Data on the Outside versus Data on the Inside

Pat Helland

Microsoft Corporation
One Microsoft Way
Redmond, WA
USA
PHelland@Microsoft.com

Abstract

Recently, a lot of interest has been shown in SOA (Service Oriented Architectures). In these systems, there are multiple services each with its own code and data, and ability to operate independently of its partners. In particular, atomic transactions with two-phase commit do not occur across multiple services because this necessitates holding locks while another service decides the outcome of the transaction. This paper proposes there are a number of seminal differences between data inside a service and data sent into the space outside of the service boundary. We then consider objects, SQL, and XML as different representations of data. Each of these models has strengths and weaknesses when applied to the inside and outside of the service boundary. The paper concludes that the strength of each of these models in one area is derived from essential characteristics underlying its weakness in the other area.

1. Introduction

Service Oriented Architectures (SOA) is an exciting topic of discussion lately. While we can easily look to the past and see examples of large enterprise solutions that we can now characterize as SOA, the discussion of this applications style as a design paradigm is relatively recent. This section attempts to describe what is meant by SOA and introduces the notions of data residing inside services and data residing outside services.

Proceedings of the 2005 CIDR Conference

1.1 Service Oriented Architectures

Service Oriented Architecture characterizes a collection of independent and autonomous services. Each *service* comprises a chunk of code and data that is private to that service. Services are different than the classic application living in a silo and interacting only with humans in that they are interconnected with messages to other services.

Services communicate with each other exclusively through messages. No knowledge of the partner service is shared other than the message formats and the sequences of the messages that are expected. It is explicitly allowed (and, indeed, expected) that the partner service may be implemented with heterogeneous technology at all levels of the stack including hardware, operating system, database, middleware, and/or application vendor or implementation team.

The essence of SOA lies in independent services which are interconnected with messaging.

1.2 Bounding Trust via Encapsulation

Services interact via a collection of messages whose formats (schema) and business semantics are well defined. Each service will only do limited things for its partner services based upon the well defined message.

The act of defining a limited set of behaviors provides a very firm encapsulation of the service. The only way to interact with the service is via the prescribed messages each of which will invoke application logic to decide if and when to access the data encapsulated within the service. Data is, in general, never allowed out of a service unless it is processed by application logic.

> When approaching your bank's ATM, you are accustomed to only having a fixed set of operations to be performed (e.g. withdrawal, deposit, etc.). Banks do not allow direct access to reads and writes of their corporate database via ATMs. This is service oriented architecture and encapsulation to bound trust.

1.3 Encapsulating Both Changes and Reads

Services encapsulate changes to their data via the application logic for the service. This is done to ensure the integrity of the data that is owned by the service and the integrity of the work performed by the service. It is essential that only the trusted application logic of the service perform changes to the data.

In addition, services encapsulate the reading of their data to control the privacy of what is exported. While the subject of business intelligence analytics is beyond the scope of this paper, most environments based on service orientation cannot avail themselves of relative simplicity of a single database when attempting to analyze their data. The placement of the underlying data on heterogeneous operating systems and databases means that there are new challenges in performing cross-service analysis.

Frequently, services will choose to export carefully sanitized subsets of their data for consumption by partner services. We will explore this phenomenon later in this paper but the main point to be made now is that an untrusted partner service cannot simply read the contents of the service's private data unless intermediated by application logic.

1.4 Trust and Transactions

To participate in an ACID transaction requires a willingness to hold database locks until the transaction coordinator decides to commit or abort the transaction. For the non-coordinator, this is a serious ceding of independence and requires a lot of trust that the coordinating system will make a decision in a timely fashion. Being constrained to hold active locks on records in the database can be devastating for the availability of a system.

In this paper we are considering the relationship across applications and data that are unwilling to trust each other enough to jointly participate in a two phase commit. The use of the word "service" in this paper is hereby clarified to carry the assumption that ACID transactions are not shared across service boundaries[1].

1.5 Data Inside and Outside Services

The premise of this paper is that data residing inside a service is different in many essential ways from data residing outside services.

