PARALLELISM, ADAPTATION AND FLEXIBLE DATA ACCESS IN ON-DEMAND ERP SYSTEMS

Vadym Borovskiy[1], Wolfgang Koch[2] and Alexander Zeier[1]
[1]*Hasso-Plattner-Institute, Potsdam, Germany*
[2]*SAP AG, Walldorf, Germany*

Keywords: ERP system architecture, ERP data provisioning, Business object query language.

Abstract: On-premise enterprise resource planning (ERP) systems are costly to maintain and adapt to specific needs. To lower the cost of ERP systems an on-demand consumption model can be employed. This requires ERP systems to support multi-tenancy and multi-threading to enable consolidation of multiple businesses onto the same operational system. To simplify the adaptation of ERP systems to customer-specific requirements the former must natively support extensions, meaning that customer-specific behavior must be factored out from a system and placed into an extension module. In this paper we propose the architecture of an ERP system that i) exploits parallelism and ii) is able to accommodate custom requirements by means of enterprise composite applications. We emphasize the importance of ERP data accessibility and contribute with a concept of business object query language that allows building fine-grained queries. All suggestions made in the paper have been prototyped.

1 INTRODUCTION

To effectively manage information and automate business processes companies use enterprise resource planning (ERP) systems. ERP systems are always associated with high cost and force companies to commit themselves to specific software for quite long time, which is not desirable. In fact, businesses require the opposite: low cost, flexible, and scalable ERP systems, which require no special skills to use them and are capable of delivering ERP data on both desktop computers and mobile devices. Because classical on-premise ERP systems (e.g. SAP R/3 and SAP Business Suite) cannot fulfill such expectations, the concept of software as a service (SaaS) penetrated ERP sector. Although offering clear advantages over existing on-premise systems, turning the later into SaaS proved to be a difficult exercise. The main reason for this is systems' architecture consisting of tightly coupled elements and based on proprietary protocols.

The current work contributes with the architecture of an ERP system that suits the SaaS model. This goal is achieved by decoupling the components of an ERP system, employing parallelism and providing flexible data access API. The importance of the last element is highly emphasized. The corner stone of the API is the business object query language (BOQL) offering both the flexibility of SQL and encapsulation of SOA. In addition to architecture of an on-demand ERP system, the paper address the issue of adapting a system to customer-specific needs, which has proved to be one of the hottest topics in ERP systems development. The adaptation is achieved by leveraging the suggested data access API with enterprise composite applications (ECAs). ECA is a means of adding customer-specific features on top of data and functionality of an ERP system.

1.1 ERP as a Service

SaaS gains more and more momentum. Subscribing to applications over the Internet and using them via industry-standard browsers or Web services clients proved to be efficient from both economic and computational points of view (Jacobs, 2005). In the SaaS model, a provider develops an application and operates the servers that host it. The provider's benefits stems from the economies of scale. The provider aggregates many users and can share all aspects of the IT infrastructure, including hardware, software, staffing, and the data center itself among a number of subscribers. From a subscriber's point of view the benefit comes from converting the fixed cost of own-

Borovskiy V., Koch W. and Zeier A. (2010).
PARALLELISM, ADAPTATION AND FLEXIBLE DATA ACCESS IN ON-DEMAND ERP SYSTEMS.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 16-22
DOI: 10.5220/0002932800160022
Copyright © SciTePress

ing and maintaining on-premise infrastructure into the variable cost of renting it on demand. On average on-premise infrastructure is underutilized, because its capacity is driven by systems' peak load. But peak loads account for a small part of systems' operating time. For example, Amazon's infrastructure has been designed to guarantee an appropriate service level during the Christmas week, but the workload during the rest of the year is significantly lower. Therefore, most of the time Amazon's infrastructure is idle. Companies make large investments in infrastructure and find it idle for most of the time. Hence, by subscribing to SaaS companies pay only for actual usage, whereas with on-premise hardware the amount of resources they pay for is driven not by actual, but peak workload. The usage-based pricing model not only reduces capital expense, but also lowers an entry barrier to computing intensive businesses. By subscribing to on-demand services companies minimize risks of entering a business and avoid sunk cost.

