Architectural Concerns for Flexible Data Management

Ionut Emanuel Subasu, Patrick Ziegler, Klaus R. Dittrich, Harald Gall
Department of Informatics, University of Zurich
{subasu,pziegler,dittrich,gall}@ifi.uzh.ch

ABSTRACT

Evolving database management systems (DBMS) towards more flexibility in functionality, adaptation to changing requirements, and extensions with new or different components, is a challenging task. Although many approaches have tried to come up with a flexible architecture, there is no architectural framework that is generally applicable to provide tailor-made data management and can directly integrate existing application functionality. We discuss an alternative database architecture that enables more lightweight systems by decomposing the functionality into services and have the service granularity drive the functionality. We propose a service-oriented DBMS architecture which provides the necessary flexibility and extensibility for general-purpose usage scenarios. For that we present a generic storage service system to illustrate our approach.

1. INTRODUCTION

Current Database Management Systems (DBMS) are extremely successful software products that have proved their capabilities in practical use all over the place. Also from a research point of view they show a high degree of maturity and exploration. Despite all progress made so far and despite all euphoria associated with it, we begin to realize that there are limits to growth for DBMS, and that flexibility is an essential aspect in DBMS architecture.

Over time DBMS architectures have evolved towards flexibility (see Figure 1). Early DBMS were mainly large and heavy-weight monoliths. Based on such an architecture, extensible systems were developed to satisfy an ever growing need for additional features, such as new data types and data models (e.g., for relational and XML data). Some of these systems allowed extensibility through application front ends at the top level of the architecture [4, 20, 22]. A similar approach is taken by aspect-oriented object database systems such as SADES [2], which support the separation of data management concerns at the top level of DBMS architecture. Based on this, further aspects can be added to the database as required. By using a monolithic DBMS structure, however, attempts to change DBMS internals will hardly succeed, due to the lack of flexibility.

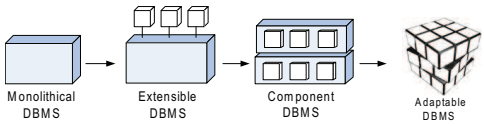

Figure 1: Evolution of DBMS architectures

In response to this, the next step in the evolution of DBMS architecture was the introduction of Component Database Systems (CDBS) [7]. CDBS allow improved DBMS flexibility due to a higher degree of modularity. As adding further functionality to the CDBS leads to an increasing number of highly dependent DBMS components, new problems arise that deal with maintainability maintainability, complexity, and predictability of system behavior.

More recent architectural trends apply concepts from interface and component design in RISC approaches to DBMS architectures [6]. RISC style database components, offering narrow functionality through well-defined interfaces, try to make the complexity in DBMS architecture more manageable. By narrowing component functionality, the behavior of the whole system can be more easily predicted. However, RISC approaches do not address the issue of integrating existing application functionality. Furthermore, coordinating large amounts of fine-grained components can create serious orchestration problems and large execution workflows.

Other approaches [13] try to make DBMS fit for new application areas, such as bio-informatics, document/content management, world wide web, grid, or data streams, by extending them accordingly. These extensions are customized, fully-fledged applications that can map between complex, application-specific data and simpler database-level representations. In this way, off-the-shelf DBMS can be used to build specialized database applications. However, adapting DBMS through a growing number of domain specific applications causes increasing costs, as well as compatibility and maintenance problems [21]. Such extensions can lead to hard wired architectures and unreliable systems [14].

The DBMS architecture evolution shows that, despite many approaches try to adapt the architecture, there is no architectural framework that is generally applicable to pro-

vide tailor made data management and directly integrate existing application functionality. In addition, none of the existing approaches can provide sufficient predictability of component behavior. Finally the ability to integrate existing application functionality cannot be achieved without requiring knowledge about component internals. These aspects, however, are essential for a DBMS architecture to support a broad range of user requirements, ranging from fully-fledged extended DBMS to small footprint DBMS running in embedded system environments. In consequence of that, the question arises whether we should still aim at an ever increasing number of system extensions, or whether it is time to rethink the DBMS approach architecturally.

In [23] we introduced a service based DBMS architecture which is based on the concepts of Service-Oriented Architecture (SOA) [15]. We argue that database services in the spirit of SOA are a promising approach to bring flexibility into DBMS architecture, as they are an adequate way to reduce or extend DBMS functionality as necessary to meet specific requirements. In this sense, we make DBMS architecture capable of adapting to specific needs, and, consequently, increase its "fitness for use".

