AN ARCHITECTURE TRADEOFF ANALYSIS OF POSTGRESQL

Xin Shen

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

March 2002

0-612-68479-2

Abstract

An Architecture Tradeoff Analysis of PostgreSQL

Xin Shen

The Architecture Tradeoff Analysis Method(ATAM) was developed by R.Kazman, M.Klein and P.Clements to evaluate early architectural decisions of software development in terms of quality attributes to avoid expensive architectural mistakes.

PostgreSQL developed in the University of California was a pioneer of many modern RDBMS systems. It is now an open source project.

We applied ATAM to the postgreSQL project in light of identifying the architectural features and possible pitfalls in the architecture of similar DBMS systems. The result shows that we identifed some sensitive points, tradeoff points and risks of postgreSQL although we did have some difficulties when extracting precise quality attributes. This work proves the effectiveness of ATAM method and explores the possible way to use ATAM on general purpose software.

Acknowledgements

I would like to thank my supervisor, Dr Gregory Butler, for his patience and valuable guidance.

I had helpful discussions with Xingguo Song.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Software Architecture

Software architecture analysis is a quite new software engineering method compared with other disciplines developed in the last few decades. Before software architecture was recognized as an important way to improve software quality and production, software products had long suffered from low reusability, compatibility and spoiled time and money budgets and many other problems difficult to cure. After the emergence and maturity of object oriented methodology, software people gained a lot of power from analyzing and organizing projects in terms of objects. This new vision of software structures and construction methods boosts the recognition for the importance of a good structure in terms of real objects and their relationships because this good structure will result in more understandable and thus maintainable and reusable software. This experience and insight lead to the evolution to an architecture centric view in software development. A lot of effort has been made to help find the effective methodology to guide the practice of building architectures and the effective process to follow to guarantee the quality of architectures, and also the right notations and tools to help software people present the architectures and communicate with each other

On the other hand, by taking advantage of the facilities provided by the object oriented languages, such as inheritance, polymorphism and delegation, we could optimize software in terms of some mature patterns and solutions. This introduced the concept of design pattern to enhance the reusability of high level design structures and solutions accumulated from experience. Also, the knowledge of some domain experts and professional developers can be shared by building software frameworks, and users of frameworks only need to add customizable modules to produce a fully functional system without a lot of effort. As a higher abstraction than design, architecture also introduced its own reuse artifact, architectural styles, which summarize mature architectural models from expertise. All in all, these evolutions of software engineering after the emergence of object oriented technology made people concentrate on how to construct a good architecture and design in the early phases of a large software project. We have too many examples from many large software projects that faced enormous challenge in the implementation phase because the development team made wrong architectural decisions. The cost for curing such architectural mistakes can be very high. Therefore, we must be able to evaluate the architectural decisions in order to find if they satisfy the system requirements. The demand for evaluation leads to the birth of architectural analysis methods in the software engineering world. Software people need to know what kind of architecture will satisfy their project's functional and nonfunctional requirements, and evaluate it against these requirements in order to avoid mistakes in architecture. This demand stimulates many studies in the software architecture analysis field and leads to many recommended solutions on this issue. ATAM is one of those methods developed in the recent years.

1.2 Architecture Tradeoff Analysis Method(ATAM)

There are many ways to develop an architecture for a new business requirement and start build software. When we build an architecture we need to consider many issues important to the success of the product. These issues can be function, performance or modifiability or some other non-functional concerns. When we have an architecture done for a given project, we need to know if this architecture addresses the requirements properly and solves them completely. This is where ATAM comes in. ATAM is a method to evaluate a software architecture against a given set of business requirements so that we can find the potential risks, sensitivity points and tradeoff points residing in the current architecture. This provides important insights about the problems in the current architecture and makes architects know what should be improved or modified in order to satisfy the users requirements. The basic issue in ATAM is to concretize quality attributes of a given software project. The quality attributes actually are just another name for nonfunctional requirements, such as performance, modifiability, etc. Concretizing quality attributes are actually eliciting use cases under each category of nonfunctional requirement. The evaluation is performed against those concretized quality attributes (use cases) in order to find discrepancies between the implementation and the requirement. The contribution of ATAM is emphasizing the impacts on architecture from correlated factors among different quality attributes. These factors are normally tradeoff points, which makes some architectural decisions critical to success because we need to make a tradeoff between two or more quality attributes. Therefore, ATAM assesses the architecture against many quality attributes at one time so that it can reveal not only the problems with one attribute but also the relationships among the impacts of many of them.

1.3 Contributions of this Thesis

In this thesis, I applied the ATAM method on the analysis of an open source RDBMS system, postgreSQL. There were many difficulties in this analysis. First, the ATAM method is only supposed to be used on the analysis of a new software in the very beginning stage of it. But postgreSQL is now quite mature software with more than one decade's history. So the purpose of the analysis is to get some lessons from such a mature product rather than identifying defects. Even if I identified some defects, it would be too late to correct them. Secondly, ATAM is supposed to be performed by many different stakeholders, for example, final users, architects, developers, managers, etc. That will guarantee we get a complete coverage on use cases. Also the method should be carried out through discussions among these stakeholders in order to stimulate ideas. However, for this analysis, I could not get many valuable points from the stakeholders and discussions among them. Although there were many difficulties, I still carried out the method successfully and obtained some results from it. The lesson I got from my analysis is that applying ATAM method to a single general-purpose software product faces the difficulty of eliciting precise quality attributes. My solution for conquering this difficulty is generalizing the quality attributes and factoring out the common features of DBMS applications, and then I could get quite useful results. However this generalization impairs the precision of the analysis because only the universal, approximate requirements can be used. This imprecision of the analysis undoubtedly reduced the credibility of ATAM on those general systems. From my practice, I found ATAM is better at analyzing application systems with precise and complete requirements than at general systems with vague requirements.

Additionally, in order to perform ATAM, I generated various use cases for postgreSQL in terms of performance, modifiability and recoverability. Furthermore, I constructed the module and runtime views from the source code, documents and manuals of postgreSQL. These results are depicted with UML notations and diagrams. Also, I described most of the mechanisms involved in the implementation of this DBMS system, and most of them follows the standard techniques important for implementing any modern RDBMS systems. So these results will be helpful for understanding and developing such systems.

1.4 Organization of the Thesis

There are 6 chapters in this thesis. The first chapter is a short introduction of the concepts of software architecture, ATAM and our contributions. Chapter 2 introduces the contents of ATAM. Chapter 3 introduces interfaces and internal mechanism of postgreSQL. Chapter 4 lists the main difficulties encountered in the analysis. Chapter 5 is our solutions to the difficulties and the ATAM steps that we performed on postgreSQL. Chapter 6 is the final conclusions we draw from the analysis.

Chapter 2

Architecture Tradeoff Analysis Method

Through many years of evolution, software engineering has evolved into such a stage that architectural issues become a main consideration in the early phase of development. The architecture always has great impact on many aspects of a software, such as performance, extensibility, security, portability, availability, etc. On the other hand, a good architecture will provide the high level reusability, which means it can be migrated into similar applications with reduced development cost. Because architecture is so important for the future's software engineering practice and theory, many people devoted and explored this little known region. One of them is the people from Carnegie Mellon's Software Engineering Institute(SEI), which has a team dedicated to studying the architectural issues in software engineering practices. Around 1999, they issued their years of study in this field and named it ATAM (Architecture Tradeoff Analysis Method), which combined several methods and concepts developed in the recent years in their institute, including the notion of architectural styles, the quality attribute analysis methods, and the Software Architecture Analysis Method(SAAM).

2.1 Software Architecture and 5 Views

Software architecture is the set of descriptions about the organization of components and connectors of a software product. Usually, defining the architecture is carried out after the analysis phase and before the design phase. Architecture serves as a "design plan" and "abstraction" as described in the book "Applied Software architecture" [4]. Although architecture has defined the basic building blocks of the system, the detail of these blocks are not elaborated until the design phase. UML [6] divides software architecture into 5 views. The use case view contains all the use cases describing the system's behavior from the view point of end user. The design view contains subsystems, classes, interfaces and their relations. The process view describes how to distribute the system into processes, threads and their synchronization mechanisms. The implementation view describes the organization of source code files. The deployment view describes how to deploy the software components among hardware platforms.

2.2 Architectural Styles

Architectural styles are patterns found in the solutions for a specific problem. When similar problems occur, these styles can be reused to solve them. They are the highest level of abstractions in program constructions. Common styles are identified in Garlan and Shaw's article [1]. Pipes and filters style is a series of filters connected by pipes, and each filter performs some actions on the data read from the input pipe and writes the result to the output pipe. Data abstraction and object-oriented organization style is to abstract artifacts into objects that encapsulate attributes and operations. Event based and implicit invocation style is organizing functions as the responses for defined events. Layered system style is wrapping and dividing the functionalities into layers and each of them only provides services to the layer above it and receives services from the layer below it. Repositories style is building a shared data structure and loosely coupled agents. Table driven interpreter style is building a virtual machine to execute pseudo-codes. These styles are extracted from many solutions of mature systems. For example, UNIX programs connected through pipes are a piles and filters style, and OSI network model is a layered system style, and Microsoft Windows is an event based and implicit invocation style.

Architectural style	**Quality Attribute**
Synchronization	Performance
Layering	Modifiability
Abstract data repository	Modifiability
Publish/subscribe	Modifiability
Simplex	Availability

Table 1: Architectural Styles and Concerned Quality Attributes

2.3 ABAS

ABAS is Attribute-Based Architectural Style [2]. ABAS associates each concerned architectural style with an analysis framework that reasons this style in terms of one quality attribute. In ABAS, an architectural style is a description of the component types and their topology of a software product. Also, it describes the pattern of how these components interact with each other in terms of data and control.

In ABAS, architectural style is treated like the counterpart of design pattern. Each architectural style is described in a given format to facilitate users to look up and reference. There are five sections, which are problem descriptions, stimulus/response, attributes measures, architectural styles and analysis.

There are five given architectural styles having been elicited and described in some ABAS documents written by Shaw and Garlan. Table 1 lists the architectural styles and their concerned quality attributes.

Associated with each architecture style, there is a reasoning framework developed to help the users analyze the possible problems which could happen when applying this style. There are two kinds of ways to analyze one given architectural style: quantitative one and qualitative one. Associated with ATAM, we would like to use qualitative analysis rather than quantitative one because the architectural information provided in ATAM is always quite rudimentary in the very beginning stage of a project.

Because architectural style represents the very high level view of the system, it also means the very high level and very efficient reuse of previous valuable experiences. This system of style can be expanded to accommodate more situations because there could be many different kinds of projects possible to be carried out. So we could extract or summerize more architectural styles from our own projects and added them into the architectureal style repertoire.

2.4 ATAM Concepts

ATAM is Architecture Tradeoff Analysis Method, which was developed by Rick Kazman, Mark Klein and Paul Clements [3] to evaluate an architecture and identify the potential risks and tradeoffs. There are several concepts important to understand ATAM.

Quality Attributes Quality attributes are the non-functional requirements, such as performance, availability, security, modifiability, etc.

Quality Attribute Characterization In order to elicit those quality attributes, in ATAM, we are asked to first perform quality attribute characterizations. The process of characterization involves three aspects: external stimuli, architectural decisions, and responses. External stimuli are the events that have impacts on the system's one quality attribute. Architectural decisions are made to solve the problems caused by the stimuli. Responses are the consequence got from adopting those architectural decisions.

Quality Attribute Related Questions The direct result of making quality attribute characterization is to stimulate the stakeholders to ask architecture related questions in each category. Those questions can be performance related issues, modifiability related issues, availability related issues or any other properties the stakeholders define and think as important. We collect the questions about the system's quality attributes in terms of stimuli, architectural decisions and response, which provide a concrete structure to elicit those questions. An example of an imaging collection application could contain the following attribute characterization questions in terms of the performance attribute:

- Is the maximum data transfer rate through serial port limited to 57600 bps? (external stimulus)
- What components are involved in the downloading process in response to the event of pressing downloading button by the users? (architectural decisions)
- How long will the decompressing component take when 100 pages of memory are processed? (architectural decisions)
- Is multiprocessing or multithreading used to improve performance, if so how many processes/threads are generated, and what is the policy to assign priorities among them? (architectural decisions)
- What will happen if the mechanism of mulitprocessing could not guarantee a hard time limit of 5 seconds? (Response)

Types of Scenarios There are threee kinds of scenarios in ATAM. They are use case scenarios, growth scenarios, and exploratory scenarios. Use case scenarios cover the typical usage of the system. Growth scenarios describe the potential modification of the given system. Exploratory scenarios are some extreme conditions the system will possibly meet or some stress tests put on the system.

Utility tree Utility tree is a hierarchy of scenarios organized in a tree. These scenarios are divided into several major branches, such as performance, availability and modifiability, etc. On each branch, the scenarios are prioritized leaf nodes. With this organization, stakeholders could easily identify the key factors in terms of each concerned quality attribute. The priorities of each scenario belongs to two categories. The first one is prioritized in terms of importance for the system's success. The second one is prioritized by the difficulties of implementation. Both of the priorities are marked with H(high), M(middle) and L(low).

Sensitivity points, Tradoff points and Risks Sensitivity points are the architectural decisions predominantly influencing one or more quality attributes. Tradeoff points are the sensitivity points having opposite effects on two quality attributes. Risks are the wrong or neglected decisions in the current architecture.

2.5 SEI's ATAM process

SEI recomments nine steps in practising ATAM. These steps are not fixed and ordered. According to different situations, some steps can be skipped, or the order of the steps can be changed. Some steps should be iterated for several times in order to get effective results.

Present the ATAM The organizer of an ATAM introduces concepts and steps of ATAM. The methods such as concretization of quality attributes and generation of utility trees will be presented along with their results.

Present business drivers The organizer presents the business background and motivation of this project. The functional and non-functional requirements and managerial factors are also elicited.

Present architecture The architects present technical constraints and basic architectural solutions adapted to fulfill these constrains.

Identify architectural approaches The architects identify the main architectural approaches and architectural styles expected to be used in the project.

Generate quality attribute utility tree The evaluation team along with managers and customer representatives identify the relavant scenarios and prioritize them in a utility tree.

Analyze architectural approaches The architects analyze the architectural decisions in respect to each high-priority utility tree requirement. The aim of this analysis is to find architectural decisions that are critical to meeting the requirements. At the same time, the architectural decisions and styles along with sensitivity points, tradeoff points and risks are recorded into some documents.

Brainstorm and prioritize scenarios An expanded group of stakeholders generate the scenarios for the system and prioritize them. The result is compared with the utility tree generated by architects. The unidentified factors in the previous steps will be found.

Analyze architectural approaches If some new high-priority scenarios are identified in the former step, architects need to analyze the architectural decisions against the new scenarios to make sure there are architectural decisions addressing the new scenarios properly. If it is not the case, some new risks are identified.

