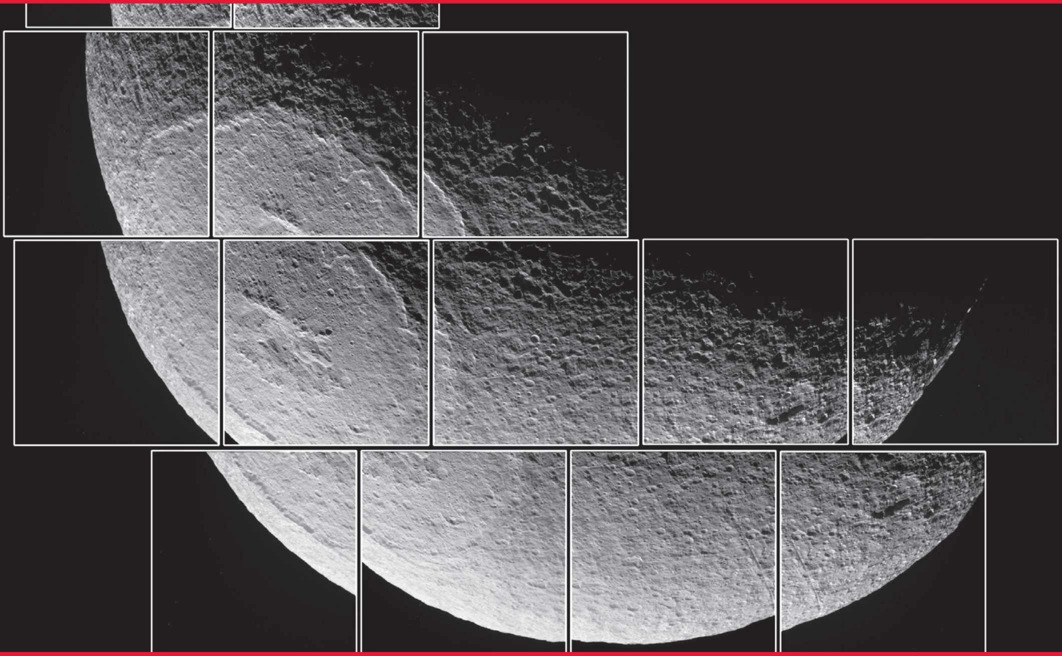
O'REILLY®

Building on Multi-Model Databases

How to Manage Multiple Schemas Using a Single Platform



Pete Aven & Diane Burley

Your data deserves better.

There's nothing wrong with your data
that a better database can't fix.

- Rely on a 100% trusted, enterprise-grade multi-model database
- Load your data *as-is*, no upfront data modeling required
- Unify your structured and unstructured data
- Build and deploy your apps in days, not months

www.marklogic.com

MarkLogic®

Building on Multi-Model Databases

How to Manage Multiple Schemas Using a Single Platform

Pete Aven and Diane Burley

Beijing • Boston • Farnham • Sebastopol • Tokyo

Building on Multi-Model Databases
by Pete Aven and Diane Burley

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Shannon Cutt
Production Editor: Melanie Yarbrough
Copyeditor: Octal Publishing Services
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2017: First Edition

Revision History for the First Edition
2017-05-11: First Release

978-1-491-97788-0

[LSI]

Table of Contents

About This Book

Purpose

CTOs, CIOs, senior architects, developers, analysts, and others at the forefront of the tech industry are becoming aware of an emerging database category that is both evolutionary and suddenly necessary: *multi-model databases*. A multi-model database is an integrated data management solution that allows you to use data from different sources and formats in a simplified way.

This book describes how the multi-model database provides an elegant solution to the problem of heterogeneous data. This new class of database naturally allows heterogeneous data, breaks down technical data silos, and avoids the complexity of integrating multiple data stores for multiple data types. Organizations using multi-model databases are discovering and embracing this class of database capabilities to realize new benefits with their data by reducing complexity, saving money, taking advantage of opportunities, reducing risk, and shortening time to value.

The intention of this book is to define the category of multi-model databases. It does make an assumption that you have at least a cursory knowledge of NoSQL database management systems.

Audience

The audience for this book is the following:

- Anyone managing complex and changing data requirements

- Anyone who needs to integrate structured, semi-structured, and unstructured data or is interested in doing so
- CTOs, CIOS, senior analysts, and architects who are overseeing and guiding projects within large organizations
- Strategic consultants who support large organizations
- People who follow analysts, such as other analysts, CTOs, CIOs, and journalists

Introduction

Database management systems (DBMS) have been around for a long time, and each of us has a set of preconceived notions about what they are, and what they can be. These preconceptions vary depending on when we started our careers, whether we lived through the shift from hierarchical to relational databases, and if we have gained exposure to NoSQL yet. Our understanding of databases also varies depending on which areas of information technology we work in, ranging from transactional processing to web apps, to business intelligence (BI) and analytics.

For example, those of us who started in the mainframe COBOL era understand hierarchical tree-structures and processing flat files whose structures are defined inside of a COBOL program. Curiously, many of us who have adopted cutting-edge NoSQL databases have some understanding of hierarchical tree structures. Working on almost any system during the relational era ensures knowledge of SQL and relational data modeling around rows, columns, keys, and joins. A more rarified group of us know ontology modeling, Resource Description Framework (RDF), and semantic or graph-based databases.

Each of these database[1] types has its own, unique advantages. As data continues to grow in volume and variety, so, too, does our need to utilize this variety of formats and databases—and often to link the various data stores together using extract, transform, and load (ETL) jobs and data transformations.

1 For simplicity, we will sometimes blur the line between a "database" and a "database management system" and use the simpler term "database" where convenient.

Unfortunately, each new data store selected becomes a "technical silo"—a new data store with boundaries between them that are both physical, because the data is stored in different places, and conceptual, because the data is stored in fundamentally different forms. Relational and non- (or not-only) relational (NoSQL) databases are different from each other, and different from graph databases, and different from other stores.

Until recently, this forced a difficult choice. Choose the relational model *or* the document model *or* graph type models; scale up *or* scale out; perform analytical *or* transactional work; or choose a few and cobble them together with ETL jobs and integration code.

Fortunately, the DBMS landscape is evolving rapidly. What organizations really want is a way to use all their data in an integrated way, so why shouldn't database products support this out of the box? Integrated data storage and access—across data types and functions —is exactly the goal of multi-model database management platforms.

> A *multi-model database* supports multiple data models in their natural form within a single, integrated backend, and uses data standards and query standards appropriate to each model. Queries are extended or combined to provide seamless query across all the supported data models. Indexing, parsing, and processing standards appropriate to the data model are included in the core database product.

This definition illustrates that simply storing various data types—as one can do in relational database management systems (or RDBMS) binary large object (or BLOB) or a filesystem directory—does not a multi-model database make. The true multi-model database can do the following:

- Index data in natural ways for the models supported
- Parse and index the inherent structure in self-describing data formats such as JSON, XML, and RDF
- Implement standards such as query languages and validation or schema definition languages for the models supported
- Provide integrated APIs that not only query the individual data models, but also query across multiple data models

- Implement data processing languages native to each supported data model

Provided these capabilities, a multi-model database does not require you to define the shape or schema of the data before loading it; instead, it uses the inherent structure in the data being stored. This makes data management flexible and adaptive, able to respond to the needs of downstream applications and changing business requirements.

With this understanding of what a multi-model database *is*, we can move on to what a multi-model database is *for* and describe use cases. That said, *any* system that stores and accesses different types of data will benefit from a multi-model database. An enterprise or complex use case involving many existing data systems will naturally encounter many different data formats, so we will focus on data integration/silo-busting as a key use case. Another scenario is the integration of structured data handling with unstructured or semi-structured data. This often has been addressed by standing up a relational or NoSQL database and manually integrating it with a search platform but can be included in one multi-model database. We will also focus on a particular multi-model combination of documents with graph structures, which is a natural model for many domains with interrelated business entities.

Some Terms You'll Need to Know

Table P-1 provides definitions to some terms that will come up frequently in this book.

Table P-1. Key terms related to multi-model databases

Term	Description
Multi-model	A multi-model database supports multiple data models in their natural form within a single, integrated backend, and uses data standards and query standards appropriate to each model. Queries are extended or combined to provide seamless query across all the supported data models. Indexing, parsing, and processing standards appropriate to the data model are included in the core database product. Document, graph, relational, and key-value models are examples of data models that can be supported by a multi-model database.
Multiquery engine	A query layer that allows multiple ways to query one data model.

Term	Description	
Query language	A language designed to identify subsets of data in a database, and often to manipulate the data as the data is retrieved through joins, subselects, or other changes. Every data model other than text has a query standard, and even text query has natural, purpose-built query syntaxes.	
	Model	**Query language**
	XML	XQuery for query XSLT for manipulation
	JSON	JavaScript for manipulation
	RDF	SPARQL
	Relational	SQL
	Text	Search
Data indexing	All databases create one suite of indexes on data as it is ingested to allow fast query of that data. True multi-model will have *one integrated suite of indexes* across data models that allows a single, composable query to quickly retrieve data across all the data models, simultaneously.	
Canonical model	A type of data model that presents data entities and relationships in a standardized, simple form. Also known as a *common data model*.	
Polyglot programming	Using several programming languages within a given application.	
Polyglot persistence	Using several data models for different aspects of a system or enterprise. The polyglot persistence approach is motivated by the idea that data should be stored in the format and DBMS that best fits the data stored and the functionality required. Traditionally, this meant choosing a different DBMS for each type of data and having the application communicate with the right data store. However, a true multi-model DBMS provides polyglot persistence with a single, integrated backend.	
Multiproduct multi-model	A multi-model system with multi-model query languages and APIs, but which are powered by a collection of separate data stores internally. These products provide one simplified API for data access, but use a façade or orchestration layer atop multiple internal databases, which adds to complexity and can affect the databases' consistency, redundancy, security, and scalability.	
Shared nothing (SN) architecture	A distributed computing architecture in which each node is independent and self-sufficient, and there is neither a single point of contention across the system, nor a single point of failure. More specifically, none of the nodes share memory or disk storage.	

Acknowledgments

A HUGE thank you to all the reviewers and contributors to this project. Thank you to Diane Burley, Damon Feldman, David Gorbet, James Kerr, Justin Makeig, Ken Krupa, Evelyn Kent, and Derek Laufenberg for all of your above-and-beyond contributions. This report would not be possible without all of your keen, discerning eyes and extraordinary additions. Thank you also to Parker Aven, my constant inspiration. Next weekend it's just you, me, Legos, and movies, sweetheart!

CHAPTER 1

The Current Landscape

Somewhere in the business, someone is requesting a unified view of data from IT for information that's actually stored across multiple source systems within the organization. This person wants a single report, a single web page, or some single "pane of glass" that she can can look through to view some information crucial to her organization's success and to bolster a level of confidence in her division's ability to execute and deliver accurate information to help the business succeed.

In addition to this, businesses are also realizing that simply having a "single view" alone is not enough, as the need to transact business across organizational silos becomes increasingly necessary. Hearing the phrase, "That's a different department; please hold while I transfer you" is tolerated less frequently by many of today's digital first consumers.

What's the reality? The data is in silos. Data is spread out across mainframes, relational systems, filesystems, Microsoft SharePoint, email attachments, desktops, local shares; it's everywhere! (See Figure 1-1.) For any one person in an organization, there are multiple sources of data available.

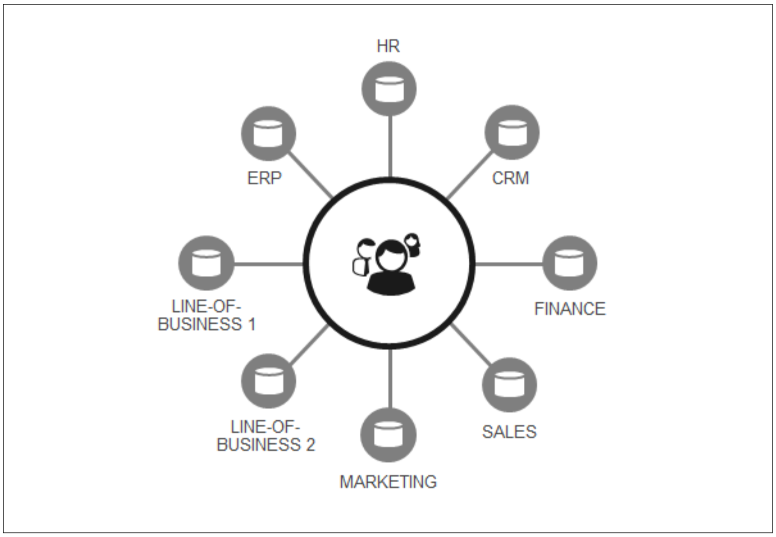


Figure 1-1. What we have: data in silos

Because the data isn't integrated, and reports still need to be created, we often find the business performing "stare and compare" reporting and "swivel chair integrations." This is when a person queries one system, cuts and pastes the results into Microsoft Excel or PowerPoint, swivels his chair, queries the next system, and repeats until he has all the information he thinks he needs. The final result is another silo of information manifested in an Excel or PowerPoint report that ultimately winds up in someone's email somewhere.

This type of integration is manual, error-prone, and takes too long to get the required answers that business can act upon. So, what happens next? Fix it! Business submits a request to IT to integrate the information. This results in a data mart and the creation of a new silo. There will be DBMS provisioning, reporting schema design, index optimizations, and finally some form of ETL to produce a report. If the mart has already been created, modifications to the existing schemas and an update to ETL processes to populate the new values will be required. And the cycle continues.

The sources for these silos are varied and make sense in the contexts they are created:

- To be able to ensure a business can quickly report its finances and other information, that business asks IT to integrate multiple, disparate sources of data, creating a new data silo. Some thought might have been given to improving or even consolidating existing systems, but IT assessed the landscape of existing data silos, and changes were daunting.

- Many silos have been stood up in support of specific applications for critical business needs, each application often coupled with its own unique backend system, even though many likely contain data duplicated across existing silos. And the data is worse than just duplicated: it's transformed. Data might be sourced from the same record as other data elsewhere, but it no longer looks the same, and in some cases has diverged, causing a master data problem.
- Overlapping systems with similar datasets and purposes are acquired as a result of mergers and acquisitions with other companies, each of which had fragmented, siloed data!

Before we look at the reality of what IT is going to face in integrating disparate and various systems, let's ask ourselves the importance of being able to integrate data by looking briefly at a few real-world scenarios. If we're not able to integrate data successfully, the challenges and potential problems to our business go far beyond not being able to generate a unified report. If we're not able to integrate enterprise resource planning (ERP) systems, are we reporting our finances and taxes correctly to the government? If we work in regulated industries such as financial services, what fines will we face if we're not able to rapidly respond to audits from financial regulatory boards on an integrated view of data requiring the ability to answer questions from ad hoc queries? If we're not able to integrate HR systems for employees, how can we be sure that an employee who has left the company for new opportunities or who has been terminated is no longer receiving paychecks and no longer has access to facilities and company computer systems? If you're a healthcare organization and you're not able to integrate systems, how can you be certain that you have all the information needed to ensure the right treatment? What if you prescribe a medicine or procedure that is contraindicated for a patient taking another medicine that you were unaware of? These types of challenges are real and are being faced by organizations—if not entire industries—daily.