Despite the clear cost advantage SaaS has a number of drawbacks. Most of them have to do with the principle of infrastructure sharing. In this regard SaaS generates privacy and security concerns, because not only a company's data are stored outside of the company, but also the company has no influence on how its ERP provider manages the data. Another drawback of infrastructure sharing is having the possibility of different subscribers affecting each other (mySql, 2007). Nevertheless, the attractiveness of the SaaS concept and the efforts of ERP software vendors outweigh the challenges and SaaS gradually enters ERP sector.

The shift to SaaS model automatically implies that existing ERP systems' architecture must be revised in order to meet new requirements. The section 2 of this paper defines what it means to an ERP system to be on-demand and how this is achieved.

2 ON-DEMAND ERP SYSTEM

What does it mean for an ERP system to be on-demand? To answer this question we need to address the cornerstone principle of on-demand systems - infrastructure sharing. According to this principle a system must be able to consolidate a big number (tens or even hundreds) of subscribers. This implies multi-tenancy in database layer and parallelism in application server layer. Multi-tenancy is basically mapping a number of single-tenant logical schemas in one multi-tenant physical schema in the database. Because the topic of multi-tenancy has received high attention in the database research community we redirect the reader to (Aulbach et al., 2008), (Jacobs et al., 2009), (mySql, 2007). The topic of parallelization is addressed in the following subsection.

2.1 Parallelism in ERP System

Because an on-demand system serves a big number of subscribers increasing the throughput of the system becomes an important goal when designing the system. In our opinion, the increase of throughput must be achieved by parallelization of request processing.

To enable efficient parallel execution of requests the later must be independent from each other, that is no task should require the results of other tasks' execution. In addition no task should compete for the system's resources with others. Task independence implies no synchronization among the threads of the system, which in turn allows to execute the threads on different processors/cores at the same time without blocking. In this case a system can dynamically increase and decrease its throughput by acquiring and releasing computational resources depending on the workload placed on the system by its users.

To minimize the number of tasks that need synchronization the principle of separation of concerns must be applied. According to this principle an information system is separated into distinct elements that overlap in functionality as little as possible. All ERP systems that we are aware of use three-tier architecture. The three-tier architecture scales well and can exploit the potential of massive parallelism. Each of the three tiers can accomplish its tasks in separate threads/processes, which will not be blocked by each other. When a tier receives a request it can spawn a child thread to process the request independently of other tasks in the system. To improve responsiveness and availability of a system asynchronous communication between the tiers can be employed.

The separation of tiers from each other rises a challenge of interfacing them. The interfaces among the tiers should enable efficient data access, meaning that the system must be able to satisfy data requests of any granularity. Efficient data access is an indicator of an ERP system's openness and plays a crucial role in enterprise composite application development. Therefore, the next two sections address this issue in details. The goal of the discussion in the sections is to understand how internals of an ERP system must be organized in order to enable efficient data access to ERP data.

2.2 Flexible Data Access

Data access API of an ERP system highly depends on internal details of the system. Not all architectures can efficiently support data access from outside of the system. In fact, most of the systems never allow such access.

2.2.1 State of the Art

A straightforward approach to data access can be to use SQL. Since ERP systems rely on relational databases, SQL statements could be issued directly against the databases to retrieve required data. Although SQL is natively supported by the underlying databases, this approach is unlikely to deliver the expected results. SQL statements need to be written against an actual schema of a database. The problem with this approach is that it violates the data encapsulation principle. Basically SQL exposes too much of control over the underlying database and greatly increases the risk of corrupting data in the system. An ERP system is not only a collection of structured data, but also a set of business rules that apply to the data. Generally these rules are not a part of the system's database. Direct access to the database circumvents the rules and implies data integrity violation. Therefore, to enforce the rules the direct access to data by any means is strictly prohibited.

An alternative to SQL can be data as a service approach. In this case a system exposes a number of Web services with strictly defined semantics. This approach has an advantage of hiding internal organization of data. Instead of a data schema a set of operations that return data are exposed by a system. By choosing operations and calling them in an appropriate sequence required data can be retrieved. Because of using Web services this approach is platform independent. However, this method has a serious disadvantage: lack of flexibility. Although an ERP system vendor can define many data accessing operations, they will never cover all possible combinations of data pieces of an ERP system. Often these operations are limited to one business object. ECAs on the other hand address very specific or fine-granular needs and deliver value by assembling information coming from different locations of a system. Therefore, the granularity of data services does not match the granularity of ECAs' operations and the services cannot provide adequate support for the ECAs. For this reason ECAs need to issue multiple service calls and combine a result set on their own. This greatly complicates the development of ECAs and undermines their performance. This situation clearly demonstrates the advantage of the SQL approach. The ability to construct fine-granular queries that fully match the information needs of ECAs makes SQL a much more flexible API than data as a service.