In this paper, we focus on the notion of flexibility in DBMS architecture. We take a broad view on database architecture and present major requirements for a flexible DBMS architecture. Based on this, we propose a service oriented DBMS architecture which provides the necessary flexibility and extensibility for general-purpose usage scenarios. Instead of taking a bottom up approach by extending the architecture when needed, we use a top down approach and specialize the architecture when required. The main feature of our architecture is its flexibility and we do not primarily focus on achieving very high processing performance.

The remainder of the paper is structured as follows. In the next section we present aspects of flexibility that are relevant for extensible and flexible DBMS architectures. Section 3 introduces our Service Based Data Management System (SBDMS) architecture designed to meet these requirements. We illustrate and discuss our approach using examples in Section 4 and finally conclude in Section 5.

2. ASPECTS OF FLEXIBILITY

From a general view of the architecture we can see three main aspects that have to be addressed to achieve our goals: (1) extend the architecture with specialized functionality, (2) handle missing or erroneous parts, and (3) optimize the architecture functionality by allowing it to do the same task in different ways, according to user requirements or current system and architecture configurations. Note that while there may be other concerns, such as security, ease of maintainability, reliability, or system performance our focus in this paper is on the three aspects mentioned above, as flexibility defines the general character of an architecture.

Following the dictionary definition, "flexibility" can be interpreted as (1) admitting of being turned, bowed, or twisted without breaking or as (2) capable of being adapted [19]. This shows that there is not an exact way or metric to measure or increase the "flexibility" of an architecture. The general character of the term can also be seen in the IEEE definition [1] which sees flexibility as the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

From a different perspective, system flexibility can be seen as the system's capability of being extensible with new components and specializations. Here, "flexibility" refers to the ease with which a system or component can be modified to increase its storage or functional capacity [1]. Extensibility itself, however, does not suffice to create a flexible architecture because it neglects the case of downsizing the architecture. Despite of this, extensibility can be considered as a subcase of flexibility. If the architecture is not able to enable the appropriate changes and adapt to the environment we say that it has limited flexibility.

To provide a systematic view of the architecture flexibility, we have to consider different aspects of flexibility [16]. From the above definitions and the general character of the architecture, the following main aspects of flexibility can be considered:

- *Flexibility by selection* refers to the situation in which the architecture has different ways of performing a desired task. This is the case when different services provide the same functionality using the same type of interfaces. In this scenario no major changes at the architectural level are required.
- *Flexibility by adaptation* handles the case of absent or faulty components that cannot be replaced. Here, existing internal instances of the architecture that provide the same functionality can be adapted to cope with the new situation.
- *Flexibility by extension* allows the system to be adapted to new requirements and optimizations that were not fully foreseen in the initial design.

System quality can be defined as the degree to which the system meets the customer needs or expectations [1]. As extensions can increase the applicability of a system, flexibility and extensibility are important aspects of the quality of an architecture, which can be described as its "fitness for use". To support tailored "fitness for use", future architectures should put more emphasis on improving their flexibility and extensibility according to user needs. In this way the architecture will have the possibility to be adapted to a variety of factors, such as the environment in which the architecture is deployed (e.g., embedded systems or mobile devices), other installed components, and further available services. Some users may require less functionality and services; therefore the architecture should be able to adapt to downsized requirements as well.

3. THE SBDMS ARCHITECTURE

Today, applications can extend the functionality of DBMS through specific tasks that have to be provided by the data management systems; these tasks are called *services* and allow interoperability between DBMS and other applications

[11]. This is a common approach for web-based applications. Implementing existing DBMS architectures from a service approach can introduce more flexibility and extensibility. Services that are accessible through a well defined and precisely described interface enable any application to extend and reuse existing services without affecting other services. In general, Service Oriented Architecture (SOA) refers to a software architecture that is built from loosely coupled services to support business processes and software users. Typically, each resource is made available through independent services that can be distributed over computers that are connected through a network, or run on a single local machine. Services in the SOA approach are accessed only by means of a well defined interface, without requiring detailed knowledge on their implementation. SOAs can be implemented through a wide range of technologies, such as RPC, RMI, CORBA, COM, or web services, not making any restrictions on the implementation protocols [10]. In general, services can communicate using an arbitrary protocol; for example, a file system can be used to send data between their interfaces. Due to loose coupling, services are not aware of which services they are called from; furthermore, calling services does not require any knowledge on how the invoked services complete their tasks.