In 2004, I was as a systems engineer for ERP-IT at Sun Microsystems. At that job, I helped integrate our ERP systems. In fact, we end-of-lifed 80 legacy servers hosting 2 custom, home-grown applications to integrate all of Sun's ERP data into what was at the time the world's largest Oracle instance. But the silos remained! I do know of people who left the company or were laid off who continued for a long time to receive paychecks and have access to campus buildings and company computer networks! This is because of the challenge of HR data being in silos, and updates to one system not propagating to other critical systems. The amazing thing is that even though I witnessed this in 2005, those same problems still exist!

Data silos are embedded in or support mature systems that have been implemented over long periods of time and have grown fragile. To begin to work our way out of this mess and survey the landscape, we frequently see mainframes, relational systems, and filesystems from which we'll need to gather data. As one CIO who had lived through many acquisitions told us, "You go to work with the architecture you have, not the one you designed."

Types of Databases in Common Use

Let's take a look at the nature of each of these typical data store types: mainframes, relational, and filesystems.

Mainframe

IBM first introduced IMS DBMS, the hierarchical filesystem, in 1966 in advance of the Apollo moon mission. It relied on a tree-like data structure—which reflects the many hierarchical relationships we see in the real world.

The hierarchical approach puts every item of data in an inverted-tree structure, extending downward in a series of parent-child relationships. This provides a high-performance path to a given bit of data. (See Figure 4-1.)

The challenge with the mainframe is that it's inflexible, expensive, and difficult to program. Queries that follow a database's hierarchical structure can be fast, but others might be very slow. There are also legacy interfaces and proprietary APIs that only a handful of people in the organization might be comfortable with or even have enough knowledge of to be able to use them.

Mainframes continue to be a viable mainstay, with security, availability, and superior data server capability topping the list of considerations.[1] But to integrate information with a mainframe, you first need to get the data out of the mainframe; and in many organizations, this is the first major hurdle.

Relational Databases

In 1970, E.F. Codd turned the world of databases on its head. In fact, the concept of a relational database was so groundbreaking, so monumental, that in 1981 Codd received the A.M. Turing Award for his contribution to the theory and practice of database management systems. Codd's ideas changed the way people thought about databases and became the standard for database management systems.

A relational database has a succinct mathematical definition based on the *relational calculus*: a means of organizing and querying data via *joins* (roughly "tables," in normal language) against a primary key.

Previously, accessing data required sophisticated knowledge and was incredibly expensive and slow. This was because it was inexorably tied to the *application* for which it was conceived. In Codd's model, the database's schema, or logical organization, is disconnected from physical information storage.

In 1970, it wasn't instantly apparent that the relational database model was better than existing models. But eventually it became clear that the relational model was more simple and flexible because SQL (invented in 1972) allowed users to conduct ad hoc queries that can be optimized in the database rather than in the application. SQL is declarative in that it asks the database for what it wants and does not inform the database how it wants the task performed. Thus, SQL became the standard query language for relational databases.

Relational databases have continued to dominate in the subsequent 45 years. With their ability to store and query tabular data, they proved capable of handling most online transaction-oriented applications and most data warehouse applications. When their rigid adherence to tabular data structures created problems, clever programmers circumvented the issues with stored procedures, BLOBs, object-relational mappings, and so on.

1 Ray Shaw, "Is the mainframe dead?" ITWire, January 20, 2017.

The arrival of the personal computer offered low-cost computing power that allowed any employee to input data. This coincided with the development of object-oriented (OO) methods and, of course, the internet.

In the 1970s, ’80s, and ’90s, the ease and flexibility of relational databases made them the predominant choice for financial records, manufacturing and logistical information, and personnel data. The majority of routine data transactions—accessing bank accounts, using credit cards, trading stocks, making travel reservations, buying things online—all modeled and stored information in relational structures. As data growth and transaction loads increased, these databases could be scaled up by installing them on larger and ever-more powerful servers, and database vendors optimized their products for these large platforms.

Unfortunately, scaling out by adding parallel servers that work together in a cluster is more difficult with relational technology. Data is split into dozens or hundreds of tables, which are typically stored independently and must be joined back together to access complete records. This joining process is slow enough on one server; when distributed across many servers, joins become more expensive, and performance suffers. To work around this, relational databases tend to replicate to read-only copies to increase performance. This approach risks introducing data integrity issues by trying to manage separate copies of the data (likely in a different schema). It also uses expensive, proprietary hardware.

Data modeling in the relational era

One of the foundational elements of Codd was the third normal form (3NF), which required a rigid reduction of data to reduce ambiguity. In an era of expensive storage and compute power, 3NF eliminated redundancy by breaking records into their most atomic form and reusing that data across records via joins. This eliminated redundancy and insured atomicity, among other benefits. Today however, storage is cheap. And although this approach works well for well-structured data sources, it fails to incorporate communication in a shape that’s more natural for humans to parse. Trying to model all conversations, documents, emails, and so on within rows and columns becomes impossible.

The challenge with ETL

But from the "when you have a hammer, everything becomes a nail" department, relational databases emerged as the de facto standard for storing integrated data in most organizations. After a relational schema is populated, it is simple to query using SQL, and most developers and analysts can write SQL queries. The real challenge, though, is in creating a schema against which queries will be issued. Different uses and users need different queries; and all too often, this is provided by creating different schemas and copies of the data. Even in a new, green-field system, there will typically be one transactional system with a normalized schema and a separate analytic data warehouse, with a separate (star or snowflake) dimensional schema.

Data integration use cases make this problem even more difficult. To appropriately capture and integrate all the existing schemas (and possibly mainframe data and text content) that you want to integrate takes a tremendous amount of time and coordination between business units, subject matter experts, and implementers. When a model is finally settled upon by various stakeholders, a tremendous amount of work is required to do the following:

- *Extract* the information needed from source systems,
- *Transform* the data to fit the new schema, and
- *Load* the data into the new schema.

And thus, *ETL*. Data movement and system-to-system connectivity proliferate to move data around the enterprise in hopes of integrating the data into a unified schema that can provide a unified view of data.

We'll illustrate this challenge in more detail when we contrast this relational approach with the multi-model approach to data integration in Chapter 2. But most of us are already very familiar with the problems with this. A new target schema for integrating source content is designed with the questions the business wants to ask of the data today. ETL works great if the source system's schemas don't change and if the target schema to be used for unification doesn't change. But what regularly happens is the business changes the questions it wants to ask of the data, requiring updates to the target schema. Source systems might also adapt and update their schemas to support different areas of the business. A business might acquire another company, and then an entire new set of source systems will need to be included in the integration design. In any of these scenarios, ETL jobs will require updates.

Often, the goal of integrating data into a target relational schema is not met because the goal posts for success keep moving whenever source systems change or the target system schema changes, requiring the supporting ETL to change. Years can be spent on integration projects that never successfully integrate all the data they were initially scoped to consolidate. This is how organizations find themselves two and a half years into a one-year integration project.

The arrows in Figure 1-2 are not by any means to scale. Initial modeling of data using relational tools can take months, even close to a year, before subsequent activities can begin. In addition, are big design trade-offs to be addressed. You can begin with a sample of data and not examine all potential sources. This can be integrated quickly but inflexibly and will require change later when new sources are introduced. Or you can aim for complex and "flexible" using full requirements; however, this can lead to poor performance and extended time to implement. The truth for the majority of data integration projects is we don't know what sources of data we might integrate in the future. So even if you have "full requirements" at a point in time, they will change in the future if any new source or query type is introduced.

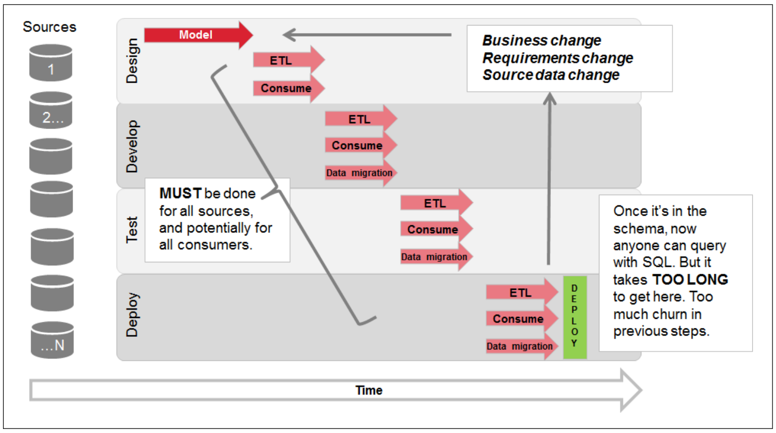

Sources
1
2...
...N
Design
Develop
Test
Deploy
Model
ETL
Consume
Data migration
Business change
Requirements change
Source data change
MUST be done for all sources, and potentially for all consumers.
DEPLOY
Once it's in the schema, now anyone can query with SQL. But it takes TOO LONG to get here. Too much churn in previous steps.
Time

Figure 1-2. Integrating data with relational

Based on the model, ETL will be designed in support of transforming this data, and the data will be consumed into the target database.

Consumer applications being developed on separate project timelines will require exports of the data to their environments so they can develop in anticipation of the unified schema. As we step through the software development lifecycle for any IT organization, we find ourselves "ETLing" the data to consume it as well as copying and migrating data and supporting data (e.g., lookup tables, references, and data for consumer projects) to a multitude of environments for development, testing, and deployment. This work must be performed for all sources and potentially for all consumers. If anything breaks in the ETL strategy along this path, it's game over: go back to design or your previous step, and start over.

When we finally make it to deploy, that's when users can begin actually asking questions of the data. After the data is deployed, anyone can now query it with SQL. But first, it takes too long to get here. There's too much churn in the previous steps. Second, if you succeed in devising an enterprise warehouse schema for all the data, that schema might be too complex for all but a handful of experts to understand. Then, you're limited to having only the few people who can understand it write applications against it. An interesting phenomenon can occur in this situation in which subsets of the data will subsequently be ETL'd out of the unified source into a more query-friendly data mart for others to access—and another silo is born!

All of our components must be completely synchronized before we deploy. Until the data is delivered in deploy, it remains inaccessible to the business as developers and modelers churn on it to make it fit the target schema. This is why so many data integration projects fail to deliver.

Schema-first versus schema-later

The problem with using relational tools, ETL, and traditional technologies for integration projects is that they are *schema-first*. These tools are *schema-driven*, not *data-driven*. You first must define the schema for integration, which requires extensive design and modeling. This model will be out of date the day you begin because business requirements and sources of data are always changing. When you put the schema first, it becomes the roadblock to all other integration efforts. You might see the familiarity of schema-first as it pertains to a master data management (MDM) project as well. But there is a solution to this problem!

A multi-model database gives us the ability to implement and realize our own *schema-later*. A multi-model database is unique in that it separates data ingestion from data organization. Traditional tools require that we define how we are going to organize the data before we load it. Not so with multi-model. By providing a system that gives us the ability to load data *as is* and access it immediately for analysis and discovery, we can begin to work with the data from the start of any project and then lay schemas on top of it when needed. This data-driven approach gives us the ability to deploy meaningful information to consumers sooner, rather than later. Schema-later allows us to work with the data and add a schema to it simultaneously! We'll be covering just how multi-model databases can do this in much greater detail in the coming chapters.

Filesystem

Now, we often don't think of text as a data model, but it very much is. Text is a form, or shape, of data that we'll want to address in a multi-model database. Structured query in a relational system is great when we have some idea of what we're looking for. If we know the table names and column names, we have some idea of where to query for what we're looking for. But when we're not sure where the data might reside in a document or record, and we want to be able to ask any question we want to find of any piece of the data to find what we're looking for, we want to be able to search text.

We might want to search text within a relational system, or we might want to search documents stored in SharePoint or on a filesystem. Most large systems have text, or evolve to include text. Organizations go to great lengths to improve query and search, either shredding documents into relational systems, augmenting their SharePoint team sites with lists, or bolting on an external search engine. It's common to see systems implemented that are a combination of structured, transactional data stored in a relational database that's been augmented with some sort of search engine for discovery, for which fuzzy relevance ranking is required.

Text has meaning in search. The more times a search term is repeated within a document, the more relevant that document might be to our search. Words can occur in phrases; they have proximity to one another; and they can happen in the context of other words or the structure of a document, such as a header, footer, a paragraph, or in metadata. Words have relevance and weight. Text is a different type of data from traditional database systems, traditionally requiring different indexes and a different API for querying. If we want to include any text documents or fields in a unified view, we'll need to address where text fits in multi-model, and it does so in the context of search. Search is the query language for text. Search engines index text and structure, and these indexes are then often integrated with the results of other queries in an application layer to help us find the data we're looking for. But as we'll find out, in a multi-model database, search and query can be handled by a single, unified API.

Many times, text is locked within binaries. It's not uncommon to find artifacts like Java Objects or Microsoft Word documents persisted as BLOBs or Character Large OBjects (CLOBs) within a relational system. BLOBs and CLOBs are typically large files. On their own, BLOBs provide no meaning to the database system as to what their contents are. They are opaque. They could contain a Word document, an image, a video—anything binary. They have to be handled in a special way through application logic because a DBMS has no way of understanding the contents of a BLOB file in order to know how to deal with it. CLOBs are only marginally better than BLOBs in that a DBMS will understand the object stored is textual. However, CLOBs typically don't give you much more transparency into the text given the performance overhead of how RDMBS databases index data.

Often, CLOB data that a business would like to query today is extracted via some custom process into columns in a relational table, with the remainder of the document stored in a CLOB in a column that essentially becomes opaque to any search or query. During search and query of the columns available, the business will stumble upon some insight that makes it realize it would like to query additional information from the CLOB'd document. It will then submit a request to IT. IT will then need to schedule the work, and then update the existing schema to accept the new data and update its extraction process to pull the new information and populate the newly added columns to the schema. The process to accomplish this task, given traditional release cycles in IT, can typically take months! But that business wants and needs answers today. Waiting months to see new data in its queries adds to the friction between business and IT. This is solely the result of using relational systems to store this data. Conversely, because Java Objects can be saved as XML or JSON and because Word documents are essentially

ZIP files of XML parts, if we use a multi-model database, we can store the information all readily available for search and query without requiring additional requests to IT to update any processes or schemas. The effort to use these data sources in their natural form is minimized.

The Desired Solution

We've addressed mainframes, relational systems, and filesystems, as these are the primary sources of data we're grappling with in the organization today. There might be others, too, and with any of these data stores comes a considerable load of mental inertia and physical behaviors required for people to interact with and make use of their data across all these different systems.

A benefit of multi-model—and why we find more and more organizations embracing multi-model databases—is that even though for any one person there are multiple sources of data (Figure 1-1), with a multi-model database, we can implement solutions with one source of data that provides different lenses to many different consumers requiring different unified views across disparate data models and formats all contained within a single database, as demonstrated in Figure 1-3.

