

Notes on Enterprise Architecture.

Tom Johnston
August, 2006.

I am a proponent of Service Oriented Architecture (SOA), less some of the buzz around that buzzword. The core principle is the same as that which guides object-oriented architectures – encapsulation, and communication only via published interfaces (the “service contract” in SOA terms).

Given an SOA high-level enterprise architecture, I am a proponent of a hub and spoke data architecture, with the data warehouse as the hub. To give some content to this statement: from this perspective, EAI (enterprise application integration) technology and tools are a band-aid that covers up an architectural injury that will require surgery – now or later. That architectural injury is a point-to-point architecture for the exchange of data among an enterprise’s business systems.

Given an SOA high-level enterprise architecture, I am a proponent of providing a near-seamless view of operational with historical data. To give some content to this statement: from this perspective, an Operational Data Store (ODS) and an historical data warehouse should not be physically distinct databases, nor should they be distinct sets of tables within the same database. Physically distinct ODS databases, that pass no-longer current data downstream to an historical data warehouse, are still being designed and built. But in most cases, they shouldn’t be.

I believe that two major facts strongly support these two architectural preferences. They are:

1. Exchanging data among IT systems (business applications) via a point-to-point data architecture is much more expensive, and much more error-prone, than exchanging data via a third-party intermediary, i.e. a data warehouse.
2. Historical data is becoming increasingly valuable to companies. It is often even more valuable when combined with operational data,

which itself is often more valuable when combined with historical data. The historical vs. operational distinction is crumbling, and we should not continue to build architectures which give a prominent role to that distinction.

Point-to-Point vs. Hub-and-Spoke.

The alternative to a point-to-point data architecture is a hub and spoke architecture, sometimes called a “data switch” architecture.

With the addition of a new IT system to a hub and spoke architecture, there are two new interfaces to build and support:

- The IT system’s interface with the hub (for both sending and receiving data); and
- The hub’s interface with the new system (for both receiving and sending data).

In contrast, with the addition of a new IT system to a point-to-point architecture, there are potentially many new interfaces to support. For example, while five fully-interconnected systems require 20 ($n * (n - 1)$) interfaces, the addition of a sixth may require up to 10 additional interfaces.¹

This difference has very real consequences. Experienced IT professionals know that when modifying or enhancing an IT system, or when replacing it wholesale with a new one, a significant part of the cost, and risk, is supporting the outflow interfaces, i.e. the batch or transactional interfaces in which other systems get data from the system in question. Since those interfaces, and their service requirements, are usually not well documented, it is not uncommon to find that when a major enhancement is deployed, one or more other systems “break” because their data retrieval interfaces with the enhanced system were overlooked.

¹ The usual formula is $[(n * (n - 1))/2]$. But this counts interfaces as arcs, whereas I describe a single arc, between one IT system and the hub, as two interfaces, thus counting interfaces as nodes, not as arcs.

With a hub and spoke architecture, only one interface needs to be supported when any system is enhanced or even replaced. That is the interface between the system and the data hub.

So from the data perspective, a hub and spoke architecture is a special kind of service-oriented architecture (SOA). It is one in which every business application has only one data consumer. That consumer is the hub. By the same token, it is one in which every business application has only one data supplier. That supplier is the hub.

Of course, the hub we are speaking of is the enterprise data warehouse (EDW). In fact, when the data being exchanged between IT systems, via the intermediary of the warehouse, is “current” data (however “current” is defined), it is the Operational Data Store aspect of the EDW that we are speaking of.

Why are legacy airlines failing?

Well, if hub and spoke is so much better than point-to-point, why are the major legacy airline carriers failing, and the newer carriers doing much better? Legacy carriers use a hub and spoke architecture. For example, Salt Lake City, Cincinnati and Atlanta are major hubs for Delta Airlines.

There are two reasons why hub and spoke airline architectures are failing. One is relevant to most IT architectures. The other is not.