In [23] we have introduced a Service Based Data Management System (SBDMS) architecture that can support tailored extensions according to user requirements. Founding the architecture on the principles of SOA provides the architecture with a higher degree of flexibility and brings new methods for adding new database features or data types. In the following we present the architecture using *components*, *connectors*, and *configurations*, as customarily done in the field of software architectures [3].

3.1 Architectural Components

Our SBDMS architecture is organized into general available functional layers (see Figure 2), where each layer contains specialized services for specific tasks:

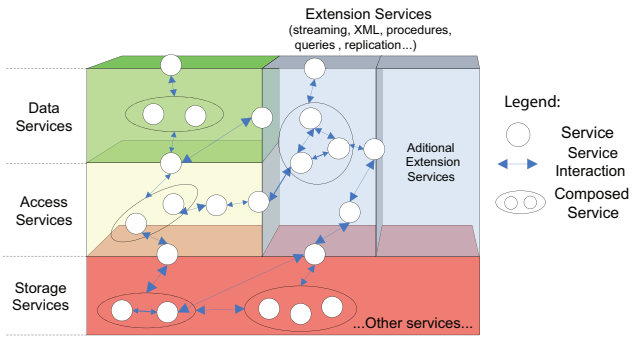

Figure 2: The SBDMS Architecture [23]

- *Storage Services* work at byte level and handle the physical specification of non-volatile devices. This includes services for updating and finding data.
- *Access Services* manage physical data representations of data records and access path structure, such as B-trees. This layer is also responsible for higher level operations, such as joins, selections, and sorting of record sets.
- *Data Services* present the data in logical structures like tables or views.
- *Extension Services* allow users to design tailored extensions to manage different data types, such as XML files or streaming data, or integrate their own application specific services.

As main components at a low-level, *functional services* provide the basic functions in a DBMS, such as storage services or query services. These services are managed by *coordinator services* that have the task to monitor the service activity and handle service reconfigurations as required. These services are handled by resource management processes which support information about service working states, process notifications, and manage service configurations. To ensure a high degree of interoperability between services, *adaptor services* mediate the interaction between services that have different interfaces and protocols. A predefined set of adapters can be provided to support standard communication protocol mediation or standard data types, while specialized adaptors can be automatically generated or manually created by the developer [17]. Service repositories handle service schemas and transformational schemas, while service registries enable service discovery.

3.2 Architectural Connectors

Connectors have the role to define the type of communication that takes place between software components [8]. Services present their purpose and capabilities through a *service contract* that is comprised of one or more service documents that describe the service [10]. To ensure increased interoperability, services are described through a *service description* document that provides descriptive information, such as used data types and semantic description of services and interfaces. A *service policy* includes service conditions of interaction, dependencies, and assertions that have to be fulfilled before a service is invoked. A *service quality* description enables service coordinators to take actions based on functional service properties. To ensure a high degree of interoperability, service contract documents should be described using open formats, such as WSDL or WS Policy. *Service communication* is done through well-defined communication protocols, such as SOAP or RMI. Communication protocols can be defined according to user requirements and the type of data exchanged between services. An important requirement is to use open protocols, rather than implementation specific technology. This allows one to achieve a high degree of abstraction and reduces implementation details in service contracts, which can reduce service interoperability.

3.3 Architectural Configurations

Configurations of the SBDMS depend on the specific environment requirements and on the available services in the system. To be adaptable, the system must be aware of the environment in which it is running and the available resources. Services are composed dynamically at run time according to architectural changes and user requirements.

From a general view we can envision two service phases: the setup phase and the operational phase. The *setup phase* consists of process composition according to architectural properties and service configuration. These properties specify the installed services, available resources, and service specific settings. In the *operational phase* coordinator services monitor architectural changes and service properties. If a change occurs resource management services find alternate workflows to manage the new situation. If a suitable workflow is found, adaptor services are created around the component services of the workflows to provide the original functionality based on alternative services. The architecture then undergoes a configuration and composition process to set the new communication paths, and finally compose newly created services. This is made possible as services are designed for late binding, which allows a high degree of flexibility and architecture reconfigurability.