As one can see both approaches have advantages and disadvantages. SQL as a data access API gives great flexibility by allowing to construct queries that match the granularity of a user's information needs. However, SQL exposes too much control over the database, circumvents business logic rules and binds ECAs to a specific implementation platform. The data as a service approach on the other hand enforces business rules by exposing a set of Web operations which encapsulate data access and hide data organization. However, the granularity of the exposed operations does not match that of a user's needs, which creates inflexibility and hits performance.

2.2.2 Business Object Query Language

In this subsection we contribute with an idea of how to combine the advantages of discussed approaches while eliminating their disadvantages and propose a concept called business object query language (BOQL).

It is clear that accessing raw data directly and circumventing business logic contradicts with data encapsulation. For this reason business objects appeared. They fully control the access to the data and protect the integrity of data. From external perspective business objects are simply a collection of semantically related data, e.g. *invoice, bill of material, purchase order*, and a number of business operations that can be performed on the data, e.g. *read, create, etc*. A business object can be represented as set of data fields or attributes, e.g. *id, count, name*, and associations or links with other business objects, e.g. a *SalesOrder* is associated with a *Customer* and *Product* business objects.

Despite the diverse semantics of business objects they all have the same structure (an array of attributes and associations) and behavior (a set of operations). After a number of experiments we have found that the most important operations to support by a business object are *Create, Retrieve, RetrieveByAssociationChain, Delete* and *Update*. Therefore, all business objects can be derived from the same base class featuring the mentioned arrays and operations. Such uniform behavior and structure allow to introduce a query language for business objects very much like SQL for relational entities. We propose the following scenario:

1. A programmer composes a query, the description of what to retrieve from the system, according to some SQL-like grammar and sends the query as

a string to the system via a generic service operation, for example *Execute*.

2. The system parses the string to detect present clauses (*from, select, where,* etc.) and builds an abstract syntax tree - an internal representation of the query. The tree is then passed for further processing to a query execution runtime.
3. Using the *from* clause the runtime obtains references to the business objects from which the retrieval must be performed: source business objects. Then the runtime traverses the tree in a specific order and converts recognized query tokens to appropriate operation calls on the source business objects. For example, tokens from a *select* clause are converted to *Retrieve* or *RetrieveByAssociationChain* operations.
4. Having extracted the values from the query string, the runtime executes the operations with the extracted values passed as input parameters and composes the result. After that the result is formatted in XML and sent back to the calling programm.

In its essence the query language performs an orchestration of calls to objects' operations based on user-defined queries. These queries are transformed to a sequence of operation calls that yield the required data. Business object query language has an advantage of supporting fine-grained queries as in the case of SQL without circumventing business rules as in the case of the data as a service approach. Such an approach is allowed by a uniform representation of business objects (in terms of the structure and behavior).

This method of accessing data can be used by both an ERP system itself and ECAs. In the later case a problem of communicating ERP data model arises. In order to develop ECAs users must have a strong understanding of the system's business object model. In other words, they need to know what business objects are in the system, what attributes the objects have and how the objects are linked among each other. To communicate this information we use oriented graphs. The vertices of a graph denote business objects and oriented edges denote associations. A set of attributes is attached to every vertex (see Figure 2). For the sake of compactness we will not list the attributes on diagrams. The graph plays the same role for business objects as the schema for a database. It depicts the structure of business data and is essential to know to compose queries.

2.3 System's Architecture

The current subsection demonstrates a possible implementation of an ERP system built according to the principles discussed in earlier in this section in a way that uses massive parallelism. The Figure 1 sketches the architecture of a prototyped system. It has five elements: user interface, a user request handing and dispatching layer, a query engine, a business object engine and a storage.