Present results The final results will be a series of documents recording the requirement scenarios, architectural decisions, sensitivity points, tradeoff points and risks found in the ATAM analysis. In the same time, some mitigation methods will be provided if the evaluation team have natural solutions for the problems they found.

2.6 Our Approach

The SEI's ATAM process is designated for the practice with a group of stakeholders in a meeting. In our project, we perform it with one or two people without any organized meeting, so we abstracted it into fewer steps. The basic steps to utilize the method are:

1. Describe system requirements and business drivers
2. Present the architectural decisions addressing those requirements and if possible, try to identify the architectural styles employed.
3. Generate the scenarios by stakeholders in order to cover the main use cases of the system and then fill the scenarios into the utility tree.
4. Evaluate architectural decisions against the utility tree in order to find the architectural defects.
5. Iterate these steps for several times and maybe with different stakeholders to join in each iteration until organizers are satisfied with the coverage and result. In this process, the scenarios are also prioritized and the most important ones are identified.

The output of the ATAM method is the potential risks, sensitivity points and tradeoff points in the system's architecture. A sensitivity point is one architectural decision that has essential influence on satisfying one or more requirements. A tradeoff point is one architectural decision that has opposite effects on several different requirements.

Risks are some problematic decisions, which can be the lack of architectural decisions addressing some requirements or the ones that were not properly done. Those analysis results will be fed back to the architects and development teams in order to correct or make improvements on the architecture. This is very cost effective because these defects are identified in the very beginning phase of a system development before a lot of effort has been made.

We will describe each step in detail in the following sections.

2.7 Step One: Describe System Requirements and Business Drivers

This step's purpose is to let all stakeholders know why we need to develop this system. This step mainly involves presenting the system requirements in terms of both functional and non-functional ones. Also, we want to present the technical, managerial, economic, or political constraints of the project and all the other important factors having impacts on constructing this system. From the above description, we could elicit the first version or view of our system's non-functional requirements in terms of performance, modifiability, availability, etc. However, we could not cover every corner of the system with just one iteration. This is a traditional problem in software engineering, how do we discover and describe the system's requirement in an accurate and complete form. The answer for this question in many software engineering books is almost the same: do it again and again, we use many iterations to achieve the expected accuracy and coverage. The recent development in object oriented paradigm contributes another solution, use case based requirement elicitation. In ATAM both of these ideas are adapted to solve the same problem. The detailed description about how to apply those ideas in ATAM approach will be discussed later in the section about scenarios.

2.8 Step Two: Present Architectural Decisions

Architectural decisions are the decisions made by the project's architects thus far. They are the perception of the architects about how the final product will work. They are generated by keeping in mind that those architectural decisions should satisfy the requirements although it does not mean they can always achieve it. Those architectural decisions should also be organized into some predefined categories as listed in the following section. The contents of this list can be flexible in order to accommodate different kinds of systems. Additionally, if the architecture is identified as the occurrence of a specific architectural style, the analysis can be carried out based on the existing materials of that architectural style.

2.8.1 Contents of Architecture Presentation

The presentation should cover the issues of hardware platforms, OS, middlewares, interactive systems, architectural approaches, etc. The following is a more detailed list.

- Driving architectural requirements
- High level architectural view
- Functional requirements
- Module/layer/subsystem
- Process/thread model
- Hardware and OS
- Architectural styles employed
- Uses of COTS
- 1-3 most important use case scenarios
- 1-3 most important change scenarios

2.8.2 Identify Architectural Styles

The result of the information obtained from the former step's attribute characterization can be fed into the architectural style model to carry out the necessary analysis. One architectural style normally only addresses one attribute (performance, modifiability, etc.). For each style, the analysis is based on the stimulus and response of the given problem, and what factors will have predominant influence on meeting the constraints. The stimulus of this attribute's characterization is the input conditions of the analysis of the given style. The analysis is performed by using some mature analytic models, such as queue, rate monotonic analysis, Markov modeling, etc. in order to find out the worst case of the response. In ATAM, most analyses are doing quite simple calculations rather than building a complex model. The last part of the analysis is some heuristic questions concerning this style and those factors that have predominant influence on the system's attribute.

2.9 Step Three: Generate the Scenarios and the Utility Tree

Non-functional system requirements, such as performance, modifiability, availability, are just dry words. In ATAM, these requirements need to be further concretized with scenarios in the given application context. Those scenarios are made by varieties of stakeholders in order to cover every usage of the target system. Scenarios actually are just use cases in the object oriented world. There are three kinds of scenarios in ATAM. All of these scenarios can be divided into different categories, such as performance, modifiability, availability, usability, security, etc. Therefore, all the scenarios can be organized into a tree structure whose primary branches are performance, availability and any other quality attributes and whose leaves are concrete scenarios. With this approach, the requirements are not dry words any more, they are concrete scenarios representing the expected system's requirements from the user's point of view. Furthermore, the scenarios are also prioritized as high, medium and low in terms of the importance and difficulty for the success of the system. Obviously, the high priority scenario for the success of the system will be studied and reasoned with greater details in the later steps of ATAM. The concretization of system requirements always yields some discovery on the requirements themselves because some new requirements will be discovered and some will be ranked as more important than they were. Figure 1 is a utility tree for an imaging collection program.

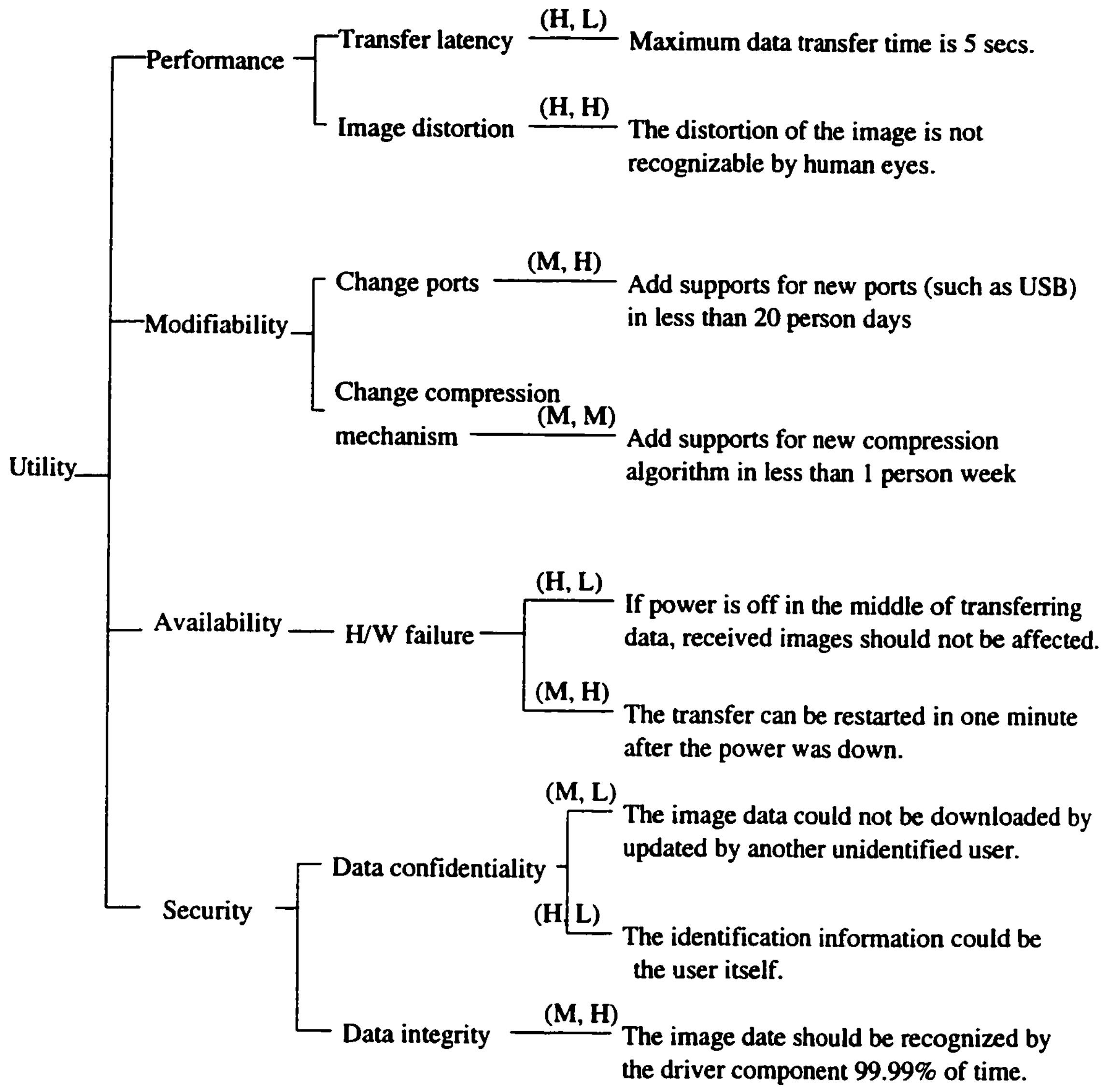

Utility
Performance
Transfer latency
(H, L)
Maximum data transfer time is 5 secs.
Image distortion
(H, H)
The distortion of the image is not recognizable by human eyes.
Modifiability
Change ports
(M, H)
Add supports for new ports (such as USB) in less than 20 person days
Change compression mechanism
(M, M)
Add supports for new compression algorithm in less than 1 person week
Availability
H/W failure
(H, L)
If power is off in the middle of transferring data, received images should not be affected.
(M, H)
The transfer can be restarted in one minute after the power was down.
Security
Data confidentiality
(M, L)
The image data could not be downloaded by updated by another unidentified user.
(H, L)
The identification information could be the user itself.
Data integrity
(M, H)
The image date should be recognized by the driver component 99.99% of time.

Figure 1: An Example Utility Tree

2.10 Step Four: Evaluate Architectural Decisions

In this step, architects present the architectural decisions having been made to address a given attribute. Because these attributes have been concretized in the former step, we could use the utility tree's leave nodes(scenarios) as the criteria for assessing the architectural decisions made thus far. As we have mentioned in the last step, the utility tree is prioritized; therefore the highest prioritized factors will be the emphasis of the evaluation. Also, the architectural decision will be described in terms of components, connectors, etc. and will be recorded in the form of diagram, which provides the basic material clarifying architects' design so far. In the mean time, some analysis is carried out in order to find sensitivity points, tradeoff points and therefore potential risks. The analysis is started from asking some attribute related questions against a given architectural decision, and then based on the answer from the architects, some discussions will be put forward and some reasoning will be made by applying simple calculations and modeling. The result will be a list of sensitivity points, tradeoff points lying in the proposed architecture and will be recorded into a template document as a report of this phase and the starting point of the next phase. At the end of this phase, architects will know the most important aspect of the entire architecture and the rationale having been made so far and potential risks, sensitivity points and tradeoff points in this architecture.

2.11 Step Five: Iterate for Several Turns

Like any software engineering process, ATAM method is an iteration process. The purpose for iteration is to cover any possible architectural issues that will impose big influence on the final product's quality attributes. Therefore, ATAM method suggests to get representative users from every possible usage area to join the process of eliciting scenarios in order to cover all the possible use cases after all the iterations. Brainstorming is recommended as a way to stimulate users to create scenarios. Those new scenarios are expected to be use case scenarios, growth scenarios and exploratory scenarios. A voting process prioritizes these new scenarios and selects the most important ones to combine into the utility tree. The order and priority of these scenarios might be different from the ones already in the utility tree. However, this is just the merit of the iteration, identifying and correcting the existing deviation of the former iterations. This difference is the possible place where architects would have misunderstood the real usage or requirements, which are clarified by end users' new version of scenarios. After adding the new high priority scenario into the utility tree, the architects will try to reason the current architecture against those new important scenarios in order to make sure the components of the current architecture has correct solutions addressing these new scenarios. If the reason shows that the new scenarios have been fully supported by the current architecture, the architecture is fine thus far. Otherwise, new architectural problems have been revealed, and then the former analysis based on the old utility tree is not correct, therefore the analysis must be redone based on the new utility tree.

2.12 The ATAM Result

The ATAM result finally should be present in a report form, which includes a list of findings of the process. These findings include the architectural styles/decisions, the scenarios and their prioritizations, the attribute specific questions, the utility tree, the risks, the sensitivity points and tradeoff points. If possible, we could offer mitigation method although this is not mandatory because the emphasis is on looking for defects in the architecture instead of solving them all together.

2.13 Deploy ATAM in a Real Project

Deploying ATAM in a new project can be done in two phases. The first phase only involves a small set of people coming from the architect team and evaluation team. The purpose of the first phase is to identify the architectural patterns and basic scenarios, which compose of the basic materials for next phase's actions. Between the first phase and the second phase, there is a hiatus of few days or few months. In that period, the materials about the ATAM method and the materials generated in the first phase will be dispatched to the stakeholders expected to attend next phase's actions. Normally the second phase is a kind of meeting of selected stakeholders. The duration will be two or three days depending on the size and complexity of the target project. These stakeholders will go through the every step of ATAM, and generate the final results in a report format.

Chapter 3

PostgreSQL

PostgreSQL is an advanced open source ORDBMS system primarily used in UNIX/Linux operating systems. It supports many modern ORDBMS features, such as transaction, sub-selection, user defined types, inheritance, etc. Those features make postgreSQL the most powerful ORDBMS in the open source world so far. This is the reason why we choose postgreSQL as the target of our ATAM analysis. In this chapter, we introduce the basic aspects of postgreSQL. We will present the history and features of postgreSQL and the SQL language. Also, we will describe the basic mechanisms of implementing query processing, transactions and concurrency control, etc. All the description is useful for clarifying architectural decisions. So the contents of this chapter is one part of architectural presentation of step 2 of chapter 5.

3.1 History of PostgreSQL

PostgreSQL originated from the University of California at Berkeley in about 1986. The original name for this project is postgres [7] [9]. In 1995, two master students Andrew Yu and Jolly Chen added SQL support on the backend of postgres and named it postgres95. Afterwards, open source community took this project and added many more features into the system and optimized its performance, and also changed its name to postgreSQL.

