presents specific challenges when integrating parameter management systems with other enterprise data sources. Mueller and Müller [24] describe approaches to database versioning and synchronization between research institutes, highlighting the challenges of maintaining consistency across systems with different update cycles.

Bhattacherjee et al. [2] discuss the principles of dataset versioning, which are relevant for synchronization between parameter management systems and other enterprise databases. Their analysis of the trade-offs between storage and recreation costs provides insights into designing efficient synchronization mechanisms that minimize both data transfer volumes and processing overhead.

Seenivasan and Vaithianathan [30] examine change data capture (CDC) techniques, which enable incremental synchronization by identifying and propagating only changed data between systems. These techniques reduce synchronization overhead compared to full dataset transfers but require reliable change detection mechanisms and careful handling of interdependent changes.

Kleppmann and Beresford [20] address the challenges of conflict resolution in distributed data systems, which are relevant for parameter management systems that must synchronize with multiple enterprise data sources. Their work on conflict-free replicated data types provides theoretical foundations for designing synchronization mechanisms that maintain consistency across distributed systems.

## 3.5 Summary and Research Gaps

The review of existing literature reveals several research gaps in the domain of database systems for automotive parameter management:

Current database versioning approaches, as described by Bhattacherjee et al. [2] and Snodgrass [33], provide general frameworks for managing time-varying data but do not specifically address the phase-based development processes common in automotive parameter management. There is a need for specialized versioning approaches that align directly with automotive development workflows while providing the traceability and auditability required for regulatory compliance.

The RBAC models described by Sandhu et al. [28] and Ferraiolo et al. [12] provide a foundation for access control but require extensions to address the specific requirements of parameter management, where access rights depend on both organizational roles and parameter-specific attributes such as module assignment and development phase.

Integration approaches documented by Hohpe and Woolf [15] and Mueller and Müller [24] provide general patterns for system integration but do not specifically address the challenges of integrating parameter management systems with automotive-specific enterprise systems such as parameter definition databases and vehicle configuration databases.

These research gaps highlight the need for domain-specific solutions that combine insights from database version control, access control models, and enterprise integration patterns with specialized knowledge of automotive development processes. The VMAP system addresses these gaps by developing a database architecture tailored to the specific requirements of automotive parameter management, as will be detailed in subsequent chapters.

# 4 Methodology and Concept Development

This chapter presents the systematic approach taken in designing the VMAP system. The methodology follows established software engineering principles to address the complex requirements of automotive parameter management. Beginning with a requirements analysis, the chapter proceeds to detail the conceptual architecture design, data model, validation mechanisms, and integration approaches developed to ensure system robustness and compatibility with existing enterprise infrastructure.

## 4.1 Requirements Analysis

The foundation of the VMAP system design was a comprehensive requirements analysis conducted through structured interviews with stakeholders, detailed examination of the existing Excel-based process, and workshops with domain experts. This multifaceted approach, following Hull's framework for requirements engineering, ensured that both functional and non-functional requirements would be thoroughly identified and prioritized [17].

### 4.1.1 Functional Requirements

The primary functional requirements were derived from direct observation of engineers' current Excel-based workflow combined with semi-structured interviews conducted with module developers and documentation specialists. The system must support the hierarchical organization of parameters within ECUs, Modules, and PIDs, mirroring the domain-specific structure of automotive electronic systems as described by Staron [34]. This hierarchical organization is essential for maintaining the logical structure of vehicle parameters and aligning with established engineering practices.

Users must be able to create variants for parameters with specific code rules determining their applicability, and define segments representing modified parameter values. If no segment exists, the system must default to Parameter Definition Database values—an approach that allows efficient storage by tracking only modifications rather than duplicating unchanged parameters, aligning with Bhattacherjee's principles of dataset versioning [2].

The system must track parameter values across four distinct release phases: Phase1, Phase2, Phase3, and Phase4, with changes in earlier phases propagating to later phases unless explicitly overridden. This phase-based approach represents a domain-specific adaptation particularly suited to automotive software development cycles as identified in Broy's research on automotive software engineering challenges [7].

All modifications require comprehensive logging with user information, timestamp, and detailed change data, supporting regulatory compliance and enabling parameter evolution tracking. The system must also provide functionality to create parameter configuration snapshots at specific points, particularly at phase transitions, for documentation purposes—a capability identified as essential for quality assurance and regulatory compliance in automotive software development by Staron [34].

### 4.1.2 Integration with External Systems

The stakeholder interviews and process analysis revealed that VMAP must integrate with two critical external enterprise systems: the Parameter Definition Database (PDD) and the Vehicle Configuration Database (VCD). The PDD serves as the authoritative source for the hierarchical structure of automotive electronic systems, containing definitions of ECUs, Modules, PIDs, and baseline parameter configurations. As noted by Pretschner et al. [25], maintaining this hierarchical structure is essential for automotive software development.

The VCD contains comprehensive vehicle specifications and configuration codes that determine which parameter variants apply to specific vehicle configurations. Integration with this system is necessary for validating the boolean code rules associated with parameter variants and supporting parameter file generation for specific vehicle configurations by resolving the applicable parameter variants based on vehicle codes. This integration requirement aligns with Staron's analysis of automotive software architectures, which emphasizes the importance of configuration management in supporting variant-rich vehicle platforms [34].

### 4.1.3 User Role Requirements

A systematic analysis of the current Excel-based workflow, coupled with contextual inquiries with engineering teams, identified four distinct user roles with specific access requirements. Module developers require write access to parameters within their assigned modules, with the ability to create and modify variants and segments. Documentation specialists need access to frozen data for documentation, comparison capabilities between phases, and comprehensive change history access. System administrators require comprehensive control over user management, release phases, and special operations like variant deletion and phase freezing. Read-only users need view access to all parameter data with parameter file generation capabilities but no modification rights.

These roles were defined based on the principle of least privilege as described by Sandhu [28], ensuring users have access only to functionality required for their specific responsibilities while enhancing system security and simplifying the user experience.

### 4.1.4 Data Management Requirements

The system must maintain distinct parameter versions across different release phases, allowing simultaneous work on multiple phases while enabling access to parameter values from any point in the development lifecycle. As highlighted by Sciore [29], data integrity requires maintaining referential integrity across all related entities, particularly ensuring variants and segments associate with valid parameters.

Multi-dimensional parameter support is essential for complex automotive parameters such as mapping tables. Operations modifying multiple related entities must function as atomic transactions to maintain data consistency—particularly important for phase transitions where numerous parameters, variants, and segments may change simultaneously, a requirement that aligns with Bhattacherjee's research on dataset versioning approaches [2].

Query performance analysis, based on projected usage patterns from the current Excel-based process, identified critical query paths including parameter retrieval by ECU, module, PID, release phase, and parameter name. These requirements influenced schema design decisions regarding normalization and indexing strategies to optimize common query patterns.

## 4.2 Use Case Modeling

Following the requirements gathering process, use case modeling was employed to formalize the system's functional requirements from a user perspective. This approach, as described by Jacobson [18], provides a structured way to represent the system's capabilities and the interactions between users and the system.

The use case diagram in Figure 4.1 illustrates the primary actors and their interactions with the VMAP system. Four primary actor types are identified, corresponding to the user roles established during requirements analysis: Module Developers, who

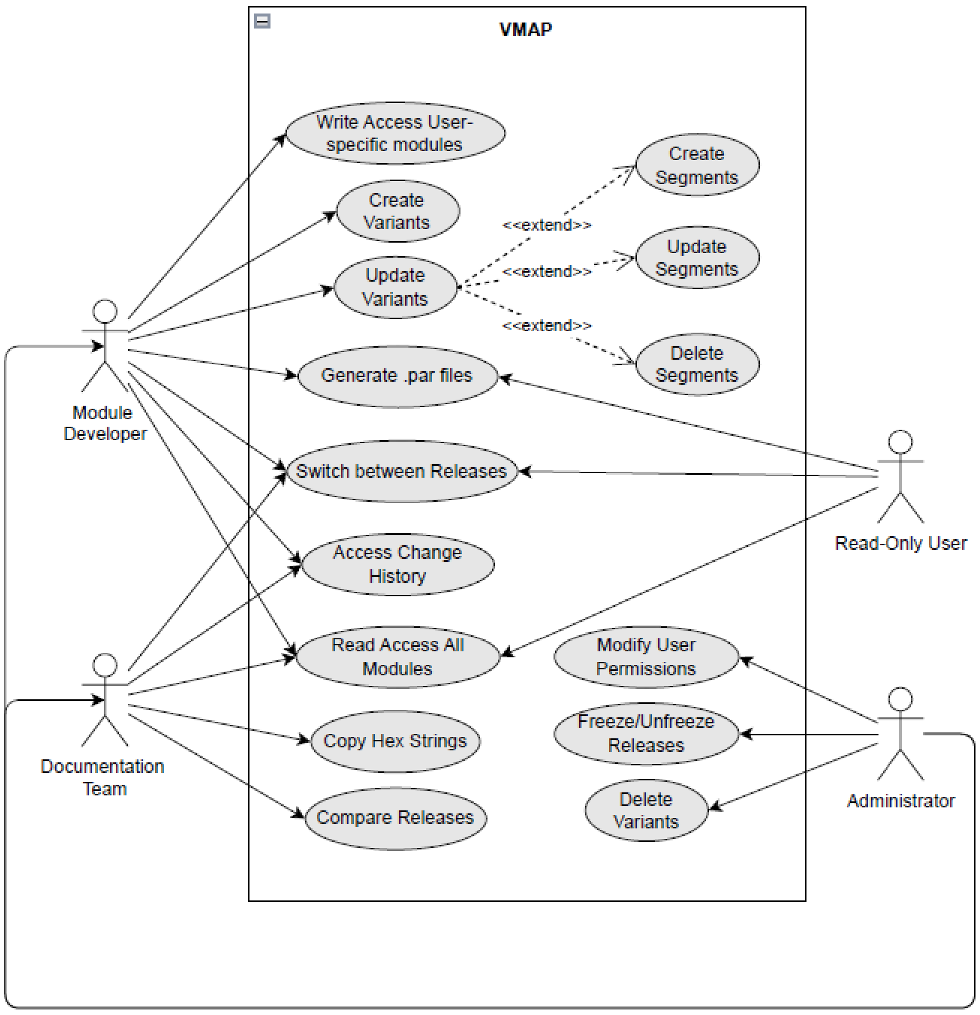


Figure 4.1: VMAP System Use Case Diagram

create and modify parameter variants; Documentation Team members, who access parameter data for documentation purposes; Administrators, who manage system settings and user access; and Read-Only Users, who view parameter data without making modifications.

This use case model provides a clear visual representation of the system's scope and functionality, serving as a bridge between user requirements and technical implementation. By mapping user roles to specific system functions, the model ensures that the database design will support all required user interactions while maintaining appropriate access controls.

## 4.3 User Management Approaches

Based on the identified user role requirements, two distinct approaches to user management were considered for the VMAP system: a traditional role-based approach and a hybrid role-permission approach. Each approach offers different advantages in terms of flexibility, administrative complexity, and alignment with organizational needs.

### 4.3.1 Traditional Role-Based Approach

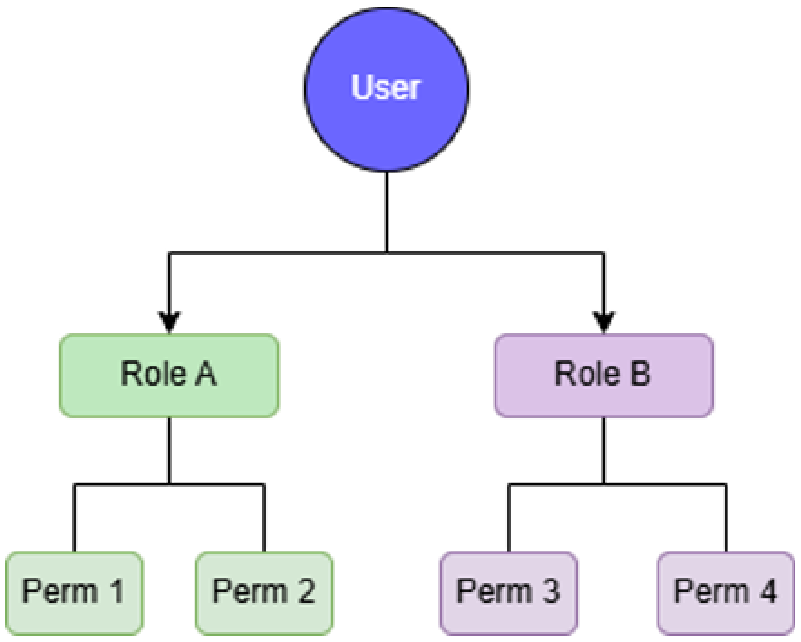


Figure 4.2: Traditional Role-Based Access Control Approach

The traditional role-based approach, illustrated in Figure 4.2, assigns users to predefined roles that contain fixed sets of permissions, following the classic RBAC model described by Sandhu [28]. In this approach, each user is assigned one or more roles, and all permissions are granted through these role assignments without individual permission adjustments.

This approach offers administrative simplicity, as user management involves only assigning appropriate roles rather than configuring individual permissions. The role structure also provides clear organizational alignment, with roles directly corresponding to job functions within the development process. A key limitation is its reduced flexibility for accommodating exceptions or specialized access requirements, potentially compromising the principle of least privilege [28].

### 4.3.2 Hybrid Role-Permission Approach

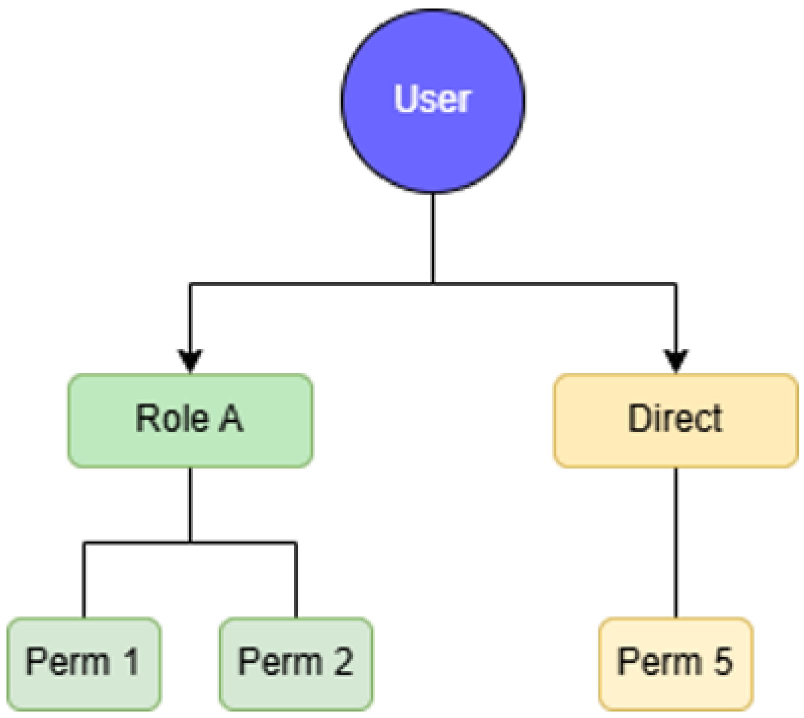


Figure 4.3: Hybrid Role-Permission Access Control Approach

The hybrid approach, illustrated in Figure 4.3, combines role-based permissions with direct permission assignments, similar to the model described by Ferraiolo et al. [12]. In this approach, users are assigned to primary roles defining their core permissions, but additional permissions can be granted on a per-user basis to address exceptional cases or specialized responsibilities.

