Data on the Outside versus Data on the Inside

DATA KEPT OUTSIDE SQL HAS DIFFERENT CHARACTERISTICS FROM DATA KEPT INSIDE.

PAT HELLAND

Recently, there has been a lot of interest in services. These can be microservices or just services. In each case, the service provides a function with its own code and data, and operates independently of partners. This article argues that there are a number of seminal differences between data encapsulated inside a service and data sent into the space outside of the service boundary.

SQL data is encapsulated within a service to ensure it is protected by application code. When sending data across services, it is outside that trust boundary.

The first question this article asks is what trust means to a service and its encapsulated data. This is answered by looking at transactions and boundaries, data kept inside versus data kept outside of services. Also to be considered

is how services compose using operators (requesting stuff) and operands (refining those requests). Then the article looks at time and service boundaries. When data in a database is unlocked, it impacts notions of time. This leads to an examination of the use of immutability in the composition of services with messages, schema, and data flowing between these boundaries.

The article then looks at data on the outside of these trust boundaries called services. How do you structure that data so it is meaningful across both space and time as it flows in a world *not* inside a service? What about data inside a service? How does it relate to stuff coming in and out?

Finally, the characteristics of SQL and JSON (JavaScript Object Notation), and other semi-structured representations, are considered. What are their strengths and weaknesses? Why do the solutions seem to use both of them for part of the job?

ESSENTIAL SERVICES

Services are essential to building large applications today. While there are many examples of large enterprise solutions that leverage services, the industry is still learning about services as a design paradigm. This section describes how the term *service* is used and introduces the notions of data residing inside services and outside services.

Services

Big and complex systems are typically collections of independent and autonomous services. Each service consists of a chunk of code and data that is private to that

service. Services are different from the classic application living in application silos, in that services are primarily designed to interact with other services via messaging. Indeed, that interaction, its data, and how it all works is an interesting topic.

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 expected messages. 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, programming language, and/or application vendor or implementation team.

The essence of a service lies in its independence and how it encapsulates (and protects) its data.

Bounding trust via encapsulation

Services interact with a collection of messages whose formats (schema) and business semantics are well defined. Each service will do only limited things for its partner services based upon well-defined messages. The act of defining a limited set of behaviors provides a firm encapsulation of the service. An important part of trust is limiting the things you'll do for outsiders.

To interact with a service, you have to follow its rules and constraints. Each message you send fits a prescribed role. The only way to interact with data in another service is through its rules and business logic. Data is, in general, never allowed out of a service unless it is processed by application logic.

For example, when using your bank' s ATM, you expect to have only a few supported operations such as withdrawal, deposit, etc. Banks do not allow direct access to database connections via ATMs. The only way to change the bank's database is through the bank's application logic in the ATM and the back-end system. This is how a service protects its data.

Encapsulating both changes and reads

Services encapsulate changes to their data with their application logic. The app logic ensures the integrity of the service's data and work. Only the service's trusted application logic can change the data.

Services encapsulate access to read their data. This controls the privacy of what is exported. While this autonomy is powerful, it can also cause some challenges.

Before your business separated its work into independent services, all of its data was in a big database. Now you have a bunch of services, and they have a bunch of databases running on a bunch of computers with a bunch of different operating systems. This is awesome for the independent development, support, and evolution of the different services, but it's a royal hassle when you want to do analytics across all your data.

Frequently each service will choose to export carefully sanitized subsets of data for consumption by partner services. Of course, this requires some work ensuring proper authorization to see this data (as well as authenticating the curious service). Still, the ability to sanitize and control the data being exposed is crucial.

Trust and transactions

Participating in an ACID (atomicity, consistency, isolation, durability) transaction means one system can be locked up waiting for another system to decide to commit or abort the transaction. If you are stuck holding locks waiting for another system, that can really cause trouble for your availability. With rare exceptions, services don't trust other services like that.

In the late 1990s, there were efforts to formalize standards for transaction coordination across trust boundaries. Fortunately, these standards died a horrible death.

Data inside and outside services

The premise of this article is that data residing inside a service is different in many essential ways from data residing outside or between 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 has always been considered "normal"—at least in your database class in college. 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. This includes messages, files, and events. It's not your classic SQL data.