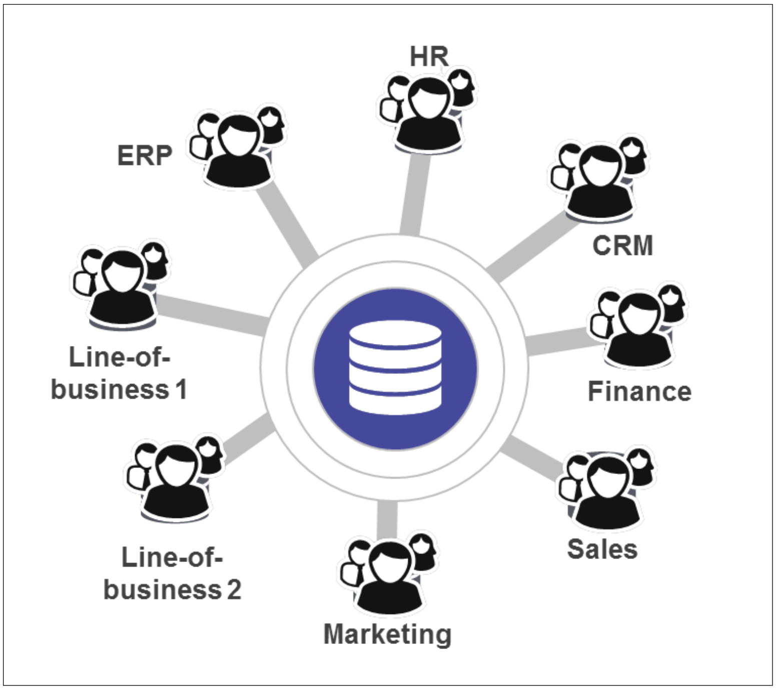


Figure 1-3. The goal: a multi-model database in action

Figure 1-3 illustrates the desired end state for many solutions. As the glue that binds disparate sources of information into a unified view, multi-model databases have emerged and are being used to create solutions in which data is aggregated quickly and can provide delivery to multiple consumers via multiple formats in the form of a unified view. In the following chapters, we'll dig deeper into how a multi-model database can get us to this desired end state quickly. We'll also cover how we work with data using a multi-model approach, and how this contrasts with a relational approach. We'll also examine the significant benefits realized in terms of the reduction in time and level of effort required to implement a solution in multi-model as well as the overall reduction of required ETL and data movement in these solutions.

CHAPTER 2

The Rise of NoSQL

In early 2009, Johan Oskarsson organized an event to discuss "open source distributed, nonrelational databases," during which he and a friend coined the term *NoSQL* (not SQL). The acronym attempted to label the emergence of an increasing number of nonrelational, distributed data stores, including open source clones of Google's BigTable/MapReduce and Amazon's Dynamo.

The rise of these systems can be attributed to the rise of big data; the volume, variety, and velocity of data in industries began rapidly increasing with the rise of the internet and the increase of power and proliferation of mobile and computing devices. Rows and columns alone weren't going to cut it anymore, and so new systems emerged to tackle some of the use cases for which traditional relational approaches were no longer a good fit for the data models or fast enough or scalable enough to meet the increased demand.

NoSQL technologies might not be databases in the traditional sense, meaning that many of them do not provide both transactional integrity and real-time results, and some of them provide neither. Each resulted from an effort to alleviate specific limitations found in the RDBMS world that were preventing their architects from completing a specific type of task, and they all made trade-offs to get there. Whether it was tackling "hot row" lock contention, horizontal scale, sparse-data performance problems, or single-schema induced rigidity, they are much more narrowly focused in purpose than their RDBMS predecessors. Many weren't designed to be enterprise platforms for building ACID transactional, real-time applications. But one of the core motivations was building them to scale.

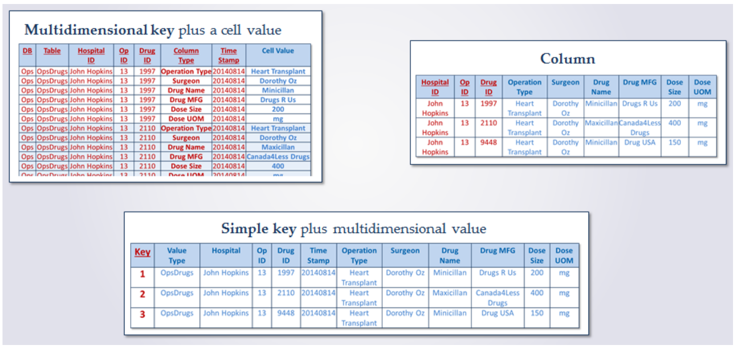
And although NoSQL suggested exclusion of SQL, the meaning transmogrified into "not only SQL," a clear indication that SQL can be included under the NoSQL banner. An exciting time to be sure. These new databases provided agility within their logical and physical data models, providing repositories that allowed for the rapid ingest and query of data in new formats and in new languages. This, in turn, provided organizations that had to this point been continually bogged down in an all-encompassing, traditional relational approach with opportunities for rapid application development. Under the banner of NoSQL, the following four major types of databases emerged, paving the way for the more encompassing type known as multi-model:

- Key-value
- Wide column
- Document
- Graph

A multi-model database is designed to support these and other data models against a single, integrated backend. With that in mind, let's begin by taking a closer look at some of the different models that may be supported by multi-model in isolation. It's all about using the right tool at the right time for the right job.

Key-Value

A key-value store is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn can have many different fields within them, each containing data. These records are stored and retrieved by using a key that uniquely identifies the record, and is used to quickly find the data within the database. Figure 2-1 presents some typical examples.


Multidimensional key plus a cell value

DB	Table	Hospital ID	Op ID	Drug ID	Column Type	Time Stamp	Cell Value
Ops	OpsDrugs	John Hopkins	13	1997	Operation Type	20140814	Heart Transplant
Ops	OpsDrugs	John Hopkins	13	1997	Surgeon	20140814	Dorothy Oz
Ops	OpsDrugs	John Hopkins	13	1997	Drug Name	20140814	Minicillan
Ops	OpsDrugs	John Hopkins	13	1997	Drug MFG	20140814	Drugs R Us
Ops	OpsDrugs	John Hopkins	13	1997	Dose Size	20140814	200
Ops	OpsDrugs	John Hopkins	13	1997	Dose UOM	20140814	mg
Ops	OpsDrugs	John Hopkins	13	2110	Operation Type	20140814	Heart Transplant
Ops	OpsDrugs	John Hopkins	13	2110	Surgeon	20140814	Dorothy Oz
Ops	OpsDrugs	John Hopkins	13	2110	Drug Name	20140814	Maxicillan
Ops	OpsDrugs	John Hopkins	13	2110	Drug MFG	20140814	Canada4Less Drugs
Ops	OpsDrugs	John Hopkins	13	2110	Dose Size	20140814	400
Ops	OpsDrugs	John Hopkins	13	2110	Dose UOM	20140814	mg

Column

Hospital ID	Op ID	Drug ID	Operation Type	Surgeon	Drug Name	Drug MFG	Dose Size	Dose UOM
John Hopkins	13	1997	Heart Transplant	Dorothy Oz	Minicillan	Drugs R Us	200	mg
John Hopkins	13	2110	Heart Transplant	Dorothy Oz	Maxicillan	Canada4Less Drugs	400	mg
John Hopkins	13	9448	Heart Transplant	Dorothy Oz	Minicillan	Drug USA	150	mg

Simple key plus multidimensional value

Key	Value Type	Hospital	Op ID	Drug ID	Time Stamp	Operation Type	Surgeon	Drug Name	Drug MFG	Dose Size	Dose UOM
1	OpsDrugs	John Hopkins	13	1997	20140814	Heart Transplant	Dorothy Oz	Minicillan	Drugs R Us	200	mg
2	OpsDrugs	John Hopkins	13	2110	20140814	Heart Transplant	Dorothy Oz	Maxicillan	Canada4Less Drugs	400	mg
3	OpsDrugs	John Hopkins	13	9448	20140814	Heart Transplant	Dorothy Oz	Minicillan	Drug USA	150	mg

Figure 2-1. Examples of key-value models

Key-value stores work great as long as all you care about querying is the keys. And you must know the keys to be able to find the associated object, because the object itself will be opaque to search and query.

For example, suppose that you've created an online service for users. Users can sign in to your service using their username. You can use their username as the key. You always have this in your application because the user provides it, and you then can use that key to quickly and easily look up the user's profile so that you can decide which information to serve. The user's profile information can be stored as text or binary, or in whatever format you want to save the value. This might work fine for you.

However, key-value stores are similar to traditional file cabinets with no cross-references. You file things one way—and that's it. If you want to find things in a different way, you need to pull all the objects out and look through them. Because of this, key-value stores can't do analytics. Analytic queries require looking across multiple records. This is why analytics on key-value stores are almost always done with the help of a separate (usually batch-oriented) technology.

When planning your use case, you might not think you need analytic queries for your application at first, but you almost always do. If your problem is to find an efficient and scalable way to store and retrieve user profiles, a key-value store might look ideal. But how long will it be until you want to understand which other users are similar to another user? Or to find all the users that share certain traits?

Wide Column/Key-Value

A wide column store is a type of key-value database. It adds a table-like layer and a SQL-like language on top of an internal multidimensional key-value engine. Unlike a relational database, the names and format of the columns can vary from row to row in the same table. We can view a wide-column store as a two-dimensional key-value store. Let's look at some of its pros and cons.

Pros:

- Fast puts and gets
- Massive scalability
- Easy to shard and replicate
- Data colocation
- Sparsely populated
- List, map, and set data types

Cons:

- Must carefully design key
- Hierarchical JSON or XML difficult to "shred" into flat columns
- Secondary indexes required to query outside of the hierarchical key
- No standard query API or language
- Must handcode all joins in app

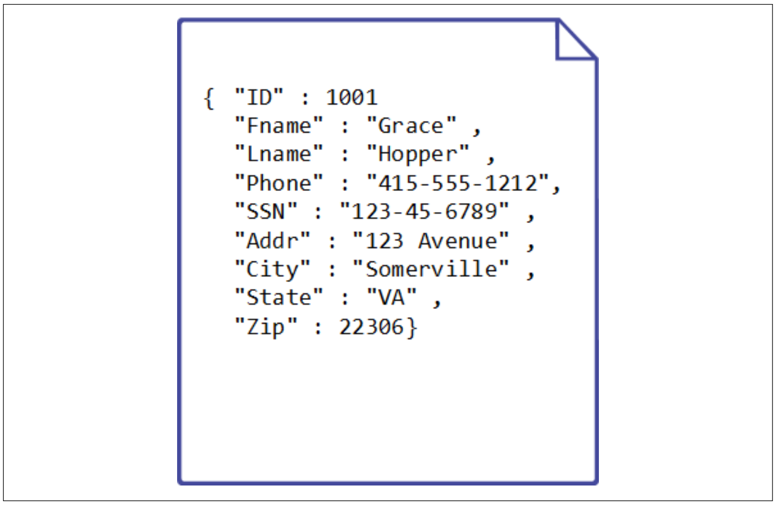
Document

A document database organizes data using self-describing, hierarchical formats such as JSON and XML. The document model often maps to business problems very naturally, and in some sense, it is a reaction to the relational model.

The main benefit of the document model is that it is human-oriented and not necessarily predetermined. In other words, all the data—no matter how long or how sparse—is stored in a document. Human beings have been using documents for a couple thousand years. For example, in literature, shipping invoices, insurance applications, and your local newspaper, there is a hierarchy. There are introductions, sections and subsections, and paragraphs and sentences, and clauses. Document models reflect how people think, whereas relational systems require people to think in terms of rows and columns.

For example, let's consider a user profile, similar to the one we discussed in our key-value example. It has a visible hierarchical structure. The profile has a summary and an experience section. Inside the experience, you probably have a number of jobs, and each job has dates. This hierarchy organizes data in a way that works for people. Just to highlight that it is a hierarchy, you can imagine the same thing as a mathematical tree structure. And if you serialize this user profile as JSON or XML, you will have a list of fields that include a name, a summary, and an experience. The interesting thing about this is that in today's hierarchical models, unlike the ones in the 1970s, everything is self-describing. Every one of these fields has a tag that indicates what it is.

Document stores can be used as key-value stores. As Figure 2-2 demonstrates, in the document model, the name of the document, often referred to as its ID or URI, is the key, and the document is the object being stored. This comes with the benefit of being able to search and query within the objects, unlike a key-value store. If you don't need to search and query within the document, this type of store might be overkill for a key-value use case. But in a document database, the text and structure are all queryable, so we can now do analytic queries across documents.



```
{ "ID" : 1001
  "Fname" : "Grace" ,
  "Lname" : "Hopper" ,
  "Phone" : "415-555-1212",
  "SSN" : "123-45-6789" ,
  "Addr" : "123 Avenue" ,
  "City" : "Somerville" ,
  "State" : "VA" ,
  "Zip" : 22306}
```

Figure 2-2. Example of a document model

Over time, the document model emerged as the most popular of the NoSQL data models to date. Documents are the most flexible of the models because they represent normal human communication. Most people think in terms of documents, be they Word documents, web forms, books, magazine articles, Java objects, messages, or emails. Information is exchanged within or between organizations in document format. This predates JSON and even XML. Electronic data interchange (EDI) is a good example. The record as document format is actually not new. Today, we can find documents everywhere, and the most prevalent types for storage are JSON and XML.

As we'll see, a multi-model database provides even more flexibility for deployment and use than document stores. Right now, let's review the pros and cons of the document model.

Pros:

- Fast development
- Schema-agnostic
- Natural for humans
- Data "de-normalized" into natural units for fast storage and retrieval without joins
- Queries everything in context

- Can query for relevance

Cons:

- Documents represent trees naturally, but not graphs. Cycles or rejoining relationships such as among people, customers, and providers cannot be captured purely in a document.
- Storage and update granularity. It's often cheaper to update one cell in a table than an entire document.

Graph

A graph database is a database that uses graph structures with nodes, edges, and properties to represent and store data (see Figure 2-4). A key concept of the system is the *graph* (or *edge* or *relationship*), which directly relates data items in the store. The relationships allow data in the store to be linked directly, and, in many cases, retrieved with a single operation. In this space, we see two major forms of graph store emerge: the *property store* and a *triple store*.

Property Graph Database

A property graph database has a more generalized structure than a triple store, using graph structures with nodes, edges, and properties to represent and store data. Property graph stores can store graphs of different types, such as undirected graphs, hypergraphs, and weighted graphs. Graph databases use proprietary languages and focus on paths and navigation from node to node, as illustrated in Figure 2-4, rather than generic queries of the graph.

Graph databases provide index-free adjacency, meaning that every element contains a direct pointer to its adjacent elements, and no index lookups are necessary. General graph databases that can store any graph are distinct from specialized graph databases such as triple stores. Property graphs are node-centric by design and allow simple and rapid retrieval of complex hierarchical structures that are difficult to model in relational systems.

Pros:

- Fast development of relationships
- Simple retrieval of hierarchical structures

- Quick and easy path traversal
- Great for path analytics (e.g., counts and aggregates)

Cons:

- No standards defined for storage and query
- No definition of semantics for edge relationships, so no ability to query the meaning of the graph
- Graphs stored in a silo, separate from the data from which it's created

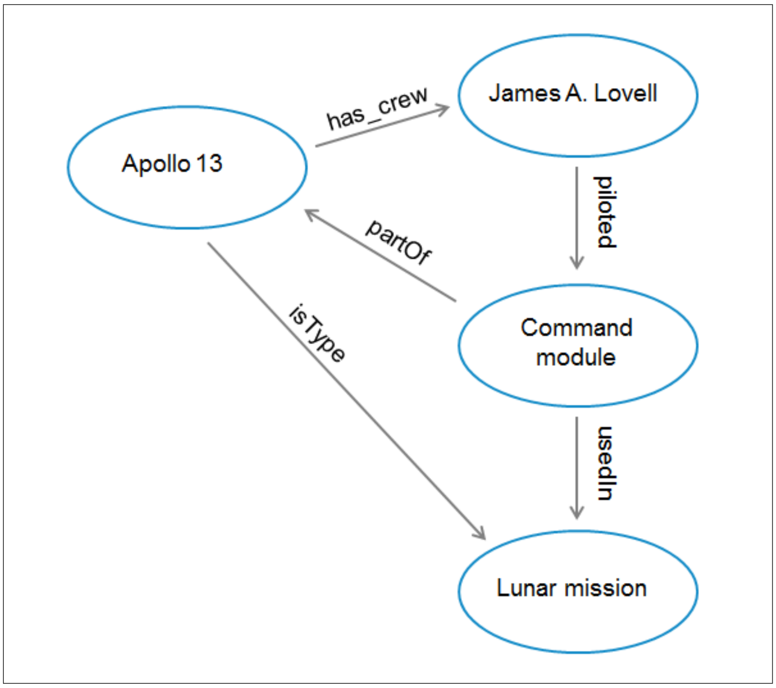


Figure 2-3. Example of data in a graph model

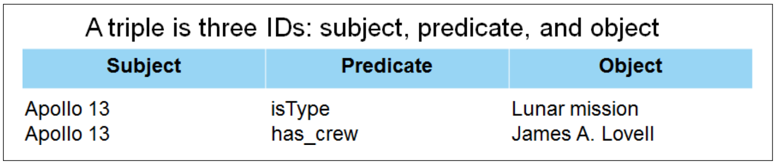
Triple Stores

Triple stores are a database used for defining and querying semantic graphs. In fact, the type of graph is a directed-label graph. Triple stores are edge-centric and based on the industry standard *Resource Description Framework* (RDF). RDF is designed for storing statements in the form of subject-predicate-object called triples (see

Figure 2-3). RDF triple stores use a list of edges, many of which are properties of a node and not critical to the graph structure itself.[1]

Triple stores query by using the WC3 standard SPARQL and can apply rules encoded in ontologies to derive new data from base facts. Triples represent the association between two entities, and the object of one triple can be the subject of another triple. They form a graph-like representation of data with nodes and edges that are without hierarchy and are machine readable—easy to share, easy to combine.