In our prototype we used Silverlight as a UI-building technology, but any other technology capable of executing Web service calls can be used. The main responsibility of the user interface it to render forms based on the information in a user's configuration profile and the actual data returned by the backend system. Note that there can be two types of client application: (i) those that connect directly the backend system and profile storage and execute BOQL queries on their own and (ii) those that use intermediary request handling and dispatching layer. The applications of first type are so called fat clients and provide richer functionality and put less workload of the backend system, but are more complex. The later ones are thin clients that essentially are terminals via which users access the system. A good example of a thin client is an application running on a mobile device. For long time thin clients have been considered to be easier to deploy. With the technology like Silverlight this is no longer the case. A Silverlight application is essentially a .Net application running inside a Web browser. In other words this is a fully-fledged desktop application (fat client) running on the client side and hosted by a Web browser. Every time a user opens a web page with a Silverlight application the browser downloads the application and executes it. Hence with Silverlight we can achieve the power of a fat client for the deployment price of a thin client.

To effectively support thin clients (like mobile phones and Web pages with server-side logic like ASP.NET and PHP) we had to develop a scalable user request handling and dispatching layer. The point is that the Web service to which thin clients connect can easily become a bottleneck in the system. Therefore we factored out the actual BOQL query executing from the Web service to reduce its workload and thus improve the responsiveness and performance of the overall system. For this we run so called working processes (each process can be run on any physical server to which the Web service hosting machine can establish a tcp connection). This allowed us to avoid blocks of the Web service caused by waiting for results from the query engine. Now if a block occurs (because of long query execution time) none of the other working processes is affected. To minimize the thread and connection management overhead every working process has two objects instantiated at a process's start up time, namely the thread pool and connection pool. We encapsulate thread management related code inside a separate component in order to make threading transparent for the rest of the system. Whenever our system needs a thread it simply picks up one from the pool. The same motivation is behind the connection pool: whenever the system needs a connection it acquires one from the pool. Because pool pattern is a very well know approach we redirect the reader to other works ((Richter, 2008) for the thread pool, (Kircher and Jain, 2004) for the connection pool) for more details on it.

The business object engine manages business objects and the query engine provides access to them from outside of the owning process via a query-like interface. These two elements are instances of *Bo-Engine* and *QueryEngine* classes respectively. Both are created at the system's startup time. Business object engine is instantiated first to assemble business objects and store references to them in a pool. Because business objects are independent from each other they can be instantiated in parallel. This will greatly improve the system's start up time. Then the instance of the query engine is created. It has access to the pool and thus can manipulate the objects. Query engine is another potential bottleneck of the system. Because all BOQL queries go through this element of the system it may become overloaded and slow down the work of the system for this reason we used the same approach as with the Web service: we factored out query parsing and service call execution from query engine to independent processes.

Every business object encapsulates an in-memory table to cache data. The in-memory table is populated with data taken from a private database. Every object also encapsulates logic to synchronize/backup its in-memory table with the database. To improve the responsiveness of the system the synchronization is done independently for every business object and in a separate execution thread in the background mode. To the query execution runtime an object is seen through its interface: a collection of attributes and associations to other objects and CRUD (*Create, Retrieve, Update, Delete*) operations. How those are implemented is completely hidden inside the object. Typically, attributes and associations are bound to data fields and relations of the underlying physical storage or local in-memory cache. In this prototype we concentrate on only read operations (accessors) from the interface: *Retrieve* to get attributes of a given object and *RetrieveByAssociationChain* to navigate from one object to another via specified associations. These operations retrieve data from the underlying physical storage or the local cache according to internal business logic. By default the accessors assume one-to-one correspondence between a business object's logical and physical schemas. For example, if an attribute *Attr1* of a business object *Bo1* is queried the query runtime looks for data field named *Attr1* in a table corresponding to a given business object; if an association *Assoc1* of a *Bo1* is queried the runtime looks for a foreign key relationship corresponding to the association and constructs a join.