3.4 Architectural Flexibility by Extension

Instead of hard-wiring static components we break down the DBMS architecture into services, obtaining a loosely coupled architecture that can be distributed. Such a service-based architecture can be complemented with services that are used by applications and other services allowing direct integration and development of additional extensions. In general, services are dynamically composed to accomplish complex tasks for a particular client. They are reusable and highly autonomous since they are accessed only through well-defined interfaces and clearly specified communication protocols. This standardisation helps to reduce complexity, because new components can be built with existing services. Organising services on layers is a solution to manage and compose large numbers of services. Developers can then deploy or update new services by stopping the affected processes, instead of having to deal with the whole system, as in the case of CDBS. System extensibility benefits from the service oriented basis of the architecture, due to a high degree of interoperation and reusability. In this manner future development within our architectural framework implies only low maintenance and development costs.

3.5 Architectural Flexibility by Selection

By being able to support multiple workflows for the same task, our SBDMS architecture can choose and use them according to specific requirements. If a user wants some information from different storage services, the architecture can select the order in which the services are invoked based on available resources or other criteria. This can be realised in an automated way by allowing the architecture to choose required services automatically, either based on a service description or by the user who manually specifies different workflows. Using extra information provided by other service execution plans, the service coordinators can create task plans and supervise them, without taking into consideration an extensive set of variables, because services just provide functionality and do not disclose their internal structure.

3.6 Architectural Flexibility by Adaptation

Compared with flexibility by selection, flexibility by adaptation is harder to achieve. If a service is erroneous or missing, the solution is to find a substitute. If no other service is available to provide the same functionality through the same interfaces, but if there are other components with different interfaces that can provide the original functionality, the architecture can adapt the service interfaces to meet the new requirements. This adaptation is done by reusing or generating adaptor services in the affected processes, to ensure that the service communication is done according to the service contracts. The main issue here is to make the architecture aware of missing or erroneous services. To achieve this we introduce architecture properties that can be set by users or by monitoring services when existing components are removed or are erroneous. SOA does not provide a general way to make the architecture aware and adaptable to changes in the current state of the system. Standardised solutions to this problems have been proposed by new architectures that have emerged on the foundations provided by SOA. One of these is the Service Component Architecture (SCA) [18]. SCA provides methods and concepts to create components and describe how they can work together. The interactions between components can be modeled as services, separating the implementation technology from the provided functionality. The most atomic structure of the SCA is the *component* (see Figure 3). Components can be combined in larger structures forming *composites* (see Figure 4). Both components and composites can be recursively contained.

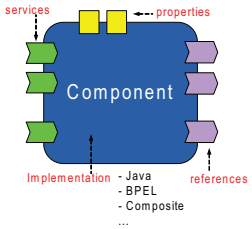

Figure 3: SCA Component [18]

Every component exposes functionality in form of one or more *services*, providing the number of *operations* that can be accessed. Components can rely on other services provided by other components. To describe this dependency, components use *references*. Beside services and references, a component can define one or more *properties*. Properties are read by the component when it is instantiated, allowing to customize its behaviour according to the current state of the architecture. By using services and references, a component can communicate with other software or components through *bindings*. A binding specifies exactly how communication should be done between the parties involved, defining the protocol and means of communication that can be used with the service or reference. Therefore a binding separates the communication from the functionality, making life simpler for developers and designers [5]. Furthermore, SCA or-

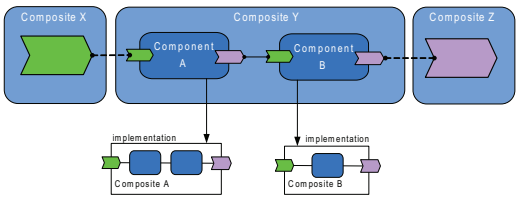

Figure 4: SCA Composites [18]

ganises the architecture in a hierarchically way, from coarse grained to fine grained components. This way of organizing the architecture makes it more manageable and comprehensible [12]. By using component properties, the adaptability character of the architecture can be easier achieved in a standardised way. For all these reasons, we include the principles of SCA into our SBDMS architecture.