1. Latency. Hub and spoke is slower than point-to-point. Flying from Atlanta to Chicago, through Cincinnati, takes longer than flying direct.
2. Cost. The successful newer airlines have chosen a small number of highly profitable point-to-point routes. As long as the number of point-to-point connections is kept relatively low, it costs more to implement a hub and spoke architecture covering the same points.

The first point is relevant. If system X sends its data to the hub on a nightly basis, but system Y needs system X data that is no more than an hour old, system Y must go directly to system X to get that data, bypassing the hub and creating a point-to-point connection with system X.

Since the hub we are speaking of is a data warehouse, this becomes the issue of real-time warehousing. The closer we get to real-time updating of an enterprise data warehouse, the fewer point-to-point connections will be necessary, because the latency in the hub and spoke is being eliminated.²

The second point is *not* relevant. The reason is that, in the set of IT systems for any enterprise, there are usually so many point-to-point connections to support that the total cost of support will always exceed the total cost of support for an equivalent hub and spoke architecture. Unlike the newer airline carriers, we in IT cannot choose to support “profitable” point-to-point connections, and ignore the rest. We must support them all.

Operational Data and Historical Data.

Inmon’s classic distinction between an ODS and an historical warehouse was made over ten years ago. It is a distinction that should not be *ignored*, but it is one that should be *demoted* in architectural importance. First, why. Then, how.

Why operational and historical data should be kept together.

The day-to-day operations of a business still require “current” data – *transactions* generated during the current accounting period, *dimensions* like customer, provider and product current in that period. Because day-to-day operations are so critical, there is still some justification for keeping some kind of distinction between current (operational) and historical data.

However, historical data has become increasingly important over the last decade. The deeper the historical basis from which trend lines are projected,

² Ian Ruston, VP of Enterprise Architecture for Matria, Inc., pointed out the latency issue to me. This was in a conversation in O’Hare airport, in June of this year.

the more reliable those projections will be. And so we see data marts in which several years' worth of transactions are stored.

But history exists as more than transactions. Persistent objects – customers, products, suppliers, regulations, competitors, etc. – these objects also have a history. A customer demographics trend analysis requires looking at either snapshots or versions of customer records.

So we can conclude that both kinds of history – transactions and snapshots/versions – belong on-line. But what is the argument against keeping operational data in one database – the ODS – and historical data in another database – the historical data warehouse?

The argument is cost of maintenance and a low level of interoperability. Often the same queries are asked of both databases. For example, “Show me all product SKUs in product class ‘flexible packaging’”, (“as of right now” being assumed). But for comparison purposes, we might also ask “Show me all product SKUs in product class ‘flexible packaging’ as of the last day of the year for each of the last five years”. Except for the time periods involved, these two are the same queries. Maintaining one query, to which a date can be supplied, is less expensive than maintaining two queries. Multiplied by the large number of queries found in most enterprises, the cost differential is significant.

Also: sometimes we want an historical query that ranges from some point in the past, up to and including the most current data we have. If the ODS is database-instance level distinct from the historical data warehouse, this requires either a federated union (if the schemas have been kept in sync), or a more complex federated query (if the schemas have drifted apart).

How operational and historical data should be kept together.

Let's distinguish three levels of separation between operational and historical data.

1. Operational data is kept in one database instance. Historical data is kept in another.

2. Operational data and historical data are kept in the same database instance. But there are two sets of tables. One is a set of tables for operational data. The other is a set of tables for historical data. For example, an operational Customer table has cust-id as its primary key, whereas the corresponding historical Customer table has cust-id + date as its primary key.
3. Operational and historical data are kept in the same tables. The date (usually 12/31/9999 for the one current instance) is the high-order part of the clustering index, because otherwise day-to-day operational queries would become very slow. Data mining queries, which would benefit from making the date the low-order part of the clustering index, can afford to run slow.

The first level of separation is the strongest. Operational and historical data are in completely separate databases. This is the traditional Inmon-style architecture, and is still the most common warehouse architecture.