The original Berkeley project aimed at integrating and experimenting some next generation relational database technologies, such as defining new types, functional attributes and inheritance. These features were used to provide support for large and complex objects. It was also developed to support a rules system, which provides constraints that the users can put on the data and relations. Other goals for that project were providing simple and fast crash recovery and utilizing some "new" hardware technology, such as writable optical disk, etc. Postgres development was carried on for about 8 years, in the third year, they delivered the first demo version, which is a mixture of C code and Lisp code. Due to the performance issues, Lisp code was replaced with C code and in about 1990, the first completely C version was released with about 90,000 lines of code. This system was developed and run on Sun's BSD UNIX version. In the following years until 1994, several versions were released, some bugs were fixed and a lot of tuning and optimization were done. The code size grew to about 180,000 lines at the second version. All of these versions were freely available and the university maintained the system until 1994 and then closed any further work on this project. In 1994, two Berkeley master students took this project and added the SQL data model and interface into the old system, which previously only supported POSTQUEL. After 1995, the open source community took this SQL version, postgres95, and enhanced its functionality and optimized its performance for mission critical applications. They gave it a new name postgreSQL. They also followed the version number from postgres, whose version was 4.2 when terminated in 1994, and then the first version of postgreSQL was 6.0. In the next 5-6 years, the open source community made many improvements in the functionality, such as adding multi-version concurrency control, write ahead logging, subselect, trigger, additional SQL compliant features, more built-in types. Performance improved 20-40%, and backend startup time decreased by 80%. At the time this thesis is written, version 7.1.2 is the newest one [10]. The code size is about 250,000 lines. Now this database server has been adapted in many commercial and research environment.

3.2 Conceptual View

3.2.1 Basic Scenarios

The typical scenario of running postgreSQL system can be described as a client server architecture, in which each client process will be served by one server process. The client process usually is the application program and the client call library, through which application program sends translated SQL commands to the server process.

The server process gets the SQL commands and executes the commands and then sends back the retrieved data to the client process. All the database manipulations are carried out on the server process. Besides the client and server processes, there is a standalone postmaster process running on the server machine. The postmaster listens on a system-defined network port, detecting the incoming request for connections. After detecting one request, it will launch one server process and then pass the communication to the server process. From then on, the postmaster will not interfere with the communication between the client and the server. The postmaster also serves to create the communication channels between the server processes because one user's operations on a database will affect other users, so this information has to be passed between the server processes. Additionally, the postmaster process always launch system service processesperiodically as child processes in order to accomplish some system service functions, such as checkpointing. The server processes and the postmaster must be running in the same machine because servers are forked off by postmaster. However, the client and the server can be running on different machines although the server might not access the files the clients are able to access in case they are running on different machines. Obviously, this architecture makes it possible that one client can connect to multiple servers on multiple different machines although these connections can not proceed in a real concurrent fashion because there is not multi threading support on either client or server side. In the following sections, we will use front end as a synonym of the client process, and back end a synonym of server process.

3.2.2 The Procedure of Processing a Query

The backend first builds a parse tree for the incoming SQL command. If the syntax of the SQL command is correct, the parse tree will be processed by the rewrite system, which will check the rules system and transform the parse tree to a form obeying the defined rules. If the query is against a view, the rewrite system will rewrite the query against the base table by applying the view description of the view. Then the planner or optimizer will take the rewritten query tree and generate all the possible paths leading to the query result. Then those paths' cost are compared in order to find the optimized path. The executor will carry out the query following the optimized path by calling the functions of the storage management system and retrieving tuples by performing sequential scan, sort and join etc.

3.3 An Introduction to SQL Language

The SQL language is the interface of postgreSQL. It defines the basic functional requirements for this RDBMS system. So we give a basic description of this interface.

3.3.1 Select Statement

The most powerful construct in SQL language is SELECT statement, with which the user program can perform all the relational operations, such as projection, production, join, union, intersect, difference, on the target tables. Some SQL select examples:

- Simple selection:

```
SELECT * FROM SALESDEP
WHERE SALARY >= 50000;
```

- Projection:

```
SELECT EMPNAME, SALARY FROM  SALESDEP
WHERE SALARY >= 50000;
```

- Production:

```
SELECT * FROM SALESDEP,  COMPANY ;
```

- Join:

```
SELECT S.SNAME, C.ADDRESS
FROM SALESDEP S, COMPANY C
WHERE S.COMPANY=C.CNAME ;
```

- Union:

```
SELECT S.SNAME, S.SALARY FROM SALESDEP S
WHERE SNAME = 'Tom'
UNION
SELECT S.SNAME, S.SALARY FROM SALESDEP S
WHERE SNAME = 'Andrew";
```

- Difference:

```
SELECT S.SNAME, S.SALARY FROM SALESDEP S
WHERE SALARY > 50000
EXCEPT
SELECT S.SNAME, S.SALARY FROM SALESDEP S
WHERE SALARY > 60000;
```

- Intersect:

```
SELECT S.SNAME, S.SALARY FROM SALESDEP S
WHERE SALARY > 50000
INTERSECT
SELECT S.SNAME, S.SALARY FROM SALESDEP
WHERE SALARY < 60000;
```

Also, the SQL92 includes many additional powerful features, such as sub selection, aggregation. Some examples:

- Sub selection in the where clause:

```
SELECT * FROM SALESDEP
WHERE SALARY > (SELECT SALARY
FROM SALESDEP
WHERE SNAME = 'Tom');
```

- Sub selection in the from clause:

```
SELECT AVG(SUBTABLE.SALARY)
FROM (SELECT SNAME, SALARY
FROM SALESDEP
WHERE SALARY > 50000);
```

- Aggregation:

```
SELECT S.SALARY COUNT(S.SALARY) FROM SALESDEP S
WHERE SALARY > 50000
GROUP BY S.SALARY
HAVING COUNT(S.SALARY) > 5;
```

3.3.2 Data Management Statements

PostgreSQL supports other table management commands, such as CREATE TABLE, INSERT, UPDATE, DELETE, DROP TABLE, etc. The similar management operations on views and triggers, such as CREATE VIEW, CREATE TRIGGER, DROP VIEW, DROP TRIGGER, etc, are also supported. Some examples:

- Create table:

```
CREATE TABLE SALESDEP
( SNO     INTEGER,
  SNAME   VARCHAR(20),
  SALARY  INTEGER,
  COMPANY VARCHAR(20) );
```

- Create index:

```
CREATE INDEX SINDEX ON SALESDEP ( SNO);
```

- Create view:

```
CREATE VIEW SVIEW AS
SELECT * FROM SALESDEP
WHERE SALARY > 50000;
```

- Drops:

```
DROP TABLE SALESDEP;
DROP  INDEX  SINDEX;
DROP  VIEW SVIEW;
```

- Insert records:

```
INSERT INTO SALESDEP
VALUES( 1, 'Tom', 'Some Company');
```

- Update records:

```
UPDATE SALESDEP
SET COMPANY = 'Another Company'
WHERE SNAME = 'Tom';
```

- Delete records:

```
DELETE FROM SALESDEP
WHERE SNAME = 'Tom';
```

3.3.3 Rules System

Another important modern DBMS feature is rules system, which puts some constraints on the attributes of a table or the relationship between attributes. For example, in an EMPLOYEE table, EMPLOYEE(EMPNAME, AGE, SALARY). We require the salary for each employee older than 40 to be 50, 000. Rule system can put this constraint on the column SALARY of the table EMPLOYEE. The rule should have the effect of :

```
UPDATE EMPLOYEE SET SALARY = 50000 WHERE AGE > 40;
```

Where some new records are inserted into the EMPLOYEE table, the age for this new record will be checked, and then the system will decide if the value of the salary satisfies the condition for forcing an update. The rules system is also the mechanism allowing the implementation of triggers and views possible. The triggers are just some rules defined in response to some events, such as insert, update, etc. The view is also defined as a rule executed by system to make a selection or projection on a base table or several base tables. Therefore, whenever an operation is requested on a view, it will be transformed by applying the given rules into the base table and then perform the real operation on the underlying base tables. In postgreSQL, the rules are defined as some SQL operations accompanying a given action to satisfy a given constraint. An example:

```
CREATE RULE INS_RULE AS
ON INSERT TO SALESDEP
DO INSTEAD
NOTHING;
```

The above example just disables the INSERT action on the table SALESDEP. Another example:

```
CREATE RULE  INS_RULE AS
ON UPDATE TO SALEDEP
DO
INSERT INTO SALELOG (old.SNAME, old.SNO);
```

This example illustrates adding a update rule, which inserts the old value of updated tuple into a log table.

3.3.4 Advanced Features

PostgreSQL supports some advanced features beyond the SQL 92 standard and also supports some object oriented database features. PostgreSQL supports inheritance in defining tables, storage of large objects such as image files, and subdivision of columns.

3.4 Backend Processes a SQL Command

The backend of a postgreSQL server is a process that keeps listening on one port for requests from the client side. The client side sends a text string containing some SQL constructs to the server through the port. When the server gets the SQL construct, it will parse it according to the grammar. If the SQL command is a correct one, the result of the parsing will be a parse tree representing the SQL command. Each node of this tree will represent a structure or entity in the command. Also, in the parsing process, the parser will store the retrieved values into the correspondent nodes as attributes in preparation for the following steps. PostgreSQL parser was built by Lex and Yacc [17]. The grammar for SQL is described in terms of the Yacc grammar descriptions. There are two parts in a Yacc grammar description, the grammar itself and the actions following the grammar. The actions written in

C are used to construct the parse tree and to fill in correspondent information. In the parse tree building process, there are no operations on the underlying physical storage structures because the parsing is done without the knowledge of the target object's states, such as if the transaction has aborted, etc. The parser program is generated automatically by Lex and Yacc from the description files.

3.4.1 The Execution Path Generation

The parse tree generated by SQL parser will be rewritten by applying rules. After that, the execution plan will be generated based on the rewritten tree. First of all, all the possible paths being able to carry out the query will be generated together with the cost it will take. We should notice that every SQL actions regardless of whether they are UPDATES, INSERTS or SELECT will be carried out as a query(SELECT) because they need to know where to find the tuple and then be able to do the appropriate actions. If the query is done on a single table, it is much simpler. The system only needs to consider the costs with an index and without it. However, if the query is done by joining multiple tables, there are three possible ways to implement a execution. They are nested loop, merge join and hash join. The nested join sequentially scans another file for each tuple in one file. If there are M tuples in the A file, and N tuples in the B file, then A join B will have M times scan of the B files. For merge join, the two tables must be first sorted, and then the join is performed. With the former example, only one scan for each file is needed. For hash join, the second table's tuples are hashed. The first file is sequentially scanned, looking for the corresponding tuples in the second table only involves looking up the hash table of the second table, which guarantees only pages containing these tuples will be accessed. Therefore, for a join table action, three execution paths will be generated with their costs, then the costs are compared with each other in order to find the most efficient one. The cost includes two parts: the startup cost and total cost. The startup cost includes the time the program must take before performing the real operation on it. For example, in the case of merge join, the startup cost includes the time consumed by sorting the two tables. The cost values are estimated based on the times of disk page accesses and some other heuristics. The final path is an action tree with each node representing an action, for example, nested loop joining table A and B, or sequential scan of table A, etc.

3.4.2 Optimization and Execution Plan Generation

The rewritten parse tree is handed to the optimizer and the optimizer will generate different execution paths for a given query. The optimizer creates one structure for each table involved in the query. All the possible paths for a given operation on this table will be recorded in this structure. If only one table is involved, the optimizer chooses the optimized path in this structure as the execution path for the query and passes it to the executor. If multiple tables are involved, the optimizer will first generate one structure for each table, and then it will create a join structure to record the possible combinations of joining tables among all these tables. PostgreSQL does not exhaust all the possible combinations for joining tables. The evaluated join combinations are derived from the SQL statement after the WHERE keyword. All the possible paths for multiple table join will be recorded in a list of the join structure. The final plan will be the optimized one among all of these paths.

3.4.3 Executor Executes a Plan

The plan for a given query is stored in a tree data structure. Each node in this tree represents one action, such as joining tables, sequentially scanning tables, etc. The executor follows the route from root nodes down to the leaves. In each path node, the executor will call the corresponding access routine to perform the actions recorded in this node. After the executor reaches the leaf, there will be a tuple returned. Repeating these steps, the executor will fetch all the tuples satisfying the query conditions one by one.

3.5 Transaction Support

In this section, we will discuss how postgreSQL implements transactions and its crash recovery strategy. Also, we will discuss how it keeps the data consistency among concurrent transactions and its multi-version concurrency implementation.

3.5.1 Transaction and Crash Recovery

In a modern DBMS, the transaction concept is well supported. A transaction is an atomic action the system performs in the database. Atomic means not divisible, which means the effect of the action should be fully reflected on the database, or not at all. If the effect of only a part of the transaction was seen, the database would be in an inconsistent state. Therefore, every transaction will transform the system from one stable state to another stable state. In postgreSQL, the system by default starts a transaction before executing a query and closes it after finishing. Also, the system provides explicit commands, BEGIN WORK and COMMIT WORK to explicitly start and close a transaction. This is advantageous because in many cases, many queries are performed together to guarantee the system is in a consistent state. Before a transaction is committed, the system is in a series of unstable(active) states whose effects are not guaranteed to be kept in stable storage. Consequently, an uncommitted transaction can be rolled back to a former committed state, whereas a committed one could not be rolled back. There is a ROLLBACK command available in postgreSQL. Additionally, postgreSQL provides the command ABORT WORK to drop a transaction in the middle.

PostgreSQL uses write ahead logging (WAL) [14] to implement transactions. When doing any updates on the real data, the system will record the operation in a system log, which is always written into the stable storage prior to the real data. Every action on the real data should be recorded in the log file. The values put into the log should contain the old value of each data item and the new value. If the user aborted a transaction in the middle, the log file will be discarded, and then both the log and the data will not be written into the stable storage. If the system crashes, the system must first undo all the transactions that are not recorded as committed in the reverse order, and then the system must redo all the committed transactions when it is brought up again. So in the both cases, the system's data in the stable storage only reflects the result of committed transactions. Another advantage of WAL is that after a transaction is committed and the log file is written, the system does not need to update the real data immediately. Only when the system needs more buffers will the real data be written into the stable storage. This reduces the disk accesses. Additionally, the log file is written sequentially, and the data file is written randomly, therefore writing to the log is much cheaper than writing to the data file. This is another advantage of WAL.

3.5.2 Concurrency Support

PostgreSQL uses multi-Version concurrency control to support concurrent transactions. Whenever a transaction writes or updates a tuple, a new version of the tuple will be inserted. The old version of the tuple is still visible to the previous transactions. Therefore, there is no need to block writing or updating of a tuple when some others are reading it. This approach reduces locking operations significantly. However, postgreSQL still supports a lot of types of locks on the database items, such as tables, rows. Because postgresql can not guarantee the tuples selected by a transaction will not be deleted or updated before this transaction is committed, it must let users to do that by themselves. PostgreSQL provides explicit LOCK TABLE statement. By using this statement, users can prevent others from updating or deleting tuples when they are doing queries.

3.5.3 SQL92 Concurrency Requirements

In SQL92 [10], three types of phenomena must be prevented in concurrent transactions. They are:

- dirty reads: A transaction reads some data written by other concurrent transaction.
- non-repeatable reads: At two occurrences, data read by a transaction are different because other concurrent transactions have committed recently.
- phantom reads: At two occurrences, the same query returned different result sets because other concurrent transactions have committed recently.