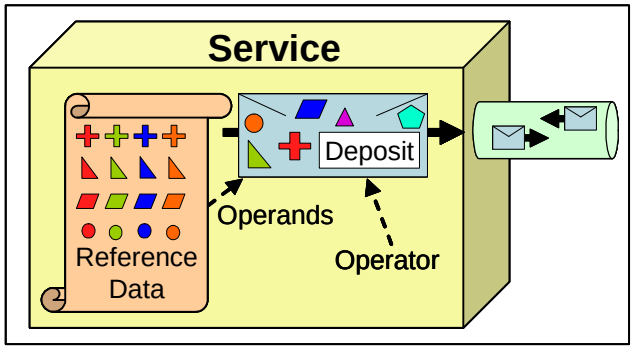
Data on the Inside refers to the encapsulated private data contained within the service itself. As a sweeping statement, this is the data that we have always considered as "normal data". The classic data contained in a SQL database and manipulated by a typical application is inside data.

Data on the Outside refers to the information that flows between these independent services. Since we defined services as being connected by messaging, it is reasonable to consider all outside data as being transmitted with a message[2].

1.6 Operators and Operands

When messages flow between services, they contain operators. Operators correspond to the intended purpose for the message. Frequently, the operator reflects a business function in the domain of the service. For example, a service implementing a banking application may have operators in their messages for deposits, withdrawals, and other such banking business functions. Sometimes, operators reflect more mundane reasons for sending messages such as "here's Tuesday's price-list".

Messages may contain operands to the operators. The operands are additional information required by the operator message to fully qualify the intent of the sending service. Operands are typically obtained from reference data published by a service to facilitate its later invocation with an operator message. We will address the topic of reference data in more depth below.



[1] Sometimes, the word "service" is used in a fashion that includes the potential for shared ACID transactions. Notably, the proposed WS-Transaction standard facilitates this. There will always be environments that are willing to share ACID transactions and others which are not. Hence, this becomes a debate about the definition and implications of the nomenclature and specifically, the interpretation of the word "service".

[2] Think about the role of paper and humans in connecting applications running in their own silos. Looking back at many enterprise applications, data and business operations flow between the applications using printouts and people. It is reasonable to consider the contents of a printout as outside data and apply the arguments about the nature of outside data made in this document equally to paper representations. For the purposes of discussion with this document, we will stick to an assumption of messages as the means for sharing outside data. That is not meant to preclude other representations of outside data but rather to sharpen our focus for discussion.

2. Data: Then and Now

This section examines the temporal implications of not sharing ACID transactions across services. We retrospectively examine the nature of work inside the boundaries of an ACID transaction and observe that this provides a crisp sense of "now" for operations against inside data.

However, it is different for data on the outside of the service. The fact that it is unlocked means that the data is no longer in the "now". Furthermore, operators are requests for operations which have not yet occurred and actually live in the future (assuming they come to fruition).

Finally, we consider the fact that different services live in their own private temporal domains and that this is an intrinsic part of service oriented architecture. It carries implications on the way we must think about applications.

2.1 Transactions, Inside Data, and "Now"

Transactions have been historically defined using ACID properties (Atomic, Consistent, Isolated, and Durable)[3]. These properties reflect the semantics of the transaction. Much work has been done to describe transaction serializability in which executing transactions on a system or set of related systems perceive their work is applied in a serial order even in the face of concurrent execution.[4]

Transactional serializability makes you feel alone. A rephrasing of serializability is that each transaction sees all other transactions in one of three categories:

- Transactions whose work preceded this one,
- Transactions whose work follows this one, or
- Transactions whose work is completely independent of this one.

This looks just like the executing transaction is all alone.

ACID transactions live in the "now". As time marches forward and transactions commit, each new transaction perceives the impact of the transactions that preceded it. The executing logic of the service lives with a clear and crisp sense of "now".

2.2 Outside Data: a Blast from the Past

Messages may contain data extracted from the local service's database. While it is processed by application logic, frequently the contents of a message are derived from the contents of data inside the service's database. This data will be unlocked as the outgoing message is created, allowing it to be changed.

By the time a partner service sees a message whose contents are based on the sender's data, the unlocked data may, in fact, be changed by subsequent transactions. It is no longer known to be accurate. The contents of a message are always from the past! They are never from "now".

> There is no simultaneity at a distance!
> -- Similar to the speed of light bounding information
> -- By the time you see a distant object, it may have changed!
> -- By the time you see a message, the data may have changed!