One motivation for this architecture is that it was the natural “next step” in the evolution of the availability of historical data. In the 70’s and 80’s, most historical data was archived onto tape. The only data that was available on-line was operational, current period, data. That’s all that we could afford to keep on-line, back then.

In the late 80’s and early 90’s, as hardware resources increased in power and storage capacity, and decreased in price, the natural next step was to restore those tape backups (at least some of them), and load a separate historical database with that data. Bill Inmon named this new database the “data warehouse”.³

But more and more companies are finding that historical data is more valuable when it “continues” up to and includes current data, and that

³ However, I believe that Barry Devlin, of IBM Ireland, contests Inmon’s claim to be the “father” of data warehousing.

current data is more valuable when seen in the “perspective” of the data that went before it. Indeed, this is the whole idea behind “real-time data marts”.

But most of today’s data marts, real-time or not, view a history of *transactions* from the perspective of the current state of *persistent objects* – customers, products, etc. (No more on this point for now. It gets into the as-is vs. as-was distinction, and how many data marts should support both, but can only support one.)

{following are rough notes. They haven’t reached even the first draft of an article stage of completion. And there’s a good deal of repetition.}

The Value of an EDW.

Single source of the truth. The “system of record” function.

With the EDW as a single source of data, data marts and other reporting databases do not independently source and update themselves. This independence meant that those databases were completely unsynchronized. There would be an unknown number of inconsistencies among them – a member’s true SSN, how many pharmacy claims a member submitted last month, etc. This problem is eliminated with a single source of the truth. No more inconsistent reporting of who is on what list, what a member’s last name really is, or the last time she was pregnant.

Simplified inter-system data exchanges. The “data hub” function.

This guarantees that individual systems can be modified or replaced without affecting other systems, as long as the data sent to the EDW (and the “contractual terms” of that relationship) remain the same. In place of impacting potentially dozens of other systems, there is only *one* impact to consider. In my experience, this is another couple of orders of magnitude time and cost savings over maintaining or replacing systems that are point-to-point connected with other systems.

To do: build a data hub, and replace all point-to-point systems interconnects with spoke and hub interconnects.

Painless derivation of new views. The “data repository” function.

Suppose a hub and spoke architecture is implemented, and nearly all data generated by OLTP systems, or brought in by those systems, is in the EDW.⁴

Any view using data in the EDW can be created from this single physical repository, and does not need to be assembled from multiple sources. This work has to be done *once*. It does not have to be done every time a new data source is needed, which would be required if an EDW were not involved.

Sourcing data once is obviously cheaper and faster than sourcing it as many times as there are legacy systems that are the original source of the data needed, and reporting systems that need that data. But in addition, sourcing data from an EDW, once the physical interconnect is initially established, is simply a matter of creating *views* over the base tables of the EDW. To obtain the same data, by going to each of the original source systems, requires building new physical data feeds, and is generally an order of magnitude more expensive than getting it via one view on the EDW.

For example, with an EDW, an Electronic Health Record (EHR) could be ready for access in a matter of days. All that would be required would be to write a SQL statement to create a view over the EDW that looked like a physical table of EHRs. No maintenance would be needed for that (or any other) view because maintenance is applied to the underlying physical tables – the “base layer” tables – as they are periodically updated from source systems. In comparison, without an EDW, assembling that data into an EHR database by copying data from multiple source systems, would take months.

⁴ The “nearly” would go away if all updates to the EDW were real-time transactional. But 100% real-time transactional updates will probably never be worth the cost. “As current as the business needs” is the currency objective for the EDW, not “100% real-time”.

The “pre-fabricated parts” concept.

What I have been describing is the concept of a data warehouse as a warehouse of pre-fabricated parts. The traditional way of building new IT systems is to do so from “raw material” – which means going back to the individual data sources and creating data feeds from each one of them. This is an expensive and slow process.

