Chapter 5

Solve Performance Problems with FastSOA Patterns

The previous chapters described the FastSOA patterns at an architectural level. This chapter shows FastSOA mid-tier service and data caching architecture applied in three real-world scenarios. The scenarios show how to accelerate SOA performance and mitigate performance problems through mid-tier service caching, native XML persistence, and mid-tier data transformation, aggregation, and federation.

5.1 Three Use Cases and the FastSOA Pattern

In this chapter, I describe three use cases where FastSOA is an appropriate solution for SOA performance and scalability challenges. Each use case shows how pure XML technology used in the mid-tier mitigates and solves performance and scalability problems and delivers flexibility unavailable with object and relational technology.

While there are many (sometimes contradictory) definitions for SOA, the majority of software developers and architects I have gotten to know over the years recognize and support SOA as a pattern built around consumers, services, and brokers. Figure 5-1 shows this relationship.

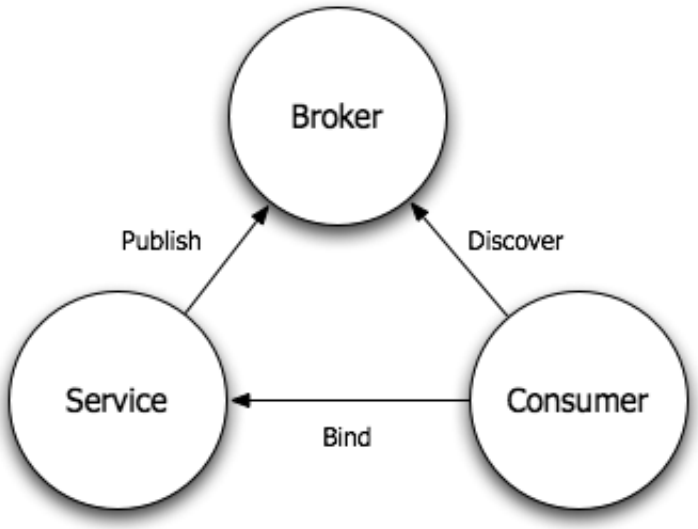

Figure 5-1 *The basic SOA pattern.*

The basic SOA patterns make sense for developers building services, Web services, and composite applications and data services. The pattern allows a consumer that makes a request to learn the location and interface message schema of a service. The consumer binds to the service by sending a request message. The service returns a response message to the consumer's request. The service makes its location known by publishing ontology of its functions and interface message schema to the broker. SOA is an abstract architecture—for instance, SOA does not define a specific protocol such as SOAP in the Web Services standard—but most SOA designs I have seen use XML as the message format between consumer, broker, and service.

To understand the SOA pattern in practice, we will look at three scenarios and show how FastSOA solves scalability, performance, and flexibility challenges in each.

- Accelerating service interface performance and scalability
- Improving SOA performance to access services
- Flexibility needed for semantic web, service orchestration, and services dynamically calling other services

5.2 Scenario 1: Accelerating Service Interface Performance and Scalability

In this scenario, a business operates a parts ordering service for customers. The service provides a Web browser user interface to enter a new order and learn the status of an existing order. Behind the user interface is a SOAP-based Web Service using ebXML message schemas to track order status from a vendor's legacy inventory system. The service stores orders and customer information in a relational database. Figure 5-2 illustrates a typical use case.

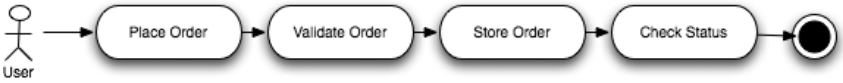

Figure 5-2 *A typical use case for a parts ordering service.*

The use case begins with a customer placing an order. The service validates the order against the current inventory to make sure the part being ordered is in the parts catalog. The service stores the order until the company places a consolidation of all the orders in a nightly batch process with the parts vendor. The service ends the use case by checking the status of the order.