This approach offers greater flexibility for accommodating exceptions without creating specialized roles, essential in environments where organizational structures evolve over time. It provides more granular permission control, allowing precise tailoring of access rights to individual responsibilities. However, this flexibility comes at the cost of increased administrative complexity. The hybrid approach is particularly valuable in the automotive parameter management context, where development responsibilities can vary between projects and temporary access adjustments may be needed for specific tasks or during transition periods.

## 4.4 Parameter Synchronization Approaches

Integration with the PDD represents a critical aspect of the VMAP system, requiring careful consideration of synchronization approaches. Two different conceptual approaches were explored for maintaining parameter data across the release phases: the change-based approach and the phase-based approach.

### 4.4.1 Change-Based Synchronization Approach

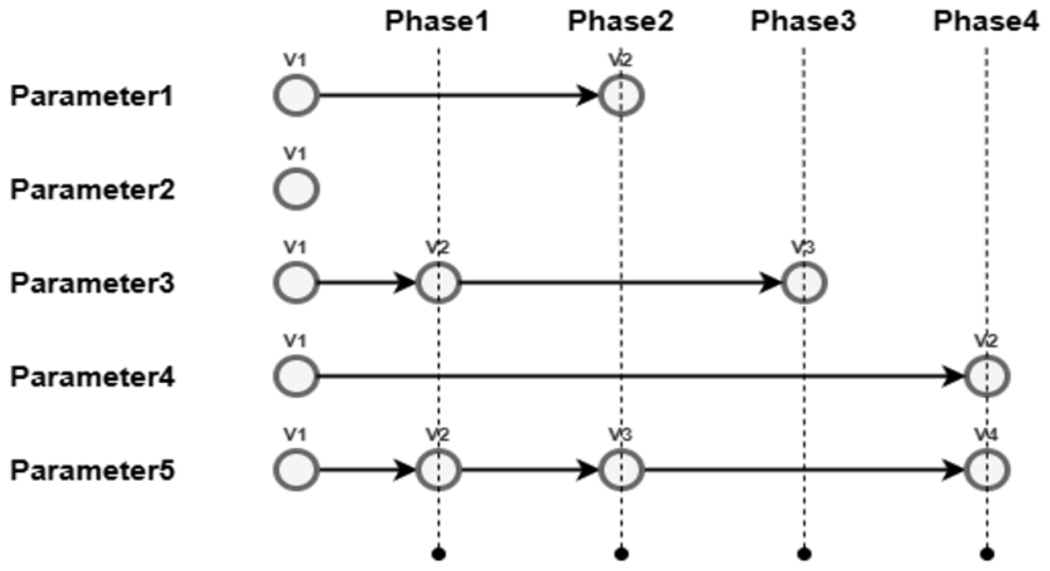


Figure 4.4: Change-Based Parameter Synchronization Approach

The change-based approach, illustrated in Figure 4.4, maintains parameter values by recording changes between phases rather than storing complete parameter sets for each phase. Parameters are initially created before the first phase, and subsequent modifications are recorded as change entries associated with specific phases.

This approach is conceptually aligned with traditional version control systems as described by Bhattacherjee et al. [2], where efficiency is achieved by storing only the differences between versions rather than complete copies. While potentially offering storage efficiency advantages, this approach introduces conceptual complexity for retrieving parameter values in a specific phase, requiring reconstruction of parameter states from change history.

### 4.4.2 Phase-Based Synchronization Approach

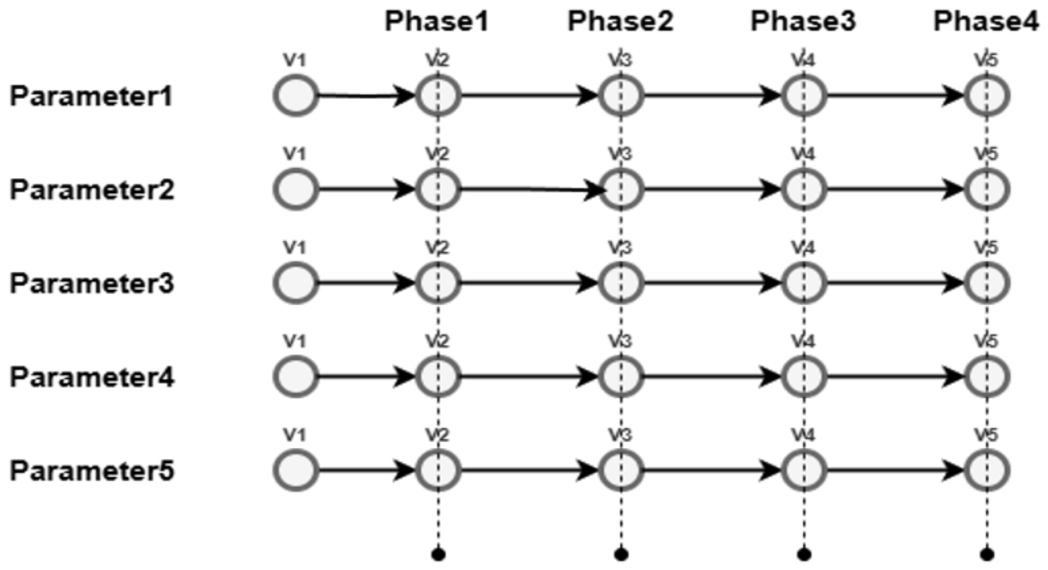


Figure 4.5: Phase-Based Parameter Synchronization Approach

The phase-based approach, illustrated in Figure 4.5, maintains complete parameter sets for each phase independently. When parameters are initially created before Phase1, each parameter has its specific version. When transitioning to a new phase, all parameters are copied forward, even if they haven't changed.

This approach aligns more directly with the phase-oriented structure of automotive development described by Broy [7], where distinct development milestones form the primary organizational principle. The approach simplifies conceptual understanding and parameter retrieval, as parameters for a specific phase can be accessed directly without reconstructing their values from change history.

The phase-based approach also simplifies phase inheritance by copying parameter configurations forward during phase transitions, allowing subsequent modifications in each phase without affecting previous phases. While requiring increased storage due to parameter duplication across phases, this trade-off provides benefits in terms of conceptual clarity, query simplicity, and alignment with the automotive development workflow.

## 4.5 Database System Considerations

Selecting an appropriate database management system for VMAP required consideration of different options against the specific requirements of automotive parameter management. The requirements analysis identified several critical database capabilities needed for effective parameter management: support for complex data types including arrays for multi-dimensional parameters, robust transaction support for maintaining data consistency, advanced indexing capabilities for optimizing common query patterns, extensibility for implementing domain-specific operations, comprehensive access control mechanisms, and efficient storage and retrieval of historical data for audit and traceability purposes.

Table 4.1: Comparison of Database Systems for Automotive Parameter Management

| **Feature** | **PostgreSQL** | **Oracle** | **SQL Server** | **MySQL** |
|---|---|---|---|---|
| **Complex Data Types** | Excellent support for arrays, JSON, custom types [31] | Good support, additional licensing for advanced features [1] | Limited built-in support, extensions required [40] | Limited support, improved in recent versions [5] |
| **Transaction Support** | Comprehensive with serializable isolation [31] | Excellent with advanced options [1] | Robust support with multiple isolation levels [40] | Limited in some storage engines [21] |
| **Indexing Capabilities** | Diverse index types including GIN for text search [31] | Advanced indexing with optimizer hints [1] | Solid capabilities with columnstore indexes [40] | Basic indexing with some limitations [21] |
| **Extensibility** | Highly extensible with custom types and functions [31] | Extensible with proprietary mechanisms [1] | Extensible through .NET integration [40] | Limited extensibility [5] |
| **Licensing** | Open source, PostgreSQL License [31] | Commercial, complex licensing model [1] | Commercial with edition-based pricing [40] | Dual licensing: GPL and commercial [5] |

Based on the comparative analysis presented in Table 4.1, PostgreSQL was selected as the database platform for the VMAP implementation. This decision was driven by PostgreSQL's native support for arrays, JSON/JSONB, and custom data types, which provides essential capabilities for representing multi-dimensional parameters and complex variant structures [31]. PostgreSQL's diverse indexing capabilities, including GIN indexes for full-text search and partial indexes for conditional indexing, provide optimal support for the complex query patterns common in parameter management. The open-source nature of PostgreSQL eliminates licensing costs while providing enterprise-grade capabilities, which was particularly important for the research context of this thesis.

## 4.6 Entity-Relationship Model

Based on the requirements analysis and architectural considerations, a comprehensive ER model was developed to capture the complex relationships between system entities. This model follows the approach described by Chen [8], providing a conceptual foundation for the database implementation.

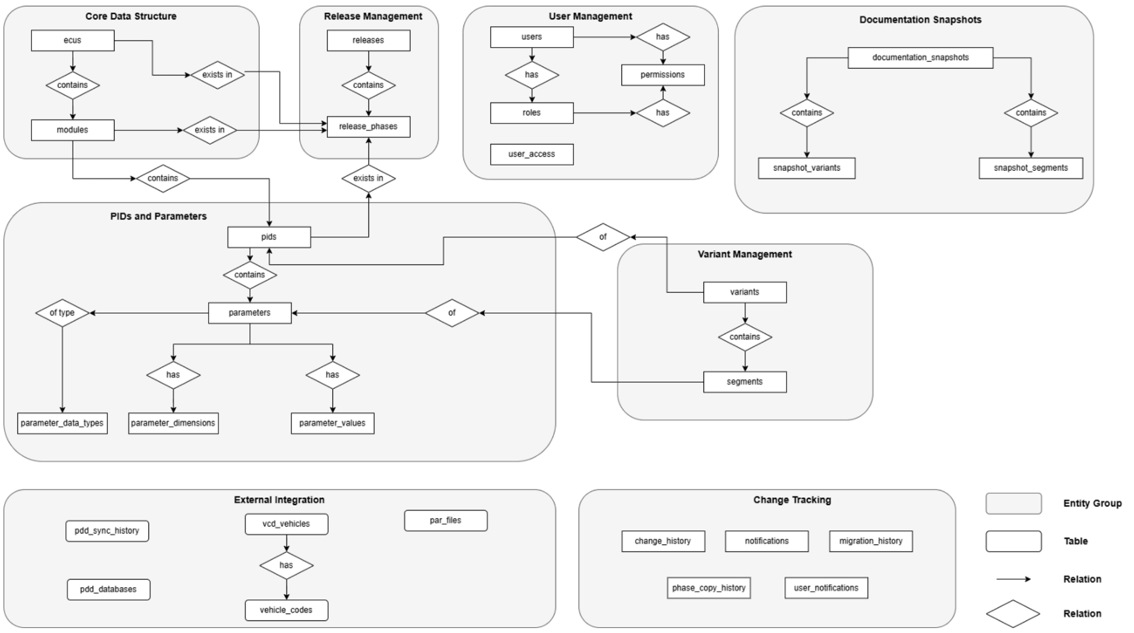


Figure 4.6: Comprehensive Entity-Relationship Diagram for the VMAP System

Figure 4.6 illustrates the complete entity-relationship model for the VMAP system, showing the logical organization of entities into functional groups and the relationships between them. The diagram captures the hierarchical structure of automotive parameter data, the phase-based versioning approach, the role-permission access control model, and the mechanisms for external system integration and change tracking.

### 4.6.1 Core Data Entities

The ER model includes several categories of entities representing different aspects of the system. User management entities include Users, Roles, Permissions, and their relationships, implementing a role-permission model for access control. Release management entities encompass Releases, Release Phases, and ECU Phase mappings, providing the foundation for the phase-based version control approach. Parameter structure entities include ECUs, Modules, PIDs, Parameters, and Parameter Dimensions, representing the hierarchical organization of vehicle electronic systems as described by Staron [34].

Variant management entities comprise Variants, Segments, and their relationships to parameters, implementing the core parameter customization functionality. Documentation entities include Documentation Snapshots, Snapshot Variants, and Snapshot Segments, supporting the preservation of historical parameter states for documentation and regulatory compliance. Integration entities consist of Synchronization Records, Vehicle Configurations, and Parameter File Records, supporting connectivity with external systems. Audit entities encompass Change History, Transaction Records, and Phase Copy History, providing comprehensive traceability for all significant operations within the system.

### 4.6.2 Relationship Structure

The relationships between entities in the ER model reflect the complex interactions between different aspects of automotive parameter management. Key relationships include hierarchical relationships between ECUs, Modules, PIDs, and Parameters, representing the structural organization of automotive electronic systems. Many-to-many relationships between parameters and phases are implemented through direct association rather than temporal versioning, supporting the phase-based versioning approach and allowing efficient retrieval of parameters for specific phases.

Complex relationships between variants, parameters, and segments capture the parameter customization process, ensuring that segments are associated with valid variants and parameters while supporting efficient resolution of effective parameter values based on vehicle configuration. Temporal relationships for audit and history entities capture the evolution of parameter configurations over time, supporting both regulatory compliance and diagnostic capabilities.

The ER model was developed using data normalization principles to minimize redundancy while maintaining data integrity, following the approach described by Codd [10].

The model generally adheres to Third Normal Form (3NF), ensuring that non-key attributes depend on the primary key rather than on other non-key attributes. Strategic denormalization was considered in specific areas to optimize performance for common operations, following the principles described by Churcher [9].

## 4.7 Validation Mechanisms

To ensure data integrity and consistency, multiple validation mechanism layers were conceptualized for the VMAP system, from basic constraints to sophisticated business rule validation. These mechanisms work together to maintain parameter data quality and reliability throughout the system lifecycle.

Database-level constraints form the foundation for enforcing basic integrity rules, following the principles described by Sciore [29]. These constraints include primary key constraints ensuring unique entity identifiers, foreign key constraints maintaining referential integrity between related entities, not-null constraints ensuring required fields contain values, unique constraints preventing duplicate values in specified columns, and check constraints enforcing domain-specific rules such as valid date ranges and parameter value ranges.

Domain-specific business rules are implemented through database triggers and stored procedures, providing a second layer of validation beyond basic constraints. These rules include parameter range validation automatically checking modified values against defined minimum and maximum bounds, phase status validation preventing modifications to frozen phases, segment validation ensuring segments reference valid parameters and variants, and user access validation ensuring users can only modify parameters, variants, and segments for modules to which they have been granted access.

Comprehensive audit and traceability mechanisms are essential for regulatory compliance and quality assurance in automotive parameter management. The core of this capability is the change history tracking mechanism, which automatically captures both before and after states for entity modifications. For each change, the system records the entity being modified, the type of change, the user making the change, the timestamp, and detailed before/after values. To optimize performance, selective filtering of change data excludes non-essential fields such as timestamps and large binary data, while asynchronous audit recording for bulk operations reduces the performance impact on high-volume operations while ensuring that all changes are eventually recorded.

# 5 Implementation

This chapter presents the technical implementation of the VMAP system database architecture described in Chapter 4. The implementation transforms the conceptual design into a functional PostgreSQL database system, focusing on key architectural decisions and novel technical contributions that enable efficient parameter versioning in automotive software development.