> Services, transactions, and locks bound simultaneity!
> -- Inside a transaction, things are simultaneous
> -- Simultaneity exists only inside a transaction!
> -- Simultaneity exists only inside a service!

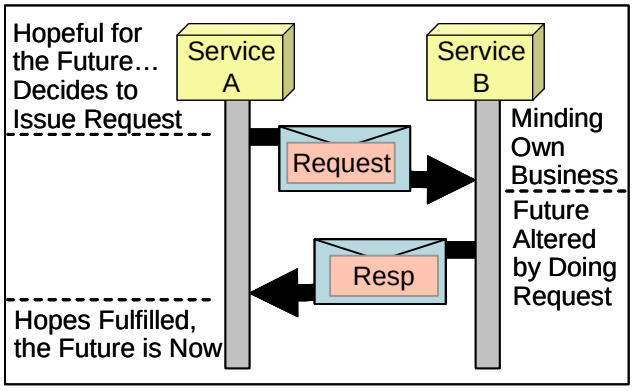
All data seen from a distant service is from the "past". By the time you see data from a distant service, it has been unlocked and may change.

Each service has its own perspective. Its inside data provides its framework of "now". Its outside data provides its framework of the "past". My inside is not your inside just as my outside is not your outside.

> Going to SOA is like going from Newton's physics to Einstein's physics.
> -- Newton's time marched forward uniformly with instant knowledge at a distance.
> -- Before SOA, distributed computing strove to make many systems look like one with RPC, 2PC, etc.
> -- In Einstein's universe, everything is relative to one's perspective.
> -- SOA has "now" inside and the "past" arriving in messages.

2.3 Operators: Hope for the Future

Message operators define requests for work from a service. If Service-A sends a message with an operator request to Service-B, it is hopeful that Service-B will do



the requested operation. It is hopeful for the future.

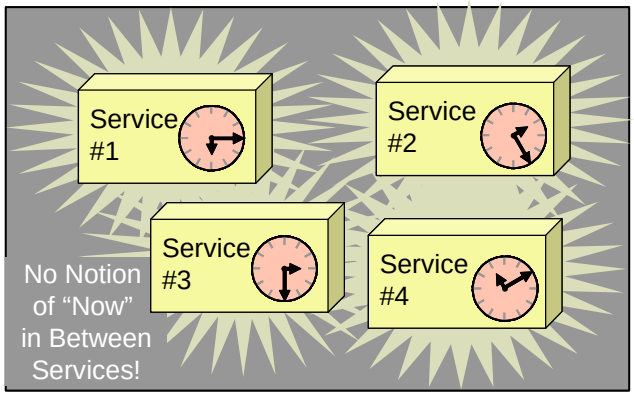
If Service-B complies and performs the work, that work becomes part of Service-B's future and its state is forever changed.

Once Service-A receives a reply back describing either success or failure of the operation, Service-A's future is changed.

[3] See [Gray and Reuter].
[4] See [Bernstein, Hadzilacos, Goodman].

2.4 Between Services: Life in the "Then"

Operands may live in either the past or the future depending on their usage pattern. They live in the past if they have copies of unlocked information from a distant service. They live in the future if they contain proposed values that hopefully will be used if the operator is successfully completed.



Between the services, life is in the world of "then". Operators live in the future. Operands live in either the past or the future. Life is always in the "then" when you are outside the confines of a service. This means that data on the outside lives in the world of "then". It is past or future but it is not now.

Each separate service has its own separate "now". The domains of transaction serializability are disjoint and each has its own temporal environment. The only way they interact is through data on the outside which lives in the world of "then".

2.5 Services: Dealing with "Now" and "Then"

Services must cope with making the "now" meet the "then". Each service lives in its own "now" and interacts with incoming and outgoing notions of "then". The application logic for the service must reconcile these!

Example#1: Accepting an Order
-- A business publishes daily prices.
-- It probably wants to accept yesterday's prices for a while after midnight!
-- The service's application logic must manually cope with the differences of prices during the overlap.
Example#2: "Usually ships in 24 hours"
-- Order processing has old information.
-- Available inventory is deliberately fuzzy.
-- Both sides can cope with different time domains.