Figure 5-3 illustrates an n-Tier architecture often recommended in the Java development community to implement the parts ordering service.

The architecture divides into three portions: A presentation tier, an application tier, and a data tier. The presentation tier uses a Web browser with AJAX and RSS capabilities to create a rich user interface. The browser makes a combination of HTML and XML requests to the application tier. Also at the presentation tier is a SOAP-based Web Services interface to allow a customer system to access the parts ordering functions. At the application tier, an Enterprise Java Bean (EJB) or plain old Java object (pojo) implements the business logic to respond to the request. The EJB uses a model, view, and controller (MVC) framework—for instance, Struts or Tapestry—to respond to the request by generating a response Web page. The MVC framework uses an object/relational (O/R) mapping framework—for instance, Hibernate or Spring—to store and retrieve data in a relational database.

There are three problem areas that cause scalability and performance problems when using Java objects and relational databases in XML environments. Figure 5-4 illustrates these problems.

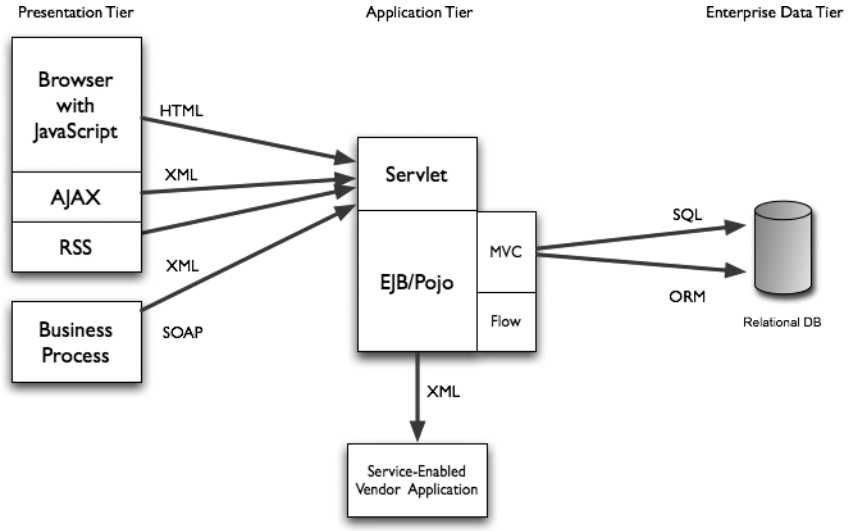

Figure 5-3 *Building the parts ordering service using the Domain pattern.*

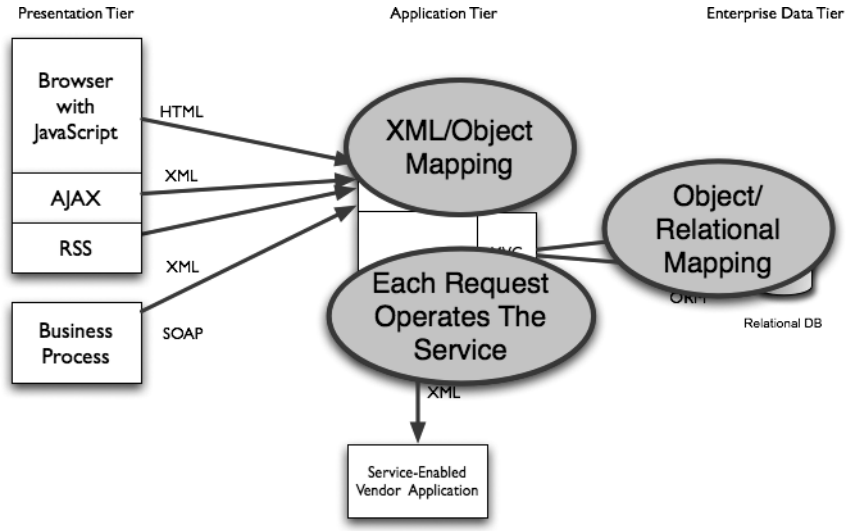

Figure 5-4 *The source of scalability and performance problems… all that mapping and transformation.*

Using the Java/relational architecture leads to performance and scalability problems as the XML messages grow in complexity and size.