3.7 A Storage Service Scenario

To exemplify our approach, we assume a simple storage service scenario and demonstrate how the three main aspects of flexibility from Section 2 can be realised using our proposed architecture. Figure 5 depicts the situation of adding a new

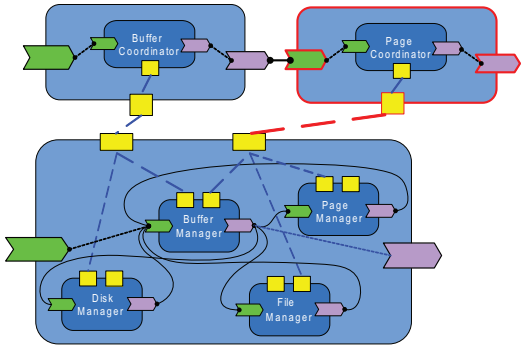

Figure 5: Flexibility by extension

service to the architecture. The user creates the required component (e.g., a Page Coordinator, as shown in Figure 5) and then publishes the desired interfaces as services in the architecture. From this point on, the desired functionality of the component is exposed and available for reuse. The service contract ensures that communication between services is done in a standardised way. In this way we abstract from implementation details and focus on the functionality provided by the system components. This allows ease of extensibility.

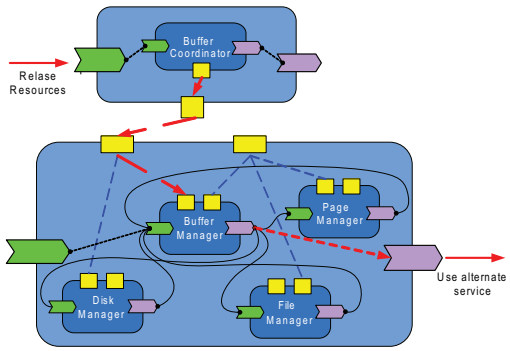

Figure 6: Flexibility by selection

At some point in time, special events may occur. Assume that some service S requires more resources. In this case, S invokes a "Release Resources" method on the coordinator services to free additional resources (see Figure 6). In our architecture component properties can then be set by users or coordinator services to adjust component properties according to the current architecture constraints. In this manner, other services can be advised to stop using the service due to low resources.

Coordinator services also have the task to verify the availability of new services and other resources. In the example in Figure 6 a service requests more resources. The Buffer Coordinator advises the Buffer Manager to adapt to the new situation, by setting the appropriate service properties. In this case the Buffer Manager can use an alternate available workflow by using other available services that provide the same functionality. Every component behaves as defined by the alternate workflows which are managed by service coordinators. If services are erroneous or no longer available, and

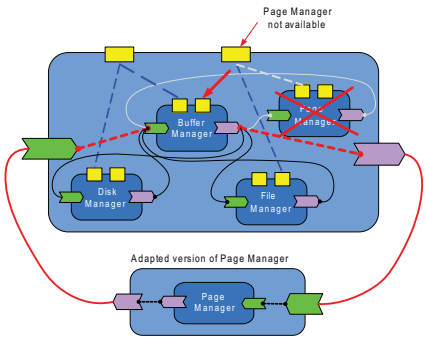

Figure 7: Flexibility by adaptation

other services can provide the same functionality, these can be used instead to complete the original tasks (see Figure 7). Even if performance may degrade to to increased work load, the system can continue to operate. If the service interfaces are compatible,coordinator services will create alternate processes that will compose the equivalent services to complete the requested task, in this way the architecture recomposes the services. Otherwise adaptor services have to be created to mediate service interaction.

4. DISCUSSION

To discuss and further illustrate our architectural concepts, we present two contrasting examples: a fully-fledged DBMS bundled with extensions and a small footprint DBMS capable of running in an embedded system environment.

Users with extensive functional requirements benefit from the ability of our architecture to integrate existing services that were developed by the user to satisfy his needs. Application developers can reuse services by integrating specialized services from any architectural layer into their application. For example developers may require additional information to monitor the state of a storage service (e.g., work load, buffer size, page size, and data fragmentation). Here, developers invoke existing coordinator services, or create customised monitoring services that read the properties from the storage service and retrieve data. In large scale architectures multiple services often provide the same functionality in a more or less specialised way. Since services are monitored by coordinators that supervise resource management and because service adaptors ensure correct communication between services, changes or errors in the system can be detected and alternate workflows and process compositions can be generated to handle the new situation.

If a storage service exhibits reduced performance that no longer meets the quality expected by the user, our architecture can use or adapt an alternative storage service to prevent system failures. Furthermore, storage services can be dynamically composed in a distributed environment, according to the current location of the client to reduce latency times. To enable service discovery, service repositories are required. For highly distributed and dynamic settings, P2P style service information updates can be used to transmit information between service repositories [9]. An open issue remains which service qualities are generally important in a DBMS and what methods or metrics should be used to quantify them.