RDF also has a formal semantics which allows graph-matching queries. You can find all the matching graph patterns where a person knows a person named Bob, where Bob works for a company that is a subsidiary of a company named Hooli. This is typically done via the SPARQL query language.



A triple is three IDs: subject, predicate, and object

Subject	Predicate	Object
Apollo 13	isType	Lunar mission
Apollo 13	has_crew	James A. Lovell

Figure 2-4. Data in a triple

Pros:

- Unlimited flexibility—model any structure
- Runtime definition of types and relationships
- Relate an entity to anything in any way
- Query relationship patterns
- Use standard query language: SPARQL
- Create maximal context around data

Cons:

- Highly normalized
- Many joins required to reconstruct entities represented as triples

1 Davide Alocci et al., "Property Graph vs RDF Triple Store: A Comparison on Glycan Substructure Search", Plos One 10, no. 12 (2015), doi:10.1371/journal.pone.0144578.

- Many joins required for nontrivial query and projection

For more information on the various types of NoSQL, see Adam Fowler's excellent reference book *NoSQL for Dummies* (Wiley). He also tracks the NoSQL space and regularly self-publishes NoSQL update reports.

Multi-Model

A multi-model database is designed to support multiple data models against a single, integrated backend. Document, graph, relational, and key-value models are examples of data models that can be supported by a multi-model database.

The first time the word "multi-model" was associated with databases has been attributed to Luca Garulli in May 2012 during his keynote address "NoSQL Adoption—What's the Next Step?"[2] at the NoSQL Matters conference. Luca envisioned the evolution of first generation NoSQL products into new products with more features able to be used by multiple use cases. He had the foresight to articulate that a single, integrated NoSQL product to maintain and manage and develop for was much more beneficial than plumbing and kludging different, separate NoSQL systems together to provide a similar result. One product providing many faces on data can allow us to realize goals more quickly and consistently than stitching things together manually and then having to constantly maintain and update our code for those disparate resources while having to address each system's differences in abilities to scale and perform, as well as model, access, and secure data.

Table 2-1 shows the database types that some vendors in the multi-model space handle.

Table 2-1. The database types supported by multi-model vendors

DBMS	Relational model	Document model	Key-value store model	Wide-column model	Graph model
ArangoDB	N	Y	Y	N	Y
Couchbase	N	Y	Y	N	N

2 Luca Garulli, "Multi-Model storage 1/2 one product" (presented at the NoSQL Matters 2012 keynote).

DBMS	Relational model	Document model	Key-value store model	Wide-column model	Graph model
MarkLogic	N	Y	Y	N	Y
OrientDB	N	N	Y	N	Y
PostgreSQL	Y	Y	Y	N	N

Some of these systems existed long before Luca's presentation, and the level of support for the various systems noted in Table 2-1 varies widely, including the ability to query across models, fully index the internal structure of a model, transactional support (ACID/BASE), and optimization for query planning across models.

Multi-model databases can support different models either within a single engine or via layers on top of an engine. Underlying a layered architecture, each data model is supported by a separate component. Layers may abstract the underlying data store or even additional engines in support of multiple models.

Although native support for relational models isn't supported for some of the aforementioned databases, this really just means that you can't create tables and/or store a model in the traditional relational sense. Taken further, this translates to you not being able to do efficient indexed joins with relational data within these systems. However, a row in a relational table becomes a document in a document database, and although the document model allows you to use entities without normalizing them to fit a relational model, document indexing and APIs allow developers using systems such as Couchbase and MarkLogic to model views for relational consumers. You can then query documents in those systems using SQL. Couchbase provides SQL capabilities through its N1QL API, and MarkLogic provides SQL capabilities through standard SQL-92 SQL.

Keeping in mind all the models we've covered so far, let's revisit our multi-model database definition:

> *A database that supports multiple data models in their natural form within a single, integrated backend, and uses data standards and query standards appropriate to each model. Queries are extended or combined to provide seamless query across all the supported data models. Indexing, parsing, and processing standards appropriate to the data model are included in the core database product.*

To be a "real" multi-model DBMS then, these products must perform the following additional tasks for at least two different data models:

- Allow ingesting of data *as is* and storing data in its most natural form:
  - Documents
  - Business objects
  - Text
  - Graphs
  - Tables
  - Geospatial coordinate
- Provide storage in native form:
  - JSON
  - Text
  - RDF
  - XML
  - Binary
- Index all text and structure of information on ingest:
  - Search indexing should not be a bolt-on or separate process but fully integrated into the system and usable out of the box.
- Provide a simple unified interface to clients while querying data in native form:
  - Unified, standards-based APIs
- Use schemas where needed (relational data) and desired (latent schemas in document data, triples for semantic data, XSDs for XML data):
  - Schemas might not be required as a condition prior to loading data, but they are still important because consumers of a multi-model database will adhere to schemas.
- Maintain high performance for queries across all data models supporting operating/transactional use cases:
  - Unified database engine with unified indexes
- Make security easier by using one security model to cover everything rather than multiple, different security models for multiple data stores.

- Improve scalability and fault tolerance through caching and clustering of a single storage repository.

That's quite a list. But despite what you may have heard from most NoSQL vendors, there are multi-model databases that provide all these capabilities in a single product. MarkLogic Server is one such database.

Is Hadoop a Multi-Model Database?

No. Hadoop is a Java-based programming framework that supports the processing and storage of extremely large datasets in a distributed computing environment. Hadoop stores many types of data using a metadata layer and a filesystem layer. This does not meet our definition of a database, and Hadoop lacks tools to search, query, and modify the data without considerable assembly. Hadoop distributions are provided as a suite of tools, loosely integrated, with considerable effort required to assemble, depending on your use case. Consequently, people use a variety of tools to achieve those goals, and the subsequent proliferation of technologies in the Hadoop ecosystem can create levels of complexity and silos of data that mirror traditional enterprise architectures.

Readings in Database Systems, aka The Red Book, provides an up-to-date compilation of the latest papers in DBMS applications reviewed by database experts Peter Bailis, Joseph Hellerstein, and Michael Stonebreaker. In the fifth edition, the authors discuss many of the limitations of Hadoop, along with the state of databases and their purpose today.[3] It is a very interesting read. In the Red Book, the authors point out that Hadoop is load *as is*, as you can load anything to a filesystem. Developers can then decide what to do with the data later. But the insight from the Red Book is that Hadoop essentially allows you to create a traditional data warehouse on inexpensive storage from a library of modular data warehouse parts. These parts, or course, require assembly by programmers. As a result, although data agility and heterogeneity provided by Hadoop Distributed File System (HDFS) are of interest, Hadoop's utility with regard to use as a database or for any distributed applications remains to be established.

3 Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker, *Readings in Database Systems*, 5th Edition (Cambridge: MIT Press, 2015).

Unfortunately, Hadoop arrived with much hype and was put under the NoSQL umbrella due to some of the components it ships with. However, Hadoop is not a database and not a multi-model database. It is a platform that has some utility with analytics and observe-the-business type functions at scale, for which requirements for real-time transactions are not a factor.

With regard to data integration, Hadoop helped solve the data ingest problem, given that a uniform schema is not required to load data to a filesystem; so in this respect, it allows you to add your schema later. But a tremendous amount of effort and skill is needed to transform the data and assemble systems on top of the data to make it useful. Multi-model databases can actually provide an excellent complement to Hadoop because they have the ability to rapidly pull in data from HDFS and make it actionable with much less effort than using what comes with Hadoop.

Can NoSQL Be ACID?

Yes. NoSQL systems provide the ability to load data quickly and scale out massively, and when applied to the correct use case can be very powerful tools. They are not without some limitations though. Used incorrectly, they can solve a short-term problem successfully while adding another data silo to the overall architecture over the long term. Vendors attempting to define this rapidly emerging space unfortunately spread some misinformation as well.

Many of the new NoSQL solutions were designed for distributed computing and sacrificed the four basic tenets of RDBMS known by the acronym *ACID*. ACID stands for atomicity, consistency, isolation, and durability. ACID defines a set of properties that will guarantee reliability in the world of database transactions. Let's take a closer look at these properties:

Atomicity
: This means that if you have multiple statements in a transaction, all parts of the transaction need to be successful; if any one of them fails, the entire transaction will also fail.

Consistency
: A guarantee that the database will go from one valid state into another when a transaction commits.

Isolation
: If you execute your transactions concurrently, you can generally treat those transactions as being unaware of each other, as if they are being executed serially.[4]

Durability
: If a transaction commits (inserts data into the database, updates a document, or deletes something), those changes are going to persist, even in the case of a system failure.

ACID is what allows a database to guarantee that users are looking at consistent and correct data.

One alternative to ACID is *BASE*: basic availability, soft-state, and eventual consistency.

Many NoSQL technologies use the BASE model. Out of this was born an incorrect notion that NoSQL systems could not be transactional—a must for enterprise computing. This suggested that NoSQL databases had to be used offline or in research, observe-the-business capacities. To run the business on these systems, additional application logic would be required to ensure transactional consistency and that no data is lost. NoSQL and ACID are, however, orthogonal. One does not cause the other. FoundationDB (before it was acquired by Apple),[5] MarkLogic, and Neo4J all have implemented ACID-supporting NoSQL database systems. In his MarkLogic World 2013 keynote address, CEO Gary Bloom explained that MarkLogic was built to be ACID from the ground up by its founder, Christopher Lindblad, as part of being enterprise-ready.[6] This spawned the term *Enterprise NoSQL*.

NOTE

Some systems will claim ACID, but ACID across one document is not the same as ACID across the dataset. Likewise, *eventual consistency* and *strong consistency* are not the same as *formal consistency*.

4 David Gorbet, "I is for Isolation, That's Good Enough for Me!" MarkLogic.com, January 19, 2016.

5 Shira Ovide, "Apple Acquires FoundationDB", the *Wall Street Journal*, March 24, 2015.

6 "MarkLogic World 2013 Keynote: Gary Bloom", YouTube video, posted by MarkLogic, April 9, 2013.

With Enterprise NoSQL databases capable of performing many of the essential functions of relational databases, and with an exponential rise in the volume and variety of data being stored, the popularity of NoSQL has exploded since 2009.

The Modern DBMS

You can find a great primer to the motivations and goals for all these types of NoSQL systems in a paper "MAD Skills: New Analysis Practices for Big Data," presented at a scholarly database conference, VLDB 09.[7] The insights and goals for the system the authors describe still resonate and are incredibly relevant today. They argue that with data acquisition and storage becoming increasingly affordable, a new type of system would be required to empower data analysts to examine and use data outside of the traditional enterprise data warehouse and business intelligence (BI) tools commonly used. They focused explicitly on data warehouses, but it's easy to see the relevance to operational and transactional systems as well. The authors stated that a modern system capable of keeping up with the increasing pace of data would need to be *magnetic*, *agile*, and *deep*. Let's examine these further:

Magnetic
: Traditional relational systems tend to repel new data sources. If information doesn't fit a preexisting relational schema, it's dropped on the floor. A considerable amount of effort is required to load data into the warehouse for use by BI tools, with design, cleansing, and preparation required before data is even loaded. Given the pace of data, a system should be magnetic in that it allows you to load all data despite its format, model, or data quality niceties.

Agile
: Data warehousing orthodoxy is based on long-range, careful design and planning. Given the growing numbers of data sources and increasingly sophisticated and mission-critical data analyses, a modern system must instead allow analysts to easily ingest, digest, produce, and adapt data at a rapid pace. This requires a database that's physical and logical contents can be in

7 Jeffrey Cohen et al., "Mad Skills: New Analysis Practices for Big Data" (presented by Ritwika Ghosh at VLDB 2009, November 10, 2015).

continuous rapid evolution. And even though the data models can be iterated upon, how those models are accessed must also be flexible. While SQL is the sole point of entry for relational query, an agile system would allow you to interact with models from a variety of languages and methods.

Deep

Due to system constraints on storage and compute, analysts often work with a subset, or sample of the data. Analysts still might bring subsets of data into a single-client, memory-bound application for analysis such as R or Excel. A deep system would allow users to store everything, and analyze everything, without limiting data to scale, size, or scope. The analysis should be able to be pushed to the database itself. The database should serve as a runtime engine for algorithm analysis.

To be clear, multi-model was not born out of the MAD Skills paper, but the paper certainly provided some keen observations on the challenges and complexity of working with data in silos and certainly inspired many in the NoSQL movement and likely some of the multi-model databases we see emerging today. There was a year or so during which you couldn't attend a NoSQL meetup without hearing someone reference this paper. I've always appreciated the requirements it set for modern data management systems. Therefore, I include magnetic, agile, and deep here as good standards for us to evaluate multi-model systems against.

The multi-model database has emerged as a present-day modern DBMS to let us load information *as is* (magnetic), iterate on the physical and logical models of the data in place through a variety of methods and languages (agile), and scale out horizontally to store information at scale and perform analysis at scale (deep).

We know the problem: our data is in silos. We see a solution: a multi-model database. We understand some of the history and characteristics of a multi-model database. For our solution, we'll aim to use a multi-model database that is hardened, provides ACID capabilities, security, and consistency for multiple models of data within a single engine. We'll next take a look at how we work to integrate data in a multi-model database. This will help to explain some of the preceding concepts and illustrate why people are so excited to learn more about multi-model databases and how they can use them.

CHAPTER 3

A Multi-Model Database for Data Integration

So, you've decided to use a multi-model database for data integration. What can you expect? How should it live in your infrastructure? Well, let's take a look.

Your data has already been modeled. Very smart people in your organization modeled it to meet the requirements of various applications and business users. The data is just in silos. To integrate that data, you don't want to remodel all the data to fit some new model. You want to take advantage of what you've already modeled *as is* and integrate it quickly to deliver results in a unified view.

Source systems for your integration might not go away. The data integration project is not necessarily a rip-and-replace solution. Some systems exist for very good reasons and will continue to exist; we just can't use their presently contained data in isolation to achieve meaningful business results. So, if we need to integrate a silo's data, a multi-model database can provide us an operational data hub and/or act as the glue for unifying various sources of data into a unified and consolidated layer. Let's see how this plays out by stepping through some simple examples.