- XML/Java mapping requires increasingly more processor time as XML message size and complexity grow.
- Each request operates the entire service. Many times the user will check order status sooner than any status change is realistic. If the system kept track of the most recent response's time-to-live duration, then it would not have to operate all of the service to determine the most previously cached response.
- The vendor application requires the request message to be in XML form. The data the EJB previously processed from XML into Java objects now needs to be transformed back into XML elements as part of the request message. Many Java to XML frameworks—for instance, JAXB, XMLBeans, and Xerces—require processor-intensive transformations. These frameworks challenge developers to write difficult and complex code to perform the transformation.
- The service persists order information in a relational database using an object/relational mapping framework. The framework transforms Java objects into relational rowsets and performs joins among multiple tables. As object complexity and size grow, many developers need to debug the O/R mapping to improve speed and performance.

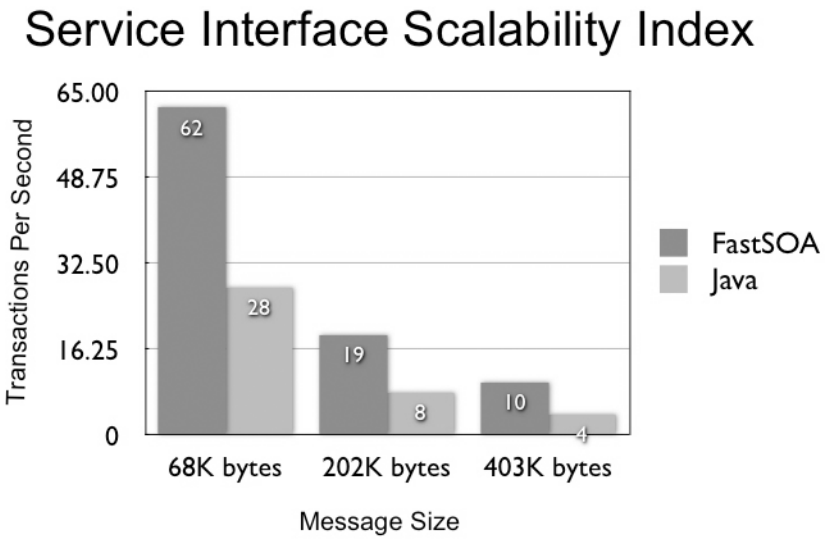

Figure 5-5 *Contrasting service interface performance between techniques.*

To give you an idea of the extent of the problem, consider the performance advantage of using native XML technology to respond to service requests. Figure 5-5 contrasts the performance difference.

The results in Figure 5-5 contrast native XML technology and Java technology to implement a service that receives SOAP requests. The test varies the size of the request message among three levels: 68 kilobytes, 202 kilobytes, and 403 kilobytes. The test measures the round-trip time to respond to the request at the consumer. The test results are from a server with dual CPU Intel Xeon 3.0-Ghz processors running on a gigabit-switched Ethernet network. I implemented the code in two ways:

1. **FastSOA technique**. Uses native XML technology to provide a SOAP service interface. I used Raining Data TigerLogic's XML query (XQuery) engine to expose a socket interface that receives the SOAP message, parses its content, and assembles a response SOAP message.
2. **Java technique**. Uses the SOAP binding proxy interface generator from a popular commercial Java application server. A simple Java object receives the SOAP request from the binding, parses its content using JAXB-created bindings, and assembles a response SOAP message using the binding.

The results show a 2 to 2.5 times performance improvement when using the FastSOA technique to expose service interfaces. The FastSOA method is faster because it avoids many of the mappings and transformations that are performed in the Java binding approach to work with XML data. The greater the complexity and size of the XML data, the greater the performance improvement.