The world is no longer flat!
-- SOA is recognizing that there is more than one computer working together!
-- Multiple machines mean multiple time domains.
-- Multiple time domains mandate we cope with ambiguity to allow coexistence, cooperation, and joint work.

3. Data on the Outside: Immutability

This section discusses some interesting properties of data on the outside. First, we address the need for each data item to be uniquely identified and to have immutable contents that do not changes as copies of it move around. Next, we describe anomalies that can be caused in the interpretation of data in different locations and at different times and introduce the notion of "stable" data which avoids these anomalies. After this, we move on to discuss schema and the messages it describes. This leads us to the mechanisms by which one piece of outside data can refer to another piece of data and the implications of immutability. Finally, we examine what outside data looks like when it is being created by a collection of independent services each in their own temporal domain.

3.1 Immutable and/or Versioned Data

Data may be *immutable*. Once immutable data is written and given an identifier, the contents of the data will always remain the same for that identifier. Once immutable data is written, it cannot be changed. In many environments, the immutable data may be deleted and the identifier will subsequently be mapped to an indication of "no present data" but it will never return data other than the original contents.

Immutable data is the same no matter when it is referenced and no matter where it is referenced.

Versioned data is immutable. If you specify a specific version of some collection of data, you will always get the same contents.

In many cases, a *version independent identifier* is used to refer to a collection of data. An example is the New York Times. A new version of the Times is produced each day (and, indeed, due to regional editions, multiple versions are produced each day).

To bind a version independent identifier, to the underlying data, it is necessary to first convert to a version dependent identifier. For example, the request for a recent New York Times is converted into a request for the New York Times on January 4th, 2005, California Edition. This is a version dependent identifier which yields the immutable contents of that region's edition of that day's paper. The contents of this edition for that day will never change no matter when you request it or where you request it. Either the information about the contents of that specific newspaper is available or it is not. If it is available, the answer is always the same.

3.2 Immutability, Messages, and Outside Data

One reality of messaging is that messages sometimes get lost. To ensure their delivery, they must be retried. It is essential that retries of the messages have the same contents. The message itself must be immutable.

Once a message is sent, it cannot be unsent anymore than the President of the United States can un-say

something on television. It is best to consider each message as uniquely identified and that identifier yields immutable contents for the message. This means the same bits are always returned for the message.

3.3 Stability of Data

Immutability isn't enough to ensure a lack of confusion. The interpretation of the contents of the data must be unambiguous. *Stable data* has an unambiguous and unchanging interpretation across space and time.

> The words "President Bush" have a different meaning in 2005 than they did in 1990. These words are not stable in the absence of additional qualifying data.

To ensure the stability of data, it is important to design for values that are unambiguous across space and time. One excellent technique for the creation of stable data is the use of time-stamping and/or versioning. Another important technique is to ensure that important identifiers are never reused.

> Observation: A monthly bank statement is stable data. Its interpretation is invariant across space and time.
> Advice: Don't recycle customer-IDs.
> Observation: Anything called "current" (e.g. current-inventory) is not stable.

3.4 Schema and Immutable Messages

As discussed above, when a message is sent, it must be immutable and stable to ensure the correct interpretation of the message. In addition, the schema for the message must be immutable. For this reason, it is recommended that all message schemas be versioned and each message use the version dependent identifier of the precise definition of the message format.

3.5 References to Data, Immutability, and DAGs

Sometimes it is essential to refer to other data. When referencing other data from outside data, it is essential that the identifier used for the reference specifies data that is, itself, immutable.

> If you find an immutable document that tells you to read "today's New York Times" to find out more details, that doesn't do you any good without more details (specifically the date and region for the paper).

As new data is generated, it may have references to complex graphs of other data items, each of which is immutable and uniquely identified. This creates a DAG (Directed Acyclic Graph) of referenced data items. Note that this model allows for each data item to refer to its schema using simply another arc in the DAG.

Over time, independent services, each within their own temporal domain, will generate new data items blithely ignorant of the recent contributions of other services. It is the creation of new immutable data items which are interrelated by their membership in this DAG that gives outside data its special charm.

4. Data on the Outside: Reference Data

Reference data refers to a type of information that is created and/or managed by a single service and published to other services for their use.

Each piece of reference data has both a version independent identifier and multiple versions, each of which is labeled with a version dependent identifier. For each piece, there is exactly one publishing service.