Entities and Relationships

Suppose that we have silos of customers, transactions, and products that we'd like to integrate into a unified view. Conceptually, the entities we care about and their relationships might look like Figure 3-1.

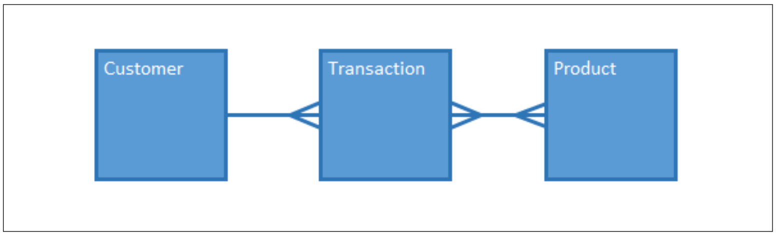


Figure 3-1. Entities and relationships

A customer can have many transactions; this is a one-to-many relationship. Transactions can contain many products; this is a many-to-many relationship. But we know the devil is in the details. For each system of tables that models these entities and their relationships, we're going to have to deal with potentially very different schemas. So starting with a set of tables from System A, we might see a model similar to that shown in Figure 3-2.

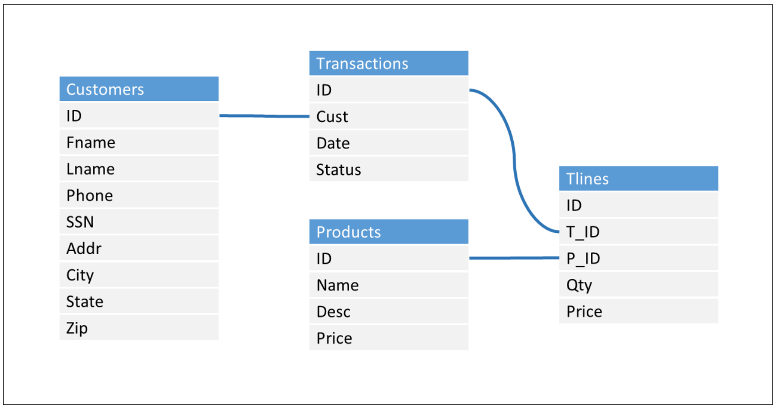


Figure 3-2. Sample customer data from System A

Yes, this example is overly simplistic, but it still helps make our point. Relational schemas are not self-documenting. In Figure 3-2, we see a set of attributes, with cryptic naming conventions, and we can make some assumptions: but when we look at a field such as `Addr`, we're not really sure what's in this field. There are attributes for

`City`, `State`, and `Zip`, so maybe `Addr` is the street address? And is this the address for shipping or billing? Well, suppose that we want to integrate this set of tables with a set of tables from a System B that contains the same type of information, but in a model that is shaped a little differently, as depicted in Figure 3-3.

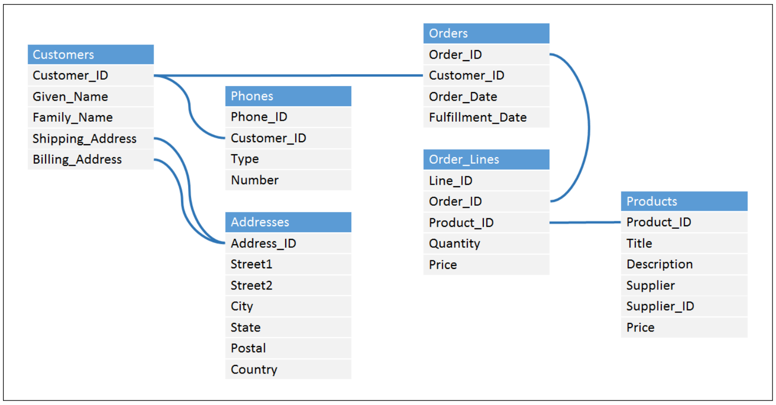


Figure 3-3. Sample customer data from System B

In Figure 3-3, we see multiple addresses and phone numbers per customer. Just like that, our integration task is quickly becoming more complex. We also see that attributes with similar semantic meaning have different labels. A customer ID in system A is `ID`, whereas in System B, it's labeled `Customer_ID`. System A contains a `Zip` field; System B labels it `Postal`. We also see the introduction of new attributes and data from System B such as `Supplier_ID`; what does this map to?

If we were going to integrate this data using another relational system, we'd need to understand all these attribute names, their meanings, relationships, and mappings to other systems before we could come up with a schema to superset our source schemas. The mappings between systems are baked in to the application code, PL/SQL procedures, and the subject matter experts' heads. Aggregating that information is very time-consuming, and designing the schema to superset all schemas also requires a considerable amount of time and effort. Although these examples are naive and simple, we know that the number of systems we'll need to integrate and their varying models are going to present us with much more complexity, *as is* demonstrated in Figure 3-4.

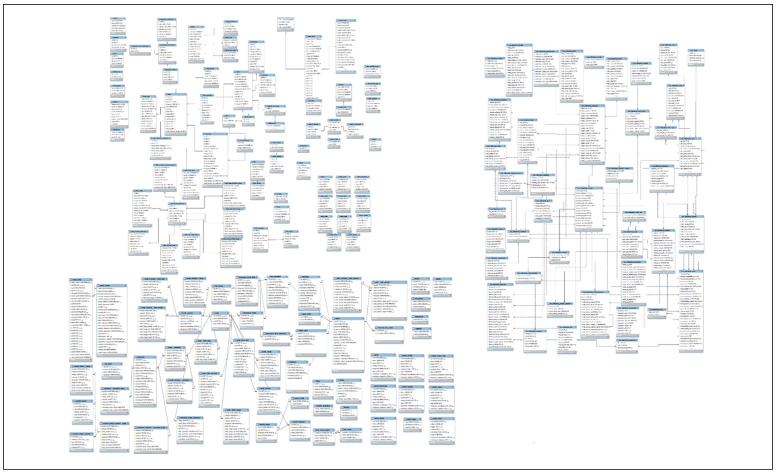


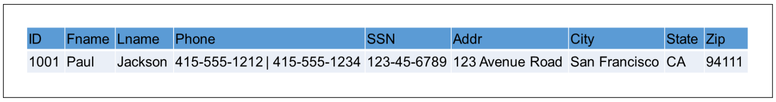
Figure 3-4. The reality of the challenge is greater than we anticipate

Figure 3-4 shows three sets of tables from three different systems. However, it's not uncommon for organizations to have a requirement to integrate hundreds of different schemas from hundreds of different systems!

This is the crux of the problem when integrating using a relational approach: you can't fit differently shaped things into the same database and query across them. You must work all of the data into a unified shape. With this approach, schemas never become simpler; they only grow more complex. In practice, to ease the burden and time cost of data integration, you find that some data is left behind. We might not include `Supplier_ID` and its associated data if we don't think we will need it at the time for the integration project in front of us. Leaving data on the floor often comes back to haunt us, though, requiring updates to our schema and to the ETL that will feed our super schema. Though we might spend considerable time and effort to create the superset schema, what we've unintentionally created is just another data silo.

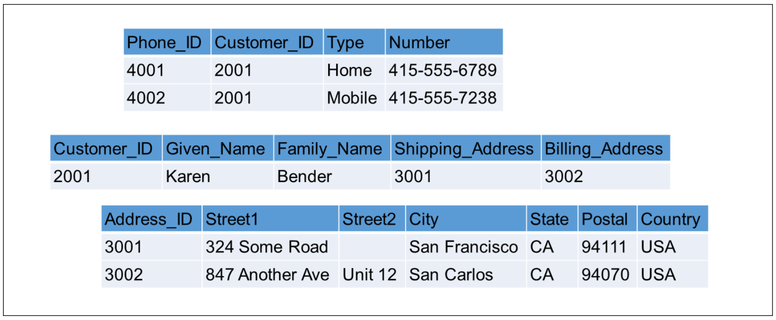
So how do we go about using our customers, transactions, and products in a multi-model database?

Let's begin with just customers. Figure 3-5 shows the shape of a customer from System A, and Figure 3-6 shows the shape of a customer from System B.



ID	Fname	Lname	Phone	SSN	Addr	City	State	Zip
1001	Paul	Jackson	415-555-1212 | 415-555-1234	123-45-6789	123 Avenue Road	San Francisco	CA	94111

Figure 3-5. Customer from System A



Phone_ID	Customer_ID	Type	Number
4001	2001	Home	415-555-6789
4002	2001	Mobile	415-555-7238

Customer_ID	Given_Name	Family_Name	Shipping_Address	Billing_Address
2001	Karen	Bender	3001	3002

Address_ID	Street1	Street2	City	State	Postal	Country
3001	324 Some Road		San Francisco	CA	94111	USA
3002	847 Another Ave	Unit 12	San Carlos	CA	94070	USA

Figure 3-6. Customer from System B

A multi-model database separates data ingest from data organization and therefore allows us to load information *as is*. This flexibility is great for being able to load data quickly, but we don't want to recreate a relational problem in a NoSQL system, and we want to be cautious to not create yet another data silo. To aid us here, a multi-model database also allows us to model entities as documents.

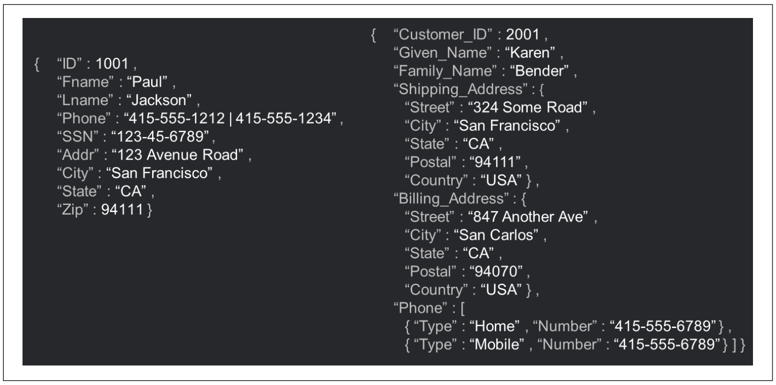
The first thing we do when loading data into a multi-model database is identify the entities. This sounds complicated, but it's not. An entity is any object in the system that we want to model and whose information we want to store. Entities are usually recognizable concepts, such persons, places, things, or events, that have relevance to our business and the data we've stored in a database. Entities to us might look like customers, orders, products, invoices, invoice lines, patients, locations, accounts, and so on. The entities are already defined; they're just normalized in third normal form (3NF) across many tables in the source systems.

If our multi-model database acts as a document database, we can easily store the entities as documents in our database. We denormalize the relational rows to create entity documents. We can think of this as a no-op transform when migrating from relational. We're storing the same data and the same entities from the relational system in a multi-model system, just in a different shape. We can carry out the denormalization to create the entities (documents) prior to loading into our multi-model database or after loading.

A multi-model database is *schema-agnostic*, meaning that a schema does not need to be defined prior to loading the data. We can literally load the data *as is*.

A multi-model database is also *schema-aware*, meaning that as soon as the data is loaded, it is immediately available for us to query it, use it, and continue to model it in place.

A multi-model database provides agility with regard to the options we have for formats in which we can persist our data. A multi-model database allows us to save our data as XML, JSON, text, or binary. If we took these two denormalized customer records and loaded them into a multi-model database as JSON, we'd expect them to look similar to Figure 3-7.



```
{   "ID" : 1001 ,
    "Fname" : "Paul" ,
    "Lname" : "Jackson" ,
    "Phone" : "415-555-1212 | 415-555-1234" ,
    "SSN" : "123-45-6789" ,
    "Addr" : "123 Avenue Road" ,
    "City" : "San Francisco" ,
    "State" : "CA" ,
    "Zip" : 94111 }
```

```
{   "Customer_ID" : 2001 ,
    "Given_Name" : "Karen" ,
    "Family_Name" : "Bender" ,
    "Shipping_Address" : {
      "Street" : "324 Some Road" ,
      "City" : "San Francisco" ,
      "State" : "CA" ,
      "Postal" : "94111" ,
      "Country" : "USA" } ,
    "Billing_Address" : {
      "Street" : "847 Another Ave" ,
      "City" : "San Carlos" ,
      "State" : "CA" ,
      "Postal" : "94070" ,
      "Country" : "USA" } ,
    "Phone" : [
      { "Type" : "Home" , "Number" : "415-555-6789" } ,
      { "Type" : "Mobile" , "Number" : "415-555-6789" } ] }
```

Figure 3-7. Customers from System A and System B as documents

A multi-model database should index all structure and text upon import of data. For text, we expect all XML *element* values or JSON *property values* to be indexed. For structure, we expect all XML elements and all JSON properties and any of their parent-child relationships to be indexed. By indexing text and structure on ingest, immediately upon inserting our documents into a database we can perform a structured query such as:

```
Give me all documents where Fname equals "Paul"
```

or:

```
Give me the Customer document where Customer_ID equals 2001
```

You then would expect to get the appropriate document back for each query.

When we don't know the structure of the document, full-text search also becomes useful. With search, we also can ask:

```
Show me the documents that contain the phrase "San Francisco"
```

and we'd expect to see the record for Paul Jackson.

Thus, in a multi-model database, we load *as is*, for some definition of *as is*, where we really want to think in terms of loading entities. In multi-model, we model entities and their relationships. So far in this example, we have two very different shaped customer entities, and because of index of text and structure, we can search and query these records. But what if we want to ask the following question:

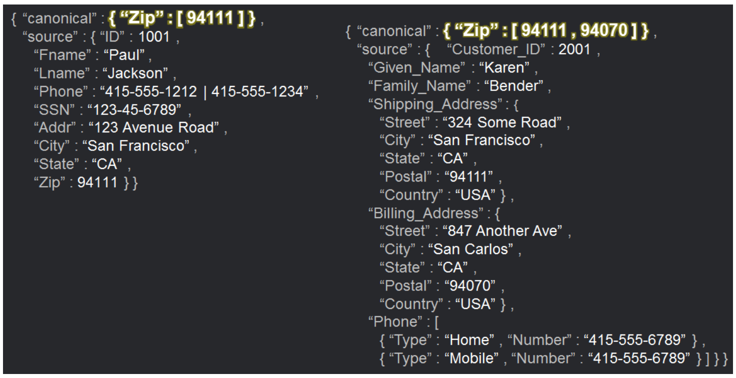
```
What are all the entities (documents) having a Zip equal
to 94111
```

Well, we'd get only Paul's record. Karen actually has an address at the same zip code, but her attribute is defined as "Postal." We could search on 94111 and potentially get both records back, along with any other document that might have 94111 in it. (What if there is a product in our database with that same sequence of numbers for its product code?) We'll want to do something to map these properties into some alignment. In a multi-model system, this is easy to do.

The Envelope Pattern

In multi-model databases, a simple yet powerful pattern has emerged that MarkLogic calls the *envelope pattern*.