FastSOA is equally applicable to improve the performance of SOA application requests that require access to data. FastSOA implements a mid-tier cache to commonly accessed data.

Caching is a powerful and proven technique for database systems. For instance, RDBMS tools vendors perform caching strategies at three points in an SOA environment, as illustrated in Figure 5-6.

RDBMS vendors recommend caching at three places. In the presentation tier, a Web cache provides cached data to Java Server Faces (JSF) dynamic pages. At the application tier, it recommends caching at the object/relational mapping (ORM) and in the connection layer to the relational database (RDBMS) using an in-memory

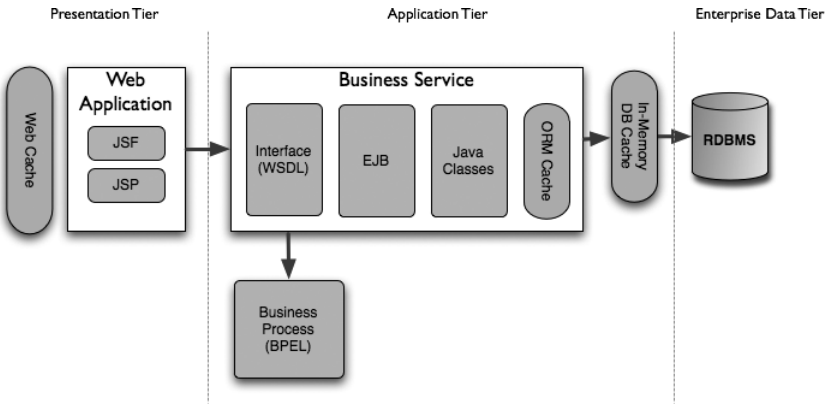

Figure 5-6 *Mitigating relational performance through caching.*

cache. Caching in this environment works well to mitigate relational performance problems.

These caching techniques should, but don't, take advantage of the unique features of XML data. Many SOA XML messages include a time-to-live value in the message itself. This gives the cache intelligence about the lifetime of data that is usually unavailable in generic data caching approaches. To achieve this cache intelligence requires a programming language and persistence engine of its own.

Figure 5-7 shows the FastSOA architecture that uses native XML technology to provide a service interface, to accelerate service performance by caching response data, and implements flexible and rapidly changed policies to operate the cache engine.

The advantage to using the FastSOA architecture as a mid-tier service cache is in its ability to store any general type of data, as well as its strength in quickly matching services with sets of complex parameters to efficiently determine when a service request can be serviced from the cache. The FastSOA mid-tier service cache architecture accomplish this by maintaining two databases.

- **The service database.** Holds the cached message payloads. For instance, the service database holds a SOAP message in XML form, an HTML Web page, text from a short message, and binary from a JPEG or GIF image.
- **The policy database.** Holds units of business logic that look into the service database contents and make decisions on servicing requests with data from the service database or passing through the request to the application tier. For

instance, a policy that receives a SOAP request validates security information in the SOAP header to validate that a user may receive previously cached response data. In another instance, a policy checks the time-to-live value from a stock market price quote to see if it can respond to a request from the stock value stored in the service database.

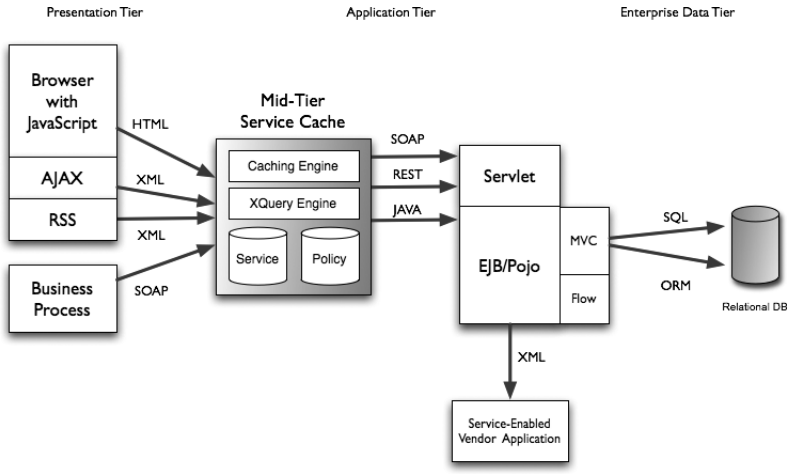

Figure 5-7 *Using XML technology to provide service acceleration through caching.*