This section will first discuss the publication of versions. Then we will move on to discuss the various usages of reference data.

4.1 Publishing Versioned Reference Data

The idea here is quite simple. A version independent identifier is created for some data. One service is the owner of that data and periodically publishes a new version which is labeled with a version dependent identifier. It is important that the version's identifier is known to be increasing as subsequent versions are transmitted.

When a version of the reference data is transmitted, it must be assumed to be somewhat out of date. The information is clearly from the "past" and not "now". It is reasonable to consider these versions as snapshots.

4.2 Usages of Reference Data

There are three broad categories of usage for reference data that I've thought of so far:

- Operands contain information published by a service in anticipation that hopefully another service will submit an operator using these values.
- Historic Artifacts describe what happened in the past within the confines of the sending service.
- Shared Collections contain information that is held in common across a set of related services that gradually evolves over time. One service is the custodian and manages the application of changes to a part of the collection. The other services use somewhat old versions of the information.

We will examine these in greater depth below.

4.3 Operands

As discussed above, messages contain operators which map to the functions provided by the service. These operators frequently require operands as additional data describing the details of the requested work.

Operands are gleaned from reference data that is typically published by the service that is being invoked.

> Example: A department store catalog is reference data used to fill out the order-form.
> Example: An online retailer's price-list, product-catalog, and shipping-cost-list are operands.

4.4 Historic Artifacts

Historic artifacts report on what happened in the past. Sometimes these snapshots of history need to be sent from one service to another.

Serious privacy issues can result unless proper care is exercised in the disclosure of historic artifacts from one service to another. For this reason, many times this usage pattern is seen across services that have some form of trust relationship.

> Example: Quarterly results of sales.
> Example: A monthly bank statement.
> Example: Inventory status at the end of the quarter.

4.5 Shared Collections

The most challenging usage pattern for reference data is the shared collection. In this case, many different services need to have a recent view of some interesting data. Frequently cited examples include the employee database and the customer database. In each of these, lots of separate services both want to examine the contents and also change the contents of the data in these collections.

> Many large enterprises experience this problem writ large. Lots of different applications think they can change the customer database and, now that these applications are running on many servers, there are many replicas of the customer database (frequently with incompatible schemas). Changes made to one replica gradually percolate to the others with information loss due to schema transformations and also due to conflicting changes.

Shared collections offer a mechanism for rationalizing the desire to have multiple updaters and allowing controlling business logic to enforce business policies on the data.

In a shared collection, there is one special service that actually owns the authoritative perspective of the collection. It enforces business rules that ensure the integrity of the data. The owning service will periodically publish versions of the collection and supports incoming requests whose operators request changes.

Note that this is NOT optimistic concurrency control. The owning service has complete control over the changes to be made to the data. Some fields may be updateable and others may not. Business constraints may be applied as each requested change is considered.

> Consider changes to the customer's address. This is not just a simple update but complex business logic:
> -- First, you don't simply update an address, you append the new address while remembering that the old address was in effect for a range of dates.
> -- Changing the address may affect the tax location.
> -- Changing the address may affect the sales district.
> -- Shipments may need to be rerouted.

5. Data on the Inside

As described above, inside data is encapsulated behind the application logic of the service. This means that the only way to modify the data is via the service's application logic. Sometimes a service will export a subset of their inside data for use on the outside as reference data.

This section examines a number of facets of data on the inside. First, we look at the temporal environment in which SQL's schema definition language operates. Then, we consider how outside data is handled as it arrives into a service. Finally, we consider by the extensibility we see in data on the outside and the challenges with storing copies of that data inside in a shredded fashion to facilitate its use in relational form.

5.1 SQL, DDL, and Serializability

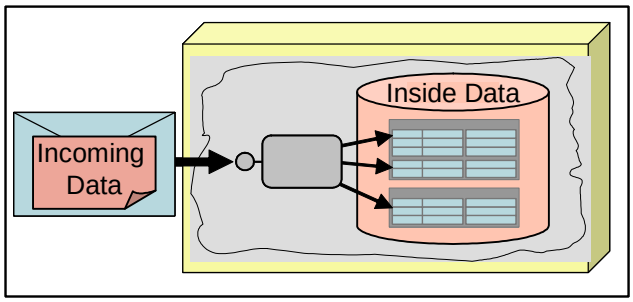
SQL's DDL (Data Definition Language) is transactional. Like other operations in SQL, updates to the schema via DDL occur under the protection of a transaction and are atomically applied. These schema changes may make significant difference in the ways that data stored within the database is interpreted.