There are four concurrency levels according to the phenomena they can prevent.

We should notice that even with the Serializable level concurrency supports, the first abnormality still can't be eliminated. That means two or more concurrent executing transactions can cover the written values of each other causing lost updating effect.

PostgreSQL supports two concurrency levels: Read committed and Serializable. The Read committed level is the default one, with which two reads(SELECTs) will see different values if there is a transaction in between updated the data and committed.

Concurrency Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible
Repeatable read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

Table 2: The Relationship Between Concurrency Levels and Data Consistency

The Serializable level can be enforced in the beginning of a transaction. With the Serializable level, a transaction will not see any updates by other transactions in the lifetime of its own. Because even with Serailizable level supports, the system still can't prevent one transaction from dropping caused by another concurrent transaction writing to the same place, cautious applications should use explicit LOCK TABLE or SELECT FOR UPDATE commands to enforce the consistency and updating success among concurrent transactions.

3.5.4 Multi-Version Concurrency Implementation

PostgreSQL uses multi version concurrency control to implement the concurrency functionality. Each row of a table have multi versions of tuples stored in the table. Each tuple's header has the following fields:

- *xmin*: transaction id of INSERT transaction
- *xmax*: transaction id of UPDATE/DELETE transaction
- *forward link*: pointer to the newer version

If the *xmin* is an ID of a transaction that is committed and *xmax* is NULL or an id of a transaction that is not committed, this tuple will be visible by the current transaction. If *xmax* is not NULL, this tuple has been deleted or updated. If the tuple is updated, the *forward link* will point to the new tuple. Following the *forward link*, the newest instance of a record always can be found. With this approach, the most recently committed transaction will be always visible by later transactions. On the other hand, if a transaction is not committed, its update will not be seen by the others. Also, updates are done by appending new tuples, so the old tuple's value is not overwritten, other concurrent transactions still can see the values prior to the newest value. So far, the Read committed level is implemented. The Serializable level is implemented by remembering the transactions which has been started (not committed) when one backend starts its own transaction. These already started transactions will not be considered as committed by the new transaction even though they do that afterwards. Also, those transactions with an ID greater than the current transaction will not be considered as committed because they are the newer transactions than the current one. Only the version of the tuple satisfied with these restrictions when the transaction is started up will be considered as the current visible tuple of the current transaction. Now two concurrency requirements are satisfied.

When concurrent transactions are running on the Read committed level. When a transaction updates the value of one row, another transaction updating the row thereafter should wait for the first transaction commits or aborts. If the first transaction commits, then the second transaction will rewrite the value updated by the first transaction. If the first transaction aborts, the value will be rolled back to the original one and then the second transaction will write the new value.

If the transactions are running on the Serializable level, after the first transaction updates one row, the second transaction also must wait for the first one commits or aborts. If the first transaction commits, the second one will abort because updating the value will cover the last update just made by the first transaction. If the first transaction aborts, the value will be rolled back to the original one, so the second transaction can update it.

3.6 Miscellaneous Utilities

For a DBMS system, the utilities of managing memory buffers and access strategies are alway important parts. In this section, we discuss how postgreSQL implements locks and pins on its data, and how postgreSQL manages the client accesses on its buffer. Also, we discuss the role of postmaster process and the implementation of the system catalog, an important meta-information database.

3.6.1 Implementation of Locks

PostgreSQL uses a hash table to store the locks held on a given object. The hash table is keyed by the object type and id. A transaction checks if a conflicting lock is held on an object by looking up the hash table. If it is, this transaction must wait on a semaphore. After the lock is released by other transactions, the waiting transaction will be wakened up.

3.6.2 The Buffer Management Mechanism

PostgreSQL keeps all the relation data or tuples shared through the buffer residing in the memory. The buffers are implemented by shared memory provided by operating systems and created by the postmaster when the system was first brought up. The size of the shared memory can be customized by giving some compiling parameters. The buffer is shared by multiple backend processes as the communication and consistency facility for the whole database. Whenever a backend wants to fetch some tuples from the stable storage device, it will first look up the memory buffer in order to find if there are intended tuples already loaded in the shared buffer. If there are no buffers hit by the searching conditions, the backend will call storage management routines to load the physical page into the buffer. After a buffer is loaded, the backend process will pin the page first, and according to the operations the backend will lock the buffer page with correspondent locks, such as shared lock or exclusive lock. Pinning a buffer page is the minimum requirement for doing anything on the buffer page. Otherwise, the page risks being deleted from the memory. Holding a pin on a page for a long interval is not a bad idea because some operations such as outer join have such a need. Also, many pins can be applied to a buffer page at one instant because many processes are able to access one page concurrently. Locking is a different case compared with pinning. A lock should be held for a short term. When a process wants to update or remove a tuple it must hold an exclusive lock on the buffer. When a process only wants to update the status bits of a tuple, it only needs to hold a shared lock on it.

3.6.3 The Daemon Process Postmaster

The postmaster is the first process forked off from the main routine, which is the daemon process living forever except when shut down by fatal errors or the system administrator. The postmaster will set up the shared memory and semaphore pool used by lock management. After all of these things are done, the postmaster will enter a loop listening on a TCP/IP port and waiting for incoming requests from client processes. Whenever some connection requests come in, the postmaster will fork off a backend process to deal with the request. The new generated backend process will do the authentication and then decide if it accepts the request and sets up the connection with the client side. One thing should be noted is that the postmaster is not involved in using shared memory and semaphores because doing that will risk itself in the case of the misused shared resources and make the system crash. The postmaster also runs some checkpoint processes with a fixed time interval. Also, the postmaster process is responsible for shutting down the whole system and canceling connection requests.

3.6.4 System Catalog

The meta-information about the tables, indexes, triggers, functions, rules, views, types, etc. is stored under one table of the system catalog database. DBMS uses the information stored in this database to keep the information about the tables and their definitions and the functions, operators defined by users and the system, etc. One important use of the system catalog is recording the types the system supports. PostgreSQL does not make any assumption about its base types. Each time it finds some type, it will look up the system catalog, *pg_type*, to get the corresponding information about this type. This mechanism makes adding new types into the system extremely easy and flexible. Also, new index access methods, new functions and new operators can be added dynamically into the system even when the system is running.

Chapter 4

Difficulties of Applying ATAM

In the process of applying ATAM to the postgreSQL project, four categories of difficulties were encountered. They were:

1. Using ATAM to evaluate a single software instead of a complete system.
2. Restrictions from applying ATAM to postgreSQL
3. Some deficiencies of postgreSQL project itself, especially it's lacking architectural level documents.
4. The incompatibility between the open source development and the ATAM working principle.

This chapter describes them separately.

4.1 Using ATAM to Evaluate a Single Software

A software product is always working along with a lot of other software components and hardware components. Evaluating software such as postgreSQL without a runtime context is impossible. In the step of presenting the architecture of ATAM, many hardware and system factors such as the CPU speed, storage devices, network and communication devices, operating systems, middleware are required. These factors are only meaningful for a whole system composing hardware and software components. For a software only DBMS system, postgreSQL can not give any presetting hardware/operating system conditions because theoretically it can be running above any hardware and operating systems. Therefore, we could not generate a "software only" requirements for postgreSQL. For example, the response time of postgreSQL might be related to the cooperating application software components and the operating system, and additionally the response time of an operating system might be related to the scheduling algorithm, clock, task slice, hardware speed, etc. This difficulty can be elaborated in two aspects, other application software components and the hardware/system software setting.

Application Software Components An application system always has several components, and only one of them is the database and DBMS system. An application implementing a particular functionality always contains many components working together to carry out the intended functionality. For example, an online shopping system probably uses CGI or server side scripts and web servers and databases. The performance or availability or modifiability issue of the whole system is decided by all of the cooperating components together. It is impossible to concretize the quality attributes for only one of the components. Let's consider the above example. Suppose we have a hard timing constraint of 2 seconds of getting response for any customer's requests. The steps the system takes to process the request are the following:

1. A client(usually by a browser) sends a HTTP request to a web server via internet connection.

2. The web server will check the type of the request. If it is a CGI request, it will launch a new process to execute the CGI script indicated in the request.

3. The CGI script will set up a connection to a database and execute some SQL commands to retrieve the required data and also generate the page sent back to the client.

4. Then the web server collects the generated page from the script process, and then sends back the page to the client issuing the request.

According to the user's requirement, the four steps' execution time should not exceed two seconds. However, only one step (step 3) involves database / DBMS operations. Obviously, we can not concretize the performance attribute only for the DBMS system unless we know how long the other steps take.

The Hardware Setting and the Operational Environment Software is always running on some hardware with a given configuration. The hardware factors, such as CPU, memory and I/O system, affect the performance of the software greatly. Running software on a 500 MHz machine will be much faster than running it on a 100MHz machine. Also, running some I/O intensive software with a parallel SCSI disk array will be much faster than running with a single IDE disk. Although the software is the same, it has different performance when running on different hardware contexts. In the case of postgreSQL, there are a lot of factors affected by the underlying hardware architecture. For example, the performance of a DBMS system is greatly affected by disk accesses. In order to reduce hard disk accesses, we want postgreSQL configured as having a large-sized shared memory buffer. But the effectiveness of the buffer is affected by the amount of physical memory available. If we make the shared memory's size bigger than the physical memory available, some shared memory will be swapped into the swap area in hard disks, then any benefits from using large shared memory will be lost. So a bigger physical memory does have significant effect on the performance of postgreSQL. Without consideration of these hardware factors, we can not give the precise performance parameters of postgreSQL. On the other hand, with the above CGI application example, the components except the DBMS, such as web servers are also highly dependent on the underlying operating system services, such as multi-process support, inter-process communication utilities and network device drivers, etc. Therefore, without giving all the configurations about the hardware, the operating system and collaborative components, we can not concretize quality attributes in terms of performance, availability, modifiability, etc.

4.2 Restrictions from applying ATAM to postgreSQL

Application areas PostgreSQL is a general-purpose DBMS system intended to be used in a vast scope of areas, such as business, banking, military, engineering, scientific research, etc. Each of these areas have their own special requirements for a DBMS system. In ATAM, quality attributes should be concretized by scenarios, which is originated from the desire of the user to achieve his special requirements. These requirements are different one from another. For example, one scientific research application may need to join a lot of tables very often but need no transaction support.

A banking application may not need to join a lot of tables, but need transaction support. These varied and sometimes even contradictory requirements makes concretize quality attributes and generate utility tree extremely difficult. Concretizing quality attributes and generating utility tree are essential for the success of ATAM. Without them, ATAM loses the aim of analysis.

Mature system ATAM is recommended by its original authors just for evaluating architectural decisions made for a newly starting project rather than for assessing a mature software product. The ATAM's results should identify the potential risks, sensitivity points and tradeoff points and make the architects aware of these things in order to make correct decisions on these issues. So the purpose of ATAM is to help the new project to avoid pitfalls in the architecture. However, in this thesis, I used ATAM to analyze a software having as long as 12 years working history spanning more than 7 major versions. Obviously, I was not intended to help on shaping postgreSQL by applying ATAM on it. I use ATAM to identify the architecture related issues in a modern RDBMS system like postgreSQL. This usage of ATAM is not recommended by the original authors of ATAM, so there are certainly a lot of risks of not getting the expected results from it.

Organization The default setting for applying ATAM is a group of stakeholders from every aspect of the project. They have some meetings following the procedures outlined in the ATAM document to elicit the quality attributes, user scenario, prioritized utility tree, and do brainstorming and reasoning for many iterations. Therefore, ATAM's results come from many stakeholders and many iterations. This guarantees the effectiveness of ATAM. In my case, I have only one stakeholder, myself, devoted to the analysis. I lack many views from other aspects. I did my best to play multiple roles in order to recover other views of different stakeholders. Also, I did many iterations and found every time I clarified some points I missed in the previous ones. However, I am just one person with limited knowledge and capability. I can not guarantee I covered every issue, even most of the important issues of postgreSQL. Certainly, this limitation in terms of organization is one negative factor to the ATAM analysis in this project.

4.3 The PostgreSQL Project

Deficient requirement analysis PostgreSQL was originally developed in the University of California for exploring new database technologies and algorithms. It was only intended to experiment on some results from academic research. After many years, the developers from many places in the world improved the whole system and added features in order to make it usable in real mission critical applications. In such a case, when the system was first started from the campus the system was intended for academic purpose rather than mission critical tasks, therefore there was not a requirement analysis for the mission critical applications. The original developers in the campus also did not know exactly what the system would perform on a real application when they were building it. They did want to develop a system with a lot of advanced and exploratory features, but advanced features do not mean they are exactly needed by a specific application. Therefore it is difficult to concrete the attributes such as performance, modifiability, availability, etc. for postgreSQL and the same difficulty applies to the generation of scenarios and utility trees.

Lacking Documentation The documentation available in the postgreSQL project does not contain enough description of the underlying architecture behind the source code. The documentation of postgreSQL include the chapters for administrators, programmers and developers, etc. However, most of them deal with the interface between the database user and the DBMS system. This user's manual talks a lot about the SQL data manipulation language and extended management tools, but little is told about the structure of the whole implementation. They do provide a lot of introduction material and tools for developers, but most of them only explain the run time or behavioral model of the system, little is told about the module view and code view. I mean how the modules of the whole system are hooked together, what are the dependencies among them. This means that developers are expected to find out everything from the source code, which forms a big directory tree with a large number of files, equal to more than 250,000 lines of code. For getting the precise architectural information, we must go through this large amount of code base and find their implementation details and then do abstraction in order to find the philosophy used to direct the development. If we use the method of dividing the architecture views from the Applied Software Architecture [4], we should use 4 views to describe a given software's architecture. The runtime architecture is only one view of them. The modular view describes the logical structure, which includes how the systems are divided into subsystems, and how subsystems are divided into modules, and how they are related to each other. According to the ATAM method, we also need to specify the architecture in terms of the objects, functions and their relationships, so without a good description about the software's architecture, it is extremely difficult to perform a satisfactory assessment. Although ATAM only needs a high level description for the system's physical organization, partly because the system usually even not designed yet, this description is essential for the future design and implementation. A poor description of the system's module structure will make ATAM much less effective than what it should be. Although the whole software of postgreSQL has good comments in the code level, and these comments maybe are useful for developers to understand the implementation details and tracing the flow of procedures, they are too detailed to be treated as architectural documents. All in all, the open source project's one big defect is the lack of high level architectural documentation(especially those on modular and code views). This makes it difficult to understand and maintain the software for newcomers.