## 5.1 Database Architecture Overview

The VMAP database implementation leverages PostgreSQL's advanced features to address the complex requirements of automotive parameter management. The architecture consists of five primary functional domains that work together to provide comprehensive parameter management capabilities. The core parameter hierarchy domain manages the structural organization of ECUs, modules, PIDs, and parameters, implementing the automotive electronic system structure through normalized relational tables. The phase-based version control domain implements the novel versioning approach aligned with automotive development cycles, using explicit phase relationships rather than generic temporal mechanisms. The variant management domain handles parameter customization through variants and segments, enabling efficient storage of parameter modifications without duplicating unchanged values. The hybrid access control domain combines traditional role-based security with module-specific permissions to accommodate complex organizational structures. Finally, the comprehensive audit system tracks all changes with detailed provenance information while the enterprise integration domain manages synchronization with external parameter definition and vehicle configuration databases.

The implementation follows PostgreSQL best practices while incorporating domain-specific optimizations for automotive parameter management workflows. Strategic denormalization improves query performance for common access patterns, while specialized indexing strategies support both hierarchical navigation and text-based parameter searches. The database schema design emphasizes data integrity through comprehensive constraint enforcement while maintaining the flexibility needed for complex automotive development processes.

## 5.2 Core Data Structure Implementation

The automotive parameter hierarchy forms the foundation of the VMAP system, implementing the logical organization of vehicle electronic systems through carefully designed relational structures. The implementation uses a combination of foreign key relationships and strategic denormalization to optimize common access patterns while maintaining data integrity.

```sql
-- ECU and Module entities with many-to-many
  ↳relationship
CREATE TABLE ecus (
    ecu_id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    external_id INTEGER UNIQUE -- PDD reference
);

CREATE TABLE modules (
    module_id INTEGER PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    external_id INTEGER UNIQUE -- PDD reference
);

CREATE TABLE ecu_modules (
    ecu_id INTEGER REFERENCES ecus(ecu_id),
    module_id INTEGER REFERENCES modules(module_id),
    PRIMARY KEY (ecu_id, module_id)
);

-- Parameters with direct phase association (strategic
  ↳denormalization)
CREATE TABLE parameters (
    parameter_id BIGINT PRIMARY KEY,
    pid_id BIGINT REFERENCES pids(pid_id),
    ecu_id INTEGER,
    phase_id INTEGER,
    name VARCHAR(255) NOT NULL,
    type_id INTEGER REFERENCES parameter_data_types(
      ↳data_type_id),
    external_id INTEGER,
    FOREIGN KEY (ecu_id, phase_id) REFERENCES
      ↳ecu_phases(ecu_id, phase_id)
);
```

Listing 5.1: Core Parameter Hierarchy Implementation

The parameter table implements strategic denormalization by including direct references to `ecu_id` and `phase_id`. While this introduces controlled redundancy, it significantly improves query performance for phase-filtered parameter operations, which represent the dominant access pattern in the system. This design decision aligns with Churcher's guidance on strategic denormalization for performance optimization [9].

Automotive parameters often represent complex data structures such as lookup tables and characteristic curves requiring multi-dimensional support. The implementation uses a specialized dimension table to handle these requirements efficiently:

```sql
CREATE TABLE parameter_dimensions (
    dimension_id BIGINT PRIMARY KEY,
    parameter_id BIGINT REFERENCES parameters(
      ↳parameter_id),
    dimension_index INTEGER NOT NULL,
    default_value NUMERIC NOT NULL,
    UNIQUE (parameter_id, dimension_index)
);
```

Listing 5.2: Multi-Dimensional Parameter Structure

This structure supports parameters ranging from scalar values with a single dimension to complex three-dimensional arrays while maintaining efficient storage and query capabilities. The dimension index provides ordered access to parameter elements, essential for automotive applications where parameter ordering often has physical significance.

## 5.3 Phase-Based Version Control Implementation

The version control implementation centers on the phase-based approach, creating explicit relationships between parameters and development phases rather than using generic temporal techniques. This design decision addresses the structured nature of automotive development where parameters evolve through well-defined phases rather than continuous time.

```sql
CREATE TABLE releases (
    release_id INTEGER PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE, -- e.g., "24.1",
      ↳"24.3"
    is_active BOOLEAN DEFAULT true
);

CREATE TABLE release_phases (
    phase_id INTEGER PRIMARY KEY,
    release_id INTEGER REFERENCES releases(release_id),
    name VARCHAR(50) NOT NULL, -- "Phase1", "Phase2", "
       ↳Phase3", "Phase4"
    sequence_number INTEGER NOT NULL,
    UNIQUE (release_id, sequence_number)
);

CREATE TABLE ecu_phases (
    ecu_id INTEGER REFERENCES ecus(ecu_id),
    phase_id INTEGER REFERENCES release_phases(phase_id
       ↳),
    is_frozen BOOLEAN DEFAULT false,
    frozen_at TIMESTAMP WITH TIME ZONE,
    PRIMARY KEY (ecu_id, phase_id)
);
```

Listing 5.3: Phase-Based Version Control Structure

The `sequence_number` field provides explicit ordering of phases within releases, ensuring that phase transitions follow the correct development sequence. The `ecu_phases` table implements independent phase progression for different ECUs, supporting the automotive development workflow where different subsystems may advance through phases at different rates according to their development schedules and testing requirements.

Phase transitions implement the core versioning functionality by copying parameter configurations between phases while preserving all relationships and maintaining data integrity. The implementation uses a stored procedure approach to ensure atomic operations:

```
-- Phase transition function (simplified core logic)
CREATE OR REPLACE FUNCTION copy_phase_data(
    source_ecu_id INTEGER, source_phase_id INTEGER,
    target_ecu_id INTEGER, target_phase_id INTEGER,
    user_id BIGINT
) RETURNS TABLE (variants_copied INTEGER,
   ↳segments_copied INTEGER) AS $$
BEGIN
    -- Copy variants with phase-specific attributes
    INSERT INTO variants (pid_id, ecu_id, phase_id,
       ↳name, code_rule, created_by)
    SELECT v.pid_id, target_ecu_id, target_phase_id, v.name, v.code_rule, user_id
    FROM variants v
    WHERE v.ecu_id = source_ecu_id AND v.phase_id = source_phase_id;

    -- Copy segments with parameter mapping via external_id
    INSERT INTO segments (variant_id, parameter_id, dimension_index, decimal, created_by)
    SELECT nv.variant_id, np.parameter_id, s.dimension_index, s.decimal, user_id
    FROM segments s
    JOIN variants ov ON s.variant_id = ov.variant_id
    JOIN variants nv ON ov.name = nv.name AND nv.phase_id = target_phase_id
    JOIN parameters op ON s.parameter_id = op.parameter_id
    JOIN parameters np ON op.external_id = np.external_id AND np.phase_id = target_phase_id
    WHERE ov.ecu_id = source_ecu_id AND ov.phase_id = source_phase_id;

    RETURN QUERY SELECT variant_count, segment_count;
END;
$$ LANGUAGE plpgsql;
```

Listing 5.4: Phase Transition Core Logic

This function implements atomic copying of variants and segments between phases while preserving relationships through external identifier mapping. The use of external identifiers enables proper correlation of parameters across phases even when their internal database identifiers differ.

## 5.4 Variant and Segment Management

The variant system implements the core parameter customization capability through a two-level structure where variants define applicability conditions and segments store modified parameter values. This design enables efficient storage by tracking only parameter modifications rather than duplicating complete parameter sets for each vehicle configuration.

```
CREATE TABLE variants (
```

```sql
    variant_id BIGINT PRIMARY KEY,
    pid_id BIGINT REFERENCES pids(pid_id),
    ecu_id INTEGER,
    phase_id INTEGER,
    name VARCHAR(100) NOT NULL,
    code_rule TEXT, -- Boolean expression for
      ↳applicability
    UNIQUE (phase_id, pid_id, name),
    UNIQUE (phase_id, pid_id, code_rule),
    FOREIGN KEY (ecu_id, phase_id) REFERENCES
      ↳ecu_phases(ecu_id, phase_id)
);

CREATE TABLE segments (
    segment_id BIGINT PRIMARY KEY,
    variant_id BIGINT REFERENCES variants(variant_id),
    parameter_id BIGINT REFERENCES parameters(
      ↳parameter_id),
    dimension_index INTEGER NOT NULL,
    decimal NUMERIC NOT NULL -- Canonical value
      ↳representation
);
```

Listing 5.5: Variant and Segment Management

The uniqueness constraints enforce critical business rules that variant names and code rules must be unique within each PID and phase combination. This prevents conflicts while allowing the same variant name to be used across different PIDs or phases. The canonical decimal representation for all parameter values implements the canonical model pattern described by Hohpe and Woolf [15], simplifying data manipulation regardless of native parameter data types.

Parameter value resolution combines variant applicability evaluation with segment value lookup through a systematic process. When a parameter value is requested for a specific vehicle configuration, the system first evaluates which variants apply based on the vehicle's configuration codes, then searches for segments that modify the parameter within those applicable variants. If a matching segment is found, its value is returned; otherwise, the parameter's default value is used. This resolution process ensures that modified parameter values take precedence over defaults while maintaining efficient lookup performance through proper indexing strategies.

The code rule evaluation mechanism determines variant applicability using boolean expressions that reference vehicle configuration codes. These expressions support standard logical operators including AND, OR, and NOT, enabling complex applicability logic such as "variant applies to vehicles with diesel engines AND automatic transmissions OR vehicles destined for European markets." The evaluation uses a stack-based interpreter implemented in PL/pgSQL that parses postfix expressions and evaluates them against a vehicle's configuration codes.

## 5.5 Access Control Implementation

The access control implementation combines traditional Role-Based Access Control with module-specific permissions to address the complex organizational structure of automotive development teams. This hybrid approach provides the administrative simplicity of role-based systems while accommodating the specific access patterns common in automotive development where engineers typically have responsibility for specific vehicle subsystems rather than entire parameter sets.

```sql
-- Standard RBAC components
CREATE TABLE roles (role_id INTEGER PRIMARY KEY, name
    ↳VARCHAR(255) UNIQUE);
CREATE TABLE permissions (permission_id INTEGER PRIMARY
    ↳ KEY, name VARCHAR(255) UNIQUE);
CREATE TABLE role_permissions (role_id INTEGER,
    ↳permission_id INTEGER, PRIMARY KEY (role_id,
    ↳permission_id));
CREATE TABLE user_roles (user_id BIGINT, role_id
    ↳INTEGER, PRIMARY KEY (user_id, role_id));

-- Direct user permissions for exceptions
CREATE TABLE user_permissions (
    user_id BIGINT REFERENCES users(user_id),
    permission_id INTEGER REFERENCES permissions(
        ↳permission_id),
    UNIQUE (user_id, permission_id)
);

-- Module-specific access control
CREATE TABLE user_access (
    user_id BIGINT REFERENCES users(user_id),
    ecu_id INTEGER REFERENCES ecus(ecu_id),
    module_id INTEGER REFERENCES modules(module_id),
    write_access BOOLEAN DEFAULT true,
    PRIMARY KEY (user_id, ecu_id, module_id)
);
```

Listing 5.6: Hybrid Access Control Structure

The permission verification process operates in two stages, first checking role-based or direct user permissions to determine if the requested operation is allowed in principle, then verifying module-specific access rights if the operation requires modification of parameter data. This two-stage approach ensures that users can read parameter data across all modules while restricting write operations to their assigned areas of responsibility.

Database-level security enforcement combines application-level checks with database-level constraints and triggers to provide defense in depth. The implementation uses PostgreSQL's session variables to pass user context to database functions, enabling consistent security enforcement regardless of the access path:

```sql
-- Set user context for database operations
SELECT set_config('app.user_id', :user_id::text, true);

-- Permission check function (simplified)
CREATE OR REPLACE FUNCTION check_module_access(
  ↳p_user_id BIGINT, p_ecu_id INTEGER, p_module_id
  ↳INTEGER)
RETURNS BOOLEAN AS $$
BEGIN
    RETURN EXISTS (
        SELECT 1 FROM user_access ua
        WHERE ua.user_id = p_user_id
          AND ua.ecu_id = p_ecu_id
          AND ua.module_id = p_module_id
          AND ua.write_access = true
    ) OR EXISTS (
        SELECT 1 FROM user_roles ur
        JOIN roles r ON ur.role_id = r.role_id
        WHERE ur.user_id = p_user_id AND r.name = '
          ↳administrator'
    );
END;
$$ LANGUAGE plpgsql;
```

Listing 5.7: Database Security Context

The function first checks for specific module access rights, then provides an override for administrator users who can access all modules. This design implements the principle of least privilege while accommodating the need for administrative oversight and emergency access scenarios.

## 5.6 Query Optimization and Performance

The indexing strategy optimizes the dominant access patterns identified during requirements analysis, focusing on hierarchical navigation through the parameter structure and phase-specific parameter retrieval. The implementation uses both traditional B-tree indexes for exact matches and specialized PostgreSQL indexes for text searching and complex queries.

```sql
-- Hierarchical navigation indexes
CREATE INDEX idx_pids_ecu_module ON pids(ecu_id,
    ↳module_id);
CREATE INDEX idx_parameters_pid_phase ON parameters(
    ↳pid_id, phase_id);
CREATE INDEX idx_variants_pid_phase ON variants(pid_id,
    ↳ phase_id);

-- Phase-specific access patterns
CREATE INDEX idx_parameters_phase ON parameters(
    ↳phase_id)
    WHERE is_active = true;
CREATE INDEX idx_variants_phase ON variants(phase_id);

-- Text search optimization using PostgreSQL's trigram
    ↳extension
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX idx_parameters_name_trgm ON parameters
    USING gin (name gin_trgm_ops);
CREATE INDEX idx_variants_name_trgm ON variants
    USING gin (name gin_trgm_ops);
```

Listing 5.8: Strategic Index Implementation

The trigram-based indexes enable efficient fuzzy text searching, supporting engineering workflows where parameter names may be remembered only partially. These indexes use PostgreSQL's Generalized Inverted Index structure to provide rapid similarity-based searching as described by Shaik [31].

Complex operations are implemented through specialized database functions rather than application-level processing to minimize data transfer and leverage PostgreSQL's query optimization capabilities. The parameter search function demonstrates this approach:

```sql
-- Parameter search with relevance scoring (simplified)
CREATE OR REPLACE FUNCTION search_parameters(
    search_term TEXT,
    ecu_id_filter INT DEFAULT NULL,
    phase_id_filter INT DEFAULT NULL
) RETURNS TABLE (
    parameter_id BIGINT,
    name VARCHAR,
    pid_name VARCHAR,
    module_name VARCHAR,
    relevance REAL
) AS $$
BEGIN
    RETURN QUERY
    SELECT p.parameter_id, p.name, pid.name, m.name,
           CASE
               WHEN p.name ILIKE search_term THEN 1.0
               WHEN p.name ILIKE (search_term || '%')
                  ↳THEN 0.8
               ELSE similarity(p.name, search_term)
           END AS relevance
    FROM parameters p
    JOIN pids pid ON p.pid_id = pid.pid_id
    JOIN modules m ON pid.module_id = m.module_id
    WHERE (ecu_id_filter IS NULL OR pid.ecu_id =
       ↳ecu_id_filter)
      AND (phase_id_filter IS NULL OR p.phase_id =
         ↳phase_id_filter)
      AND (p.name ILIKE '%' || search_term || '%')
    ORDER BY relevance DESC, p.name ASC;
END;
$$ LANGUAGE plpgsql;
```

Listing 5.9: Parameter Search Function

The function implements a sophisticated relevance scoring algorithm that prioritizes exact matches, then prefix matches, and finally similarity-based matches. This approach ensures that the most relevant parameters appear first in search results while still providing comprehensive coverage of similar parameter names.