FastSOA uses the XQuery data model to implement policies. The XQuery data model supports any general type of document and any general dynamic parameter used to fetch and construct the document. Used to implement policies, the XQuery engine allows FastSOA to efficiently assess common criteria of the data in the service cache, and the flexibility of XQuery allows for user-driven fuzzy pattern matches to efficiently represent the cache.

FastSOA uses native XML database technology for the service and policy databases for performance and scalability reasons. Relational database technology delivers satisfactory performance to persist policy and service data in a mid-tier cache provided the XML message schemas being stored are consistent and the message sizes are small. To understand this in more depth, consider the following results from a comparison of native XML database technology to relational databases.

The test runs multiple test cases where each test case varies the number of concurrent requests made to the database host and var-

ies the size of the XML message. The test environment makes requests to a relational database and a native XML database. For the relational database the test used an XML CLOB field type. The use case is modeled around a mid-tier cache's need to rapidly persist and query an unknown quantity of data with multiple and unknown schemas.

The test environment monitors the time it takes to receive a response at the consumer. The test produces a report showing transactions per second (TPS) at each level of concurrent requests and message payload sizes. Figure 5-8 shows a scalability index report for querying a database of stored XML documents.

Figure 5-8 *Comparing query performance between a native XML database and a relational database management system.*

In the query performance test, the native XML database starts at 45 TPS and goes up to 186 TPS while the relational database stays at 15 TPS or less. Figure 5-9 compares performance characteristics while inserting XML documents into the database.

The insert document performance test shows that native XML database and relational database performances are evenly matched at 20 and 17 TPS at the lowest number of concurrent requests. At 32 concurrent requests, the native XML database performs 3.4 times (48 TPS/14 TPS) as many inserts than the relational database performs.

These results are not meant to knock relational database technology out of the running for XML persistence, because there are

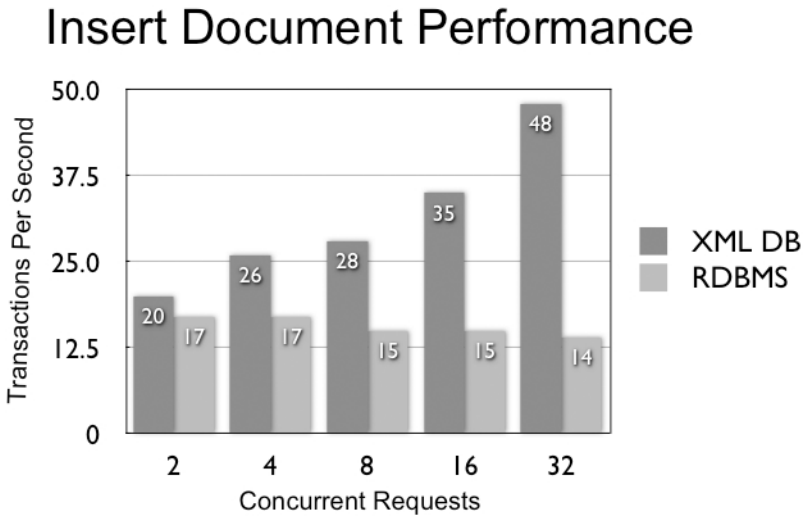

Figure 5-9 *Comparing insert performance.*

undoubtedly a large number of optimizations that could be employed to improve performance. Instead, I show these numbers to prepare you for the performance and scalability challenges in SOA environments that use an unknown variety of XML message sizes and schemas. There is no gatekeeper to the number and form of XML message schemas in an SOA environment. Mid-tier caching strategies need the best performing and most flexible database available that can handle multiple and unknown message schemas and data formats.