Operators and operands

Messages flowing between services contain *operators*, which correspond to the intended purpose of the message. Frequently the operator reflects a business function in the

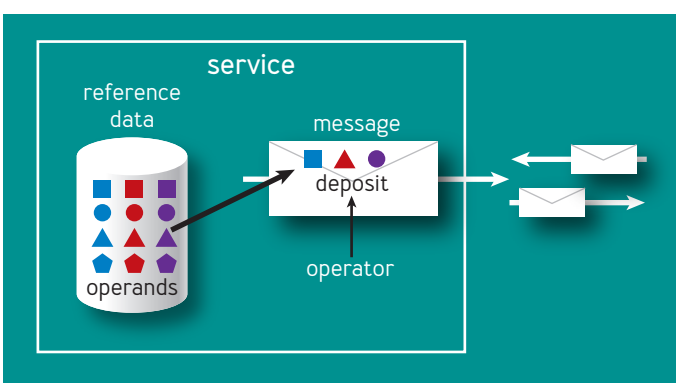
domain of the service. For example, a service implementing a banking application may have operators in its messages for deposits, withdrawals, and other banking functions. Sometimes operators reflect more mundane reasons for sending messages, such as "Here's Tuesday's price list."

Messages may contain *operands* to the operators, shown in figure 1. The operands are additional stuff needed by the operator message to qualify the intent of the message fully. Operands may be obtained from *reference data*, published to describe those operands. A message requesting a purchase from an e-commerce site may include product IDs, requested numbers to be purchased, expected price, customer ID, and more. This is covered in more detail later.

DATA: THEN AND NOW

This section examines the temporal implications of not sharing ACID transactions across services and examines

FIGURE 1: **OPERANDS**



the nature of work inside the boundaries of an ACID transaction. This provides a crisp sense of "now" for operations against inside data.

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

Different services live in their own private temporal domains. This is an intrinsic part of using distrusting services. Trust and time carry implications about how to think about applications.

Transactions, inside data and now

Transactions have been historically defined using ACID properties.[1] These properties reflect the semantics of the transaction. Much work has been done to describe transaction serializability, in which transactions executing on a system or set of related systems perceive their work as applied in a serial order even in the face of concurrent execution.[2] Transactional serializability makes you feel alone. A rephrasing of serializability is that each transaction sees all other transactions to be in one of three categories:

- Those whose work preceded this one.
- Those whose work follows this one.
- Those 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.

Blast from the past

Messages may contain data extracted from the local service's database. The sending application logic may look in its belly to extract that data from its database. By the time the message leaves the service, that data will be unlocked.

The destination service sees the message; the data on the sender's service may be changed by subsequent transactions. It is no longer known to be the same as it was when the message was sent. The contents of a message are always from the past, 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. Likewise, 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.

Using services rather than a single centralized database

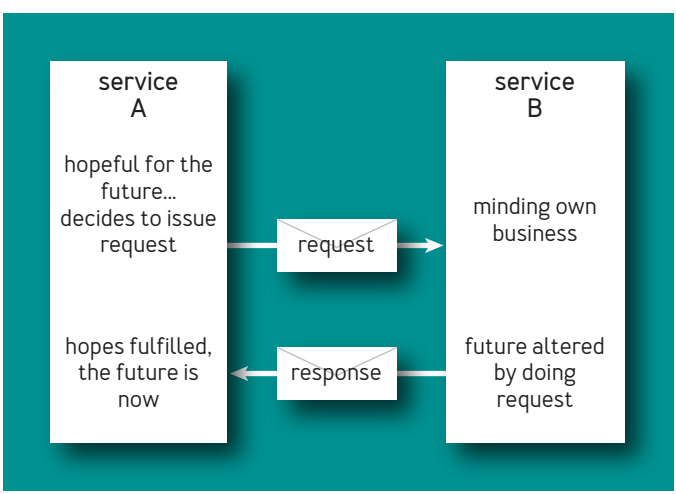
is like going from Newton's physics to Einstein's physics:

- Newton's time marched forward uniformly with instant knowledge at a distance.
- Before services, distributed computing strove to make many systems look like one, with RPC (remote procedure call), two-phase commit, etc.
- In Einstein's universe, everything is relative to one's perspective.
- Within each service, there is a "now" inside, and the "past" arriving in messages.

Hope for the future

Messages contain operators that define requests for work from a service, shown in figure 2. If Service A sends a message with an operator request to Service B, it is

FIGURE 2: **REQUESTS FOR WORK**



hopeful that Service B will do the requested operation. In other words, 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 describing either success or failure of the operation, Service A's future is changed.

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.