But with an EDW, all the data that is needed is already assembled in one place, and can be fabricated into useful “sub-assemblies” at a truly negligible cost. These pre-fabricated sub-assemblies are database *views* of the data in the EDW, and even highly complex views can be written and debugged in a matter of a few days. These views are what replace all the effort of going back to original sources, creating data feeds, and integrating the results. This is a cost, time and reliability savings of one to two orders of magnitude, a few man-days instead of several man-months.

To do: build a data assembly plant, and develop new systems from pre-fabricated sub-assemblies instead of from basic parts obtained from multiple locations.

Immediate availability of history.

All reportable history should be kept in the EDW. This means that not only transaction history, but versioned/snapshotted history of persistent objects, would be “immediately available”, meaning that it would be available from one physical source, without procedural programming, by means of SQL only.

Transaction history is best reported on with OLAP-structured data marts. Instead of physically assembling multiple sources to feed each data mart, the EDW would be the single source for all of them. And a new data mart “cube”, whether physical or virtual, could be created with SQL alone.

Non-transaction history consists of versions and/or snapshots of persistent objects – things that exist over time, such as members, insurance plans, contracts or healthcare surveys. With a full history of changes to all such objects contained in the EDW, both a full history and a view of any object as

of any point in time, is immediately available. And once again, only SQL is needed.

This treats history as “first class data”. It stores history in the same tables as current data, not in separate history tables. This means that historical eligibility data, historical demographic data, as well as a history of claims transactions, can be retrieved just by supplying the date you are interested in. No separate tables, no separate code, no separate SQL. History is seamlessly integrated with current data.

An analogy. Physical warehouses.

Suppliers don't ship directly to Home Depot stores. They ship to warehouses (actually to a cascade of warehouses). Home Depot then ships from the warehouses to the stores.

Neither Home Depot, nor any other multi-outlet merchant, could afford to ship directly from each supplier to each store. But the traditional way of developing new business applications is to create a direct data supply chain from each data supplier (legacy system) directly to that new application!

In other words, warehouses are “load consolidation and assembly points”. Suppliers ship to the warehouses. Home Depot (a) creates sub-assemblies at those warehouses, and (b) puts together shipments to stores that contain product from multiple suppliers. The EDW (as a single physical database, or as a federation of distributed databases) is a “load consolidation and assembly point”.

Three roles for data warehouses.

A support structure for an integrated set of data marts.

In Ralph Kimball's terms, the means by which data mart dimensions are conformed. For information derived from rolling up and “slicing and dicing” metrics derived from transactions, Kimball's is the best architecture.

The flaw in Kimball's architecture is its incompleteness. It does not directly support non-transactional trend-over-time analysis, e.g. analysis of the changing demographic profile of a population. It does not directly support analysis of interrelationships among persistent objects, e.g. changing cardinalities in customer to salesperson relationships.

An integrated union of operational (current) data from legacy system OLTP databases.

In Bill Inmon's terms, an Operational Data Store (ODS). An important trend is for ODSs to become increasingly real-time, updated transactionally and not just in batch mode.

Ultimately, except for internal transactions needed to "run the machinery" of OLTP systems, all such systems would interface only with the ODS, sending the business data they create to the ODS, and obtaining the business data they need from the ODS. This would replace the current application point-to-point architecture with a "spoke and hub", or "data switch" architecture. It is the ultimate service oriented architecture, permitting specific applications to be modified or even unplugged and replaced, just as long as the data feeds into the ODS remain supported.

Modifications to a system would affect only one interface! No risk of screwing up other systems with unanticipated repercussions!

An historical repository of snapshots and versions of once-operational data.

Commonly known as the Historical Data Warehouse. The source for trend-over-time projections and data mining.

For example, trend lines established for ICD9 codes would indicate diseases on the rise, or perhaps on decline. Associating the data points on these curves with NDC codes from pharmacy claims would indicate the relative effectiveness of different medications for a disease.

Odds and Ends.

Some historical data must be available in real-time.