Implementation language The whole project of postgreSQL is implemented with the C programming language, which has built-in support for the structured programming. There are no object oriented facilities provided by the C language. Also, because the foundation of postgreSQL was built at late 1980s, and object oriented design and architecture concepts were not well developed yet, the design of postgreSQL was not well organized by today's standard. For C, it is possible to adapt the idea of object oriented design and result in a better structured project with high modifiable and maintainable code. Actually, most good C programs were using implicit object oriented ideas even before object oriented concepts were invented [15]. However, there are no built-in facilities supporting and enforcing this kind of design, so a C program is always much easier to be abused by careless developers, who put everything onto the global level and then everything is dependent on everything else. The postgreSQL has high dependencies among subsystems, this makes it hard to identify the potential data dependencies and functional dependencies and thus fully understand the whole project.

4.4 Open Source Development

The attitude of the open source community The open source development has a lot of advantages over the conventional company centric and source closed development. It has wider range of developers and testers and users to find the bugs and fix them with great enthusiasm and efficiency. However, when the software grows larger and more valuable, the hackers always do not want to introduce the details of their implementation or maybe they do not have time to do that. They only want to talk to the experts of the system, but reluctant to introduce the details of the implementation to the new comers and novices. You might say the source code is open, you can find every thing you want by perusing the codes. Well, that is true if you face some thousands of lines of code. But when you face hundreds of thousands of lines of code, understanding them is not an easy task, especially there are so many secrets and conventions you need to get acquainted with. Hackers might have spent much time on studying the code, and they customized the code to the way they want it to be, but they do not provide the necessary documentation about how the whole program are organized, and they do not describe the architecture or design. It may be too difficult to describe because there are many different authors. This is the opposite of the company based development, which has much better documentation and training program to let newcomers become experts and then enhance the strength for further explorations. Also, the company based development has better written documentation and centralized architecture and design strategies to ensure quality and modifiability. In the hacker's model, I found it was really difficult to get some useful information from the project if I did not read its code.

Chapter 5

Solutions and Results

5.1 General Solutions

Although it is impossible to generate business drivers for postgreSQL, I am still able to generate the requirements for RDBMS systems because essentially each RDBMS system needs to do queries, joins, relational operations, table creations, triggers, etc. [14] Also, we can elicit quality attributes with the same method. For example, the performance quality attribute is actually how fast a system can finish its operations and send back the results to the users. By analyzing the general properties of RDBMS systems, I can generate the rough requirement for postgreSQL, which is essentially a special instance of a general RDBMS system. Following this route, I tried to generate the use case scenarios, assigned them into different quality attributes and generated the utility tree. Based on the quality attributes, I analyzed the architectural styles deployed in the project and applied other ATAM analysis methods in light of finding sensitivity points, risks and tradeoff points in the architecture. Finally, I found I did get some results from the analysis. This approach proves to be effective for analyzing the general purpose system like postgreSQL. The only defect for such an approach is that it is impossible to elicit precise quantitative criteria for quality attributes like performance. For example, I can not put a constraint of 2 seconds response time here for postgreSQL because without a software and hardware context, this quantitative value is meaningless. This difficulty has been elaborated in the previous chapter with an e-business example. In this thesis, all the evaluation is done without quantitative analysis, and only tendency and possibility are pointed out in the form of sensitivity and tradeoff points. Lacking a good architecture-oriented document is one of the biggest obstacles in the analysis. I must read some parts of the code to find out what is going on there. Also, I made some tools in Linux and use a lot of *greps* and *finds* to track down the dependencies and function flows. However, I must admit I did not read most of the code, a lot of information is based on the deduction from the part of the code I read, so I can not say everything I described on postgreSQL is free of error. It is just my view of this project. The next several sections are the steps I adopted in the ATAM analysis. The theory of all the steps has been described in the chapter 2 of this thesis. Both results and difficulties encountered are recorded. The quality attributes we are interested in are performance, modifiability, availability and recoverability because for a DBMS system, whether it can query fast and whether it can be used in a adaptive environment and whether it can recover from a system crash are principle to its success.

5.2 Describe System Requirements and Business Drivers

The functional requirements can be presented by the SQL language features, or in other words, the basic relational operations supposed to be implemented by every relational database management system. The purpose of developing this system in the university was exploring some new DBMS technologies in an academic setting. After transferred to the open source community, the purpose became develping a practical RDBMS system capable of undertaking mission-critical tasks. The functional requirements and the project backgroud have been described in chapter 3. The managerial constraints for the current project come from the nature of the open source development. Developers are not gathered in an institution to devote to the project. Instead, they are scattered around the world and communicate with each other by emails. The economic factors can be ignored because the software is developed by volunteers and can be used without any charge. The stakeholders are also widely spread to every area where relational DBMS systems have petential usage.

5.3 Presenting the Architecture

In this section, we describe the module view and the execution view of postgreSQL. In the module view, we divide the system into subsystems and describe their services. In the execution view, our desciption is based on the client-server architecture.

5.3.1 Driving Architectural Requirements

In general, performance, modifiability, availability and recoverability are our concerns in the case of postgreSQL project.

5.3.2 High Level Architectural Views

PostgreSQL can be divided into the server part and the client part. In the server part, there are a postmaster daemon process and backend processes. In the client part, there are libpq and a series of library modules to support C, C++, tcl, ODBC, JDBC, perlDBD interfaces [13]. Also, postgreSQL has psql, a client-side terminal interface. Our interest is mainly on the backend process because all the real work, such as parsing, rewriting, optimizing, storage managing, are done in the backend process. Also, most of the modules of postgreSQL deal with the backend process. The backend process basically is a big C program linked together from a large number of modules. The architecture of this big backend can be illustrated in Figure 3 with a rough module view. All of these backend processes share a common buffer, which contains the mirrored data from the disks, transaction information, logging information, etc. Because there is only one shared copy of all the above data, the consistency is achieved among multiple instances of backend processes.

5.3.3 Modules and Subsystems

PostgreSQL can be decomposed into several subsystems in terms of their functional divisions. They are query processing subsystem, storage management subsystem and utilities subsystem. There are also some modules responsible for coordination and initialization, and they could not be combined into any subsystems. The standalone modules are bootstrap, postmaster, libpq(client libraries), tcop(Traffic control), etc.

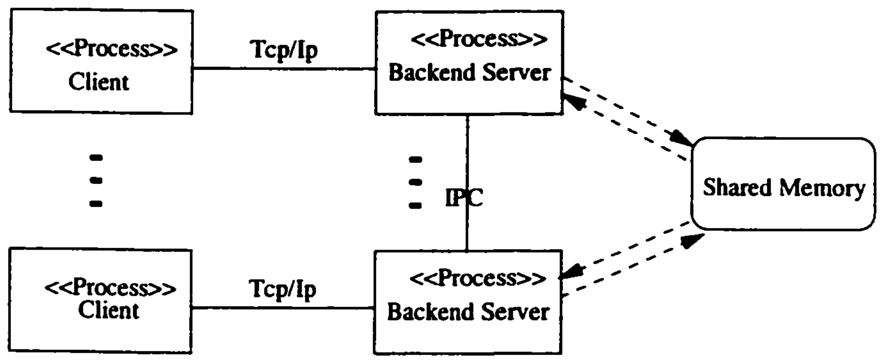
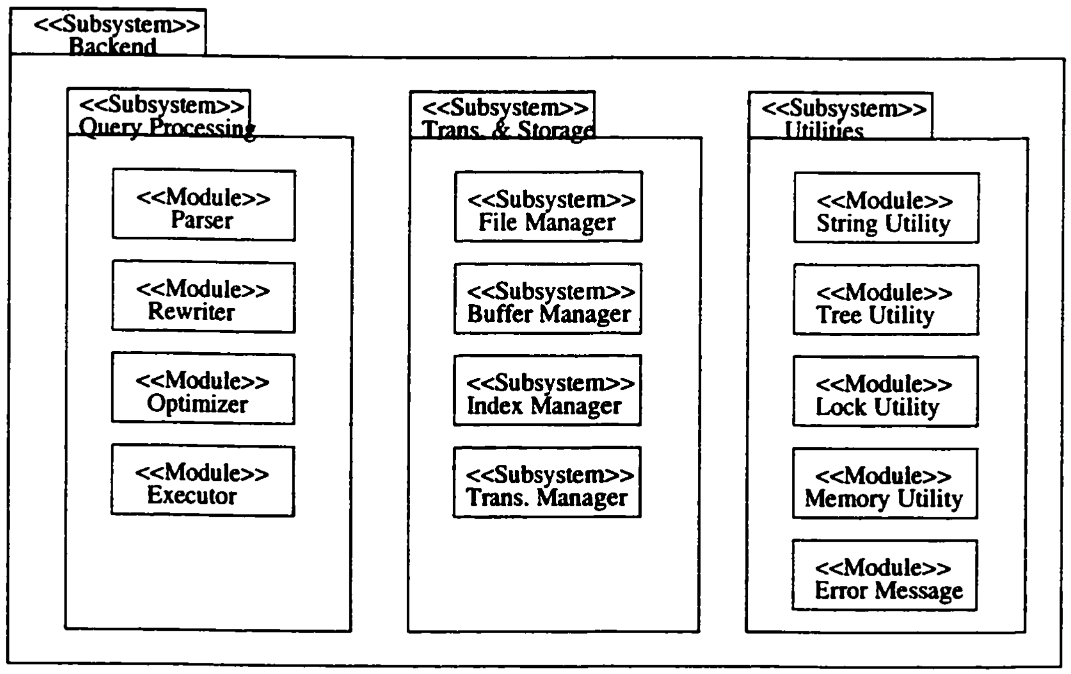


Figure 2: Execution View of PostgreSQL



<<Subsystem>> Backend
<<Subsystem>> Query Processing
<<Module>> Parser
<<Module>> Rewriter
<<Module>> Optimizer
<<Module>> Executor
<<Subsystem>> Trans. & Storage
<<Subsystem>> File Manager
<<Subsystem>> Buffer Manager
<<Subsystem>> Index Manager
<<Subsystem>> Trans. Manager
<<Subsystem>> Utilities
<<Module>> String Utility
<<Module>> Tree Utility
<<Module>> Lock Utility
<<Module>> Memory Utility
<<Module>> Error Message

Figure 3: Module View of PostgreSQL

Some description in the following part is quoted from the postgreSQL's developer documentation [12].

The functionality of three subsystems is described in the following part.

1. Query processing subsystem:

   Parser - converts SQL query to query tree

   rewrite - rewrites the parsing tree according to the rules system

   Optimizer - creates paths and plans

   Optimizer/path - creates paths from the parser's output

   This module takes the parser query output, and generates all possible paths of executing the request by considering different methods. It examines table join order, where clause restrictions, and optimizer table statistics to evaluate each possible execution method, and assigns a cost to each.

   Optimizer/geqo - genetic query optimizer

   This module is adapted to conquer the problem of searching too many combinations among a large number of joined tables. The Genetic Query Optimizer considers each table separately, then figures out the most optimal order to perform the join. For a few tables, this method takes longer, but for a large number of tables, it is faster. There is an option to control whether this feature is used.

   Optimizer/plan - optimize path output

   This module takes the optimizer/path output, then chooses the path with the least cost, and then creates a plan for the executor.

   Executor - executes complex node plans from optimizer

   This module handles select, insert, update, and delete statements. The operations required to handle these statement types include heap scans, index scans, sorting, joining tables, grouping, aggregates, and uniqueness.

2. Storage management subsystem:

   Storage - provides an interface among many different storage access methods.

   Storage/buffer : shared buffer pool manager

   Storage/file : file manager

   Storage/ipc : semaphores and shared memory manager

   Storage/lmgr : lock manager

   Storage/page : page manager

   Storage/smgr : storage/disk manager

   Access/hash : hash access

   Access/heap : heap of storing data rows

   Access/index : index types

   Access/nbtree : Lehman and Yao's btree management algorithm

   Access/rtree : index of 2-dimensional data

   Access/transam : transaction manager (BEGIN/ABORT/COMMIT)

3. Utilities subsystem:

   Utils : support routines

Utils/adt : built-in data type routines

Utils/cache : system/relation/function cache routines

PostgreSQL supports arbitrary data types, so no data types are hard-coded into the core backend routines. When the backend needs to find out about a type, it does a look-up of a system catalog table. Because the system table is referred too often, a cache is maintained to speed lookups. There are a system relation cache, a function/operator cache and a relation information cache. This last cache maintains information about all recently-accessed tables, not just system ones.

Utils/errors : error reporting routines

Reports backend errors to the front end.

The following desciption is about the functionality of standalone modules.

- Bootstrap - create the initial system tables and templates in the first call to initdb.
- Main: The entry point of the whole program It also passes parameters to the following postmaster or postgres processes.
- Postmaster: the daemon process listens for the connection requests and starts and terminates backend servers. Also, it creates the shared memory pool used by backends.
- Libpq: backend libpq library routines, which handle the communication to clients.
- Tcop: this traffic controller dispatches requests to proper modules. This module makes calls to the parser, optimizer, executor, and commands functions.
- Commands: commands that do not require the executor

  This module processes SQL commands that do not require complex handling. It includes vacuum, copy, alter, create table, create type, and many others. The code is called with the structures generated by the parser. Most of the routines do some processing, then call lower-level functions in the catalog directory to do the actual work.

- Catalog: system catalog manipulation

  This module contains functions that manipulate the system tables or catalogs. Table, index, procedure, operator, type, and aggregate creation and manipulation routines are here. These are low-level routines, and are usually called by upper routines that pre-format user requests into a predefined format.

5.4 Architectural Styles

Identifying the architectural styles deployed in postgreSQL is a part of the architecture presentation. Because the part contains a lot of contents, I put them into a separate section. In this section, I used some sentences and paragraphs from the original ABAS document [2] because it is the best way to keep the description precise. At the end of each ABAS style, I present our result of applying the ABAS analysis.

5.4.1 Synchronization Architectural Style

PostgreSQL is a client-server architecture, which means for each client connection, there is exactly one backend process to serve it. Many client connections could initiate the same number of backend processes on the server side. All of the backend processes will run concurrently when scheduled by the operating system. These processes share data in the shared memory. This situation is exactly what the synchronization architectural style is suited for as a performance style in ABAS. This style is used to enhance the performance by encouraging concurrency and parallel executions.

5.4.1.1 What is Synchronization ABAS

Criteria for Choosing this ABAS This ABAS will be relevant if the problem inherently has real-time performance requirements and consists of multiple processes on a single processor that share a resource.