In resource restricted environments, our architecture allows to disable unwanted services and to deploy small collections of services to mobile or embedded devices. The user can publish service functionality as web services to ensure a high degree of compatibility, or can use other communication protocols that suit his specific requirements. Devices can contain services that enable the architecture to monitor service activity and functional parameters. In case of a low resource alert, which can be caused by low battery capacity or high computation load, our SBDMS architecture can direct the workload to other devices to maintain the system operational. Disabling services requires that policies of currently running services are respected and all dependencies are met. To ensure this, service policies must be clearly described by service providers to ensure proper service functioning.

5. CONCLUSIONS

In this paper we proposed a novel architecture for DBMS that achieves flexibility and extensibility by adopting service-orientation. By taking a broad view on database architecture, we discussed aspects of flexibility that are relevant for extensible and flexible DBMS architectures. On this foundation, we designed our SBDMS framework to be generally applicable to provide tailored data management functionality and offer the possibility to directly integrate existing application functionality. Instead of taking a bottom up approach by extending an existing DBMS architecture, we use a top down approach and specialize our architecture to support the required "fitness for use" for specific application scenarios, ranging from fully-fledged DBMS with extensive functionality to small footprint DBMS in embedded systems. We exemplified how essential aspects of flexibility are met in our architecture and illustrated its applicability for tailor-made data management.

In future work we are going to design the proposed architecture in more detail and define a foundation for concrete implementations. We plan to take existing light weight databases, brake them into services, and integrate them into our architecture for performance evaluations. Testing with different levels of service granularity will give us insights into the right tradeoff between service granularity and system performance in a SBDMS.

6. REFERENCES

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 10 Dec 1990.

[2] R. Awais. SADES - a Semi-Autonomous Database Evolution System. In *ECOOP '98: Workshop on Object-Oriented Technology*, pages 24–25. Springer, 1998.

[3] L. Bass and others. *Software Architecture in Practice*. AWLP, USA, 1998.

[4] M. Carey et al. The EXODUS Extensible DBMS Project: An Overview. In *Readings in Object-Oriented Database Systems*, pages 474–499. MKP, 1990.

[5] D. Chappel. Introducing SCA. Technical report, Chappell & Associates, 2007.

[6] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. *The VLDB Journal*, pages 1–10, 2000.

[7] K. Dittrich and A. Geppert. *Component Database Systems*. Morgan Kaufmann Publishers, 2001.

[8] S. Dustdar and H. Gall. Architectural Concerns in Distributed and Mobile Collaborative Systems. *Journal of Systems Architecture*, 49(10-11):457–473, 2003.

[9] S. Dustdar and W. Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[10] T. Erl. *SOA Principles of Service Design*. PTR, 2007.

[11] A. Geppert et al. KIDS: Construction of Database Management Systems Based on Reuse. Technical Report ifi-97.01, University of Zurich, 1997.

[12] M. Glinz et al. The Adora Approach to Object-Oriented Modeling of Software. In *CAiSE 2001*, pages 76–92, 2001.

[13] J. Gray. The Revolution in Database System Architecture. In *ADBIS (Local Proceedings)*, 2004.

[14] T. Härder. DBMS Architecture – New challenges Ahead. *Datenbank-Spektrum (14)*, 14:38–48, 2005.

[15] S. Hashimi. Service-Oriented Architecture Explained. Technical report, O'Reilly, 2003.

[16] P. Heinl et al. A Comprehensive Approach to Flexibility in Workflow Management Systems. *WACC '99*, pages 79–88, 1999.

[17] H. R. Motahari Nezhad et al. Semi-automated adaptation of service interactions. In *WWW '07*, pages 993–1002, 2007.

[18] OASIS. *SCA Service Component Architecture, Specification*. 2007.

[19] Oxford Online Dictionary. `http://www.askoxford.com`.

[20] H. Schek et al. The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE TKDE*, 2(1):25–43, 1990.

[21] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time has Come and Gone. In *ICDE '05*, pages 2–11, 2005.

[22] M. Stonebraker et al. The Implementation of POSTGRES. *IEEE TKDE*, 2(1):125–142, 1990.

[23] I. E. Subasu et al. Towards Service-Based Database Management Systems. In *BTW Workshops*, pages 296–306, 2007.