It should be as easy to produce an SQL result set showing objects as of any past point of time, as it is to produce an SQL result set showing current data. It should be just as easy to ask for a result set that shows all changes to all a client's members over the past eighteen months as it is to ask for the current version of those members.

The reason is not simply traceability. Traceability can be satisfied with a link back to the original source file.

One reason is to support on-line applications in which personnel must answer client member questions when those questions involve historical information. This is a real-time need for history, not an "I'll get back to you later" need.

Another reason is to support trend-over-time analysis. It should not be necessary to involve IT personnel to get a set of data describing changes to objects over time. It should be possible to get this data with a simple SQL query. The enterprise data warehouse should provide this, instead of requiring SAS and other types of programmers to build their own departmental-level historical databases.

A third reason is to support as-was as well as as-is reporting in data marts. Data marts are "star schema", slice-and-dice rollups of transactional metric data, e.g. how many prescriptions for diabetes in a certain population, by month, for the last twenty-four months. If the definition of "prescriptions for diabetes" changed recently, e.g. if new medications were added to that set, then the above query has two forms. In one form, it totals over twenty-four months using the old set of diabetes medications. In the other form, it totals using the new set. The former is an as-was query; the latter is an as-is query.

Some data must be allowed into an EDW even if it is incomplete and/or incorrect.

One level of error is row-level. This includes inserting “dummy” rows to be the target of foreign keys that would otherwise be null.

Another level of error is column-level. This involves inserting rows into tables when one or more of the data elements in the source transaction contain bad data. An example would be an invalid ICD9 code on a medical claim.

When incorrect or incomplete data is allowed into the EDW, it must be identified as such.

The need for IT Operations intervention to get to historical data must be minimized.

This is essential for historical data that must be real-time available, i.e. available while a person is sitting at a screen waiting for the data.

It will be helpful on other occasions, as well, e.g. to restore a database state in order to rerun a set of reports, especially if those reports were not run on a full database backup boundary. If this process can be automated, as it can with versioned tables, IT Operations will be free to concentrate on other responsibilities. In addition, the possibility of error will be reduced.

Objective 1. The Single Source of Truth.

An EDW should be the “single source of the truth”, insuring consistency across all enterprise reporting and querying. This means that:

- All data marts and reporting databases should draw their data from the EDW. They should not draw their data directly from any other source.
- Legacy OLTP systems should supply the EDW with the data they obtain or create, and pull from the EDW the data they require.

Without a single source of the truth, data marts and reporting databases refresh themselves by independently pulling the data they need from various sources, often from the same source but at different points in time. OLTP systems gradually develop slightly different sets of customers (clients, members), update their standard reference tables at slightly different times, etc. Such variation is the root cause of inconsistency across reports and data marts.

Objective 2. A Hub in a Service-Oriented Architecture.

An EDW should be a single “hub” (or “data switch”) for the entire enterprise. This is an architecture completely opposed to a point-to-point architecture. With point-to-point architectures, application systems supply data to other systems, and receive data from other systems. Although this is a traditional architecture for IT systems, and still the prevailing one, it is inferior to a data switch architecture.

Point-to-point data exchanges among application systems create a spaghetti code tangle, at the systems level. When applications exchange data with one another, they become *tightly coupled*. Consequently, when any one of them is replaced or modified, the development team must be careful to identify all interfaces for which the application is a data supplier, and be sure that the modified or new system continues to support the “service level agreement” for each of those interfaces.

With a hub and spoke (or data switch) architecture, every application is a data supplier for only one target – the enterprise data warehouse. Modification or replacement of the system is guaranteed not to disturb any other systems as long as the one interface to the data warehouse is supported.

This architecture makes the EDW the heart of a Service Oriented Architecture. This is how you get to a component architecture in which you can unplug one component and replace it with another. It’s not just a matter of fitting tools and systems together. It’s a matter of common *semantics*, guaranteed by a common source of data for all IT systems.