Here's how it works: take your source entity, and make it the subdocument of a parent document. This parent document is the "envelope" for your source data. As a sibling to your source data within the envelope, you add a header section where you start to add your standardized attribute names, attribute values, and attribute structure. In our example, we've standardized the "Zip" and "Postal" attributes as "Zip" properties in both envelopes (see Figure 3-8). As a result, we can now issue a structured query of the type "Show me all customers having a zip code equal to 94111" and get both entities back.



```
{ "canonical" : { "Zip" : [ 94111 ] } ,
  "source" : { "ID" : 1001 ,
    "Fname" : "Paul" ,
    "Lname" : "Jackson" ,
    "Phone" : "415-555-1212 | 415-555-1234" ,
    "SSN" : "123-45-6789" ,
    "Addr" : "123 Avenue Road" ,
    "City" : "San Francisco" ,
    "State" : "CA" ,
    "Zip" : 94111 } }
```

```
{ "canonical" : { "Zip" : [ 94111 , 94070 ] } ,
  "source" : {   "Customer_ID" : 2001 ,
    "Given_Name" : "Karen" ,
    "Family_Name" : "Bender" ,
    "Shipping_Address" : {
      "Street" : "324 Some Road" ,
      "City" : "San Francisco" ,
      "State" : "CA" ,
      "Postal" : "94111" ,
      "Country" : "USA" } ,
    "Billing_Address" : {
      "Street" : "847 Another Ave" ,
      "City" : "San Carlos" ,
      "State" : "CA" ,
      "Postal" : "94070" ,
      "Country" : "USA" } ,
    "Phone" : [
      { "Type" : "Home" , "Number" : "415-555-6789" } ,
      { "Type" : "Mobile" , "Number" : "415-555-6789" } ] } }
```

Figure 3-8. The envelope pattern in action

The benefits to this approach are numerous:

- We retain our source entity *as is*. This is a big benefit for highly regulated environments that come in and request to see the entities as they were originally imported into our database.
- We standardize only what we need, when we need it. We don't require all the mappings up front. Now we iteratively manage our data in place.
- We can continue to update our model later without reingesting.
- We can divide and conquer the standardization process across development.

MarkLogic refers to this process of standardizing only the attributes we need, as we need them, as *harmonization*. In this light, harmonizing entities includes wrapping entities in envelopes and placing their standardized field names and values in the envelope, as a sibling to the source data.

Enterprise-wide data models as a notion failed to take hold in the 1990s due to the complexity of dealing with tools that only allow you to work schema-first. The task at hand to identify all the attributes from all the source systems and develop an integration schema to contain everything identified repeatedly proved itself an insurmountable task. But, by working through this process of harmonization in a multi-model system that allows us to add our own schema later, as a sidecar to the source data in its original format, we can begin to iterate to an enterprise-wide model bottom up. You'll frequently hear this general, standardized enterprise-wide model referred to as the *canonical model*.

Keep in mind that although we can load data without defining a schema first, schemas are still important. We're going to integrate data and then deliver results to consumers that adhere to schemas. Validation against business rules is still important and useful, but not necessarily required for getting started with our data. We can load and use our information *as is*. We can envelope our data and harmonize it to get entities in some alignment for digestion by consumers. We can perform harmonization in place, after we've loaded the data. Likewise, we can choose to validate and enforce schemas after we've loaded the data. If we're using XML, MarkLogic supports validating documents by *.xsd*. In a multi-model database, we often do schema validation when delivering results requested by consumer systems, but not necessarily before.

The multi-model approach to data integration

Silos of data often have silos of people working above them. To integrate silos in the relational approach, you need to get all the developers, analysts, and business stakeholders from all the silos in the room and get them to agree on the uber-schema before you begin to load it. In a multi-model database, different groups can standardize properties for their particular division, and both "canonical" models can be present in the envelope simultaneously. Different groups query the part of the document they require to feed their apps: but by meeting regularly, these groups can make comparisons and merge similarities so that over time they can iterate to a single canonical model for the enterprise derived from across all entities in the database. This iterative, divide-and-conquer approach has demonstrated time and again to complete integrations in weeks instead of months, and months instead of years.

Let's contrast how we work with data in multi-model for data integration (Figure 3-9) with how we work with it using the relational tools in Chapter 1. (Refer to Figure 1-2 to see the relational approach.)

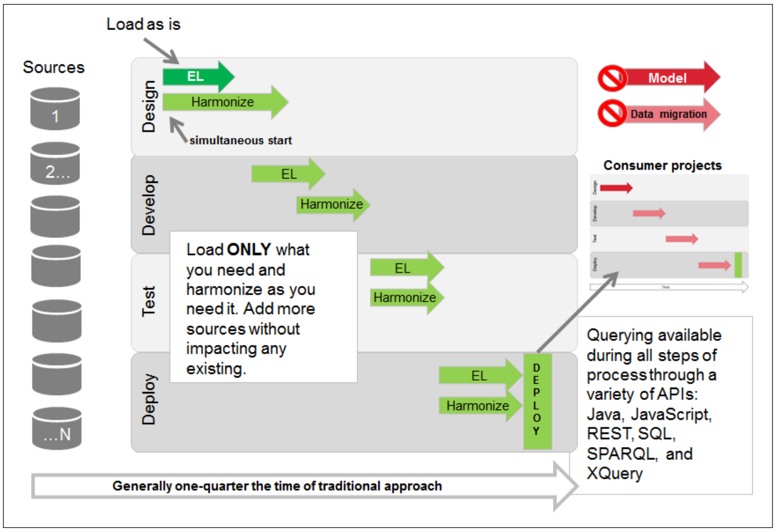

Load as is
Sources
1
2…
…N
Design
EL
Harmonize
simultaneous start
Develop
EL
Harmonize
Test
EL
Harmonize
Deploy
EL
Harmonize
DEPLOY
Load ONLY what you need and harmonize as you need it. Add more sources without impacting any existing.
Model
Data migration
Consumer projects
Querying available during all steps of process through a variety of APIs: Java, JavaScript, REST, SQL, SPARQL, and XQuery
Generally one-quarter the time of traditional approach

Figure 3-9. Integrating data by using a multi-model approach

Here are some of the benefits of the multi-model approach:

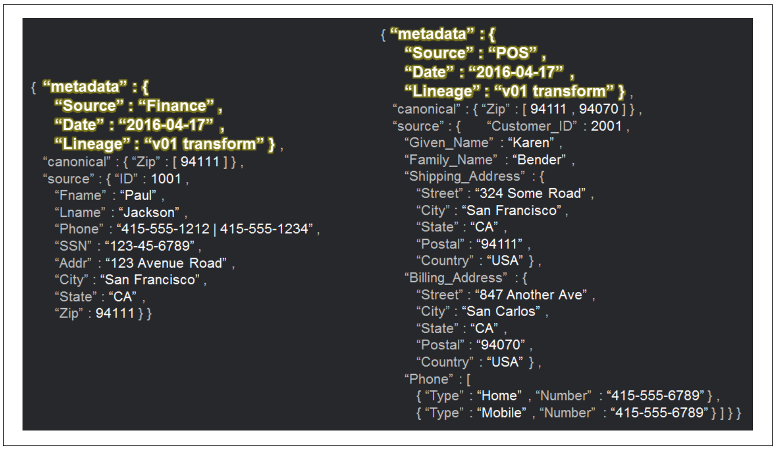
- Eliminates upfront modeling requirement.
- Eliminates data migrations and store mappings and dependencies with the data.
- Querying and data access available during all steps of the process through a variety of APIs: Java, JavaScript, REST, SQL, SPARQL, and XQuery.
- Consumer projects using existing harmonizations are not affected by changes.
- Multiple versions of data can exist within the database at the same time. An envelope can persist multiple headers, allowing teams to work in parallel.

When we load *as is*, we've eliminated the time for upfront modeling. We eliminate the "T" from ETL, lowering the time and cost for getting data into the system. As data is indexed on insert, we now can begin to assess and standardize/harmonize the data in place. We can load data and begin to harmonize simultaneously. This is a direct result of separating data organization from data ingest in multi-model. We don't need to load all the sources; we can begin with just one or as many as we like. And because a multi-model database allows us to store all of the metadata and mappings and different versions and shapes and schemas of the data all together, we eliminate data migration. Now we're working with agility in a cycle of loading and harmonizing, iterating to deployment. Deployment doesn't require that everything be mapped or completed before something useful is deployed, either. And for consuming applications that use existing harmonizations, they don't need to change until they want or need to. We can achieve data integration victory much quicker and with less risk by using a multi-model database.

In fact, complex data integrations done using a multi-model database, such as MarkLogic, are four times faster than when done using a relational approach. Completion of these projects includes not only integrating the data from source systems but also delivering results in multiple formats to multiple consumers.

Adding more information to envelopes

The envelope pattern is incredibly useful in that it allows us to also store metadata, such as provenance, lineage, and any other meaningful information that we'd like, in the envelope (see Figure 3-10). With a true multi-model database, we can reshape the data and add an envelope with additional data and have that additional structure and data immediately available for search and query. This is the benefit of a multi-model database being schema-agnostic and schema-aware.



```
{ "metadata" : {
    "Source" : "Finance" ,
    "Date" : "2016-04-17" ,
    "Lineage" : "v01 transform" } ,
  "canonical" : { "Zip" : [ 94111 ] } ,
  "source" : { "ID" : 1001 ,
    "Fname" : "Paul" ,
    "Lname" : "Jackson" ,
    "Phone" : "415-555-1212 | 415-555-1234" ,
    "SSN" : "123-45-6789" ,
    "Addr" : "123 Avenue Road" ,
    "City" : "San Francisco" ,
    "State" : "CA" ,
    "Zip" : 94111 } }
```

```
{ "metadata" : {
    "Source" : "POS" ,
    "Date" : "2016-04-17" ,
    "Lineage" : "v01 transform" } ,
  "canonical" : { "Zip" : [ 94111 , 94070 ] } ,
  "source" : {      "Customer_ID" : 2001 ,
    "Given_Name" : "Karen" ,
    "Family_Name" : "Bender" ,
    "Shipping_Address" : {
      "Street" : "324 Some Road" ,
      "City" : "San Francisco" ,
      "State" : "CA" ,
      "Postal" : "94111" ,
      "Country" : "USA" } ,
    "Billing_Address" : {
      "Street" : "847 Another Ave" ,
      "City" : "San Carlos" ,
      "State" : "CA" ,
      "Postal" : "94070" ,
      "Country" : "USA" } ,
    "Phone" : [
      { "Type" : "Home" , "Number" : "415-555-6789" } ,
      { "Type" : "Mobile" , "Number" : "415-555-6789" } ] } }
```

Figure 3-10. Adding more information to the envelope

Here are some additional benefits:

- Preserve and query lineage, provenance, and other metadata.
- New schemas don't impact existing schemas or applications.

Adding a completely differently shaped entity to our database—perhaps a social media feed—would have no impact to our standardized, enveloped entities at all. It becomes another entity we can search and query *as is*, or another candidate for us to envelope and standardize with our existing entities.

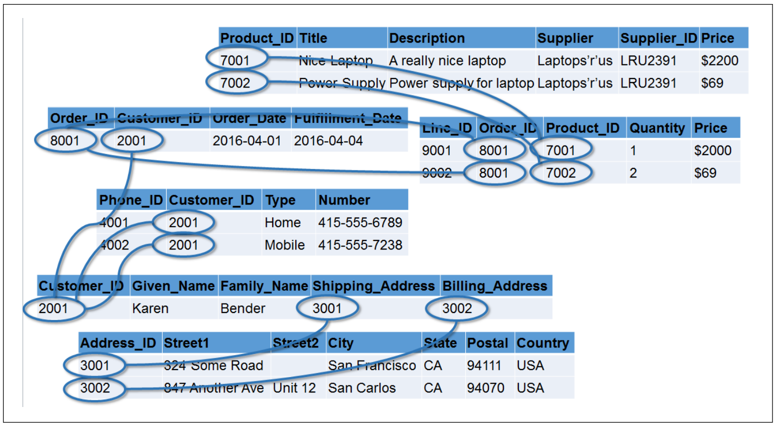
As stated previously, in a multi-model database, we think of entities as documents. Where do entities come from? They come from the entities we've already modeled and saved in our existing source systems. Previously, they lived in rows across several tables. We denormalize to create the entities by topics of interest for our data integration applications. If we care about customers, we create customer entities. If we want to know about orders for these customers, we create order entities, and so on. We store the entities as documents.

But in multi-model, and in data integration, we want to begin by keeping the data as close to its original source form as possible. By doing this, we always have a snapshot, which we could normalize if we so wish, of what the data looked like when it was received. We wrap our entity in an envelope document and add any standardization of element/property names, element/property values, element/property structure, metadata, lineage, provenance, and anything else we might want to capture along in the envelope sibling to the source.

A closer look at modeling relationships

Now, let's take it another step: what about relationships among our entities?

If we look at a given set of tables for our customers, transactions, and products, we see a slew of relationships occurring, as depicted in Figure 3-11.



Product_ID	Title	Description	Supplier	Supplier_ID	Price
7001	Nice Laptop	A really nice laptop	Laptops'r'us	LRU2391	$2200
7002	Power Supply	Power supply for laptop	Laptops'r'us	LRU2391	$69

Order_ID	Customer_ID	Order_Date	Fulfillment_Date
8001	2001	2016-04-01	2016-04-04

Line_ID	Order_ID	Product_ID	Quantity	Price
9001	8001	7001	1	$2000
9002	8001	7002	2	$69

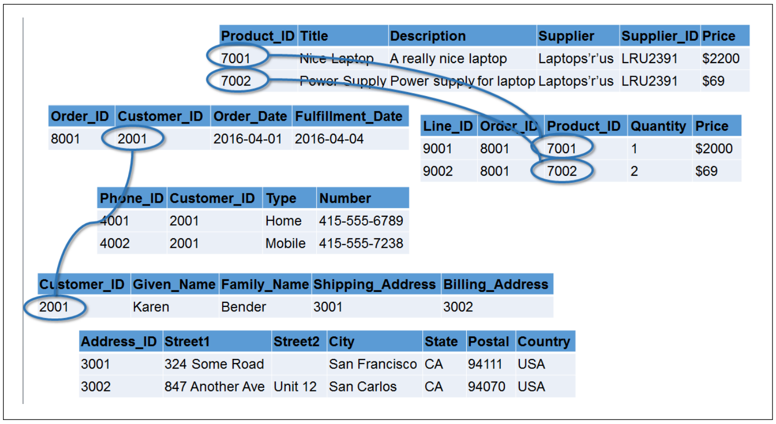
Phone_ID	Customer_ID	Type	Number
4001	2001	Home	415-555-6789
4002	2001	Mobile	415-555-7238

Customer_ID	Given_Name	Family_Name	Shipping_Address	Billing_Address
2001	Karen	Bender	3001	3002

Address_ID	Street1	Street2	City	State	Postal	Country
3001	324 Some Road		San Francisco	CA	94111	USA
3002	847 Another Ave	Unit 12	San Carlos	CA	94070	USA

Figure 3-11. Only business entity concept relationships are meaningful; others are spurious

But many of these relationships are misleading. If we strip away the relationships maintained as an artifact of modeling data in 3NF and just leave the relationships that represent actual business context, the relationships we care most about are actually fewer, as highlighted in Figure 3-12.



Product_ID	Title	Description	Supplier	Supplier_ID	Price
7001	Nice Laptop	A really nice laptop	Laptops'r'us	LRU2391	$2200
7002	Power Supply	Power supply for laptop	Laptops'r'us	LRU2391	$69

Order_ID	Customer_ID	Order_Date	Fulfillment_Date
8001	2001	2016-04-01	2016-04-04

Line_ID	Order_ID	Product_ID	Quantity	Price
9001	8001	7001	1	$2000
9002	8001	7002	2	$69

Phone_ID	Customer_ID	Type	Number
4001	2001	Home	415-555-6789
4002	2001	Mobile	415-555-7238

Customer_ID	Given_Name	Family_Name	Shipping_Address	Billing_Address
2001	Karen	Bender	3001	3002

Address_ID	Street1	Street2	City	State	Postal	Country
3001	324 Some Road		San Francisco	CA	94111	USA
3002	847 Another Ave	Unit 12	San Carlos	CA	94070	USA

Figure 3-12. Capture the correct relationships

Ironically, relational databases are bad at modeling relationships. The meanings of these relationships aren't in our data or in the schema, they are often found in application code. In this data, a customer placed an order for some product. We don't necessarily know if the customer has actually purchased the items in the order; we just know there is relationship between the customer and the order. We have no idea what the meaning of the relationship is unless we're provided further information.