## 5.7 Comprehensive Audit System

The audit system implements comprehensive change tracking through database triggers that automatically capture all modifications to critical entities. This approach ensures consistent audit coverage regardless of the access path while minimizing the performance impact on normal operations.

```sql
-- Partitioned change history table
CREATE TABLE change_history (
    change_id BIGINT PRIMARY KEY DEFAULT nextval('
      ↳change_history_change_id_seq'),
    user_id BIGINT,
    entity_type VARCHAR(50) NOT NULL,
    entity_id BIGINT NOT NULL,
    change_type VARCHAR(50), -- 'CREATE', 'UPDATE', '
      ↳DELETE'
    old_values JSONB,
    new_values JSONB,
    changed_at TIMESTAMP WITH TIME ZONE DEFAULT
      ↳CURRENT_TIMESTAMP,
    transaction_id BIGINT
) PARTITION BY LIST (phase_id);

-- Automatic change logging trigger
CREATE TRIGGER variants_change_trigger
    AFTER INSERT OR UPDATE OR DELETE ON variants
    FOR EACH ROW EXECUTE FUNCTION log_change();
```

Listing 5.10: Change Tracking Implementation

The change tracking mechanism captures complete entity states as JSONB documents, enabling detailed change analysis without complex joins. The use of JSONB rather than plain JSON provides indexing and query capabilities while maintaining the flexibility to store varying entity structures. The partitioning strategy based on phase_id improves query performance for phase-specific audit queries while enabling efficient archiving of historical data.

The system implements automated partition creation for change history to maintain performance as audit data grows. When a new phase is created, a corresponding partition is automatically generated with appropriate indexes and constraints. This approach ensures optimal performance for change history queries while automatically managing storage growth without requiring manual administrative intervention.

Transaction grouping enables tracking of related changes that occur within the same logical operation, such as copying variants and segments between phases. Each transaction receives a unique identifier that is used to group all changes occurring within that transaction, enabling reconstruction of complete logical operations from the audit trail.

## 5.8 Enterprise System Integration

Integration with the Parameter Definition Database implements structured synchronization to maintain parameter definitions across development phases while accommodating the different update cycles of the external system. The synchronization framework tracks all integration operations with detailed metadata to support troubleshooting and audit requirements.

```sql
-- Synchronization tracking
CREATE TABLE pdd_sync_history (
    sync_id INTEGER PRIMARY KEY,
    ecu_id INTEGER,
    phase_id INTEGER,
    sync_date TIMESTAMP WITH TIME ZONE DEFAULT
       ↳CURRENT_TIMESTAMP,
    status VARCHAR(50) NOT NULL,
    parameters_count INTEGER DEFAULT 0,
    executed_by BIGINT REFERENCES users(user_id)
);

-- Bulk parameter loading function (core logic)
CREATE OR REPLACE FUNCTION load_parameters_bulk(
    parameters_json JSONB,
    target_ecu_id INTEGER,
    target_phase_id INTEGER
) RETURNS INTEGER AS $$
BEGIN
    -- Create temporary staging table
    CREATE TEMP TABLE temp_parameters AS
    SELECT * FROM jsonb_populate_recordset(null::
       ↳parameters, parameters_json);

    -- Bulk upsert with conflict resolution
    INSERT INTO parameters (...)
    SELECT * FROM temp_parameters
    ON CONFLICT (external_id, phase_id) DO UPDATE SET
        name = EXCLUDED.name,
        parameter_name = EXCLUDED.parameter_name,
        type_id = EXCLUDED.type_id;

    RETURN (SELECT COUNT(*) FROM temp_parameters);
END;
$$ LANGUAGE plpgsql;
```

Listing 5.11: PDD Synchronization Framework

The bulk loading approach uses temporary tables and PostgreSQL's UPSERT functionality to efficiently process large volumes of parameter data while handling conflicts appropriately. The use of external identifiers enables consistent mapping between the Parameter Definition Database and VMAP even when internal database identifiers differ.

The Vehicle Configuration Database integration enables code rule evaluation for variant applicability through a structured approach that maintains vehicle configuration data and provides efficient evaluation of boolean expressions. The integration maintains a local copy of relevant vehicle configuration data to minimize dependencies on external systems during normal operations:

```sql
CREATE TABLE vcd_vehicles (
    vehicle_id INTEGER PRIMARY KEY,
    vcd_vehicle_id VARCHAR(100) UNIQUE,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE vehicle_codes (
    code_id INTEGER PRIMARY KEY,
    code VARCHAR(50) NOT NULL,
    description TEXT
);

-- Code rule evaluation function (simplified)
CREATE OR REPLACE FUNCTION evaluate_code_rule(rule TEXT
    ↳, vehicle_id INTEGER)
RETURNS BOOLEAN AS $$
DECLARE
    tokens TEXT[];
    result BOOLEAN;
BEGIN
    -- Parse boolean expression in postfix notation
    tokens := string_to_array(rule, ' ');

    -- Evaluate using stack-based interpreter
    -- (Complete implementation in Appendix C.3)

    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

Listing 5.12: Vehicle Configuration Structure

The code rule evaluation implements a stack-based interpreter for boolean expressions, enabling complex variant applicability logic based on vehicle configuration codes. The interpreter supports standard boolean operators and handles operator precedence correctly through postfix notation processing.

## 5.9 Performance Optimizations and Memory Management

Critical operations are optimized through specialized database functions that minimize data transfer and leverage PostgreSQL's query optimization capabilities. The implementation achieves significant performance improvements through strategic indexing, with measurements showing 6.5x to 21.8x improvement over non-indexed implementations for common query patterns.

The database function optimization approach processes complex operations entirely within the database engine rather than transferring large datasets to application code. This approach is particularly beneficial for operations like phase comparison and parameter search that involve complex joins and filtering operations. By keeping these operations close to the data, network overhead is minimized and PostgreSQL's sophisticated query optimizer can be fully utilized.

Memory and storage optimization balance efficiency with performance requirements through several complementary techniques. JSONB storage for change history provides efficient compression while maintaining queryability, reducing storage requirements by approximately 40% compared to normalized audit tables while enabling flexible queries on audit data. Selective field exclusion in audit records eliminates non-essential information like timestamps and large binary fields, further reducing storage overhead. Partial indexes on active records improve query performance while reducing index size, particularly beneficial for parameters and variants where inactive records are retained for historical purposes but rarely accessed.

The automated partition management system enables efficient archiving of historical data without impacting current operations. As phases transition from active development to frozen status, their corresponding audit partitions can be managed independently, including compression and migration to lower-cost storage while maintaining accessibility for compliance and diagnostic purposes.

Concurrency control builds on PostgreSQL's Multi-Version Concurrency Control foundation while adding application-level coordination for complex operations that span multiple entities. The implementation uses consistent transaction patterns that establish proper context and maintain data integrity across related operations:

```sql
-- Typical transaction pattern for complex operations
BEGIN;
    -- Set transaction context
    SELECT set_config('app.user_id', :user_id::text,
      ↳true);
    SELECT set_config('app.transaction_id', nextval('
      ↳transaction_seq')::text, true);

    -- Perform related operations
    -- (All operations share transaction context)