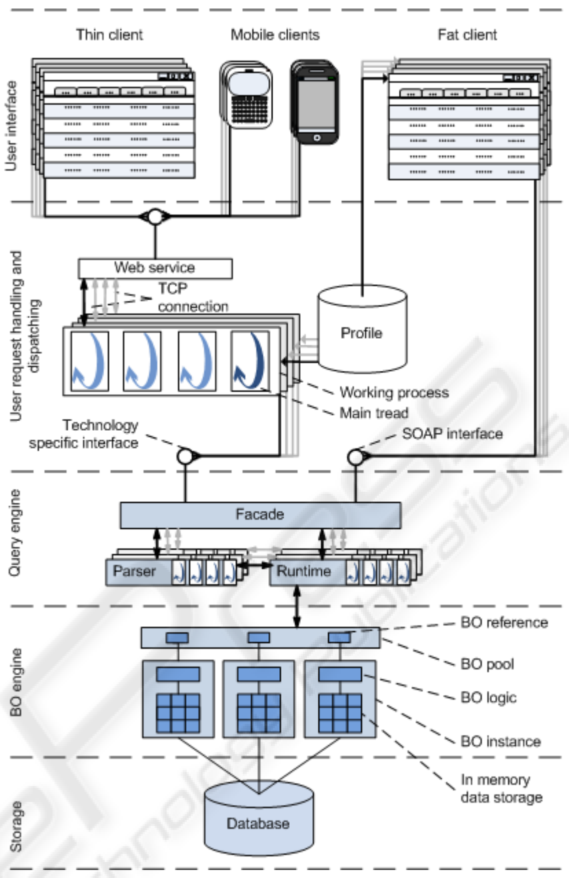

Figure 1: The architecture of the test system.

Neither a business object nor its in-memory cache nor database tables can be directly accessed outside of the owning process. The direct access to the data is prohibited to enforce integrity rules and internal business logic implemented by business objects. To access the business data an external application must use the standardized query-like interface exposed by the query engine. When the latter receives a query it acquires a thread object from the thread pool, parses the query and transforms recognized tokens to corresponding operation invocations. Then the work is handed over to the runtime engine that is responsible for performing actual service calls and constructing the result set. Whenever is possible the service calls are done in parallel. To figure out what calls to execute at the same time we analyze the abstract syntax tree built by the parser for a given query and issue every call, which does not rely on yet not retrieved data, in a separate thread. The result of these invocations is assembled in a single XML document and sent back to a client application.

As an implementation platform for the prototype we chose .NET. The system is implemented as a Windows service and the query interface is published as a Web service hosted by Internet Microsoft Information Services (IIS). The Web service is meant to dispatch a query to the system and serves as a request entry point. There is no other way to invoke or access the system except for issuing a call to the Web service. The physical data storage is implemented as a Microsoft SQL Server 2005 database.

2.4 Enterprise Composite Applications as Extension Modules

A composite application is an application generated by combining content, presentation, or application functionality from desperate Web sources. CAs aim at combining these sources to create new useful applications or services (Yu et al., 2008). An enterprise composite application is a CA application which has an ERP system as one of its sources[1]. CA access their sources via thoroughly specified application programming interface (API). The key characteristics of a CA are its limited/narrowed scope and straightforward result set. CAs often address situational needs and provide replies to fine-grained information requests. They are not intended to provide complex solutions for general problems rather they offer compact answers to clear-cut questions.

CAs create value by pulling all data and services a user needs to perform a task on a single screen. These data and services can potentially come from many sources, including an ERP system. Very often users are confronted with a problem of having necessary information and functionality distributed across many forms. By creating a CA that assembles them on the same screen users can substantially increase the productivity of their work. Additional benefit here is that a CA can present information in a way that meets personal preferences of a user.

[1]From now on we consider only enterprise composite applications. The terms "CA" and "ECA" for the sake of brevity are considered to mean the same in the rest of the paper.

The architecture we suggest natively supports CA. The main enabler of composite applications is the BOQL, which basically provides a mechanism for query-like invocation of business objects' services. BOQL allows CAs to manipulate ERP data from outside of a system without violating internal business logic. Because the query engine supports SOAP protocol, CAs can be developed and executed on any platform that is suitable for a user and has support for XML.

The process of developing CAs we see as follows.

1. A user[2] figure out on which business objects they want to perform custom operations. This depends on the actual task and application domain. Then the user composes BOQL queries that will return the data from the business objects. To compose the queries the user can use Object Explorer and Schema Explorer tools described earlier.
2. Using SOAP interface of the query engine the user executes the queries and retrieves ERP data.
3. Using selected programming language the user develops code that operates on the selected data and performs the required operations.
4. In case the CA needs to change the data in the source ERP system it composes BOQL queries that do so and executes the queries using the same SOAP interface of the query engine.