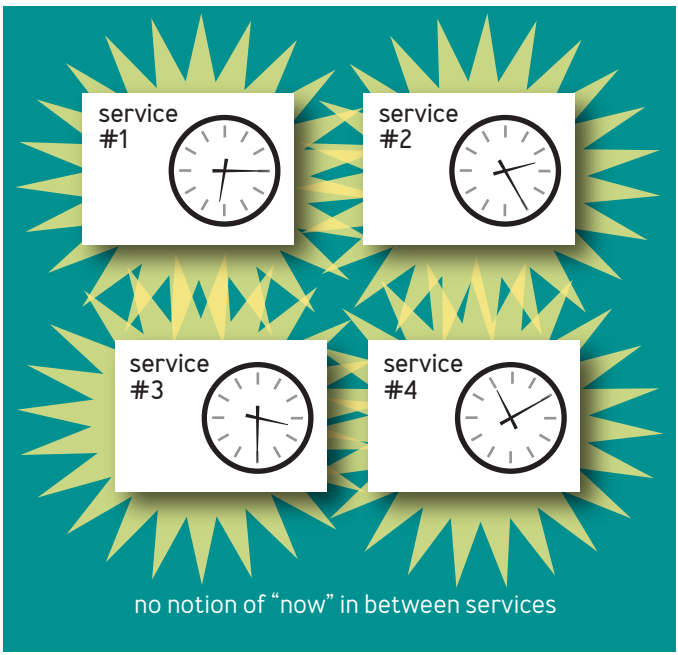
Between the services, life is in the world of "then." Operators live in the future. Operands live in either the past or the future. Life is always in the then when you are outside the confines of a service. This means that data on the outside lives in the world of then. It is past or future, but it is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

Each separate service has its own separate "now," illustrated in figure 3. 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.

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.

FIGURE 3: **SERVICES WITH DIFFERENT "NOW"S**



Consider, for example, what's involved when a business accepts an order: The business may publish daily prices, but it probably wants to accept yesterday's prices for a while after midnight. Therefore, the service's application logic must manually cope with the differences in prices during the overlap.

Similarly, a business that says its product "usually ships in 24 hours" must consider the following: Order processing has old information; the available inventory is deliberately fuzzy; both sides must cope with different time domains.

The world is no longer flat:

- Services with private data support more than one computer working together.
- Services and their service boundaries mean multiple trust domains and different transaction domains.
- Multiple transaction domains mean multiple time domains.
- Multiple time domains force you to cope with ambiguity to allow coexistence, cooperation, and joint work.

DATA ON THE OUTSIDE: IMMUTABILITY

This section discusses properties of data on the outside. First, each data item needs to be uniquely identified and have immutable contents that do not change as copies of it move around. Next, anomalies can be caused in the interpretation of data in different locations and at different times; the notion of "stable" data avoids these anomalies. The section also discusses schemas and the messages they describe. This leads to the mechanisms by which one piece of outside data can refer to another piece of data and the implications of immutability. Finally, what does outside data look like when it is being created by a collection of independent services, each living in its own temporal domain?

Immutable and/or versioned data

Data may be immutable. Once immutable data is written and given an identifier, its contents will remain the same for that identifier. Once it 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 or 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 newspaper is produced each day (and, indeed, because of regional editions, multiple versions are produced each day). To bind a version-independent identifier to the underlying data, it is necessary first to 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 4, 2005, California edition.

This is a version-dependent identifier that 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 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.

Immutability, messages, and outside data

One reality of messaging is that messages sometimes get lost. To ensure delivery, the message must be retried. It is essential that retries have the same contents. The message itself must be immutable. Once a message is sent, it cannot be unsent any more than a politician can unsay something on television. It is best to consider each message as uniquely identified, and that identifier must

yield immutable contents for the message. This means the same bits are always returned for the message.

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.

For example, a monthly bank statement is stable data. Its interpretation is invariant across space and time. On the other hand, the words *President Bush* had a different meaning in 2005 than they did in 1990. These words are not stable in the absence of additional qualifying data. Similarly, anything called *current* (e.g., current inventory) is not stable.

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 such as customer IDs are never reused.

Immutable schema and immutable messages

As discussed previously, when a message is sent, it must be immutable and stable to ensure its correct interpretation. 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. Alternatively, the schema can be embedded in the message. This is popular when using JSON

or other semi-structured formats.

References to data, immutability, and DAGs

Sometimes it is essential to refer to other data. When referencing from outside data, the identifier used for the reference must specify data that is 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 of 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 its own temporal domain, will generate new data items blithely ignorant of the recent contributions of other services. The creation of new immutable data items that are interrelated by membership in this DAG is what gives outside data its special charm.

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 discusses the publication of versions, then moves on to the various uses of reference data.

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 that 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.

Uses of reference data

There are three broad usage categories for reference data, at least so far:

- Operands contain information published by a service in anticipation that 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 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 older versions of the information.