It is an essential quality of DDL that transactions that precede a DDL operation are based on the schema that existed before and transactions that follow the DDL operation are based on the schema as changed by the operation. In other words, changes to the schema participate in the serializable semantics of the database.

SQL and DDL live in the "Now". Each transaction is meaningful only within the context of the schema defined by the preceding transactions. This notion of "now" is the temporal domain of the service comprising the service's logic and its data contained in this database.

5.2 Storing Incoming Data

When data arrives from the outside, most services copy it inside their local SQL database. While inside data is not, in general, immutable, most services choose to implement a convention by which they immutably retain the data. It is not uncommon to see the incoming data syntactically converted to a more convenient form for the service. .



Many times, an incoming message kept as an exact binary copy for auditing and non-repudiation while still converting the contents to a form easier to use within the service itself.

5.3 Extensibility versus Shredding

Frequently, the outside data is kept in a hierarchical representation like XML. XML has a number of wonderful qualities for this including extensibility. XML's extensibility allows for other services to add information to a message that was not declared in the schema for the message. Basically, the sender of the message added stuff that you didn't expect when the schema was defined. Extensibility is in many ways like scribbling on the margins of a paper form. It frequently gets the desired results but there are no guarantees.

As incoming outside data is copied into the SQL database, there are advantages to shredding it. Shredding is the process of converting the hierarchical data into a relational representation. It is interesting to note that normalization is not of the same importance for copied incoming data since the intent is to simply append the information and never change it. Shredding is, however, of great interest for business analytics. The better the relational mapping, the better you will be able to analyze the data.

It is interesting that extensibility fights shredding. It is hard to map unplanned extensions to planned tables. Many times, partial shredding is performed wherein the incoming information that does comply with well known and regular schema representations is cleanly shredded into a relational representation and the remaining data (including extensions) is kept without shredding.

6. Representations of Data

Let's consider the characteristics of three prominent representations of data: XML, SQL, and Objects.

6.1 Representing Data in XML

XML (eXtensible Markup Language)[5] is a standard for representing hierarchical collections of data as XML-Documents. Its foundation is the InfoSet standard which defines the abstract dataset prescribing the semantics of parents, children, elements, attributes, and the rest of the details of the way data can be held in a tree in XML. While the InfoSet does define the semantics, it does not define the syntax of a message transmission. This may be of a proprietary form or may comply with the syntactic representation whose angle-brackets we know and love.

In addition, XML-Schema[6] defines a datatype library and schema definition mechanism. The cool thing about XML-Schema is the composeability of schemas. It is expected that the definition of a schema will likely include other definitions and this is made easy and convenient with embedded identifiers (URIs –Universal Resource Identifiers[7]) that can easily be used to reference immutable documents.

It is this combination of hierarchy, explicit and well defined identifiers (URIs), clear mechanism for leveraging old schema within the new schema, and extensibility that has given XML its prominence in representing outside data.

6.2 Representing Data in SQL

SQL represents relationships by values contained in fields. Being value-based allows it to "relate" different records to each other by their value. This is the essence of the "relational" backbone of SQL. It is precisely this value-based nature of the representation that enables the amazing query technology we have seen emerge over the last few decades. SQL is clearly the leader as a representation for inside data.

6.3 Bounded and Unbounded Data Representations

It is illustrative to contrast SQL's value-based mechanism with XML's identity and reference-based mechanism.

Relational representations must be bounded. For the value-based comparisons to work correctly there must be both temporal and spatial bounds. Value-based comparisons are only meaningful if the contents of both records are defined within the same schema. Multiple schemas can only have well defined meaning when they can (and are) updated within the same temporal scope (i.e. updated with ACID semantics in the same serializability domain). This effectively yields a single schema. SQL is semantically based on a centrally managed single schema.