The benefits to a business running a FastSOA mid-tier service cache include:

- Less CPU licenses for commercial application servers
- Less network overhead
- Improved performance as compared with other mid-tier cache architectures
- Advanced SOAP in-line processing for a 2–3 times performance improvement over binding proxies created with Java application server utilities
- More efficient relational to XML transformation processing

The next scenario shows how FastSOA is used as a service data cache to improve data retrieval performance.

5.3 Scenario 2: Improving SOA Performance to Access Services

In this scenario, a business operates a portal for employees to sign up for a retirement plan, medical insurance plan, and other programs. The portal application interfaces to an external database using a JDBC driver to retrieve company news. It also interfaces using REST protocols (XML over HTTP) for employee benefits data from the human resources service. The portal allows single sign-in to these services by interoperating with the corporate directory using LDAP protocols. Figure 5-10 illustrates a typical use case.

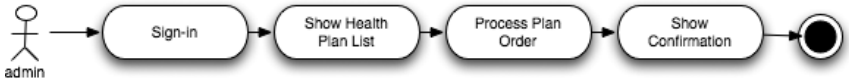

Figure 5-10 *The employee portal use case.*

The use case begins with an employee signing in. The portal application validates the user credentials. Validated users sign up for alerts to enable the system to send an email notification when new healthcare plans become available. The employee uses the portal to browse healthcare plans and choose a plan. The service ends the use case by confirming the plan choice.

Figure 5-11 illustrates an architecture often recommended in the Java development community to implement the employee portal.

The architecture divides into three portions: a presentation tier, an application tier, and a data tier. The presentation tier uses a Web browser with AJAX and RSS capabilities to create a rich user interface. The portal application also presents data to systems in other departments using SOAP and Java interfaces. At the application tier, an Enterprise Java Bean (EJB) implements the business logic to respond to the request. The EJB interoperates with a relational database, security manager, and human resources service. Corporate mandates require the system to store the plans offered to the employees in its original XML form.

This architecture puts a lot of emphasis on the EJB. The EJB must produce aggregated views of the healthcare plans according to the user's position in the company. The views are assembled from XML data sources (the human resources system) and LDAP security provider. There are three problem areas that cause scalability, performance, and developer productivity problems when using Java objects and relational databases in what is otherwise an XML environment in this scenario:

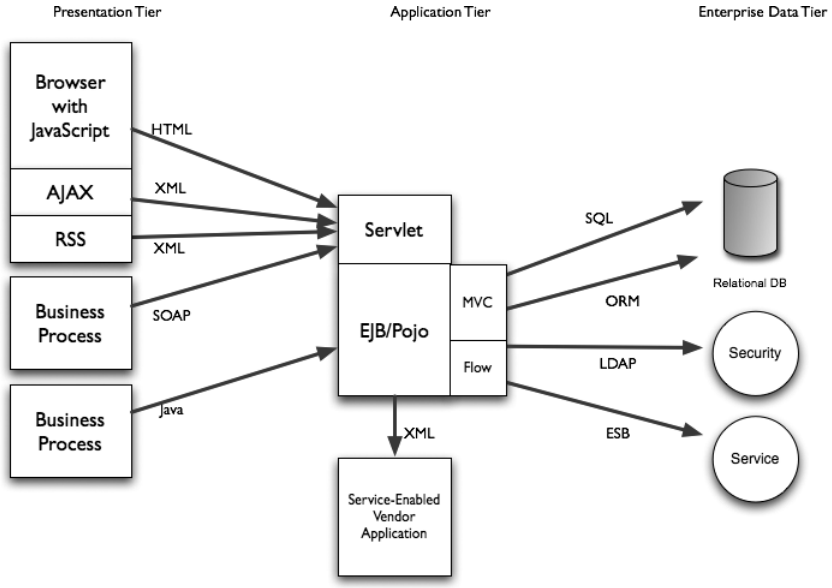

Figure 5-11 *The employee portal built with Java and relational technologies*

- **Slow object/relational mapping.** The views from the aggregated XML services need to be mapped into objects and then mapped into relational rowsets. This requires much coding and processing.
- **Every request accesses the services and database slowly.** There are only so many types of employees. Yet, the EJB assembles the views into the plan data each time a user requests a page.
- **Schema changes cause coding changes.** Every change to the human resources system message schema requires you to be right back in the code for the EJB.

Figure 5-12 shows an architecture that uses XML-centric technology to efficiently and rapidly create aggregate views of data from multiple data sources, accelerate data source performance by caching commonly used view data in the mid-tier, and implement flexible and rapidly changed policies to operate the data cache.

The mid-tier data cache stores XML, non-XML (such as binary), and other general types of data in two databases:

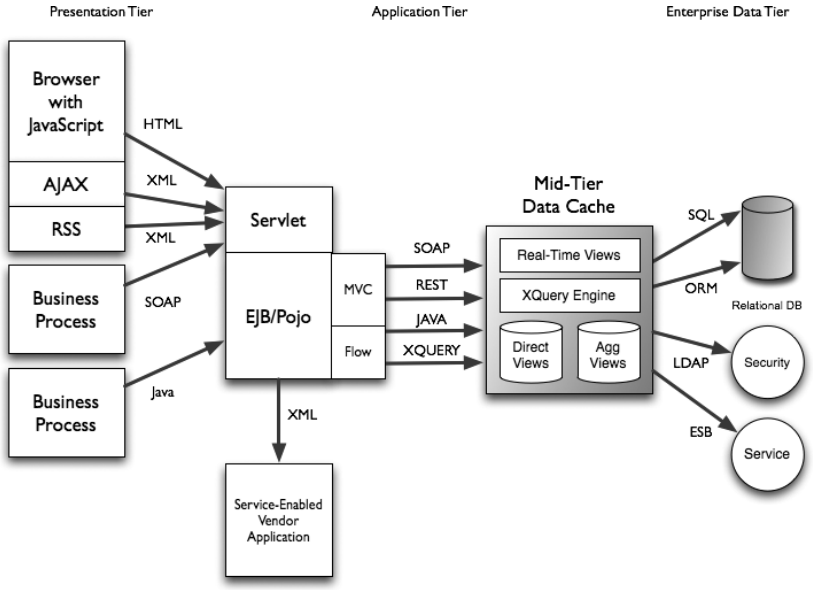

Figure 5-12 *Implementing a data cache for aggregate views*

1. **Direct views database.** Holds views of the data from the upstream data source providers. For instance, when the human resources health care plan services go off-line, the mid-tier data cache still services the last available view of the data. In another instance, the EJB may need a healthcare plan using a more modern schema and the direct views database holds a transformed healthcare plan from the human resources service emitting plans in an older schema. The direct views database may also hold an aggregated view of two plans and an archive of how the plans have changed over recent time.
2. **Aggregate views database.** Holds data views that are intelligently composed as data is served to the EJB. For instance, in the employee portal scenario the aggregate views database stores the most recent plans viewed, keeps a list of popular plans, and keeps a list of high-quality plans as ranked by employee feedback.

The mid-tier data cache uses a set of policies to determine the contents of the direct and aggregate views databases. Each policy holds

the business logic to look into upstream data sources and make decisions on which data to update and how frequently. The policy system is efficient at processing the business logic and storing the resulting XML views of the data from its use of native XML technology.