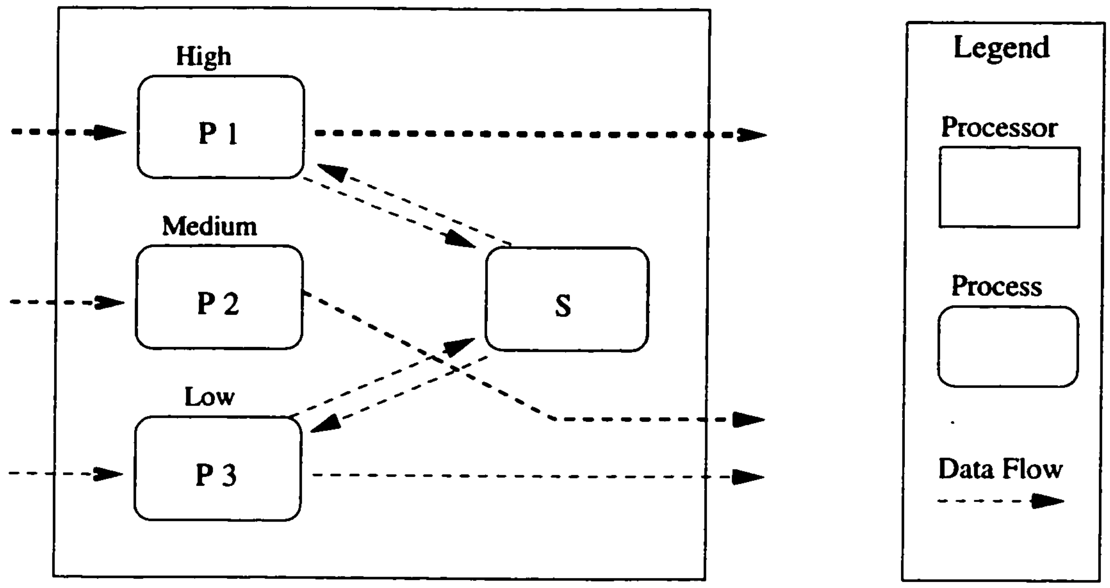


Figure 4: Synchronization Style

5.4.1.2 Stimulus/Response Attribute Measures

We characterize the important stimuli and their measurable, controllable responses as follows:

- Stimuli: two or more periodic or sporadic input streams
- Response: end-to-end, worst-case latency

"End-to-end" refers to a measure beginning at the point of message input, through all stages of computation to its final output.

5.4.1.3 Architectural Style

The synchronization style is shown in Figure 4 in a concurrency view mapped onto a hardware view. In this ABAS, there is a single processor and a set of processes with the associated known (or estimated) properties, listed in Table 3, that are transforming input streams into output streams. Some of these processes need to synchronize to share a resource controlled by S, the "server" process.

Performance Architectural Parameters
Topology: star
Preemption policy: priority based
Execution time for each process associated with processing each input
Period associated with each process
Scheduling discipline: fixed priority
Synchronization protocol including: • The queuing discipline (e.g., FIFO or priority) for the server process • How the priority is managed during the critical section (e.g., the section of code during which other processes are locked out)

Table 3: Architectural Decisions for the Synchronization ABAS

5.4.1.4 Analysis and Design Heuristics

Even if one does not build a formal analytic model of the latency in a synchronization ABAS, a designer should keep in mind that the latency of a process that synchronizes to access shared data is very sensitivity to the:

- Prioritization strategy
- How does your choice of priority assignment impact latency?
- Is there another prioritization strategy that might reduce latency?
- Sources of blocking
- Has blocking been accounted for in estimating latency?
- Are there sources of blocking in black-box components that have not yet been accounted for?

- Priorities used during the critical section
- Does an unbounded priority inversion situation exist?
- Can blocking time be reduced by using a different technique for managing the priority of a critical section?

5.4.1.5 Utilization in Our Analysis

In postgreSQL, every backend process is a server process(denoted by P in the ABAS diagram). All the backend processes have the same priority, fixed on generation. All of the processes are scheduled by the operating system. There are not internal queuing and prioritizing mechanisms in postgreSQL itself. Therefore, the latency caused by priority assignment can be ignored. Blocking is a quite serious issue in postgreSQL, mainly coming from the following several aspects: shared memory, locks, deadlocks and disk I/O. The shared memory access is guarded by semaphores, which can block other concurrent accesses when the first process is accessing the memory. PostgreSQL uses locks extensively to achieve the data consistency among concurrent transactions. There are page level locks, row level locks, table level locks, etc. Although not all of them are exclusive locks, the process updating some data forbids other concurrent processes to access the same data. Therefore, if postgreSQL is used in an environment that has a lot of frequent data updates, the performance will definitely be degraded· more or less. Because there are locks, there are deadlocks. Deadlocks can be caused by acquiring the same set of locks with the reverse order by the concurrent processes. PostgreSQL has its own mechanism to detect deadlocks, but it is quite expensive. If the deadlock situation does happen, one of the processes will abandon its possession of all its locks and abort its own transaction to ensure the system can continue. Disk I/O operation always causes a lot of interrupts which will interrupt the execution of the current process and cause possible context switch, so this is also a contribution to blocking. We do not need to care about priority inversion if we only consider the backend processes themselves because all of them have the same priority, and it is impossible to make priority inversion happen. The postmaster process might be involved in the priority inversion situation if it competes the access to the shared memory with other backend processes. But as stated by postgreSQL developers, the access by the postmaster is minimal, so we do not need care about priority inversion at all.

5.4.2 Data Indirection ABAS

Data indirection ABAS is an architectural style used to enhance modifiability of a software product.

5.4.2.1 Problem Description

This ABAS is characterized by keeping the producers and consumers of the shared data from having knowledge of each other's existence and their implementations. This is accomplished by interposing an intermediary—a component and/or protocol—between the producer and consumers of shared data items. The general principle at work here is that modifiability is enhanced by reducing the data or control coupling between distinct components. In this ABAS, coupling is reduced by having the intermediary—typically a shared data repository—coupled to both the producers and consumers, and hence having them decoupled from each other.

5.4.2.2 Criteria for Choosing this ABAS

This ABAS will be relevant if we anticipate changes in the producers and consumers of data, including the addition of new producers and consumers. If these changes are frequent and pervasive enough to warrant concern about the cost of modification, this ABAS is relevant.

5.4.2.3 Stimulus/Response Attribute Measures

We characterize the important stimuli and their measurable, controllable responses as follows:

Stimuli

- a new producer or consumer of data
- a modification to an existing producer or consumer of data
- a modification to the internals of the data repository

Response The number of components, interfaces, and connections added, deleted, and modified, along with a characterization of the complexity of these changes, deletions and modifications.

5.4.2.4 Architectural Style

Figure 5 shows the generic structure of this ABAS. Its topology is a star, with the repository located at the center and the producers and consumers at the periphery. The data repository can be a location that is known to both producers and consumers (e.g., a file or a global data area) or it can be a separate computational component (a blackboard that is hosted in a separate process, potentially even on a separate computer). The only constraint on the repository is that it can hold data. The data repository is a persistent data store such as a shared data area in memory, a file, or a database. In addition to the repository, there are some number of data producers and some number of data consumers. A single component may be both a producer and a consumer of data. The repository has a specified data layout—a file structure, or schema—and a set of data types that are known by all producers and consumers. This layout remains consistent during a single execution of the system. The producers place their data in the repository by virtue of the fact that they know the details of the repository's layout; the consumers similarly retrieve data from the repository. The issues of how performance and concurrency control are managed (e.g., the policy and mechanisms for determining who gets to update the repository and when) are outside the scope of this ABAS. The components can be independent processes on the same processor or different processors. They could also be bundled together in a single process. Thus, there are no restrictions within this ABAS on the run-time packaging of the components.

5.4.2.5 Analysis and Design Heuristics

We should choose this ABAS if we do not expect to add, delete, or change the data types used in the repository frequently. If, for example, our most frequently anticipated change is to add a new consumer of an existing data type, this ABAS will be simple to implement (since we have not had to go to the additional trouble of defining an abstraction interface) and will easily support our anticipated palette of modifications. If we are adding new data to the repository, we should add it in such a

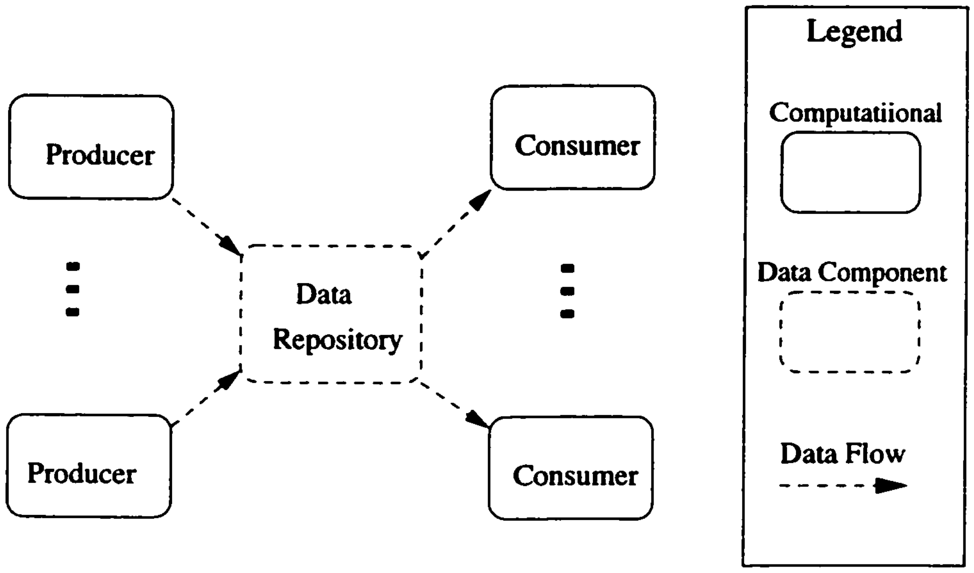


Figure 5: Data Indirection Style

Modifiability Architectural Parameters
Topology: star
Persistence of data: persistent
Client knowledge of data schema: complete knowledge
Activeness of repository: passive

Table 4: Architectural Decisions for the Data Indirection ABAS

way that it does not affect the access of existing data. This could mean, for example, adding new data to the end of an array, if arrays are used. Similarly, when removing a data type from the ABAS, the data type could be left in place as an unused field in the repository, or it can be removed, potentially causing ripple effects to other data types and their producers and consumers.

5.4.2.6 Utilization in Our Analysis

In postgreSQL, data indirection style is used to support and extend data types of the SQL language. No predefined data types are compiled with the source code. Whenever the parser or other system components need to know some information about data types, they will look up a system catalog to retrieve the required type definition. The system catalog is a set of tables kept in the database to store the meta-information about tables, types, attributes and functions, etc. Whenever we create some tables, views, functions, triggers, types, some corresponding table entries will be inserted into those tables. In the case of the type system, postgreSQL's *pg_type* catalog provides the detailed information about a type. Therefore, the system catalog is the indirection data repository, and the executor module of the backend is the producer, and the SQL parser module and other components using the catalog are the consumers. The producer and consumers are in the same process, and they pass the type information by the data repository. So the producers do not need to have any knowledge of consumers, and vice versa. The data schema is the database tables, and the access method is the abstraction language SQL, so even when the schema or the definition of the system catalog is changed, it does not introduce any changes in the producer or the consumer's code. Also, we could imagine that if there were a demand of adding some new producers or consumers, it also would not affect the existing modules. Additionally, the situation that the data schema in the system catalog does not change very often is also a good reason of adapting this ABAS.

5.4.3 Layered Architectural Style

Layered architecture is another style used to enhance modifiability of a software product. With this style, many layers are used to construct the whole system. Each layer only asks for services from the lower layer and provides services to the upper layer.

Usually, one upper layer only calls the functions just below it, and therefore this restricts the scope of dependencies. Additionally, all of these services are only accessed through well-defined interfaces between layers, and the internal implementation is transparent to the outside. So changing implementation while keeping interfaces the same will not affect other modules. Layered structure is a powerful style coping with complexity and portability. Many large projects use this approach. For example, operating system services are always wrapped into a system service layer, and we also abstract the services required by our project into a system service interface, and the functions in the interface translate the user's requests to the real operating system calls. This architecture can achieve the effect of reusing the same code base on different platforms. PostgreSQL has more than 250,000 lines of code. The scale of the project is suitable for applying layered architecture. However, postgreSQL didn't utilize a good layered structure. First of all, its source code is highly dependent on UNIX system calls. This makes porting to another platform difficult and costly because many modules need to be modified manually. Secondly, the modules are not well divided into logical layers and then they are highly coupled with each other. RDBMS systems actually have very natural functional divisions. For query processing, function calls go through the sequence: parser, rewriter, optimizer, executor and then storage manager. This logical relationship can be used to organize each functional unit into a separate layer with a well defined interface. This would reduce the coupling of modules in the global level, and consequently enhance the modifiability of the whole project. Similarly, the operating system functions also should be wrapped into a separate layer in order to facilitate the porting from one to another. On the other hand, using layered architecture will have some performance overhead because function calls need to be routed through the interface of each layer to reach the real service calls. But compared with the great benefit brought in by it, the overhead is usually tolerable.