Efforts in recent years to implement distributed database systems attempt to blur the distinction between a single system and multiple systems. By using two phase commit and other variants, they create a single temporal domain in which there is a well defined schema that can be centrally updated. This is extending the periphery of the boundary across multiple machines (at the potential risk of performance and/or availability) but does not negate this argument that relational representations work only within a clearly defined boundary.

XML is unbounded. In XML, data is referenced using URIs and not values. These URIs are universally defined and unique. They can be used on any machine to uniquely identify the referenced data. When used with the proper discipline, this can result in the creation of directed acyclic graphs of XML-documents each of which may be created by independent services living in independent temporal domains.

A further aspect to the unbounded nature of XML lies in the open schema definition which allows for the composition of schema pieces from different origins into a new schema. The ability to independently define schema without consulting other services complements

[5] See [XML].
[6] See [XML-Schema]
[7] See [URIs]

the compositional nature of XML-Schema and leads to a vibrant environment for the independent generation of interrelated documents.

6.4 Encapsulation and Anti-Encapsulation

It is interesting to consider encapsulation as it relates to SQL, XML, and the implementation of services.

- SQL is essentially about anti-encapsulation. The whole idea of allowing SELECT WHERE or UPDATE WHERE which involves joining anything in the database with anything else in the database is, by its nature, anti-encapsulation. It seems to be a deeply engrained aspect of the philosophy of databases that all data is accessible at all times.
- XML is strongly oriented towards anti-encapsulation. The whole notion of a publicly defined schema fights against the idea of keeping data private and controlling it.
- Components and objects emphasize encapsulation. Of course, there is a long tradition of cheating. The passing of references into an object which allows the manipulation of the shared object breaks encapsulation and easily introduces anomalies. The habit of different objects modifying the same data in the database blithely ignoring the fact that this is manipulating shared state is another major source of anomalous behavior.

6.5 A Service's View of Encapsulation

Services offer very strong and rigid encapsulation. The basic notion is that there is no access whatsoever to the underlying data unless it is mediated by the application logic of the service. There is no visibility to the internals of the service.

Within a service anti-encapsulation is OK in its place. SQL's penchant for anti-encapsulation is contained inside the service and only visible to the service's application logic. Hence, this has no impact on the external behavior provided by the service and does not pose a vulnerability to its periphery. XML's anti-encapsulation only applies to the messages flowing in and out of a service. Other mechanisms for providing authentication, authorization, and ensuring privacy are used to protect the messages. Once a receiving service can pass the relevant security, XML's anti-encapsulation empowers the semantic connection between heterogeneous services by easing the understanding of the intended purpose of the message through shared schema.

6.6 Characteristics of Inside and Outside Data

Let's consider the various characteristics we have discussed for inside and outside data. Please refer to the figure below:

	Outside Data	Inside Data
Immutable?	Yes	No
Identity-Based References	Yes	No
Open Schema?	Yes	No
Represent in XML?	Yes	No
Encapsulation Useful?	No	Yes
Long-Lived Evolving Data with Evolving Schema?	No	Yes
Business Intelligence Desirable over Data?	Yes	Yes
Durable Storage in SQL Inside the Service?	Yes: Copy of XML Kept in SQL	Yes

Immutability, identity-based references, open schema, and XML representation all apply to outside data and not to inside data. This is all part of a package-deal in the form of the representation of the data and it suits the needs of outside data very well. The immutable data items can be copied throughout the network and new one's generated by any service. Indeed, the open and independent schema mechanisms allow independent definition of new formats for messages, further empowering the independence of separate services.

Next, we consider encapsulation and realize that outside data is not protected by code. There is no formalized notion of ensuring that access to the data is mediated by a body of code. Rather, there is a design point that if you have access to the raw contents of a message, you should be able to understand it. Inside data is always encapsulated by the service and its application logic.

Consider data and its relationship to its schema. Outside data is immutable and, each data item's schema remains immutable. Note that the schema may be versioned and the new version applied to subsequent similar data items but that does not change the fact that once a specific immutable item is created, its schema remains immutable. This is in stark contrast to the mechanisms employed by SQL for inside data. SQL's DDL is designed to allow powerful transformations to existing schema while the database is populated.

Next, we consider the desirability of performing business intelligence analysis over the data. Experience shows that those analysis folks want to slice and dice anything they can get there hands on. Existing analytics operate largely over inside data and inside data will

certainly continue as fodder for analysis. There is little doubt of the utility of analyzing outside data, as well.