Let's look again at our entities now modeled as documents. In a multi-model database, our conceptual and physical models are much more closely aligned, as demonstrated in Figure 3-13.

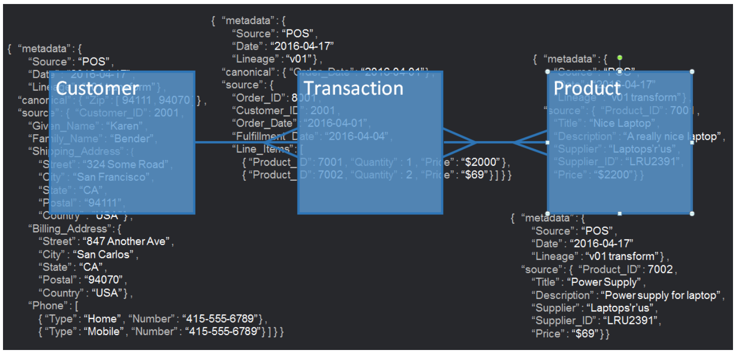
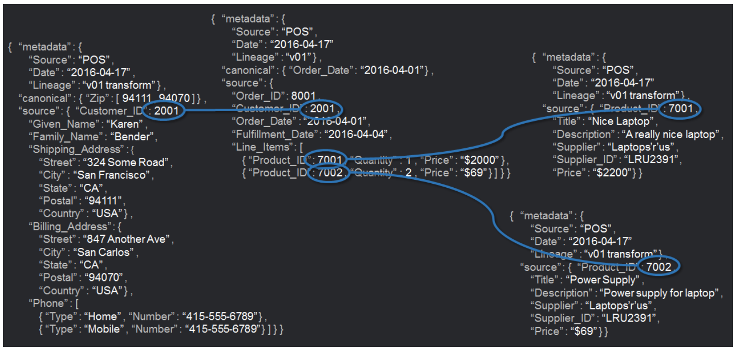


Figure 3-13. Another way to look at entities are relationships; documents are entities

And when we look closely at the entities, we can still see the available joins. In the example JSON documents (Figure 3-14), we can rely on similar property names and values, either in our harmonized headers or our enveloped source documents, to perform joins for us.



```
{ "metadata": {
    "Source": "POS",
    "Date": "2016-04-17",
    "Lineage": "v01 transform"},
  "canonical": { "Zip": [ 94111, 94070 ] },
  "source": { "Customer_ID": 2001,
    "Given_Name": "Karen",
    "Family_Name": "Bender",
    "Shipping_Address": {
      "Street": "324 Some Road",
      "City": "San Francisco",
      "State": "CA",
      "Postal": "94111",
      "Country": "USA"},
    "Billing_Address": {
      "Street": "847 Another Ave",
      "City": "San Carlos",
      "State": "CA",
      "Postal": "94070",
      "Country": "USA"},
    "Phone": [
      { "Type": "Home", "Number": "415-555-6789"},
      { "Type": "Mobile", "Number": "415-555-6789"} ] } }
```

```
{ "metadata": {
    "Source": "POS",
    "Date": "2016-04-17"
    "Lineage": "v01"},
  "canonical": { "Order_Date": "2016-04-01"},
  "source": {
    "Order_ID": 8001,
    "Customer_ID": 2001,
    "Order_Date": "2016-04-01",
    "Fulfillment_Date": "2016-04-04",
    "Line_Items": [
      { "Product_ID": 7001, "Quantity": 1, "Price": "$2000"},
      { "Product_ID": 7002, "Quantity": 2, "Price": "$69"} ] } }
```

```
{ "metadata": {
    "Source": "POS",
    "Date": "2016-04-17"
    "Lineage": "v01 transform"},
  "source": { "Product_ID": 7001,
    "Title": "Nice Laptop",
    "Description": "A really nice laptop",
    "Supplier": "Laptops'r'us",
    "Supplier_ID": "LRU2391",
    "Price": "$2200"} }
```

```
{ "metadata": {
    "Source": "POS",
    "Date": "2016-04-17"
    "Lineage": "v01 transform"},
  "source": { "Product_ID": 7002,
    "Title": "Power Supply",
    "Description": "Power supply for laptop",
    "Supplier": "Laptops'r'us",
    "Supplier_ID": "LRU2391",
    "Price": "$69"} }
```

Figure 3-14. Foreign key joins still possible, but in multi-model there's a better way

But with a multi-model database, we can do so much better. We can provide rich meaning and utility to these relationships using *semantics*.

Semantics

Semantic technologies refer to a family of specific W3C standards that facilitate the exchange of information about relationships in data, in machine-readable form. A triple statement contains a subject, predicate, and an object, as shown in Figure 3-15.

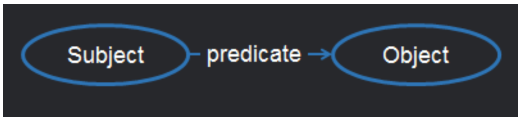


Figure 3-15. The definition of "a triple" in semantics

If this is completely new to you, the simplest way to think about these three pieces of data is as two things and the relationship between them. In terms of a graph, we can think of them as a node (subject), an edge (predicate), and a node (object). The object of one triple can be the subject for another. Triples provide an elegant way to model joins that represent business concepts, as demonstrated in Figure 3-16.

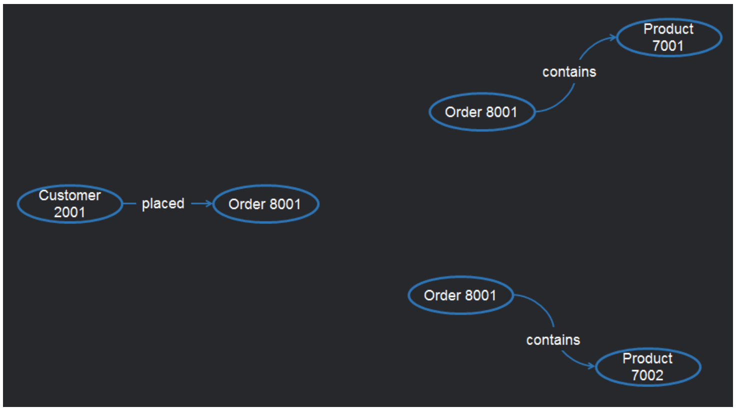
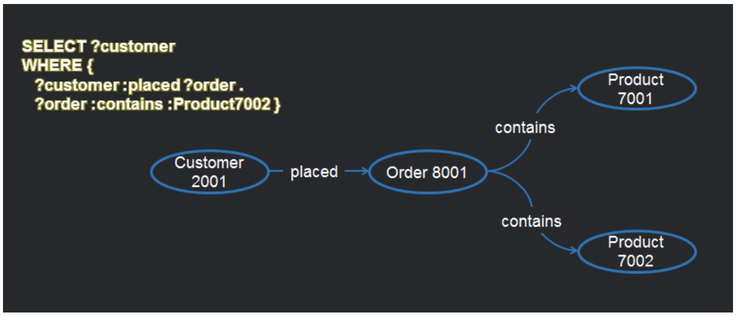


Figure 3-16. Relationships are triples

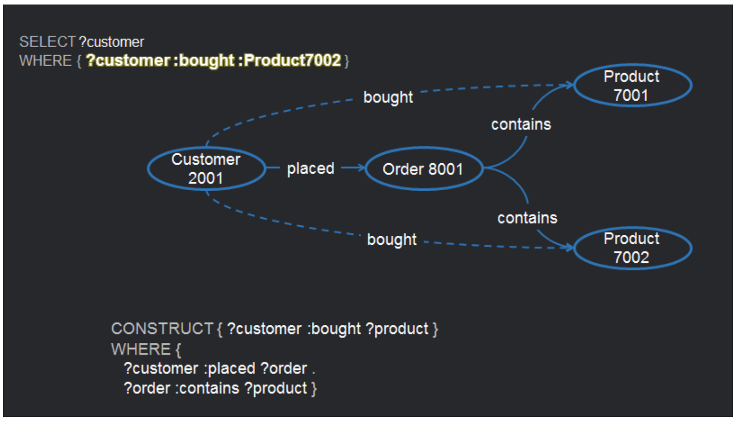
By representing data and their relationships in this way, you can assemble the triples into a graph and query it by using SPARQL (a recursive acronym naming convention which stand for SPARQL Protocol and RDF Query Language), which you can see in Figure 3-17.



```
SELECT ?customer
WHERE {
  ?customer :placed ?order .
  ?order :contains :Product7002 }
```

Figure 3-17. A SPARQL query (in the upper-left corner)

Something else triple stores allow you to do is to create rules and perform inference on these relationships. By doing this, we can assign rich meaning to the relationships within the graph. (See Figure 3-18). We can create new relationships and extract new facts by inferring meaning from existing relationships. Thus, if a person places an order, and an order contains a given product, we might know that the customer bought the product. We can create a rule to tell us that. We can then use this new fact about our data in subsequent queries.

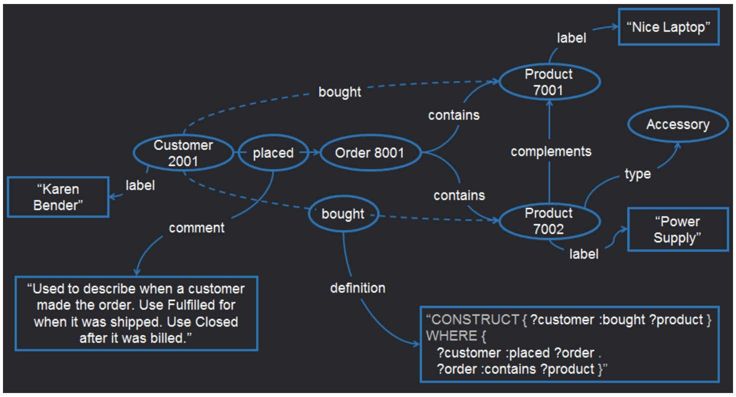


```
SELECT ?customer
WHERE { ?customer :bought :Product7002 }
```

```
CONSTRUCT { ?customer :bought ?product }
WHERE {
  ?customer :placed ?order .
  ?order :contains ?product }
```

Figure 3-18. The edges of this graph show that a customer has bought a specific product

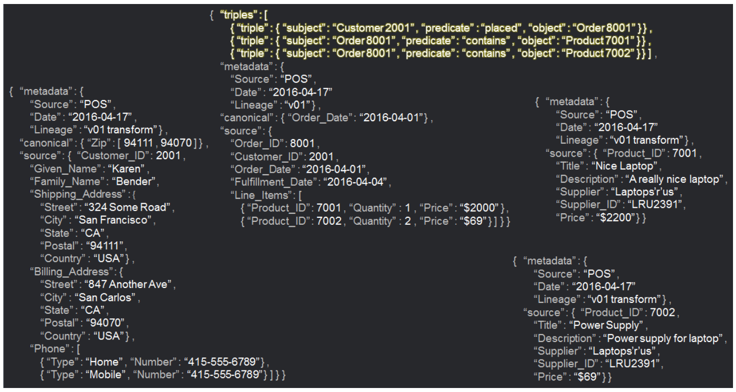
And it gets even richer. We can use triples to relate entities with other entities, but we can also use them to relate entities with concepts, and concepts with other concepts. We can use machine code or text annotations to enrich our graphs to create these relationships. This makes it possible for us to do things like find and group “like” customers or suggest “like” or “complementary” products to customers. (See Figure 3-19.)



```
"CONSTRUCT { ?customer :bought ?product }
WHERE {
  ?customer :placed ?order .
  ?order :contains ?product }"
```

Figure 3-19. Using machine code and text annotations to enrich relationships with triples

And in a multi-model database, we can store these triples *outside* of our entities, within their own documents, or by using the envelope pattern, to store them *with* our entities as well. (See Figure 3-20.)



```
{ “metadata” : {
    “Source” : “POS” ,
    “Date” : “2016-04-17” ,
    “Lineage” : “v01 transform” } ,
  “canonical” : { “Zip” : [ 94111 , 94070 ] } ,
  “source” : { “Customer_ID” : 2001 ,
    “Given_Name” : “Karen” ,
    “Family_Name” : “Bender” ,
    “Shipping_Address” : {
      “Street” : “324 Some Road” ,
      “City” : “San Francisco” ,
      “State” : “CA” ,
      “Postal” : “94111” ,
      “Country” : “USA” } ,
    “Billing_Address” : {
      “Street” : “847 Another Ave” ,
      “City” : “San Carlos” ,
      “State” : “CA” ,
      “Postal” : “94070” ,
      “Country” : “USA” } ,
    “Phone” : [
      { “Type” : “Home” , “Number” : “415-555-6789” } ,
      { “Type” : “Mobile” , “Number” : “415-555-6789” } ] } }
```

```
{ “triples” : [
    { “triple” : { “subject” : “Customer 2001” , “predicate” : “placed” , “object” : “Order 8001” } } ,
    { “triple” : { “subject” : “Order 8001” , “predicate” : “contains” , “object” : “Product 7001” } } ,
    { “triple” : { “subject” : “Order 8001” , “predicate” : “contains” , “object” : “Product 7002” } } ] ,
  “metadata” : {
    “Source” : “POS” ,
    “Date” : “2016-04-17”
    “Lineage” : “v01” } ,
  “canonical” : { “Order_Date” : “2016-04-01” } ,
  “source” : {
    “Order_ID” : 8001 ,
    “Customer_ID” : 2001 ,
    “Order_Date” : “2016-04-01” ,
    “Fulfillment_Date” : “2016-04-04” ,
    “Line_Items” : [
      { “Product_ID” : 7001 , “Quantity” : 1 , “Price” : “$2000” } ,
      { “Product_ID” : 7002 , “Quantity” : 2 , “Price” : “$69” } ] } }
```

```
{ “metadata” : {
    “Source” : “POS” ,
    “Date” : “2016-04-17”
    “Lineage” : “v01 transform” } ,
  “source” : { “Product_ID” : 7001 ,
    “Title” : “Nice Laptop” ,
    “Description” : “A really nice laptop” ,
    “Supplier” : “Laptops’r’us” ,
    “Supplier_ID” : “LRU2391” ,
    “Price” : “$2200” } }
```

```
{ “metadata” : {
    “Source” : “POS” ,
    “Date” : “2016-04-17”
    “Lineage” : “v01 transform” } ,
  “source” : { “Product_ID” : 7002 ,
    “Title” : “Power Supply” ,
    “Description” : “Power supply for laptop” ,
    “Supplier” : “Laptops’r’us” ,
    “Supplier_ID” : “LRU2391” ,
    “Price” : “$69” } }
```

Figure 3-20. We can store triples in envelopes

True multi-model systems allow us to query triples via SPARQL, search and query entire documents using APIs (JavaScript here for JSON), and even combine queries together!

How this is entirely modeled is beyond the scope of this book. But likely you'll have some set of triples stored within your enveloped entities, and another set of triples external to the entities. A multi-model database allows you to store the different data models together *in their native formats* and compose a query across them through a single API.