Choosing the FastSOA architecture for a mid-tier data cache in this scenario delivers a way to mitigate XML schema migration problems, reduce the amount of object-to-relational-to-XML mapping and transformation, and provides off-line data browsing capability. All of this comes without requiring you to be back in the EJB writing code.

The benefits to a business running a FastSOA mid-tier data cache include:

- Direct and aggregated views of the data model
- Real-time and near-time access between data and application tiers
- Two to 20 times performance advantage

In the next scenario, I will show how FastSOA is used as a platform for building high-performance and scalable dynamic services.

5.4 Scenario 3: Flexibility Needed for Semantic Web, Service Orchestration, and Services Dynamically Calling Other Services

Over the past two years, the software development community has enjoyed a renaissance of creativity from new XML-based technology, including mashups (for instance, combining Google maps with photos from Flikr), AJAX for better user interfaces, and REST for easy application-to-application interoperability. Much of this creativity pushes software development in the direction Tim Berners Lee espoused in the semantic Web.

When services communicate with other services, XML is the interoperability standard. FastSOA has much to offer a software developer working in an XML environment. Figure 5-13 illustrates an architecture that adds fast, efficient, and flexible XML capabilities to an otherwise Java and relational database architecture.

The FastSOA architecture enables developers to write business logic using pure XML technology. The FastSOA architecture provides the XML interface to the existing Java and relational database systems. For instance, when a browser makes an XML request from an

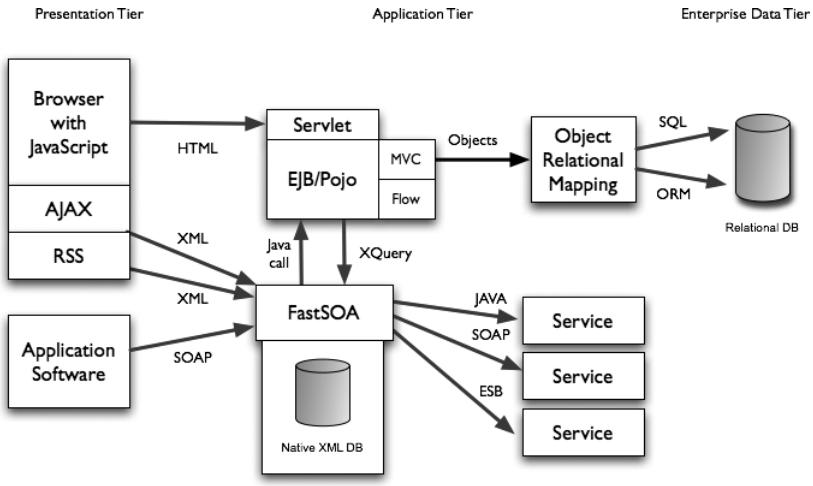

Figure 5-13 *FastSOA in a semantic Web environment.*

AJAX component, FastSOA receives the request, sees if the response is cached and still valid, and responds with the cached response. If the request requires data provided by a Java object (in the EJB), then FastSOA makes a direct Java call to the object and method.

FastSOA as a service interface and component development environment for semantic Web applications brings the following business benefits.

- One hundred percent native XML environment
- Avoiding object/relational/XML mapping reduces need for expensive application servers and network bandwidth
- Reduced software maintenance over time as message schemas change

The above three use cases show where FastSOA is an appropriate solution for XML performance and scalability challenges. We saw how native XML technology used at the mid-tier mitigates and solves performance and scalability problems and delivers flexibility unavailable with object and relational technology. The following are the business benefits for FastSOA.

- Solves SOA scalability problems for fast throughput and good scalability

- Works within existing infrastructure to avoid replacement costs
- Easy to customize with enterprise business processes using XQuery-based components
- Improves business agility and flexibility by maintaining interoperability and accelerating performance

5.5 Summary

This chapter showed three real-world scenarios where FastSOA improves performance, mitigates service bottlenecks, and improves developer productivity. In the next chapter, I show my PushToTest methodology to test and quantify performance in an SOA environment.