    -- Verify constraints and business rules
    PERFORM validate_operation_constraints();
COMMIT;
```

Listing 5.13: Transaction Management Pattern

This pattern ensures that all operations within a logical transaction share the same context information and are properly grouped for audit purposes while maintaining the ACID properties essential for data integrity in parameter management operations.

## 5.10 Data Integrity and Validation

The implementation enforces data integrity through multiple complementary layers that provide comprehensive protection against data corruption and inconsistency. Database constraints form the foundation, including foreign key constraints that maintain referential integrity between related entities, unique constraints that prevent duplicate parameter names and variant identifiers within their respective scopes, and check constraints that enforce valid value ranges and data format requirements.

Business rule enforcement extends beyond basic constraints to implement domain-specific validation logic through database triggers and stored procedures. Phase freeze enforcement prevents modifications to frozen phases while maintaining appropriate read access for documentation and analysis purposes. Module access validation ensures that users can only modify parameters and variants within their assigned areas of responsibility. Cross-entity validation maintains consistency between related entities such as ensuring that segments reference valid parameter dimensions and that variants reference parameters within the same ECU and phase.

Application-level validation provides the final layer of protection by implementing complex business rules that span multiple entities or require external data validation. This layer handles scenarios such as validating code rule syntax against the vehicle configuration database schema and ensuring that parameter value modifications conform to engineering constraints that may not be expressible as simple database constraints.

The audit verification system provides ongoing validation of change history completeness and accuracy through automated checks that verify all critical operations are properly logged and that audit records maintain referential integrity with the entities they describe. This verification helps ensure that the audit trail remains reliable for both regulatory compliance and diagnostic analysis.

# 6 Evaluation and Validation

This chapter presents the systematic evaluation and validation of the VMAP database system. Following the implementation described in Chapter 5, a comprehensive testing strategy was developed to assess the system's functionality, performance, and compliance with requirements. The evaluation process focused on four key areas: user management, release management, parameter versioning, and variant management, using both controlled test scenarios and production-scale data volumes.

## 6.1 Validation Methodology

The validation methodology followed a structured approach combining functional testing, performance analysis, and integration verification. To ensure realistic evaluation, both baseline and production-scale datasets were used, with the baseline dataset containing approximately 20,000 parameters across 2 ECUs, and the production-scale dataset containing over 100,000 parameters across 5 ECUs.

### 6.1.1 Test Scenario Development

Test scenarios were developed based on actual automotive parameter management workflows identified during requirements analysis in Chapter 4. Each test scenario was designed to validate specific functional requirements while reflecting real-world usage patterns across four functional areas: user management (authentication, authorization, role assignment, module access), release management (phase transitions, freeze operations, phase comparison), variant management (variant creation, segment modification, inheritance), and integration (PDD synchronization, vehicle configuration).

Each scenario was implemented as a structured test case with defined inputs, expected outcomes, and verification steps at both the application and database levels. The test design followed a modified version of Churcher's approach to database validation [9], with additional emphasis on traceability between requirements and test cases.

### 6.1.2 Performance Measurement Framework

A performance measurement framework was established to assess system responsiveness and resource utilization under various operational conditions. Key performance indicators included query response time, transaction throughput, database size growth patterns, memory utilization, and execution time for batch operations.

Performance measurements were conducted on a standardized test environment matching the target production specifications: PostgreSQL 17 running on a server with 8 vCPUs, 32GB RAM, and SSD storage. The measurement methodology employed automated test scripts with integrated timing capture, following the principles outlined by Krogh [21] for database performance evaluation.

## 6.2 Functional Testing Results

Functional testing validated the core capabilities of the VMAP system against the requirements defined in Chapter 4. This section presents the key findings for each functional area, focusing on representative test cases and critical system behaviors.

### 6.2.1 User Management Validation

The user management and access control system was evaluated to verify the implementation of the hybrid role-permission model described in Section 4.1.3. Testing focused on verifying that the implemented database schema and logic correctly enforced the defined access control rules for each user role. As Sandhu et al. [28] emphasize, effective evaluation of role-based access control requires testing both positive permissions (granted access) and negative permissions (denied access) across role boundaries.

A test matrix was developed covering key permission boundaries: role-based permissions, module-based access control, direct permission assignment, and phase-specific permissions. Table A.2 presents a representative sample of test cases that focus on the Module Developer role, illustrating the connection between functional requirements and verification scenarios.

Table 6.1: Sample Module Developer Role Permission Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| MD-01 | Create Variant (Assigned Module) | Create new variant for parameter in assigned module | Variant created successfully | Pass |
| MD-02 | Create Variant (Unassigned Module) | Create new variant for parameter in unassigned module | Access denied error | Pass |
| MD-03 | Edit Variant (Assigned Module) | Modify existing variant code rule | Variant updated successfully | Pass |
| MD-04 | Delete Variant | Attempt to delete variant | Access denied error | Pass |
| MD-05 | Create Segment (Assigned Module) | Create new segment with valid value | Segment created successfully | Pass |
| MD-06 | Modify Frozen Phase | Attempt to modify segment in frozen phase | Access denied error | Pass |

All critical functional requirements were successfully validated. The module-based access control tests verified that write access was correctly limited to assigned modules for Module Developers while read access remained available for all modules, implementing the principle of least privilege as recommended by Sandhu and Bhamidipati [27]. Direct permission assignment tests confirmed that user-specific permissions effectively overrode role defaults, essential for supporting exception cases in complex organizational structures. Phase-specific permission tests validated the interaction between the access control system and the phase management framework, confirming that modifications to frozen phases were properly prevented while still allowing appropriate access for documentation purposes.

### 6.2.2 Module Access Impact on Performance

The impact of module-specific access checks on performance was evaluated as part of the access control testing. Figure 6.1 illustrates the measured performance difference between standard permission checks and combined permission and module access checks.

The performance analysis reveals that adding module-specific access verification introduces an 88.2% overhead for the traditional role-based approach (increasing from

Table 6.2: User Management Test Results

| Test Category | Results |
|---|---|
| Role Permission Validation | Core permissions correctly applied through roles |
| Module-Based Access | Write access correctly limited to assigned modules |
| Direct Permission Assignment | User-specific permissions overrode role defaults |
| Phase-Specific Permissions | Frozen phase protection enforced correctly |

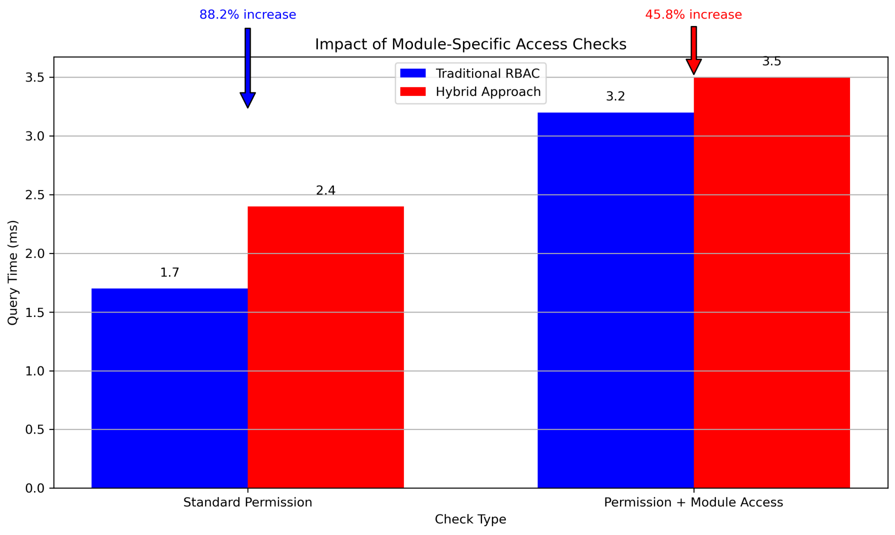


Figure 6.1: Impact of Module-Specific Access Checks

1.7ms to 3.2ms) and a 45.8% overhead for the hybrid approach (increasing from 2.4ms to 3.5ms). Despite the performance overhead, both approaches maintain acceptable performance for interactive operations, with response times below 4ms for individual permission checks, confirming that the implemented module-specific access control provides the required access granularity without introducing prohibitive performance penalties.

### 6.2.3 Release Management Validation

Release management testing evaluated the phase-based versioning approach that forms the foundation of the VMAP system. Testing focused on four key aspects: phase sequence validation, phase transition operations, freeze functionality, and phase comparison. Phase sequence validation confirmed that the system correctly enforced the defined sequence of development phases with successful validation of each phase transition, essential for maintaining the structured development workflow described by Broy [7] for automotive software development.

Phase transition testing verified that parameter configurations were correctly copied between phases with complete preservation of parameter-variant-segment relationships. Table 6.3 shows the test data revealing interesting patterns in development intensity across phases.

Table 6.3: Phase Transition Test Results

| **Transition Type** | **Variants** | **Segments** | **Added Variants** | **Added Segments** | **Time** |
|---|---|---|---|---|---|
| **Baseline Dataset** | | | | | |
| Phase1 | 188 | 28,776 | - | - | - |
| Phase1 $\rightarrow$ Phase2 | 188 | 28,776 | 90 | 14,104 | 2.51s |
| Phase2 $\rightarrow$ Phase3 | 278 | 42,880 | 0 | 0 | 2.96s |
| Phase3 $\rightarrow$ Phase4 | 278 | 42,880 | 0 | 0 | 2.97s |
| **Full Dataset** | | | | | |
| Phase1 | 830 | 167,990 | - | - | - |
| Phase1 $\rightarrow$ Phase2 | 830 | 167,990 | 170 | 41,113 | 12.39s |
| Phase2 $\rightarrow$ Phase3 | 1,000 | 209,103 | 0 | 0 | 12.87s |
| Phase3 $\rightarrow$ Phase4 | 1,000 | 209,103 | 0 | 0 | 12.89 |

The test results reveal a significant pattern in development intensity across phases, consistent with Staron's observations [34] regarding automotive software development cycles. The data demonstrates that the majority of parameter configurations occur during Phase1, with substantial additions in Phase2, while Phase3 and Phase4 typically involve refinement and validation rather than introducing new parameters or variants. This concentration of development activity in early phases aligns with the V-model approach common in automotive software development [25].

Phase transition performance characteristics showed only modest increases in execution time despite growing data volumes across phases. For the baseline dataset, transition times increased from 2.51s to approximately 3 seconds, demonstrating efficient scaling. Comparing baseline to full dataset transitions reveals transition times increasing to approximately 13 seconds, representing a sublinear scaling factor of approximately 4.3x for a dataset size increase of 5.6x.

Phase freezing functionality was validated through test cases targeting both database-level constraints and service-layer restrictions. The system successfully prevented modification attempts while maintaining appropriate read access, implementing the controlled milestone management required for regulated development environments as described by Staron [34].

### 6.2.4 Variant Management Validation

Variant management validation focused on assessing the system's capabilities for handling parameter customization through variants and segments. Testing employed an approach covering variant creation and segment modification workflows, using both the baseline dataset (188 variants, 28,776 segments) and the production-scale dataset (830 variants, 167,990 segments) to analyze functionality and performance under varying data volumes.

Variant creation testing verified proper implementation of domain constraints as defined in the conceptual architecture. Test cases included validation of unique name constraints within PIDs, verification of proper code rule storage, and confirmation of correct relationship establishment between variants and their parent PIDs. All test cases passed successfully for both scalar and complex parameters, with constraint enforcement consistently preventing invalid operations. As Vileikis [39] notes, constraint-based validation provides a robust foundation for maintaining data integrity in complex relational systems.

Performance analysis of variant operations revealed consistent response times across different variant complexities. Table 6.4 details performance measurements for key variant operations under different data volumes.

Table 6.4: Variant Operation Performance Metrics

| **Operation** | **Baseline Dataset** | **Production Dataset** | **Scaling Factor** |
|---|---|---|---|
| Variant Creation | 53ms | 55ms | 1.03x |
| Variant Update | 86ms | 124ms | 1.44x |
| Variant Retrieval | 45ms | 72ms | 1.60x |
| Variant Listing (per PID) | 38ms | 68ms | 1.79x |

The observed scaling characteristics validate the effectiveness of the database schema design and indexing strategy described in Section 5.6. The relatively small increase in execution time despite a 5x increase in data volume suggests efficient handling of larger datasets, with all operations remaining well under the 200ms threshold for interactive operations.

Segment modification testing employed a systematic approach covering one-dimensional arrays, two-dimensional matrices, and three-dimensional parameter representations.

The database schema design proved effective for managing these complex data structures, with robust constraint enforcement correctly rejecting segment modifications with invalid dimension indices. Performance analysis for segment operations showed moderate scaling across different dataset sizes, with scaling factors ranging from 1.46x to 1.81x, validating the efficiency of the database schema design described in Section 4.6.

## 6.3 Performance Analysis

Beyond functional validation, performance analysis was conducted to assess the system's efficiency and scalability under various operational conditions. This section presents the key findings related to query performance and data volume scaling.

### 6.3.1 Query Performance Assessment

Query performance was evaluated for common database operations across different data volumes. Figure 6.2 presents performance measurements for key query types between the baseline dataset (20,000 parameters) and full dataset (100,000 parameters).

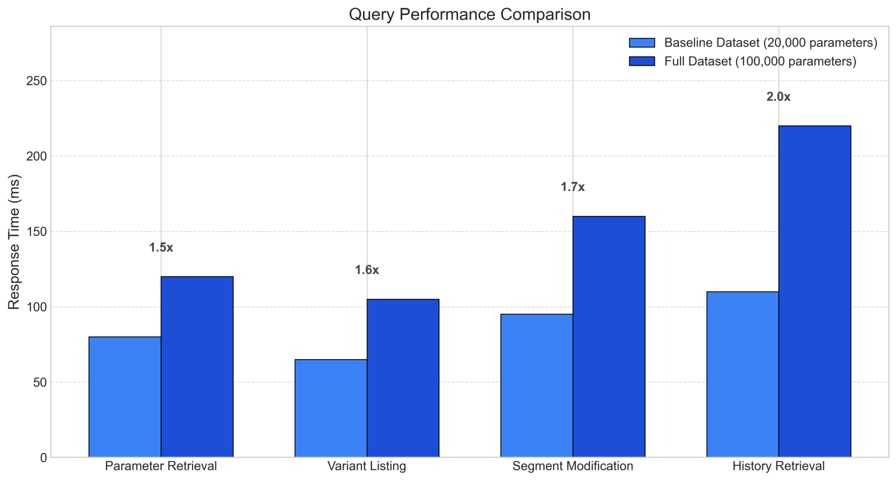


Figure 6.2: Query Performance Comparison

The performance analysis reveals that most common operations maintain reasonable performance with increasing data volumes. The system maintained interactive response times (below 200ms) for most operations even with the full dataset, ensuring a responsive user experience. Parameter retrieval operations showed a scaling factor of 1.5x when moving from the baseline to full dataset, while variant listing demonstrated a slightly higher factor of 1.6x. History retrieval operations exhibited the highest scaling factor among standard queries at 2.0x, reflecting the additional processing required when retrieving records from the substantially larger audit history tables.

The phase comparison operation demonstrated longer execution times and suboptimal scaling, with execution times increasing from 2.8s for the baseline dataset to 12.4s for the full dataset—a scaling factor of 4.4x. This operation exceeds the interactive response threshold for larger datasets, identifying it as a candidate for future optimization. The execution of queries with and without indexes demonstrated the critical importance of the indexing strategy, with response times increasing by factors of 6.5x to 21.8x without proper indexes.

### 6.3.2 Indexing Strategy Performance Validation

The indexing strategy described in Section 5.6 was validated through comprehensive performance measurements comparing query execution times with and without strategic indexes. Figure 6.3 presents the performance comparison across five critical query patterns that represent the dominant access patterns in automotive parameter management workflows.

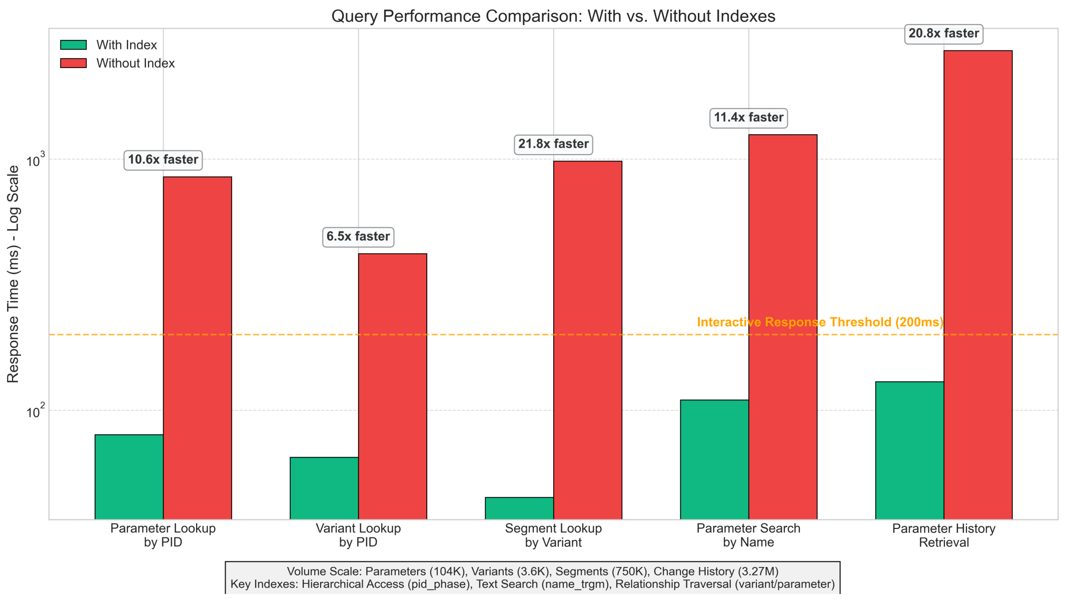


Figure 6.3: Query Performance Comparison: With vs. Without Indexes