5.5 Generate the Scenarios and the Utility Tree

As we have mentioned in the first section (concretizing quality attributes), the performance scenarios are not generated with quantitative values. I tried to generate some performance scenarios without quantitative values. For modifiability, I also generated

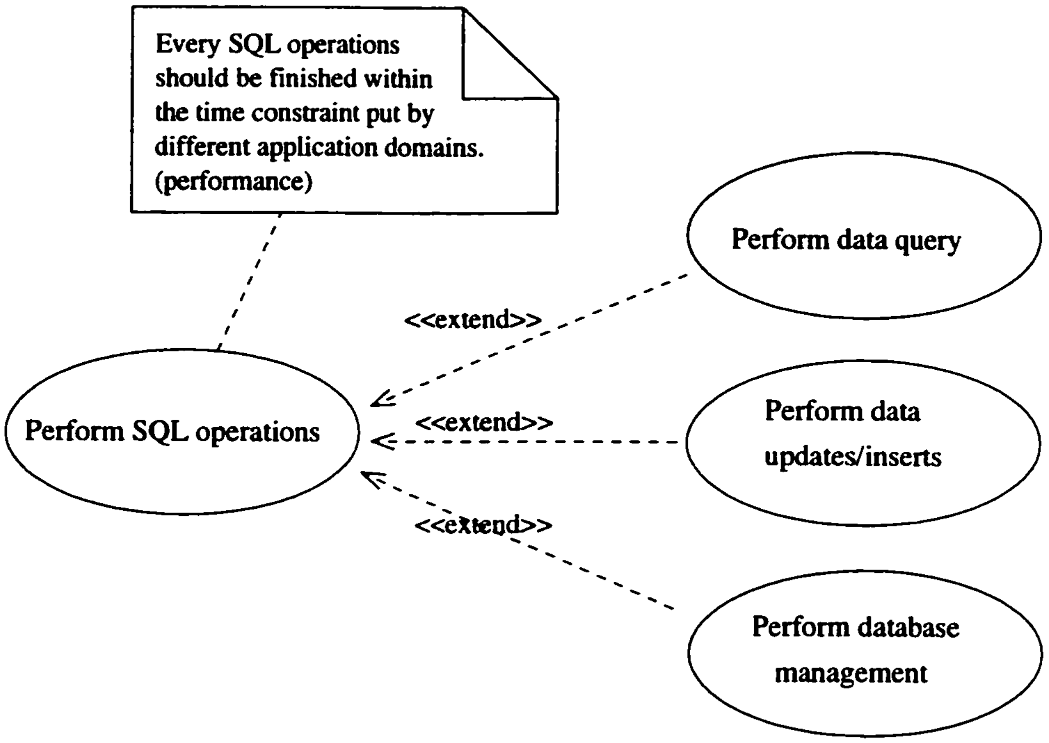

Every SQL operations should be finished within the time constraint put by different application domains. (performance)
Perform SQL operations
<<extend>>
<<extend>>
<<extend>>
Perform data query
Perform data updates/inserts
Perform database management

Figure 6: Use Case for SQL Operations

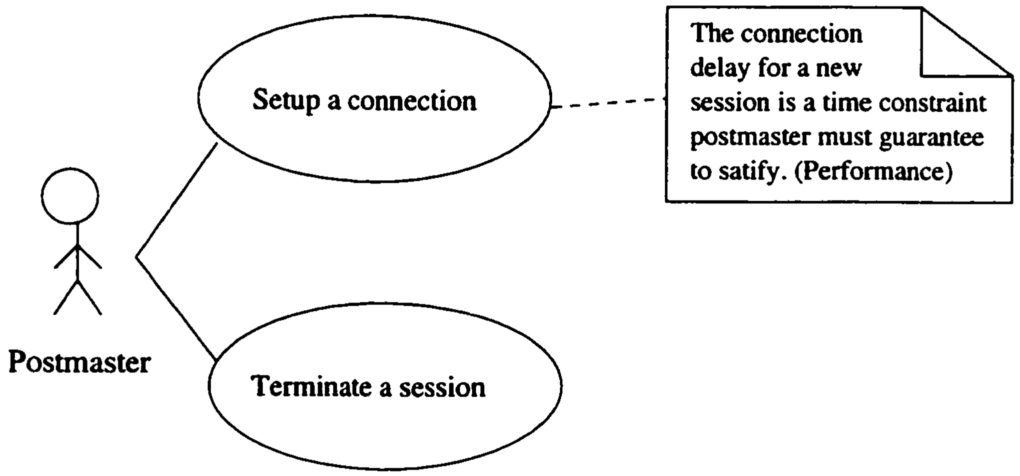

Setup a connection
The connection delay for a new session is a time constraint postmaster must guarantee to satify. (Performance)
Postmaster
Terminate a session

Figure 7: Use Case for Connections

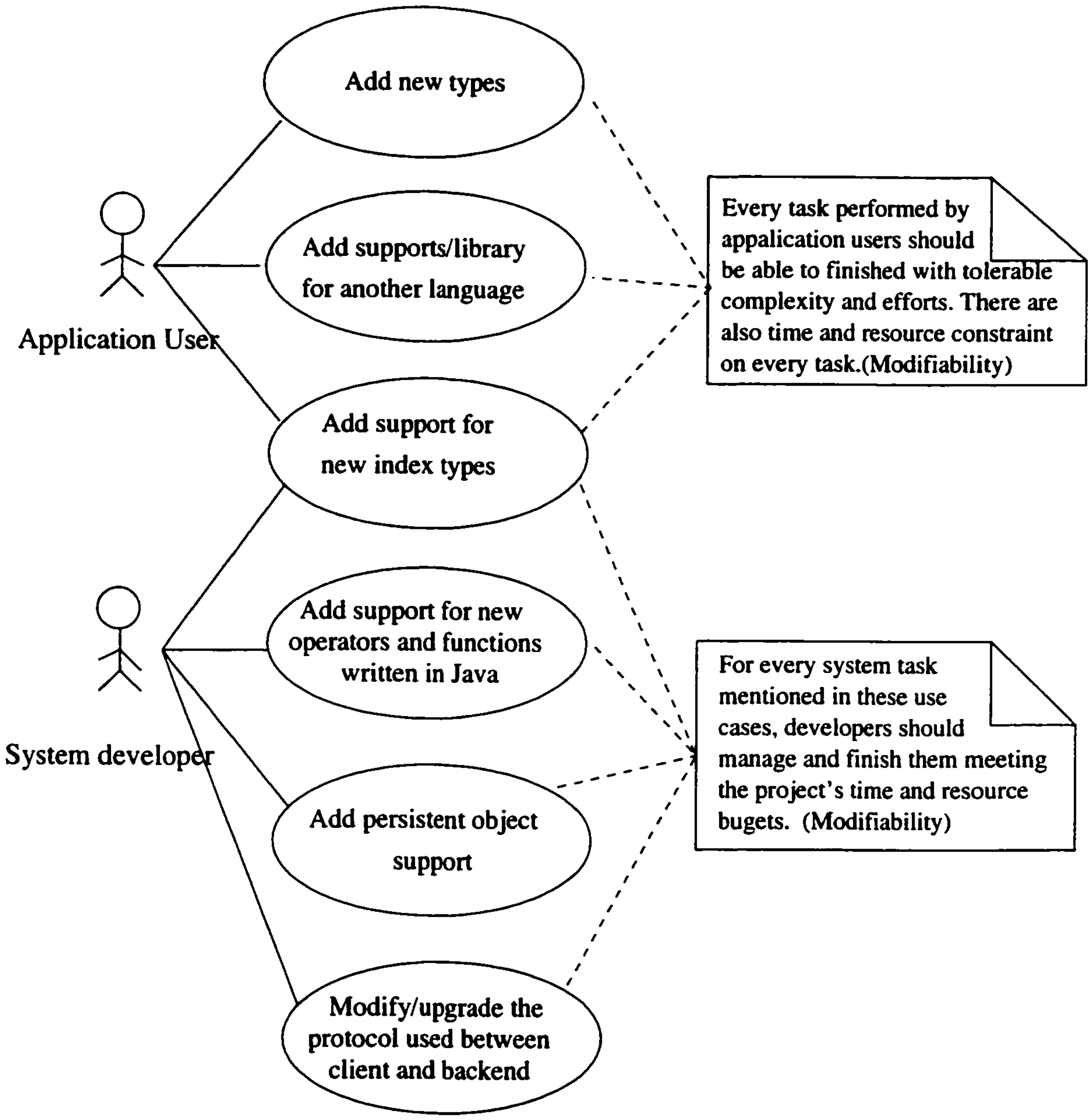

Add new types
Add supports/library for another language
Add support for new index types
Application User
Every task performed by appalication users should be able to finished with tolerable complexity and efforts. There are also time and resource constraint on every task.(Modifiability)
Add support for new operators and functions written in Java
System developer
Add persistent object support
Modify/upgrade the protocol used between client and backend
For every system task mentioned in these use cases, developers should manage and finish them meeting the project's time and resource bugets. (Modifiability)

Figure 8: Use Case for Modifiability

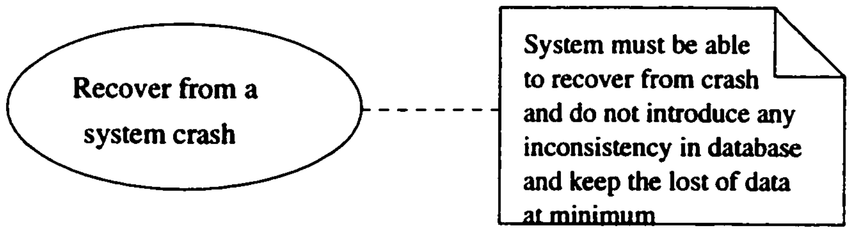

Recover from a system crash
System must be able to recover from crash and do not introduce any inconsistency in database and keep the lost of data at minimum

Figure 9: Use Case for Recoverability

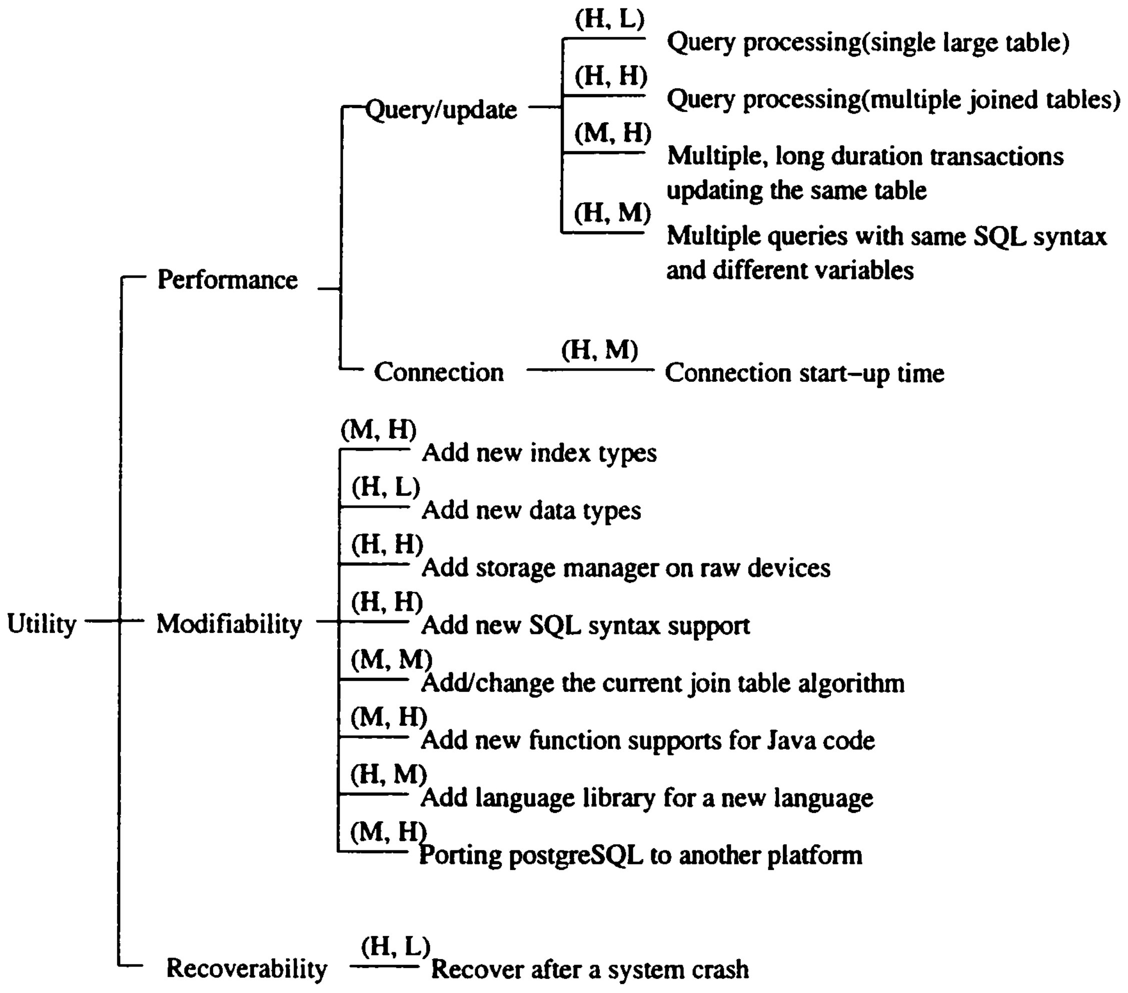

Utility
Performance
Query/update
(H, L)
Query processing(single large table)
(H, H)
Query processing(multiple joined tables)
(M, H)
Multiple, long duration transactions updating the same table
(H, M)
Multiple queries with same SQL syntax and different variables
Connection
(H, M)
Connection start–up time
Modifiability
(M, H)
Add new index types
(H, L)
Add new data types
(H, H)
Add storage manager on raw devices
(H, H)
Add new SQL syntax support
(M, M)
Add/change the current join table algorithm
(M, H)
Add new function supports for Java code
(H, M)
Add language library for a new language
(M, H)
Porting postgreSQL to another platform
Recoverability
(H, L)
Recover after a system crash

Figure 10: Utility Tree for PostgreSQL

some scenarios commonly important for a DBMS system. All of these scenarios are not measured by numbers because of the difficulty of extracting quantitative criteria on a single software component. Considering the importance of recoverability for the success of a DBMS system, I added the recoverability into the utility tree.

5.5.1 Use Case and Scenario Generation

Figure 6 and Figure 7 describe basic SQL operations and session control scenarios respectively. They describe functional requirements of the system and also have some performance requirements, such as time constraints on the system. Figure 8 contains modifiability use cases. Modifiability is a non functional requirement, so these use cases are only concerned about adding or modifying features of the system. Figure 9 describes a scenario of recoverability after system crashes.

5.5.2 Utility Tree

After identifying the use cases in terms of performance, modifiability and recoverability, we generated the utility tree as shown in Figure 10. From this utility tree, we identified that the high-priority performance factors are query processing on a single large table or multiple joined tables, repeated similar queries, and the high-priority modifiability factors are adding new data types, adding storage managers for raw devices, adding new SQL syntax and adding new language libraries, and the recoverability factor is recovering from any system crashes.

5.6 Evaluate the Architecture

Based on the module and runtime views, ABAS architectural styles, we evaluated these architectural decisions against the high-priority factors identified by the utility tree.

5.6.1 Performance Issues

In the case of query processing, we notice that the system always generates many execution paths and chooses and buffers the best one. Also, the system will fetch tuples following the best execution path. When the system scans the tables for satisfying tuples, it must first read the pages of the table into the shared memory buffer and evaluate them there. The shared buffer is flushed with LRU algorithm. As we know, the disk access is always expensive. If there is not enough shared memory, not a lot of disk pages can be kept in the memory, and then for the queries on the large tables or joins on multiple tables, there would be a lot of disk accesses. This would degrade performance significantly. So the size of the shared memory is a sensitivity point of performance.

We could verify this assertion by building a simple mathematical model as following. Suppose there are x and y data blocks in the cache and disk respectively. The relationship between them can be described as $x = ry$, in that r satisfy the condition $r > 0$. Additionally, for a random data query action, suppose the probability of finding the data in a cache block is p_x, and the probability of finding the data in a disk block is p_y. Because the chance of finding a match in the cache is much larger than in the disk, $p_x = np_y$. The n is the ratioin between two probabilities. Assume the probabilities of finding a match in all the cache disks are always equal. So we have the following equations for a random query:

$$\begin{aligned} xp_x + yp_y &= 1 \\ x &= ry \\ p_x &= np_y \end{aligned}$$

We are interested in the probability of missing a hit on the cache, yp_y, for a random query. From the above equations, we get the yp_y by solving the above equations:

$$yp_y = \frac{1}{nr+1}$$

Because the sensitivity of the probability of missing a match on the cache is the differentiation of the above equation, we have:

$$\frac{\partial(yp_y)}{\partial r} = -\frac{n}{(nr+1)^2}$$

First of all, the negative sign means the increase of the ratio between the data in the cache and in the disk will decrease the probability of missing a match in the cache. A fact is that the ratio n is normally far less than 1 because most people can not afford such a big memory that all of the database data are kept on the memory cache. Then if we have a very small number of data in the cache compared with the data kept in the disk, or in other words, we have a very small n, the sensitivity is approximately $-n$, which means the probability of missing a match to the cache is highly sensitive to the ratio between cached data and the disked data. On the other hand, if the ratio r is quite large, for example $r = 1$, which means there are half of the whole data set kept in the cache, the sensitivity will be approximately $-\frac{1}{n}$. In this case, missing a match is not sensitive to the ratio r. Therefore, if the ratio r is quite small , increasing cache size will result in a big improvement in performance. On the other hand, if the cache has kept a large amount of the database, the improvement by adding more cache will not be remarkable.