This leads us to the final column wherein we conclude that the typical storage mechanism for both inside and outside data will be inside SQL. The only twist for outside data is that it will be copied into SQL's representation. The copy of the incoming data will then be kept unchanged which provides immutable semantics.

6.7 The Ruling Triumvirate of Data Representations

Now, let's compare the strengths and weaknesses of the three representations of data, SQL, XML, and Objects.

- SQL with its bounded schema is fantastic to compare anything with anything (but only within bounds).
- XML with its unbounded schema supports independent definition of schema and data. Extensibility is cool, too.
- Objects offer encapsulated data and ensure the enforcement of the business rules via application logic. Objects also ease the composition of logic.

	Arbitrary Queries	Independent Definition of Shared Data	Encapsulation (controls data)
SQL	Outstanding	Impossible	Not via SQL (done by DBA)
XML	Problematic	Outstanding	Impossible
Objects	Impossible	Impossible	Outstanding

Consider what it takes to perform arbitrary queries:

- SQL is outstanding due to its value based nature and tightly controlled schema which ensure alignment of the values hence facilitating the comparison semantics that underlie queries.
- XML is problematic because of schema inconsistency. It is precisely the independence of the definition that poses the challenges of alignment of the values. Also, the hierarchical shape and forms of the data may, too, be a headache.
- Objects are impossible to query unless a new definition is made of how they define data to be queried. Encapsulation is all about hiding the data.

Next, consider independent definition of shared data:

- SQL is impossible because it has centralized schema. As discussed above, this is intrinsic to its ability to support value-based querying in a tightly controlled environment.
- XML is outstanding! It specializes in independent definition of schema and independent generation of documents containing the data.
- Objects are impossible since encapsulation is all about NOT sharing data.

Finally, consider encapsulation to control data access:

- SQL doesn't really do encapsulation. Realistically, all you get is table-level access control. Smart database administrators ensure this means that only the correct applications can run under the user-id of the accounts configured to access the tables. That is the coarse-grained encapsulation that is really used.
- XML is all about anti-encapsulation. Firstly, XML has no concept of code to ensure is encapsulating the document. Secondly, XML is designed to facilitate the sharing of documents and their schema. It's really not oriented towards encapsulation!
- Objects are outstanding at encapsulation! That's what they were designed for in the first place.

Each model's strength is simultaneously its weakness!
What makes SQL exceptional for querying makes it dreadful for independent definition of shared data. XML is wonderful for the independent definition and creation of data but is anti-encapsulated. Encapsulation is the key to the success of object systems and yet it prevents querying. *You cannot try harder to add features to one of these models to address the weaknesses without undermining its strengths!*

7. Conclusion

This paper describes the impact of Service Oriented Architecture (SOA) on the treatment of data. First, we introduced the notions of inside data as distinct from outside data. After discussing the temporal implications of not sharing transactions across the boundaries of services, we considered the need for immutability and stability in outside data. This led to a depiction of outside data as a DAG (Directed Acyclic Graph) of data items being independently generated by disparate services.

Following this introduction of the basic concepts behind outside data, we examined the notion reference data and its usage patterns in facilitating the interoperation of services.

Next, we presented a brief sketch of inside data with a discussion of the challenges of shredding incoming data in the face of extensibility.

Finally, we discussed XML, SQL, and Objects as representations of data and compared and contrasted their strengths. This led us to the conclusion that each of these models has strength in one usage that complements its weakness in another usage.

This conclusion should not surprise us when we realize that most application developers are pretty smart. It is common practice today to use XML to represent data on the outside, objects to implement the business logic of the services, and SQL to store the data on the inside. We simply need all three of these representations and we need to use them in a fashion that plays to their respective strengths!

8. References

[Bernstein, Hadzilacos, Goodman]
Concurrency Control and Recovery In Database Systems by Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman (see http://research.microsoft.com/pubs/ccontrol/)

[Gray and Reuter]. Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993, ISBN 1-55860-190-2

[URIs] http://www.w3.org/Addressing/

[XML] - http://www.w3.org/TR/REC-xml/

[XML-InfoSet] -- http://www.w3.org/TR/xml-infoset/

[XML-Schema] http://www.w3.org/TR/xmlschema-1/