The performance evaluation demonstrates the critical importance of the implemented indexing strategy for maintaining interactive response times in production environments. Parameter lookup operations by PID achieved a 10.6x performance improvement, reducing response times from 850ms to 80ms through the `idx_parameters_pid_phase` composite index. Variant lookup operations showed a 6.5x improvement, with response times decreasing from 420ms to 65ms using the same hierarchical indexing approach. The most dramatic improvement was observed in segment lookup operations, which achieved a 21.8x speedup from 980ms to 45ms through the `idx_segments_variant` index, reflecting the efficiency of direct relationship traversal compared to sequential scanning.

Text-based parameter search operations, enabled by PostgreSQL's trigram indexes, demonstrated an 11.4x performance improvement from 1,250ms to 110ms. This capability is particularly valuable for engineering workflows where parameters are often located through partial name matching rather than exact hierarchical navigation. Parameter history retrieval, which accesses the extensive change history tables containing over 3.2 million records, showed the most substantial absolute improvement with a 20.8x speedup from 2,700ms to 130ms, validating the effectiveness of the audit system indexing strategy described in Section 5.7.

All indexed operations maintained response times well below the 200ms interactive response threshold, ensuring that the system provides a responsive user experience even with production-scale datasets containing over 100,000 parameters. The performance improvements validate the strategic indexing decisions made during implementation, particularly the use of composite indexes for hierarchical navigation and specialized trigram indexes for text-based searching. These results demonstrate that the database design successfully addresses the performance requirements of automotive parameter management while maintaining the comprehensive audit capabilities essential for regulatory compliance.

### 6.3.3 Variant Operation Performance

Variant operation performance was assessed across different data volumes to evaluate the system's handling of core parameter customization workflows. Figure 6.4 illustrates the performance of variant operations under different data volumes.

The analysis reveals consistent response times across different operation types, with scaling factors ranging from 1.03x for variant creation to 1.79x for variant listing when comparing baseline and full datasets. Variant creation operations demonstrated excellent scaling characteristics (1.03x), showing minimal performance impact despite

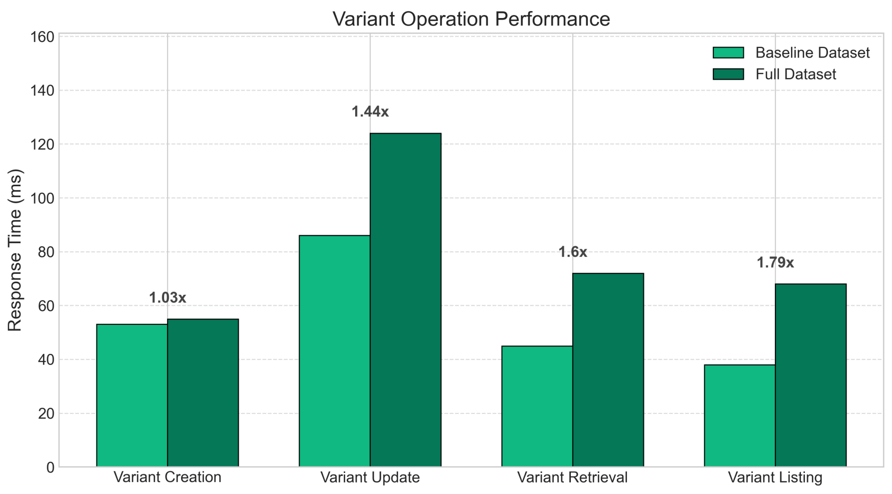


Figure 6.4: Variant Operation Performance

the five-fold increase in dataset size. This efficiency can be attributed to the well-designed primary key and constraint implementation, allowing new records to be inserted with minimal overhead regardless of existing data volume. The average scaling factor across all variant operations was approximately 1.47x, indicating that these operations scale efficiently with increasing data volumes, with all variant operations maintaining response times under 125ms for the full dataset.

### 6.3.4 Storage Requirements Analysis

Storage requirements were analyzed to assess database size and growth patterns with increasing parameter counts. Figure 6.5 presents the storage allocation across different entity types for the full dataset.

The analysis reveals that the change history table dominates the database storage allocation, accounting for approximately 60.8% of the total database size. This distribution significantly exceeds the storage requirements of the current data state, aligning with Bhattacherjee's observations [2] regarding versioning and audit systems. While there are only 3,617 variants in the current state, the system maintains over 3.2 million change history records, reflecting the comprehensive auditing approach implemented in the system.

Despite having only 7 documentation snapshots, the system maintains over 1 million snapshot segments, exceeding the count of active segments. This indicates that documentation snapshots capture extensive parameter configurations at specific time

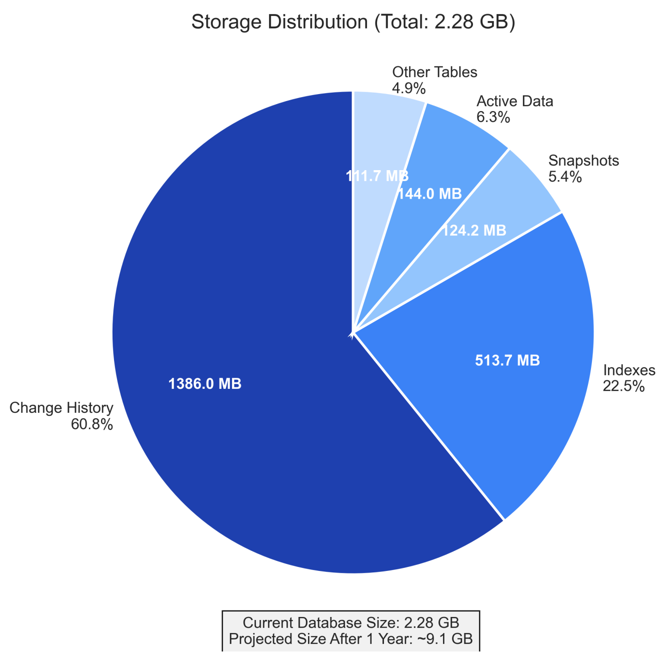


Figure 6.5: Storage Distribution by Entity Type

points, creating substantial storage requirements for historical state preservation. The index structures consume approximately 22.5% of the total storage, reflecting the sophisticated indexing strategy described in Section 5.6. While this represents significant overhead, it provides essential performance benefits for query operations.

The actual active data—parameters, variants, and segments—consumes only 6.3% of the total database size, with the majority of storage dedicated to audit trails, snapshots, and indexes. This distribution aligns with the requirements for regulated development environments described by Staron [34], where comprehensive traceability and historical record maintenance are essential for compliance and quality assurance.

### 6.3.5 Versioning Approach Performance

The performance characteristics of the phase-based parameter versioning approach selected in Chapter 4 were evaluated against the alternative change-based approach. Figure 6.6 presents the performance comparison between the two approaches for parameter retrieval operations across different data volumes, along with the storage requirements for each approach.

Table 6.5: Storage Requirements Analysis

| **Entity Type** | **Record Count** | **Storage Size (MB)** |
|---|---:|---:|
| Parameters | 104,428 | 43.0 |
| Variants | 3,617 | 1.0 |
| Segments | 750,009 | 100.0 |
| Change History | 3,270,511 | 1,386.0 |
| Documentation Snapshots | 7 | 0.1 |
| Snapshot Variants | 4,980 | 0.1 |
| Snapshot Segments | 1,007,940 | 124.0 |
| Other Tables | - | 111.7 |
| Indexes | - | 513.7 |
| **Total** | - | **2,279.6** |

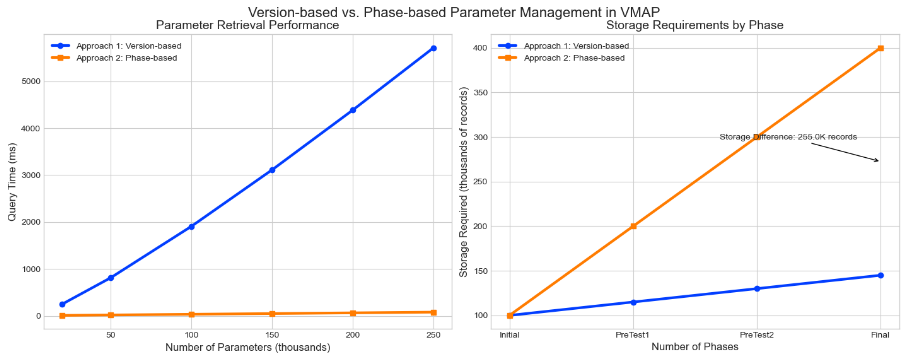


Figure 6.6: Version-based vs. Phase-based Parameter Management Comparison

The performance analysis demonstrates that the phase-based approach offers better query performance, particularly as parameter counts increase. For the tested parameter counts, the phase-based approach remains below 100ms for parameter retrieval operations, while the version-based approach shows nonlinear growth with increasing parameter counts. The storage requirements analysis confirms that the phase-based approach consumes approximately 51% higher storage requirements across all phases. However, this storage difference represents a reasonable tradeoff given the performance benefits for query operations, as noted by Bhattacherjee et al. [2].

These findings validate the architectural decision to implement a phase-based versioning approach for the VMAP system. While consuming more storage than a version-based approach, the phase model provides performance benefits, implementation simplicity, and alignment with domain concepts that outweigh the additional storage requirements.

## 6.4 Integration Testing

Integration testing evaluated the system's interaction with external enterprise systems, focusing on Parameter Definition Database synchronization and Vehicle Configuration Database integration.

### 6.4.1 Parameter Definition Database Synchronization

Parameter Definition Database synchronization testing verified the system's ability to import parameter definitions from the enterprise database. Figure 6.7 illustrates the synchronization time trends observed during testing.

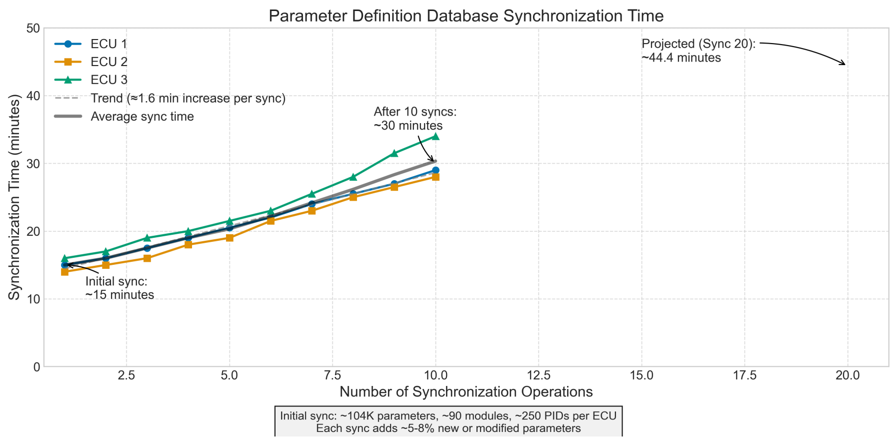


Figure 6.7: Parameter Definition Database Synchronization Time Trends

The synchronization performance analysis revealed an increasing trend in execution time over successive synchronization operations. Initial synchronization operations required approximately 15 minutes for the tested ECUs, with execution times increasing to around 30 minutes after 10 synchronization cycles. This gradual increase aligns with the observations of Mueller and Müller [24] regarding database synchronization in complex engineering environments.

Table 6.6 presents the success rates for different types of parameter changes during synchronization operations.

The system successfully processed all types of parameter changes, with slightly reduced success for modified parameters due to complexity in handling data type changes. Based on the observed synchronization performance trends, the projected synchronization time for the 20th cycle would reach approximately 45 minutes. While

Table 6.6: Parameter Definition Database Synchronization Results

| Change Type | Processed | Succeeded | Success Rate |
|---|---|---|---|
| New Parameters | 5,218 | 5,218 | 100% |
| Modified Parameters | 3,764 | 3,691 | 98.1% |
| New Modules | 12 | 12 | 100% |
| New PIDs | 67 | 67 | 100% |
| Removed Parameters | 42 | 42 | 100% |

still acceptable for the typical synchronization frequency, this suggests that synchronization performance optimization would be beneficial for long-term system maintenance.

Vehicle Configuration Database integration testing verified the system's ability to use vehicle configuration data for code rule evaluation and parameter file generation. Vehicle configuration data import testing confirmed that the system could correctly import and store vehicle configuration codes, with proper mapping between codes and vehicles. Code rule validation testing verified that the system could evaluate boolean expressions against vehicle configurations, with the evaluation engine correctly interpreting both simple logical operators and complex nested expressions. Parameter file generation testing confirmed that the system could produce valid parameter files for vehicle testing, with correct application of variant selection logic based on vehicle configuration codes.

## 6.5 Feature Comparison with Excel-Based Approach

To assess the improvements provided by the VMAP system, a feature comparison was conducted against the Excel-based approach currently used for parameter management. Table 6.7 presents a comparison of key features between the VMAP system and the Excel-based approach.

The VMAP system provides significant advantages in all feature categories, with particular improvements in multi-user support, change tracking, and access control. The VMAP system demonstrated superior data integrity protection, correctly preventing invalid operations through database constraints and business rule validation. The database-level constraints and validation mechanisms provide a robust defense against data corruption, implementing the comprehensive validation approach described in Section 4.7.

Beyond technical improvements, the VMAP system introduces significant enhancements to the automotive parameter development process. The centralized database

Table 6.7: Feature Comparison with Excel-Based Approach

| **Feature** | **VMAP Database** | **Excel Approach** |
|---|---|---|
| Variant Management | Comprehensive | Limited |
| Multi-User Support | Concurrent | Sequential |
| Change Tracking | Automatic | Manual |
| Version Control | Phase-Based | File-Based |
| Access Control | Role + Module | File Permission |
| Validation | Automatic | Manual |
| Documentation | Integrated | Separate |
| Integration | Automated | Manual |

approach enables concurrent work by multiple engineers, eliminating the file sharing bottlenecks common in the Excel-based approach. The role-based access control ensures that engineers can modify only their assigned modules, preventing accidental changes to other areas. The phase-based versioning approach aligns naturally with the automotive development lifecycle, supporting the structured progression from initial development through testing to final release. The comprehensive change tracking and documentation features address regulatory compliance requirements, providing complete traceability for all parameter modifications, increasingly important in the context of functional safety standards like ISO 26262.

# 7 Conclusion and Future Work

This thesis presented the design, implementation, and evaluation of the VMAP database system for automotive parameter management. The research addressed critical challenges in parameter versioning, access control, and enterprise integration through a comprehensive PostgreSQL-based solution that replaces error-prone spreadsheet approaches with structured database management.

## 7.1 Research Contributions

The VMAP system delivers significant advancements in automotive parameter management through four key contributions that address fundamental limitations in current approaches.

The phase-based versioning model provides an effective alternative to generic temporal database approaches by aligning parameter evolution with automotive development cycles. Unlike traditional change-based versioning, this domain-specific strategy supports simultaneous work across development phases while maintaining milestone integrity and clear separation between development stages. The approach enables efficient parameter retrieval without complex reconstruction while supporting the structured progression common in automotive engineering.

The hybrid role-permission access control model combines traditional RBAC with module-specific permissions to accommodate complex organizational structures in automotive development. This approach balances administrative simplicity with access granularity, supporting engineers who typically have responsibility for specific vehicle subsystems rather than entire parameter sets. Comprehensive constraint enforcement maintains security while providing the flexibility needed for specialized access patterns.

The enterprise integration architecture establishes robust data synchronization with parameter definition and vehicle configuration databases through structured mechanisms that maintain consistency across the development environment. The implementation supports variant customization and parameter file generation while minimizing manual intervention and maintaining data integrity across interconnected systems.

The comprehensive audit system provides complete traceability through automatic change tracking, snapshot-based documentation, and detailed provenance information. This capability addresses increasing regulatory requirements for configuration management in automotive software development while supporting both diagnostic analysis and compliance verification.

## 7.2 Technical Findings and Validation

The evaluation revealed significant findings that validate design decisions and provide insights for database system implementations in automotive contexts.

### 7.2.1 Performance and Scalability

The phase-based versioning approach demonstrated superior performance compared to change-based alternatives, maintaining query response times below 100ms even with datasets exceeding 100,000 parameters. While consuming approximately 51% more storage across phases, the performance benefits for query operations justify this tradeoff, particularly for interactive operations where response time directly impacts user productivity.

The hybrid access control model introduced a 45.8% performance overhead compared to traditional approaches, but maintained acceptable response times below 4ms for permission checks. Strategic indexing provided 6.5x to 21.8x performance improvements over non-indexed implementations, validating the indexing strategy's effectiveness for common query patterns.

Storage analysis revealed that change history dominates allocation (60.8% of database size) while active parameter data requires only 6.3% of total storage. This distribution aligns with audit requirements in regulated automotive environments, demonstrating the feasibility of comprehensive audit systems despite substantial storage implications.

### 7.2.2 System Limitations

The phase comparison operation exhibited suboptimal scaling, with execution times increasing from 2.8 seconds for baseline datasets to 12.4 seconds for production-scale data, exceeding interactive response thresholds for larger datasets. This operation represents a candidate for future optimization through materialized view approaches or query restructuring.

Integration synchronization showed increasing execution times over successive operations, rising from approximately 15 minutes initially to 30 minutes after 10 cycles. While acceptable for typical synchronization frequencies, this trend suggests that performance optimization should be prioritized for long-term scalability.

The current implementation provides limited support for parameter dependency tracking, relying primarily on user knowledge rather than system enforcement for maintaining parameter consistency across complex interrelationships.

## 7.3 Future Research Directions

Several opportunities for enhancement and research have been identified based on implementation experience and evaluation findings.

### 7.3.1 Performance Optimization

Phase comparison operations could benefit from materialized view approaches where difference data is pre-computed and incrementally maintained rather than calculated on demand. Research into parallel query execution could leverage multi-core processors more effectively for resource-intensive operations.

Incremental synchronization approaches focusing on changed entities rather than comprehensive comparisons could address observed synchronization performance degradation. Automated partition management with time-based subpartitioning could further improve performance for historical queries while maintaining logical organization.