Additionally, because most modern operating systems support virtual memory, if the shared memory can not be kept on the main memory, it will greatly degrade performance because of swapping. So the physical memory available for the shared memory space is also a sensitivity point of performance.

Anther issue in performance is the transaction support of the serializable level and the read committed level. If there are concurrent transactions updating the same data, the serialization level transaction support will cause other transactions except the first one drop. When some transactions find one concurrent transaction has updated the shared data, they must wait for the commit of the first transaction. If the first transaction takes a long time to complete, it will block all the other transactions for a long time, even though most possibly that transaction will commit and the waiting transactions will abort themselves anyway. Therefore, the serializable level transaction support will affect the performance of long duration concurrent transactions. So the concurrency strategy is a sensitivity point of performance

The third issue in performance is that postgreSQL uses UNIX's file system as the lowest level storage entity. It does not have any direct accesses and manipulations on the physical storage devices. So every operation on the data is called through the UNIX system calls. This indirection makes postgreSQL difficult to control the real disk access and the real cost for each disk operation. As we know, UNIX file system is implemented by inodes and data blocks [16]. In each inode, there are only 10 direct data blocks available for sequential and direct access, 11 and 12 are pointers to the following data blocks and 13 is the second level pointers. Therefore, if the data is larger than 10 data blocks, the cost for searching data blocks will go higher and this is especially true if most of these data accesses are in a random way. So the physical storage form is a sensitivity point of performance. Also, postgreSQL can not control the buffering made by the operating system, so there is no guarantee that writing to a file means the real data does get written into the physical device. This could be a possible recoverability hole because it might cause some data lost in the operating system's buffer in a crash. So the physical storage form is also a sensitivity point of recoverability.

The fourth issue in performance is that postgreSQL uses processes for connections rather than threads. Processes definitely consume much more resources, such as memory and other system structures, compared with threads. This makes maintaining a large number of concurrent connections difficult. When the system's resources are exhausted, there will not be any new connections. Also for each new session, forking a new process to take care of the session makes the start-up time quite long compared with the threading model. There are two possible usage scenarios for a DBMS system, multiple concurrent long-duration sessions and short-duration frequent sessions. In the second case, long start-up time will damage performance greatly because most transaction time is wasted on start-up. On the other hand, the possibility that one transaction can block other concurrent transactions which are running on different sessions will simply lock up much more resources. Additionally, the process model makes IPC the only communication method among the backends, otherwise, with the threading model in the same process space, the backends can communicate directly by the global memory space, which is definitely much faster. So process/thread selection is a sensitivity point of performance. Additionally, the threading model always means the functionality can be divided into different threads communicating with each other through messages, so threading leads to much more modifiable system because the loose coupling among modules, so the process/thread selection is a sensitivity point of modifiability.

The fifth issue in performance is locking strategy. In postgreSQL, if a user locks a table, there is no way to explicitly unlock a data item until the transaction commits.

Other users can not do any operations on the locked table. This certainly reduces the concurrency level and degrades performance. Locking is always a tradeoff point between consistency and performance. Also, the granularity of locking affects performance. For example, if a transaction updates a few rows in a table, and the system locks the whole table, this will forbid the transactions that even do not write the same rows to access the whole table and thus reduce concurrency and damage performance. So locking strategy and the granularity of locking are sensitivity points of performance.

The sixth and seventh sensitivity points for a RDBMS are deadlock detection strategy and join strategy. Also, deadlock detection strategy is a sensitivity point for availability. Running deadlock detection routines frequently enhances the availability of the system but damages performance, so deadlock detection strategy is a tradeoff point between performance and availability.

5.6.2 Modifiability

The first issue on modifiability is that the dependencies between subsystems are too complicated. Almost every subsystem is dependent on other subsystems. Those dependencies make modifying the program very difficult because the impact of a little change can be spread to other parts of the whole program. For example, if we want to add the support for a new SQL syntax, we might need to change the parser, rewriter, optimizer, executor, access, etc. If we want to add some utilities for this new syntax, maybe we need to make some changes on the storage manager. Similarly, if we add the support for a new indexing method in the access subsystem, we need to change the storage manager, executor, or even parser. This is far more difficult and complicated than what it should be. The only benefit of such an approach is some kinds of performance gain because of the direct access to the lower level or higher level functions. But in the context of the fast evolving hardware, the real value of this gain is questionable. Because the dependency issue might affect both performance and modifiability, it is the sensitivity point of modifiability, and also is a tradeoff point between modifiability and performance.

The second issue on modifiability is the layered architecture. Without a layered architecture, for software as large as postgreSQL, it is hard to manage when more features are added or the existing code is modified. Adding more code also brings in more and more complicated dependency. Additionally, there is not a layer to wrap operating system services, so it will be hard to port the system to another platform. PostgreSQL is highly dependent on the UNIX operating system, it uses UNIX's file system as its storage management, and uses UNIX's socket as its communication media, and uses UNIX's semaphore, shared memory, and other IPCs as locks, buffers, etc. The dependency on one platform degrades portability. So the layered architecture is a sensitivity point of modifiability. The operating system service wrapper is a sensitivity point of modifiability or portability.

The third issue on modifiability is the data indirection approach. This issue is also recovered from the architecture style analysis. The system catalog is a set of tables used by the system to record and check the meta-information of the whole system. The system catalog separates the producers and consumers from calling each other directly. Obviously, this approach enhances modifiability and gives the system the flexibility of choosing the way to grow in the future. On the other hand, the format and availability of the system catalog will be essential for the normal operations of producers and consumers. So the system catalog is a sensitivity point of both modifiability and availability.

5.6.3 Availability

PostgreSQL is essentially a program built upon UNIX operating system, so the operation of postgreSQL is dependent on the services of the UNIX operating system. If one of the services is not available or abnormal, postgreSQL will not work well. Especially, if the file system is damaged or halted, the data in the database will not be available. Also, if the socket behaves abnormally, no connection can be set up between the client and the database server. So the UNIX operating system's service is a sensitivity point of availability.

5.7 Results from ATAM Analysis

In this last step, we summarize the results we got from the evaluation of former steps and categories them in terms of sensitivity points, tradeoff points and risks. This result is shown in table 5.

	Performance	Modifiability	Availability
Sensitivity points	Shared memory Concurrency strategy Physical storage Thread/process Locking strategy Locking granularity Deadlock detection Join algorithm	Module dependency Layered architecture OS service wrapper System catalog Thread/process	Physical storage System catalog OS services
Tradeoff points	Dependency(M) Deadlock detection(A) Locking strategy(C)	Dependency(P)	Deadlock detection(P)
Risks		No layered architecture	

Table 5: Sensitivity Points, Tradeoff Points and Risks in PostgreSQL

5.7.1 Sensitivities and Tradeoffs and Risks

The sensitivity points on performance are the size of the shared memory and the physical memory, the concurrency strategy, the physical storage form, the thread/process selection, the locking strategy, the locking granularity, the deadlock detection strategy, the join strategy. Although we find many sensitivity points on performance, we should notice that some of them are only sensitivity under certain conditions. For example, the concurrency strategy and the deadlock detection strategy are only sensitivity for applications with extensive transaction activities. The join strategy is only sensitivity for applications with many table joins. However, in general, the shared memory, the physical storage form, the locking strategy or maybe the thread/process selection are always important issues affecting performance.

The sensitivity points on modifiability are the dependency complexity, the layered architecture, the operating system service wrapper, the system catalog and the thread/process selection. All of these sensitivity points on modifiability are important when the system evolves.

The sensitivity points on recoverability is the physical storage form. The physical storage form is quite important because it is a sensitivity point on both performance and recoverability. If the system manages the physical storage device by itself instead of by the file system, it can use methods such as clusters and extends to gain speed on the disk access and also can avoid the danger caused by the file system buffering.

The sensitivity points on availability are the system catalog and the UNIX operating system services.

The dependency complexity is a tradeoff point between performance and modifiability. The subsystem dependency should be reduced in most cases. The performance gain got from messing up dependency is not significant, and the lost modifiability is much bigger than the performance gained from it.

The deadlock detection strategy is a tradeoff point between performance and availability. Deadlocks can be avoided by carefully acquiring locks with the same sequence. So this tradeoff point is only applicable for limited situations. The locking strategy is a tradeoff point between performance and consistency.

Lacking layered architecture is a potential risk and a true danger of the system.

Chapter 6

Conclusions

In this thesis, I investigated how to apply ATAM, a modern architecture analysis method, to an open source RDBMS project, postgreSQL. ATAM is supposed to be an effective method to spot architectural weaknesses in the early phase of software development. PostgreSQL is not in its early phase of development, so applying this method to postgreSQL is a new way of using the method, and this brings in many difficulties that were not considered in the original method. Additionally, because postgreSQL is a general purpose DBMS system used with many other application components in various hardware contexts, it is very difficult to elicit precise requirements and scenarios to concretize the quality attributes for itself only, and then implementing the other steps of ATAM becomes difficult too.

Although there are some difficulties of applying ATAM to postgreSQL, ATAM's general ideas are applicable to the architectural analysis of postgreSQL. I made some experiments and tried to elicit the quality attributes and the utility tree. My solution is making an abstraction on the general requirements of RDBMS systems, and using the generalized requirements and scenarios to concretize quality attributes and the utility tree. Every RDBMS system needs to do relational operations, such as select, join, union, intersection, etc. Every modern RDBMS system needs to support the SQL language as their query language. So the relational operations and the SQL language are actually the functional requirements and the performance of accomplishing these functions is actually the performance quality attribute, and so on. So we can use these generalized requirements and scenarios to concretize the quality attributes of a specific system, in our case, that is postgreSQL. This makes concretizing all the quality attributes, such as performance, modifiability, availability, etc. possible and reasonable. Consequently, I successfully applied ATAM method's general ideas to the analysis on postgreSQL. In this analysis, I followed the steps mentioned in the ATAM's original document and elicited the module and subsystem architecture of postgreSQL and applied the ABAS architectural style to the analysis and finally found a series of sensitivity points, tradeoff points and risks on the current postgreSQL architecture. On the other hand, because I can not elicit the software requirements with certain accuracy, I can not perform some important analysis on performance or availability issues. For example, even when I identified that postgreSQL uses pipeline model to foster performance, I could not say if it can meet the performance requirements in an e-business application using postgreSQL as its backend database. It is impossible to produce accurate requirement criteria for a single software component without considering the runtime environment.

My conclusion on this analysis is that ATAM is a useful method to evaluate system architecture and direct developments although it faces some difficulties when used to evaluate a single general purpose software component. If we generalize the requirements for this kind of software components, we still can apply this method and identify the possible pitfalls and trends although we lose some precision.

Additionally, in the process of implementing ATAM, I also clarified module and runtime views of postgreSQL, and identified architectural styles deployed and pitfalls in the current architecture. In general, postgreSQL supports most features provided by many modern commercial systems. These features include the standard SQL language interface, rules system, user-defined functions, types, procedure language, transaction controls, crash recovery, multiple language interface libraries, etc. So we should say postgreSQL's architecture is successful for fulfilling its purpose as a pioneer of modern ORDBMS systems. However, this success does not mean its architecture is a good example of such a system. By investigating its implementation, I found the modules and subsystems are highly coupled with each other. The size of the project now is well beyond 250,000 lines of code. The highly coupled dependencies make adding new features or modifying the existing ones very difficult. Also, the whole system is highly dependent on the UNIX system call interface, which makes porting to another platform very hard. These dependencies are the result of lacking a good architectural design in the early phase of building it. This is a lesson we should learn. This also makes it clear that architecture is important for software developments and ATAM is an important method we should apply when starting a new project. Although doing ATAM seems tedious in the first place, finally it will prove worthwhile for what you get from it.

Bibliography

[1] Garlan, D., Shaw, M., *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, 1993, pp. 1–39.

[2] Klein, M., Kazman, R., *Attribute-Based Architectural Styles*, Software Engineering Institute, Carnegie Mellon University, 1999.

[3] Kazman, R., Klein, M., Clements, P., *ATAM: Method for Architecture Evaluation*, Software Engineering Institute, Carnegie Mellon University, 2000.

[4] Hofmeister, C., Nord, R., Soni, D., **Applied Software Architecture**, Addison-Wesley, 2000.

[5] Jacobson, I., Booch, G., Rumbaugh, J., **The Unified Software Development Process**, Addison-Wesley, 1999.

[6] Booch, G., Rumbaugh, J., Jacobson, I., **The Unified Modeling Language User Guide**, Addison-Wesley, 1999.

[7] Stonebraker, M., Rowe, L.A., *The design of POSTGRES*, Proceedings of ACM SIGMOD'86. International Conference on Management of Data, 1986, pp. 340–355.

[8] Stonebraker, M., *The design of the POSTGRES storage system*, Proceedings of the Thirteenth International Conference on Very Large Data Bases, 1987, pp. 289–300.

[9] Stonebraker, M., Rowe, L.A., Hirohama, M., *The implementation of POSTGRES*, IEEE Transactions on Knowledge and Data Engineering, 1990, pp. 125–142.

[10] PostgreSQL Global Development Group, *PostgreSQL 7.1 Documentation*, PostgreSQL Interactive Documentation, 1996–2001.

[11] Lane, T., *Transaction Processing in PostgreSQL*, Open Source Database Conference, 2000.

[12] Momjian, B., *A Tour of PostgreSQL Internals*, Open Source Database Conference, 2000.

[13] Momjian, B., **PostgreSQL: Introduction and Concepts**, Addison-Wesley, 2000.

[14] Korth, H.F., Silberschatz, A., **Database System Concepts**, McGraw-Hill, 1986.

[15] Stroustrup, B., **The C++ Programming Language**, Addison-Wesley, 2000.

[16] Tanenbaum, A.S., Woodhull, A.S., **Operating System Design and Implementation 2nd Ed.**, Prentice Hall, 1997.

[17] Axel, T., Schreiner, H., Friedman, G.Jr., **Introduction to compiler construction with UNIX**, Prentice Hall, 1985.