2.4.1 Business Case

In this section we present a business case, which benefits from the flexible data access of the suggested approach. Consider a Web retailer that sells items on-line and subcontracts a logistics provider to ship sold products to customers. The retailer operates in a geographically large market (e.g. the US or Europe[3]). In this situation the consolidated shipment of items can generate considerable savings in delivery and thus increase the profit of the retailer. Consolidation means that a number of sold items is grouped in a single bulk and sent as one shipment. The bigger the shipment size, the higher the bargaining power of the retailer when negotiating the shipment with a logistics provider. The savings come from price discounts gained from higher transportation volume[4]. In this way the retailer can lower the transportation cost per sold product. The consolidation of shipment is done anyway by all logistics providers in order to minimize their operating costs. By controlling the delivery of

[2]power user or application programmer

[3]The greater the territory and the higher the sales volume, the more relevant the case.

[4]So called economies of scale in transportation

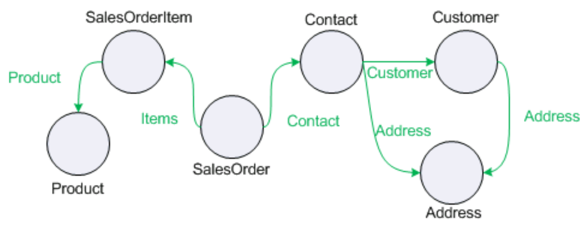

Figure 2: Schema of the Web retailer's CRM.

sold items explicitly, the retailer captures the savings that otherwise go to a logistics provider.

Let the retailer use a system with a business object graph as Figure 2 presents to manage their sales. We assume that the system exposes a query-like Web service interface as described in the section 2.3. The query returning the shipping address for all sales order items that are to be delivered looks as follows:

```
SELECT
  SO˜id, SO.Contact.Customer˜name,
  SO.Contact.Address˜FormattedAddress
FROM
  SalesOrder As SO
WHERE
  SO.Status = "ToDeliver"
GROUP BY
  SO.Contact.Address˜city
```

By invoking the query-liked Web service and passing the above query to it, a third-party application consolidates the items by their destination. The next step for the application is to submit a request for quote to a logistics provider and get the price of transporting each group of items. Many logistics providers have a dedicated service interface for this, so the application can complete this step automatically. Once the quote has been obtained and the price is appropriate the products can be packaged and picked up by the logistics provider.

To enable applications like the one just described the system must expose a flexible data access API. That is, the system must be able to return any piece of data it stores and construct the result set in a user-defined way. As mentioned in the section 2.2.1 traditional APIs cannot completely fulfill this requirement: SQL against views circumvents the business rules enforced outside the database; Web services limit the retrievable data to a fixed, predefined set. The architecture suggested in the current work overcomes the existing limitations and offer the necessary degree of data access flexibility.

3 CONCLUSIONS

On-demand model offers a better pricing option for enterprises than the traditional on-premise approach. For ERP software vendors a shift to ERP as a service implies challenge of making an ERP system able to consolidate multiple business customers. The challenge is intensified by the need of adapting standard ERP software to specific needs of customers.

The current work contributes with an architecture of an ERP system that exploit massive parallelism in order to cope with high workload put onto a system by many concurrent users and argues in favor of ECAs as an adaptation tool. The ability to support ECAs comes from the high accessibility of a system's data, which is achieved with the help of business object query language. We outlined the major components of the architecture and prototyped the system using Microsoft platform. We also demonstrated how an ECA can be developed.

REFERENCES

Aulbach, S., Grust, T., Jacobs, D., Kemper, A., and Rittinger, J. (2008). Multi-tenant databases for software as a service: schema-mapping techniques full. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1195–1206.

Jacobs, D. (2005). Enterprise software as service. *Queue*, 3(6):36 – 42.

Jacobs, D., Aulbach, S., Kemper, A., and Seibold, M. (2009). A comparison of flexible schemas for software as a service. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 881–888.

Kircher, M. and Jain, P. (2004). *Pattern-Oriented Software Architecture: Patterns for Resource Management*. Wiley.

mySql (2007). Anatomy of mysql on the grid. http://blog.mediatemple.net/weblog/2007/01/19/anatomy-of-mysql-on-the-grid/.

Richter, J. (2008). *Windows Via C/C++*. Microsoft Press.

Yu, J., Benatallah, B., Casati, F., and Daniel, F. (2008). Understanding mashup development. *IEEE Internet Computing*, pages 44–52.