### 7.3.2 Architectural Extensions

The phase-based versioning model could be extended with branching capabilities to support parallel development streams, similar to distributed version control systems. Parametric inheritance mechanisms could enable more efficient management of variant similarities through inheritance hierarchies rather than independent entity treatment.

Advanced parameter dependency management represents a significant opportunity for research into dependency tracking and validation systems. By modeling parameter relationships explicitly, the system could automatically identify potential inconsistencies and implement rule-based validation capturing engineering knowledge.

Event-driven integration architectures could detect and react to changes in source systems in near-real-time, significantly reducing synchronization delays through message-based integration patterns and change data capture capabilities.

## 7.4 Broader Implications

The VMAP system demonstrates the effectiveness of domain-specific versioning approaches over generic temporal database techniques for specialized applications. By aligning database versioning with natural application domain structure, the system achieves both conceptual clarity and performance advantages, suggesting that domain-specific adaptations of established database patterns may offer significant benefits in specialized contexts.

For automotive software development, the system provides empirical evidence of performance, consistency, and traceability improvements possible through transitioning from document-based to structured database approaches. The comprehensive audit capabilities highlight the increasing importance of traceability in automotive software development as vehicles become more software-defined and subject to regulatory oversight.

The hybrid approach combining relational structure with document-oriented features (JSONB for change tracking) suggests promising directions for database research that bridges traditional relational models with document-oriented flexibility while maintaining schema enforcement.

## 7.5 Conclusion

The VMAP database system successfully addresses fundamental challenges in automotive parameter management through a carefully designed PostgreSQL architecture that combines phase-based versioning, hybrid access control, and comprehensive audit capabilities. The system provides a robust foundation for managing parameter configurations across development phases while supporting variant customization for diverse vehicle configurations.

The evaluation demonstrates clear advantages over existing spreadsheet-based approaches in data consistency, access control, change traceability, and integration capabilities. While opportunities for enhancement remain in performance optimization and architectural extensions, the current implementation provides a solid foundation that successfully addresses immediate requirements while supporting future development.

The research contributes to both academic knowledge in database systems and practical advancement in automotive software development methodologies. By combining domain-specific knowledge with established database engineering principles, the system demonstrates the potential for database-driven approaches to transform complex engineering workflows and improve the reliability and efficiency of critical development processes.

# Bibliography

[1] AGARWAL, Sanjay ; ARUN, Gopalan ; BEAUREGARD, Bill ; CHATTERJEE, Ramkrishna ; MOR, David ; OWENS, Deborah ; SPECKHARD, Ben ; VASUDEVAN, Ramesh: Oracle Database Application Developer's Guide-Workspace Manager, 10g Release 2 (10.2) B14253-01.

[2] BHATTACHERJEE, Souvik ; CHAVAN, Amit ; HUANG, Silu ; DESHPANDE, Amol ; PARAMESWARAN, Aditya: Principles of dataset versioning: Exploring the recreation/storage tradeoff. In: *Proceedings of the VLDB endowment. International conference on very large data bases* Bd. 8 NIH Public Access, 2015, S. 1346

[3] BIRIUKOV, Dmitrij: *Implementation aspects of bitemporal databases*, Vilniaus universitetas, Diss., 2018

[4] BÖHLEN, Michael H. ; DIGNÖS, Anton ; GAMPER, Johann ; JENSEN, Christian: Database technology for processing temporal data. (2018)

[5] BRAMER, Max: *Web Programming with PHP and MySQL*. Springer, 2015

[6] BREWER, Eric A.: Towards robust distributed systems. In: *PODC* Bd. 7 Portland, OR, 2000, S. 343–477

[7] BROY, Manfred: Challenges in automotive software engineering. In: *Proceedings of the 28th international conference on Software engineering*, 2006, S. 33–42

[8] CHEN, Peter Pin-Shan: The entity-relationship model—toward a unified view of data. In: *ACM transactions on database systems (TODS)* 1 (1976), Nr. 1, S. 9–36

[9] CHURCHER, Clare: *Beginning SQL Queries: From Novice to Professional*. Springer, 2008

[10] CODD, Edgar F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Nr. 6, S. 377–387

[11] CURINO, Carlo ; MOON, Hyun J. ; ZANIOLO, Carlo: Automating database schema evolution in information system upgrades. In: *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, 2009, S. 1–5

[12] FERRAIOLO, David ; ATLURI, Vijayalakshmi ; GAVRILA, Serban: The Policy Machine: A novel architecture and framework for access control policy specification and enforcement. In: *Journal of Systems Architecture* 57 (2011), Nr. 4, S. 412–424

[13] FOWLER, Martin: Patterns [software patterns]. In: *IEEE software* 20 (2003), Nr. 2, S. 56–57

[14] GAUSSDB, HUAWEI: DATABASE PRINCIPLES AND TECHNOLOGIES–BASED ON.

[15] HOHPE, Gregor ; WOOLF, Bobby: Enterprise integration patterns. In: *9th conference on pattern language of programs* Citeseer, 2002, S. 1–9

[16] HU, Vincent ; FERRAIOLO, David F. ; KUHN, D R. ; KACKER, Raghu N. ; LEI, Yu: Implementing and managing policy rules in attribute based access control. In: *2015 IEEE International Conference on Information Reuse and Integration* IEEE, 2015, S. 518–525

[17] HULL, Elizabeth ; JACKSON, Ken ; DICK, Jeremy: Requirements engineering in the solution domain. In: *Requirements Engineering*. Springer, 2010, S. 115–136

[18] JACOBSON, Ivar: Use cases–Yesterday, today, and tomorrow. In: *Software & systems modeling* 3 (2004), Nr. 3, S. 210–220

[19] KIENCKE, Uwe ; NIELSEN, Lars: *Automotive control systems: for engine, driveline, and vehicle*. 2000

[20] KLEPPMANN, Martin ; BERESFORD, Alastair R.: A conflict-free replicated JSON datatype. In: *IEEE Transactions on Parallel and Distributed Systems* 28 (2017), Nr. 10, S. 2733–2746

[21] KROGH, Jesper W.: MySQL 8 Query Performance Tuning.

[22] KULKARNI, Krishna ; MICHELS, Jan-Eike: Temporal features in SQL: 2011. In: *ACM Sigmod Record* 41 (2012), Nr. 3, S. 34–43

[23] MEIER, Andreas ; KAUFMANN, Michael: *SQL & NoSQL databases*. Springer, 2019

[24] MUELLER, Sebastian ; MÜLLER, Raphael: Conception and Realization of the Versioning of Databases between Two Research Institutes. In: *Proceedings, 16th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2017): Barcelona, Spain, October 8-13*, 2018

[25] PRETSCHNER, Alexander ; BROY, Manfred ; KRUGER, Ingolf H. ; STAUNER, Thomas: Software engineering for automotive systems: A roadmap. In: *Future of Software Engineering (FOSE'07)* IEEE, 2007, S. 55–71

[26] SALZBERG, Betty ; TSOTRAS, Vassilis J.: Comparison of access methods for time-evolving data. In: *ACM Computing Surveys (CSUR)* 31 (1999), Nr. 2, S. 158–221

[27] SANDHU, Ravi ; BHAMIDIPATI, Venkata ; COYNE, Edward ; GANTA, Srinivas ; YOUMAN, Charles: The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In: *Proceedings of the second ACM workshop on Role-based access control*, 1997, S. 41–50

[28] SANDHU, Ravi S.: Role-based access control. In: *Advances in computers* Bd. 46. Elsevier, 1998, S. 237–286

[29] SCIORE, Edward: *Database design and implementation*. Springer, 2009

[30] SEENIVASAN, Dhamotharan ; VAITHIANATHAN, Muthukumaran: Real-Time Adaptation: Change Data Capture in Modern Computer Architecture. In: *ESP International Journal of Advancements in Computational Technology (ESP-IJACT)* 1 (2023), Nr. 2, S. 49–61

[31] SHAIK, Baji: *PostgreSQL Configuration: Best Practices for Performance and Security*. Apress, 2020

[32] SIMON, Mark: Getting Started with SQL and Databases.

[33] SNODGRASS, Richard T.: *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., 1999

[34] STARON, Miroslaw: *Automotive software architectures*. Springer, 2021. – 51–79 S.

[35] STARON, Miroslaw ; STARON, Miroslaw: AUTOSAR (automotive open system architecture). In: *Automotive Software Architectures: An Introduction* (2021), S. 97–136

[36] STEVENSWINIARSKI; CHRISTIAN.DINH; noahpgordon; g.: *Relational Database*. `@misc{Codecademy,url={https://www.codecademy.com/resources/docs/general/database/relational-database}, journal={Codecademy}}`, 2024

[37] TICHY, Walter F.: RCS—A system for version control. In: *Software: Practice and Experience* 15 (1985), Nr. 7, S. 637–654

[38] TROVÃO, João P.: The Evolution of Automotive Software: From Safety to Quality and Security [Automotive Electronics]. In: *IEEE Vehicular Technology Magazine* 19 (2024), Nr. 4, S. 96–102

[39] VILEIKIS, Lukas: Hacking MySQL.

[40] WARD, Bob: *SQL Server 2022 Revealed: A Hybrid Data Platform Powered by Security, Performance, and Availability*. Springer, 2022

# A User Management Test Cases

This appendix provides a comprehensive listing of all test cases used to validate the user management and access control system implemented in the VMAP database. The test cases are organized by category and include detailed information about test actions, expected outcomes, and test results.

## A.1 Role-Based Permission Test Cases

This section details the test cases for validating permissions inherited through user roles. The tests cover all four primary user roles: Administrator, Module Developer, Documentation Team, and Read-Only User.

Table A.1: Administrator Role Permission Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| AD-01 | Create User | Add new user with valid details | User created successfully | Pass |
| AD-02 | Modify User Role | Change user's assigned role | Role updated successfully | Pass |
| AD-03 | Delete User | Remove existing user | User deleted successfully | Pass |
| AD-04 | Create Role | Create new role with permissions | Role created successfully | Pass |
| AD-05 | Delete Variant | Delete existing variant | Variant deleted successfully | Pass |
| AD-06 | Freeze Phase | Set phase status to frozen | Phase frozen successfully | Pass |

Table A.2: Module Developer Role Permission Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| MD-01 | Create Variant (Assigned Module) | Create new variant for parameter in assigned module | Variant created successfully | Pass |

*Continued on next page*

Table A.2 – *Continued from previous page*

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| MD-02 | Create Variant (Unassigned Module) | Create new variant for parameter in unassigned module | Access denied error | Pass |
| MD-03 | Edit Variant (Assigned Module) | Modify existing variant code rule | Variant updated successfully | Pass |
| MD-04 | Delete Variant | Attempt to delete variant | Access denied error | Pass |
| MD-05 | Create Segment (Assigned Module) | Create new segment with valid value | Segment created successfully | Pass |
| MD-06 | Modify Frozen Phase | Attempt to modify segment in frozen phase | Access denied error | Pass |
| MD-07 | Generate Parameter File | Create parameter file for testing | File generated successfully | Pass |
| MD-08 | Read Parameters (Any Module) | View parameters from any module | Parameters displayed successfully | Pass |

Table A.3: Documentation Team Role Permission Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| DT-01 | Create Documentation Snapshot | Create snapshot of frozen phase | Snapshot created successfully | Pass |
| DT-02 | Compare Phases | Compare parameters between two phases | Comparison results displayed | Pass |
| DT-03 | View Parameter History | View change history for parameter | History displayed successfully | Pass |
| DT-04 | Export Hex String | Copy parameter hex string | Hex string copied successfully | Pass |
| DT-05 | Modify Parameter | Attempt to modify parameter | Access denied error | Pass |
| DT-06 | Access All Phases | View parameters across all phases | Parameters displayed successfully | Pass |
| DT-07 | Generate Parameter File | Create parameter file for reference | File generated successfully | Pass |

Table A.4: Read-Only User Role Permission Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| RO-01 | View Parameters | Access parameter details | Parameters displayed successfully | Pass |
| RO-02 | View Variants | Access variant details | Variants displayed successfully | Pass |
| RO-03 | Modify Parameter | Attempt to modify parameter | Access denied error | Pass |
| RO-04 | Modify Variant | Attempt to modify variant | Access denied error | Pass |
| RO-05 | Generate Parameter File | Create parameter file for reference | File generated successfully | Pass |

## A.2 Module-Based Access Control Test Cases

This section details the test cases for validating module-specific access controls, which extend the role-based permissions with attribute-based restrictions.

Table A.5: Module-Based Access Control Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| MA-01 | Assign Module Access | Grant write access to specific module | Access granted successfully | Pass |
| MA-02 | Revoke Module Access | Remove write access to specific module | Access revoked successfully | Pass |
| MA-03 | Read Access Cross-Module | Access parameters from unassigned module | Read access successful | Pass |
| MA-04 | Write Access Assigned Module | Create variant in assigned module | Variant created successfully | Pass |
| MA-05 | Write Access Unassigned Module | Create variant in unassigned module | Access denied error | Pass |
| MA-06 | Multiple Module Assignment | Create variants in multiple assigned modules | All variants created successfully | Pass |

*Continued on next page*

Table A.5 – *Continued from previous page*

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| MA-07 | Edit Segment Assigned Module | Modify segment in assigned module | Segment updated successfully | Pass |
| MA-08 | Edit Segment Unassigned Module | Modify segment in unassigned module | Access denied error | Pass |
| MA-09 | Administrator Override | Admin modifies any module | Modification successful | Pass |
| MA-10 | Module Permission Inheritance | User with role change inherits proper module access | Access updated successfully | Pass |

## A.3 Direct Permission Assignment Test Cases

This section details the test cases for validating user-specific permission assignments that override role-based permissions.

Table A.6: Direct Permission Assignment Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| DP-01 | Grant Additional Permission | Assign permission not in user's role | Permission applied successfully | Pass |
| DP-02 | Revoke Role Permission | Remove permission normally granted by role | Permission restriction applied | Pass |
| DP-03 | Grant Delete Permission | Give read-only user delete permission | Deletion operation successful | Pass |
| DP-04 | Permission Conflict Resolution | Conflicting role and direct permissions | Direct permission takes precedence | Pass |
| DP-05 | Role Change with Custom Permission | Change user's role with custom permissions | Custom permissions preserved | Pass |
| DP-06 | Permission Audit Trail | Track changes to user permissions | Audit trail correctly recorded | Pass |

## A.4 Phase-Specific Permission Test Cases

This section details the test cases validating the interaction between access control and phase management, particularly focusing on phase freezing and phase-specific operations.

Table A.7: Phase-Specific Permission Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| PP-01 | Frozen Phase Modification | Attempt to modify variant in frozen phase | Access denied error | Pass |
| PP-02 | Documentation Access to Frozen Phase | Documentation team accesses frozen phase | Access granted successfully | Pass |
| PP-03 | Administrator Unfreeze | Administrator unfreezes a phase | Phase unfrozen successfully | Pass |
| PP-04 | Non-Administrator Freeze Attempt | Module developer attempts to freeze phase | Access denied error | Pass |
| PP-05 | Read Access to Frozen Phase | Read-only user accesses frozen phase | Access granted successfully | Pass |
| PP-06 | Phase Transition Permission | Module developer initiates phase transition | Transition completed successfully | Pass |

## A.5 Boundary Case Test Cases

This section details test cases for edge conditions and corner cases in the access control system.

Table A.8: Boundary Case Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| BC-01 | No Role Assignment | User with no assigned role attempts access | Access limited to public content | Pass |
| BC-02 | Multiple Role Assignment | User with multiple roles attempts action | Most permissive role takes effect | Pass |
| BC-03 | Role With No Permissions | Assign user to empty role | No permissions granted | Pass |
| BC-04 | Session Timeout Handling | Session expires during operation | User properly redirected to login | Pass |

## A.6 Test Implementation Details

Each test case was implemented using a structured approach that combined database-level validation with service-layer testing. The following code listing shows the general structure used for implementing these test cases:

```
[Test]
public void TestCaseID_Description_ExpectedOutcome()
{
    // Arrange: Set up test environment
    var testUser = CreateTestUser("[UserRole]");
    var testEntity = CreateTestEntity();

    // Configure specific test conditions
    ConfigureTestConditions();

    // Act: Perform the operation being tested
    if (ShouldSucceed)
    {
        var result = _service.PerformOperation(
           ↳testEntity, testUser.UserId);

        // Assert: Verify operation succeeded
        Assert.IsNotNull(result);
        Assert.That(result.Status, Is.EqualTo(
           ↳OperationStatus.Success));

        // Verify database state reflects the change
        var dbEntity = _database.QuerySingleOrDefault<Entity>(
            "SELECT * FROM entities WHERE id = @Id",
            new { Id = testEntity.Id });
        Assert.IsNotNull(dbEntity);
        Assert.That(dbEntity.Property, Is.EqualTo(testEntity.Property));
    }
    else
    {
        // Assert: Verify operation is denied with appropriate error
        var exception = Assert.Throws<PermissionDeniedException>(() =>
            _service.PerformOperation(testEntity, testUser.UserId));
        Assert.That(exception.Message, Contains.Substring("expected error message"));

        // Verify database state was not modified
        var dbEntity = _database.QuerySingleOrDefault<Entity>(
            "SELECT * FROM entities WHERE id = @Id",
            new { Id = testEntity.Id });
        Assert.That(dbEntity, Is.Null().Or.Property("Property")
                               .Not.EqualTo(testEntity.Property));
    }
}
```

Listing A.1: Test Case Implementation Template

This standardized approach ensured consistent validation across all test cases while providing clear evidence of both successful permission grants and appropriate permission denials. Each test verified both the immediate operation result and the resulting database state, ensuring comprehensive validation of the access control system.

## A.7 Role Permission Matrix

Table A.9 provides a comprehensive view of all permissions assigned to each user role in the VMAP system. This matrix formed the basis for the permission validation test cases.

Table A.9: Role Permission Matrix

| **Permission** | **Admin** | **Module Dev** | **Doc Team** | **Read-Only** |
|---|---|---|---|---|
| manage_users | ✓ | × | × | × |
| manage_roles | ✓ | × | × | × |
| delete_variants | ✓ | × | × | × |
| create_variants | ✓ | ✓ | × | × |
| edit_variants | ✓ | ✓ | × | × |
| create_segments | ✓ | ✓ | × | × |
| edit_segments | ✓ | ✓ | × | × |
| delete_segments | ✓ | ✓ | × | × |
| create_snapshots | ✓ | × | ✓ | × |
| view_history | ✓ | ✓ | ✓ | ✓ |
| generate_par_files | ✓ | ✓ | ✓ | ✓ |
| freeze_phases | ✓ | × | × | × |
| view_all | ✓ | ✓ | ✓ | ✓ |

Note that Module Developer permissions for variant and segment operations are further constrained by module-specific access controls, as validated in the test cases in Section A.2.

# B Variant Management Test Cases

This appendix provides a comprehensive listing of all test cases used to validate the variant management functionality implemented in the VMAP database. The test cases are organized by category and include detailed information about test actions, expected outcomes, and test results.

## B.1 Variant Creation Test Cases

This section details the test cases for validating variant creation functionality across different parameter types and constraints.

Table B.1: Variant Creation Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| VC-01 | Basic Variant Creation | Create variant with valid name and code rule | Variant created successfully | Pass |
| VC-02 | Duplicate Variant Name | Create variant with name that already exists in PID | Name uniqueness error | Pass |
| VC-03 | Empty Variant Name | Create variant with empty name | Validation error | Pass |
| VC-04 | Special Characters in Name | Create variant with special characters in name | Variant created successfully | Pass |
| VC-05 | Maximum Name Length | Create variant with 100-character name (maximum length) | Variant created successfully | Pass |
| VC-06 | Exceed Name Length | Create variant with name exceeding 100 characters | Validation error | Pass |
| VC-07 | Valid Code Rule | Create variant with syntactically valid code rule | Variant created successfully | Pass |

*Continued on next page*

Table B.1 – *Continued from previous page*

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| VC-08 | Complex Code Rule | Create variant with complex rule containing multiple operators | Variant created successfully | Pass |
| VC-09 | Invalid PID Reference | Create variant with non-existent PID | Foreign key constraint error | Pass |
| VC-10 | Creation in Frozen Phase | Create variant in a frozen phase | Phase frozen error | Pass |
| VC-11 | Variant in Inactive PID | Create variant for parameter in inactive PID | Validation error | Pass |
| VC-12 | Null Code Rule | Create variant with null code rule | Variant created successfully | Pass |
| VC-13 | Variant Audit Trail | Create variant and verify audit trail | Audit record created correctly | Pass |
| VC-14 | Variant for Boolean Parameter | Create variant for parameter with boolean type | Variant created successfully | Pass |
| VC-15 | Variant for Enum Parameter | Create variant for parameter with enumeration type | Variant created successfully | Pass |
| VC-16 | Concurrent Variant Creation | Create variants concurrently from multiple sessions | All variants created successfully | Pass |
| VC-17 | Transaction Rollback | Begin transaction, create variant, then force rollback | No variant created | Pass |
| VC-18 | Permission Verification | Create variant with insufficient permissions | Permission denied error | Pass |

## B.2 Segment Modification Test Cases

This section details the test cases for validating segment modification functionality across different parameter dimensions and value types.

Table B.2: Segment Creation Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| SC-01 | Create Scalar Segment | Create segment for scalar parameter | Segment created successfully | Pass |
| SC-02 | Create Array Segment (1D) | Create segment for 1D array parameter | Segment created successfully | Pass |
| SC-03 | Create Matrix Segment (2D) | Create segment for 2D matrix parameter | Segment created successfully | Pass |
| SC-04 | Create 3D Array Segment | Create segment for 3D array parameter | Segment created successfully | Pass |
| SC-05 | Invalid Dimension Index | Create segment with out-of-bounds dimension index | Validation error | Pass |
| SC-06 | Invalid Parameter Reference | Create segment with non-existent parameter ID | Foreign key constraint error | Pass |
| SC-07 | Integer Parameter Value | Create segment with integer parameter type | Segment created successfully | Pass |
| SC-08 | Float Parameter Value | Create segment with float parameter type | Segment created successfully | Pass |
| SC-09 | Boolean Parameter Value | Create segment with boolean parameter type | Segment created successfully | Pass |
| SC-10 | Minimum Value Boundary | Create segment with minimum allowed value | Segment created successfully | Pass |
| SC-11 | Maximum Value Boundary | Create segment with maximum allowed value | Segment created successfully | Pass |
| SC-12 | Below Minimum Value | Create segment with value below minimum | Validation error | Pass |
| SC-13 | Above Maximum Value | Create segment with value above maximum | Validation error | Pass |
| SC-14 | Creation in Frozen Phase | Create segment in a frozen phase | Phase frozen error | Pass |

*Continued on next page*

Table B.2 – *Continued from previous page*

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| SC-15 | Duplicate Parameter-Dimension | Create segment for already modified parameter dimension | Unique constraint error | Pass |
| SC-16 | High Precision Value | Create segment with high precision decimal value | Segment created successfully | Pass |

Table B.3: Segment Update Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| SU-01 | Update Scalar Segment | Modify existing scalar segment value | Segment updated successfully | Pass |
| SU-02 | Update 1D Array Element | Modify element in 1D array segment | Segment updated successfully | Pass |
| SU-03 | Update 2D Matrix Element | Modify element in 2D matrix segment | Segment updated successfully | Pass |
| SU-04 | Value Range Verification | Update segment with value outside valid range | Validation error | Pass |
| SU-05 | Update in Frozen Phase | Modify segment in a frozen phase | Phase frozen error | Pass |
| SU-06 | Concurrent Updates | Update same segment from multiple sessions | Last update preserved with proper locking | Pass |
| SU-07 | Update Non-Existent Segment | Update segment that doesn't exist | Not found error | Pass |
| SU-08 | Change to Default Value | Update segment to match default parameter value | Segment updated successfully | Pass |

Table B.4: Segment Deletion Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| SD-01 | Delete Single Segment | Remove existing segment | Segment deleted successfully | Pass |

*Continued on next page*

Table B.4 – *Continued from previous page*

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| SD-02 | Delete Non-Existent Segment | Delete segment that doesn't exist | Not found error | Pass |
| SD-03 | Delete in Frozen Phase | Delete segment in a frozen phase | Phase frozen error | Pass |
| SD-04 | Cascade Delete via Variant | Delete variant and verify segments cascade | All segments deleted | Pass |
| SD-05 | Cascade Delete via Parameter | Delete parameter and verify segments cascade | All segments deleted | Pass |
| SD-06 | Segment Deletion Audit | Delete segment and verify audit trail | Audit record created correctly | Pass |
| SD-07 | Permission Verification | Delete segment with insufficient permissions | Permission denied error | Pass |
| SD-08 | Transaction Rollback | Begin transaction, delete segment, then force rollback | Segment not deleted | Pass |

## B.3 Performance Test Cases

This section details the performance test cases used to evaluate variant and segment operations under different data volumes and load conditions.

Table B.5: Variant and Segment Performance Test Cases

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| VP-01 | Baseline Variant Creation | Create 10 variants and measure time | $<$ 2 seconds total time | Pass |
| VP-02 | Baseline Segment Creation | Create 100 segments and measure time | $<$ 10 seconds total time | Pass |
| VP-03 | High Volume Variant Creation | Create 100 variants for single PID | $<$ 20 seconds total time | Pass |

*Continued on next page*

Table B.5 – *Continued from previous page*

| ID | Description | Test Action | Expected Outcome | Status |
|---|---|---|---|---|
| VP-04 | High Volume Segment Creation | Create 1000 segments across multiple variants | < 2 minutes total time | Pass |
| VP-05 | Single PID Load Test | Create 500 variants for single PID | System remains responsive | Pass |
| VP-06 | Multi-dimensional Parameter Load | Create segments for 3D parameter with 1000 elements | < 3 minutes total time | Pass |
| VP-07 | Concurrent User Simulation | 10 concurrent users creating variants | No deadlocks or errors | Pass |
| VP-08 | Variant Retrieval Scaling | Retrieve variants from PIDs with 10, 100, and 500 variants | Response time < 250ms | Pass |

## B.4 Test Implementation Details

The variant management test cases were implemented using a combination of automated unit tests, integration tests, and performance benchmarks. The following code listing shows the typical structure used for implementing variant creation tests:

```
[Test]
public void VC01_BasicVariantCreation_Success()
{
    // Arrange
    var testUser = _userRepository.GetTestUser("module_developer@example.com");
    var testPid = _pidRepository.GetTestPid();

    var variant = new VariantCreationPayload
    {
        PidId = testPid.PidId,
        EcuId = testPid.EcuId,
        PhaseId = _activePhaseId,
        Name = "Test Variant " + Guid.NewGuid().ToString().Substring(0, 8),
        CodeRule = "A AND (B OR C)"
    };

    // Act
    var result = _variantService.CreateVariant(variant, testUser.UserId);

    // Assert
    Assert.IsNotNull(result);
    Assert.That(result.VariantId, Is.GreaterThan(0));

    // Verify database state
    var dbVariant = _database.QuerySingleOrDefault<Variant>(
        "SELECT * FROM variants WHERE variant_id = @VariantId",
        new { VariantId = result.VariantId });

    Assert.IsNotNull(dbVariant);
    Assert.That(dbVariant.Name, Is.EqualTo(variant.Name));
    Assert.That(dbVariant.CodeRule, Is.EqualTo(variant.CodeRule));
    Assert.That(dbVariant.CreatedBy, Is.EqualTo(testUser.UserId));

    // Verify audit trail
    var auditRecord = _database.QuerySingleOrDefault<ChangeRecord>(
        "SELECT * FROM change_history WHERE entity_type = 'variants' " +
        "AND entity_id = @VariantId AND change_type = 'CREATE'",
        new { VariantId = result.VariantId });

    Assert.IsNotNull(auditRecord);
    Assert.That(auditRecord.UserId, Is.EqualTo(testUser.UserId));
}
```

Listing B.1: Variant Creation Test Implementation Example

Similarly, segment modification tests followed this structure but with appropriate adaptations for the specific operations:

```
[Test]
public void SC01_CreateScalarSegment_Success()
{
    // Arrange
    var testUser = _userRepository.GetTestUser("
        ↳module_developer@example.com");
    var testVariant = _variantRepository.GetTestVariant
        ↳();
    var testParameter = _parameterRepository.
        ↳GetScalarParameter(testVariant.PidId);

    var segment = new SegmentCreationPayload
    {
        VariantId = testVariant.VariantId,
        ParameterId = testParameter.ParameterId,
        DimensionIndex = 0,
        Decimal = 42.5m
    };

    // Act
    var result = _segmentService.CreateSegment(segment,
        ↳ testUser.UserId);

    // Assert
    Assert.IsNotNull(result);
    Assert.That(result.SegmentId, Is.GreaterThan(0));

    // Verify database state
    var dbSegment = _database.QuerySingleOrDefault<
        ↳Segment>(
        "SELECT * FROM segments WHERE segment_id =
            ↳@SegmentId",
        new { SegmentId = result.SegmentId });

    Assert.IsNotNull(dbSegment);
    Assert.That(dbSegment.VariantId, Is.EqualTo(segment
        ↳.VariantId));
    Assert.That(dbSegment.ParameterId, Is.EqualTo(
        ↳segment.ParameterId));
    Assert.That(dbSegment.DimensionIndex, Is.EqualTo(
        ↳segment.DimensionIndex));
    Assert.That(dbSegment.Decimal, Is.EqualTo(segment.
        ↳Decimal));
    Assert.That(dbSegment.CreatedBy, Is.EqualTo(
        ↳testUser.UserId));

    // Verify parameter value is within valid range
    var parameterRange = _database.QuerySingleOrDefault
        ↳<ParameterRange>(
        "SELECT * FROM parameter_values WHERE
            ↳parameter_id = @ParameterId",
        new { ParameterId = testParameter.ParameterId
```

```
            ↳});

    if (parameterRange != null)
    {
        Assert.That(segment.Decimal, Is.
            ↳GreaterThanOrEqualTo(parameterRange.
            ↳ValueRangeBegin));
        Assert.That(segment.Decimal, Is.
            ↳LessThanOrEqualTo(parameterRange.
            ↳ValueRangeEnd));
    }
}
```

Listing B.2: Segment Modification Test Implementation Example

Performance tests were implemented using a benchmarking approach that measured execution time across multiple iterations:

```
[Test]
public void VP01_BaselineVariantCreation_Performance()
{
    // Arrange
    var testUser = _userRepository.GetTestUser("
        ↳module_developer@example.com");
    var testPid = _pidRepository.GetTestPid();
    var variants = new List<VariantCreationPayload>();

    for (int i = 0; i < 10; i++)
    {
        variants.Add(new VariantCreationPayload
        {
            PidId = testPid.PidId,
            EcuId = testPid.EcuId,
            PhaseId = _activePhaseId,
            Name = $"Perf Test Variant {i}_{Guid.
                ↳NewGuid().ToString().Substring(0, 8)}",
            CodeRule = "A AND B"
        });
    }

    // Act
    var stopwatch = new Stopwatch();
    stopwatch.Start();

    foreach (var variant in variants)
    {
        _variantService.CreateVariant(variant, testUser
            ↳.UserId);
    }

    stopwatch.Stop();

    // Assert
    Assert.That(stopwatch.ElapsedMilliseconds, Is.
        ↳LessThan(2000));
    Console.WriteLine($"Time to create 10 variants: {
        ↳stopwatch.ElapsedMilliseconds}ms");
}
```

Listing B.3: Performance Test Implementation Example

This standardized approach ensured comprehensive validation of the variant management functionality while providing detailed performance metrics for system evaluation.

## B.5 Test Environment Configuration

All variant management tests were conducted in a controlled test environment with the following specifications:

- PostgreSQL 17 running on Windows Server 2022
- Database server: 8 vCPUs, 32GB RAM, SSD storage
- Application server: 4 vCPUs, 16GB RAM
- Database containing baseline dataset (20,000 parameters, 188 variants, 28,776 segments)
- Testing conducted with both the baseline dataset and scaled dataset (100,000 parameters, 830 variants, 167,990 segments)
- Network latency between application and database servers < 1ms
- PostgreSQL configuration optimized for test environment with appropriate memory allocation for shared buffers, work memory, and maintenance work memory

The test environment was reset to a known state between test runs using database snapshots, ensuring consistent starting conditions for each test execution.