Operands

As previously discussed, messages contain operators that 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 being invoked. A department store catalog, for example, is reference data used to fill out the order form. An online retailer's price list, product catalog, and shipping-cost list are operands.

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, this usage pattern is often seen across services that have some form of trust relationship, such as quarterly results of sales, a monthly bank statement, or inventory status at the end of the quarter.

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 want both to examine and to 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 caused by schema transformations and conflicting changes. A shared collection offers a mechanism for rationalizing the desire to have multiple updaters and allowing controlling business logic to enforce policies on the data. A shared collection has 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 periodically publishes 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 updatable, and others may not. Business constraints may be applied as each requested change is considered.

Consider changes to a customer's address. This is not just a simple update but complex business logic:

- 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.

DATA ON THE INSIDE

As previously described, 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 its inside data for use on the outside as reference data.

This section examines the following facets of data on the inside: (1) the temporal environment in which SQL's schema definition language operates; (2) how outside data is handled as it arrives into a service; and (3) the extensibility seen in data on the outside and the challenges inherent in storing copies of that data inside in a shredded fashion to facilitate its use in relational form.

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 a significant difference in the ways that data stored within the database is interpreted.

It is essential that transactions preceding a DDL operation be based on the existing schema, and those that follow the DDL operation be based on the schema as changed by the operation. In other words, changes to the schema participate in the serializable semantics of the database.

Both 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 consisting of the service's

logic and its data contained in this database.

Storing incoming data

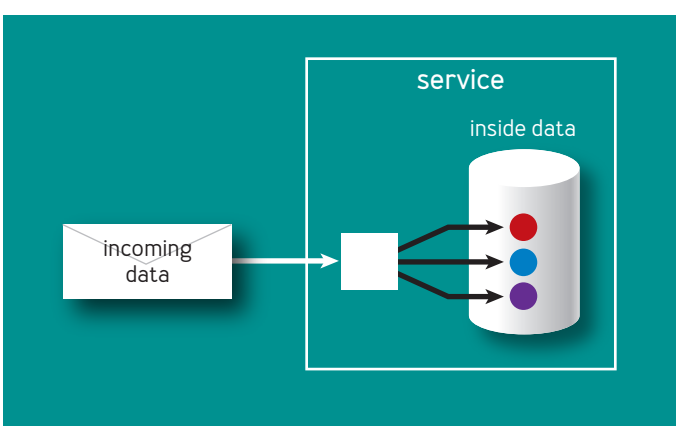
When data arrives from the outside, most services copy it inside their local SQL database. Although inside data is not, in general, immutable, most services choose to implement a convention by which they immutably retain the data. It is not uncommon to see the incoming data syntactically converted to a more convenient form for the service. This is called *shredding* (figure 4).

Many times, an incoming message is 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.

Extensibility versus shredding

Frequently the outside data is kept in a semi-structured

FIGURE 4: **SHREDDING**



representation such as JSON, which has a number of wonderful qualities for this, including extensibility. JSON's extensibility allows other services to add information to a message that was not declared in the schema for the message. Basically, the sender of the message has 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 semi-structured data into a relational representation. Normalizing the incoming outside data is not a priority. Normalization is designed to eliminate or reduce update anomalies. Even though you're stuffing the data into a SQL database, you're not going to update it. You are capturing the outside data in a fashion that's easier to use inside SQL. 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. Mapping unplanned extensions to planned tables is difficult. 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.

REPRESENTATIONS OF DATA

Let's consider the characteristics of these two prominent representations of data: JSON and SQL.

Representing data in JSON

JSON is a standard for representing semi-structured data. It is an interchange format with a human-readable text for storing and transmitting attribute-value pairs. Sometimes a schema for the data is kept outside the JSON document. Sometimes the metadata is embedded (as attribute-value pairs) into the hierarchical structure of the document. JSON documents are frequently identified with a URL (universal resource locator), which gives the document a unique identity and allows references to it.

It is this combination of human readability, self-describing attribute-value pairs, and global identity through URLs that make JSON so popular. Of course, its excellent and easy-to-use libraries in multiple languages help too.

Representing data in SQL

SQL represents relationships by values contained in cells within rows and tables. 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 that has emerged over the past few decades. SQL is clearly the leader as a representation for inside data.

Bounded and unbounded

Let's contrast SQL's value-based mechanism with JSON'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 meaningful only if the contents of both records are defined within the same schema. Multiple schemas can have well-defined meaning only when they can be (and are) updated within the same temporal scope (i.e., with ACID semantics in the same database). This effectively yields a single schema. SQL is semantically based on a centrally managed single schema.

Attempts over the past 20 years to create distributed SQL databases are fine but must include a single transactional scope and a single DDL schema. If not, the semantics of relational algebra are placed under pressure. SQL only works inside a single database.

JSON is unbounded. In JSON, data is referenced using URIs (uniform resource identifiers) and not values. These URIs are universally defined and unique. Of course, every URL is a legitimate URI so they're cool, too. URIs 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 DAGs of JSON documents, each of which may be created by independent services living in independent temporal (and schema) domains.

Characteristics of inside and outside data

Let's consider the various characteristics discussed so far for inside and outside data, as shown in figure 5.

FIGURE 5: **INSIDE AND OUTSIDE DATA**

	OUTSIDE DATA	INSIDE DATA
Immutable?	yes	no
Identity based references	yes	no
Open schema?	yes	no
Represent in JSON or other semi-structured fashion	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	yes

Immutability, identity-based references, open schema, and JSON representation apply to outside data, 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 well. The immutable data items can be copied throughout the network and new ones 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, 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 says 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.

Finally, let's 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 their hands on. Existing analytics operate largely over inside data, which will certainly continue as fodder for analysis. But there is little doubt about the utility of analyzing outside data as well.

The dynamic duo of data representations

Now, let's compare the strengths and weaknesses of these two representations of data, SQL and JSON:

- SQL, with its bounded schema, is fantastic for comparing anything with anything (but only within bounds).
- JSON, with its unbounded schema, supports independent definitions of schema and data. Extensibility is cool too.

	ARBITRARY QUERIES	INDEPENDENT DEFINITION OF SHARED DATA
SQL	Outstanding!	Impossible. SQL data definition is centralized, not independent
JSON	Problematic	Outstanding!

Consider what it takes to perform arbitrary queries: SQL is outstanding because of its value-based nature and tightly controlled schema, which ensure alignment of the values, hence facilitating the comparison semantics that underlie queries.

JSON 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 also be a headache. Still, you *can* project consistent schema in a form easily queried. It might be a lossy projection where not all the knowledge is available to be queried.

Next, consider independent definition of shared data: SQL is impossible because it has centralized schema. As already discussed, this is intrinsic to its ability to support value-based querying in a tightly controlled environment. *JSON is outstanding.* It specializes in independent definition of schema and independent generation of documents containing the data. That is a huge strength of JSON and other semi-structured data representations.

Each model's strength is simultaneously its weakness. What makes SQL exceptional for querying makes it dreadful for independent definition of shared data. JSON is wonderful for the independent definition, but it stinks for querying. *You cannot add features to either of these models to address its weaknesses without undermining its strengths.*

Related articles

Beyond Relational Databases
There is more to data access than SQL.
Margo Seltzer
https://queue.acm.org/detail.cfm?id=1059807

The Singular Success of SQL
SQL has a brilliant future as a major figure in the pantheon of data representations.
Pat Helland
https://queue.acm.org/detail.cfm?id=2983199

A co-Relational Model of Data for Large Shared Data Banks
Erik Meijer and Gavin Bierman
Contrary to popular belief, SQL and noSQL are really just two sides of the same coin.
https://queue.acm.org/detail.cfm?id=1961297

CONCLUSION

This article describes the impact of services and trust on the treatment of data. It introduces the notions of inside data as distinct from outside data. After discussing the temporal implications of not sharing transactions across the boundaries of services, the article considers the need for immutability and stability in outside data. This leads to a depiction of outside data as a DAG of data items being independently generated by disparate services.

The article then examines the notion of reference data and its usage patterns in facilitating the interoperation of services. It presents a brief sketch of inside data with a discussion of the challenges of shredding incoming data in the face of extensibility.

Finally, JSON and SQL are seen as representations of data, and their strengths are compared and contrasted. This leads to the conclusion that each of these models has strength in one usage that complements its weakness in another usage. It is common practice today to use JSON to represent data on the outside and SQL to store the data on the inside. Both of these representations are used in a fashion that plays to their respective strengths.

This is an update to the original paper by the same name presented at CIDR 2005 (Conference on Innovative Data Systems Research). At that time, XML was more commonly used than JSON. Similarly, SOA (service-oriented architecture) was used more then, while today, it's more common to say simply, "service." In this article, "service" is used to mean a database encapsulated by its service or application code. It does not mean a microservice. That's for a separate paper. Nomenclature aside, not much has changed.

References

1. Bernstein, P. A., Hadzilacos, V., Goodman, N. 1987. *Concurrency Control and Recovery In Database Systems.* Addison-Wesley (http://sigmod.org/publications/dblp/db/books/dbtext/bernstein87.html).
2. Gray, J., Reuter, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.*