In our previous examples, we illustrated storing our data as entities (documents) derived from relational data, but the same rules can apply for other data sources. If we're importing mainframe copybook or *.vsam* files, as well as *.csv*, Excel documents, text files, and JSON and XML feeds, we load the information *as is*. It's not so much about the existing physical layout of the record, because that will change over time anyway, but we want to avoid remodeling any of our data as a condition of loading. We want to begin with our data in a shape as close to the existing data model from the source system as possible. We can summarize the process as follows:

1. We identify the entities in our source data, create the entities, and wrap the source entities in envelope documents, standardizing only the fields we require for structured queries within the envelope as we need them.
2. We use triples to define relationships among our entities.
3. We can use the name of the document we create for an entity as a key; the document itself is a value container.

In a multi-model database, we can store our entities and relationships as either XML or JSON, so for binaries, such as mainframe files or Microsoft Office documents, we'll want to convert the binary to XML or JSON so that we have something available for search and query. Open source libraries are available for this, and a multi-model database might come with built-in utilities to provide this capability as well. But what about other document types? We take a closer look at those in Chapter 4.

CHAPTER 4

Documents and Text

Up until this point, we've looked at multi-model through the lens of data integration with a focus on integrating data from relational systems. This is frequently where data integration begins, but there is tremendous value in being able to search structured data with unstructured and semi-structured data such as documents and text. Unfortunately, most people who hear documents and text in relation to a database automatically think of Binary Large OBjects (BLOBs) or Character Large OBjects (CLOBs) or the amount of shredding required to get those documents and text to fit a relational schema. But a multi-model database allows us to load them and use them *as is* because their text and structure are self-describing.

When we asked more than 200 IT professionals what percentage of their data was not relational, 44% said 1 to 25% of their data was unstructured. That is a shockingly low volume of unstructured data —for a surprisingly high number of organizations. Although it is possible that these organizations really do have a paucity of data that isn't relational, it is more likely that most companies aren't dealing with their unstructured data because it is simply too inconvenient to do so.

The ubiquity of relational databases has meant that, for many, only that which fits in a relational database management systems (RDBMS)—billing, payroll, customer, and inventory—is considered data. But this view of data is changing. Analysts estimate that more than 80% of data being created and stored is unstructured. This includes documentation, instant messages, email, customer communications, contracts, product information, and website tracking data.

So why does the survey indicate that *at most* 25% of an organization's data is unstructured? Possibly because data that is unstructured (or even semi-structured) is rarely dealt with by IT management.

That doesn't make unstructured data low-value. In fact, the inverse is true. Instead, it means this highly valuable content is *unused*.

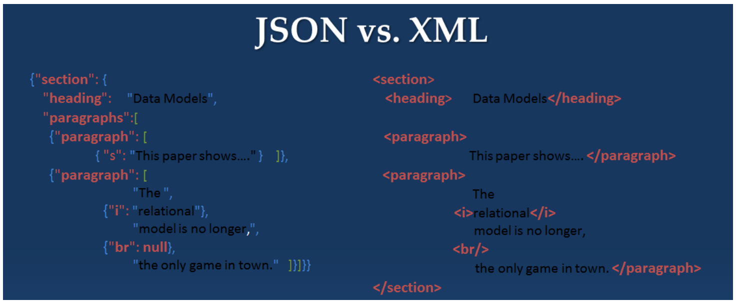
But there are options for modeling, storing, and querying unstructured data that give us insight into the 80% of the data that, for many, has remained unexplored. Further, any multi-model database that includes a structured model, such as JSON or XML, with text indexing provides the ability to query structured and unstructured data together.

Schemas Are Key to Querying Documents

XML and JSON documents are self-describing. The schema is already defined within the elements and properties already written within each document. Multi-model systems can identify this implicit schema when data is imported into the database and allow you to immediately query it.

Schemas create a consistent way to describe and query your data. As we noted earlier, in a relational database, the schema is defined in terms of tables, columns, and attributes; each table has one or more columns, and each column has one or more attributes. Different rows in a relational database table have the same schema—period. This makes the schema static, with only slow, painful, and costly changes possible. Relational systems are schema-first, in that we have to define the schema before pouring in the data. Changing the shape of the schema after data has been poured in can be painfully challenging and costly.

In a nonrelational database, the idea of a schema is more fluid. Some NoSQL databases have a concept of a schema that is similar to the relational picture, but most do not. Of those that do not, you can further divide them into databases for which the schema is *latent*, that is implicit, in the structure of each semi-structured or unstructured entity. When working with data from existing sources, your data has already been modeled. Data usually comes to us with some shape to it. When we load XML or JSON documents into a multi-model database, the latent or implicit schema is already defined within the XML elements or JSON properties of the documents, as demonstrated in Figure 4-1.


JSON vs. XML

```
{"section": {
  "heading":   "Data Models",
  "paragraphs":[
   {"paragraph": [
          { "s": "This paper shows...." }   ]},
   {"paragraph": [
                "The ",
          {"i": "relational"},
                "model is no longer,",
          {"br": null},
                "the only game in town."   ]}]}}
```

```
<section>
  <heading>    Data Models</heading>

  <paragraph>
              This paper shows.... </paragraph>
  <paragraph>
              The
          <i>relational</i>
              model is no longer,
          <br/>
              the only game in town. </paragraph>
</section>
```

Figure 4-1. Same document saved in two different formats

If the primary data entity is a document, frequently represented as an XML or JSON object, the latent schema implicit in a document might be different than that of another document in the same NoSQL database or database collection. That is to say, one document might have a title and author, whereas another has a headline and writer. Each of those two documents can be in the same NoSQL database because the schema is included in the documents themselves. In other words, in document stores, the schema is organized in each document, but it's not defined externally as it would be in a relational database. The latent schema is the shape of the data when you loaded it. In NoSQL databases, the schema for any document could, and often does, change frequently.

A true multi-model database is *schema-agnostic* and *schema-aware* so that you are not forced to cast your queries in a predefined and unchanging schema.

This is perfect for managing and querying text, and for querying text with structured data. This gives you a semantic view of varied databases and one place where you can look up every fact that you have. In that case, a multi-model database handles (at least) a document data model as well as semantic RDF data.

In the preceding documents, a multi-model database will index the JSON properties and XML elements as well as their values. In this way, the system is schema-aware, in that element and property values can be used for structured queries such as this one:

```
Show me the documents where heading equals "Data Models"
```

We also can use them as full-text search queries such as the following:

```
Show me any documents with the word "relational" in them
anywhere in the document.
```

Combining full-text search and structured query, we should be able to perform a search query of the following type:

```
Search for documents with the word "relational" anywhere in a
paragraph element/property.
```

A multi-model database management system must let you load data with multiple schemas. You shouldn't need to be concerned about lengths nor worry about cardinality of elements/properties. And, if something unexpected shows up tomorrow, you can still store it just fine.

Document-Store Approach to Search

In a document-store, documents are akin to rows, although a contract will make a lot more sense stored as a document versus stored as a row. Now, without knowing anything else about the document, you would be able to do a Boolean and full-text search against it. This approach sounds familiar if you have used Apache Lucene or Apache SOLR.

Flexible Indexing

The beauty of indexing in a document store is that it also provides for structured search, indexing the inherent structure of the document in addition to the text and values within the document. That is to say, documents have plenty of structure, including titles, section headings, book chapters, headers, footers, contract IDs, line items, addresses, and subheadings, and we can describe that structure by using JSON properties or XML elements. A multi-model database creates a generalized index of the JSON and XML values for that structure and indexes every value and every parent-child relationship that has been defined. In other words, the document model operating in a multi-model database, where anything can be added to any record (and is immediately queryable in a structured way) is far more powerful and flexible than the related-table model of a relational database for storing rich representations of entities. It also allows us to store, manage, and query relationships between those entities.

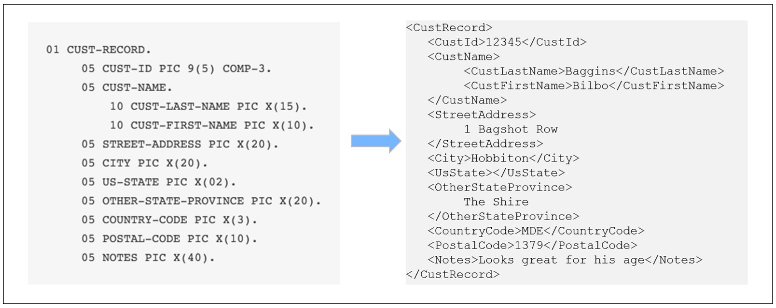
With a multi-model database, we can also extend the generalized document text and structure index with *special purpose* indexes such as range, geospatial, bitemporal, and triples.

Mapping (Mainframe) Data into Documents

Let's take a moment to look at documents that might come to us from mainframe systems. COBOL copybooks and Virtual Storage Access Method (VSAM) can actually be a very good fit for the document model.

COBOL is an acronym for the Common Business-Oriented Language. It's an older computer language designed for business use and found in many legacy mainframe applications. Data in these systems is found in copybook format, which is used to define data elements that can be referenced by COBOL programs. Due to the declining popularity of mainframes and the retirement of experienced COBOL programmers, COBOL-based programs are being migrated to new platforms, rewritten in modern languages, or replaced with software packages; and their associated data is being moved to more modern repositories such as NoSQL and multi-model databases.

Copybooks capture data resembling objects, or entities, and as a result are often a more natural fit for a document store than relational. You can transform a copybook object into an XML or JSON document in a pretty straightforward manner, as illustrated in Figure 4-2. There are open source libraries available on the web to help do this.

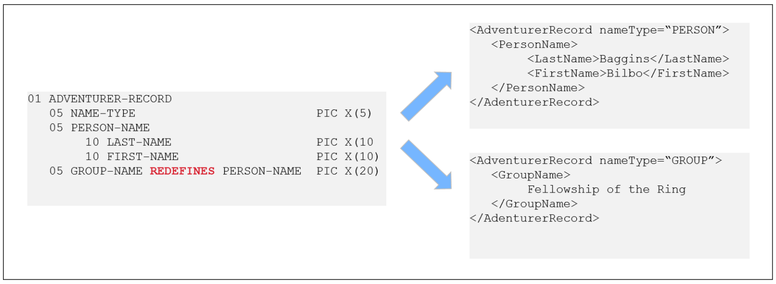


```
01 CUST-RECORD.
    05 CUST-ID PIC 9(5) COMP-3.
    05 CUST-NAME.
        10 CUST-LAST-NAME PIC X(15).
        10 CUST-FIRST-NAME PIC X(10).
    05 STREET-ADDRESS PIC X(20).
    05 CITY PIC X(20).
    05 US-STATE PIC X(02).
    05 OTHER-STATE-PROVINCE PIC X(20).
    05 COUNTRY-CODE PIC X(3).
    05 POSTAL-CODE PIC X(10).
    05 NOTES PIC X(40).
```

```
<CustRecord>
   <CustId>12345</CustId>
   <CustName>
        <CustLastName>Baggins</CustLastName>
        <CustFirstName>Bilbo</CustFirstName>
   </CustName>
   <StreetAddress>
        1 Bagshot Row
   </StreetAddress>
   <City>Hobbiton</City>
   <UsState></UsState>
   <OtherStateProvince>
        The Shire
   </OtherStateProvince>
   <CountryCode>MDE</CountryCode>
   <PostalCode>1379</PostalCode>
   <Notes>Looks great for his age</Notes>
</CustRecord>
```

Figure 4-2. COBOL copybook as XML

Additionally, the `REDEFINES` clause in COBOL creates a data polymorphism, something else that isn't simple or easy to capture in a relational system. Using entities and relationships in a multi-model system, however, this becomes a much more straightforward transformation, as shown in Figure 4-3.



```
01 ADVENTURER-RECORD
   05 NAME-TYPE                          PIC X(5)
   05 PERSON-NAME
        10 LAST-NAME                     PIC X(10
        10 FIRST-NAME                    PIC X(10)
   05 GROUP-NAME REDEFINES PERSON-NAME   PIC X(20)
```

```
<AdventurerRecord nameType="PERSON">
   <PersonName>
        <LastName>Baggins</LastName>
        <FirstName>Bilbo</FirstName>
   </PersonName>
</AdenturerRecord>
```

```
<AdventurerRecord nameType="GROUP">
   <GroupName>
        Fellowship of the Ring
   </GroupName>
</AdenturerRecord>
```

Figure 4-3. COBOL REDFINES clause simpler to manage as XML

Along with COBOL, we might find VSAM files along our mainframe data integration path. These are generally exported from their source mainframe as fixed-width text files. You'll receive the data in one set of files, and then a specification file that lets you know the offsets for the data for each row in a data file (characters 1–10 are the `customer_id`, 11–21 are `first_name`, 22–34 are `last_name`, etc.). These are generally parsed using Java and saved in XML or JSON within a multi-model or document database.

We mention this here because, again, in using multi-model for data integration, we'll be concerned with capturing entities and relationships. These are often already defined within their source systems, and those source systems might be mainframes from which we import COBOL copybooks and VSAM files. On their own, these formats are binary, so processing into a document format for search and reuse will be required. But after you've done that, they will be immediately available for search and reuse along with all your other data in a multi-model database. And this is what we see organizations are doing today. Many large organizations modernizing their architectures to replace and remove legacy mainframes are bypassing relational databases and jumping straight to NoSQL and multi-model ones.

Indexing Relationships

As previously mentioned, a multi-model database can use triples to represent relationships between entities. As a reminder, a triple is called a triple because it consists of three things: a subject, a predicate, and an object. In other words, it models two "things" and the relationship between them. This creates very granular facts.

Unlike primary/foreign key relationships in a relational database, these facts have meaning because we can infer new data based on the semantic meaning inherent in the facts we already had. In other words, triples don't just need to relate entities to other entities; they can relate entities to concepts, and even concepts to other concepts.

We can even store a data dictionary or ontology right in the database along with the data, so the meaning of the relationships can be preserved in both human- and machine-readable form.

To scale, a multi-model database should automatically index relationships between entities so that they can be joined, queried, combined with other triples, and used to build inferences.

If all indexes know the same document ID, we can compose queries against this single integrated index.

The net result is multiple data models, but one integrated index.

NOTE

Take care when dealing with vendors who have implemented their multi-model solutions by stitching together multiple disparate products (a multiproduct, multi-model database). Each product has its own queries and indexes, and there exists the potential for inconsistencies between data. Each purpose-built data store requires an API into each repository. On the development side, the burden is on the coder to write code that queries each data store, and then splices and joins the two separate results together in an application tier. More lines of code equal more points of failure in an application and more risk. Further, this type of query can never be performant; it's a join that can be performed only one way.

Scaling multiproduct, multi-model systems will also introduce more complexity, which can be difficult to overcome. Different systems require different indexes and have different scalability requirements. If developers write code to bring together a document store with a search engine and a triple store and call this multi-model, a lot of orchestration and infrastructure work must be implemented at the beginning of the project, and the system will only ever be as fast as the slowest application. All that orchestration code will also now require continued maintenance by your development staff.

A multi-model product that's delivered as a single application to include support for documents, graphs, and search will scale more effectively and perform better than a multiproduct, multi-model approach every time. Because the focus on plumbing disparate components isn't required, with a true multi-model system, developers can load data *as is* and begin building applications to deliver data immediately without having to handle the upfront cost of plumbing disparate systems together for the same purposes. We'll examine this further in upcoming chapters.

By looking at the schema-agnostic, schema-aware, and flexible indexing characteristics of a multi-model database, we've addressed the agility a multi-model database provides us in managing our conceptual and physical data models in the system as well as managing those documents as JSON, XML, binary, or text. We now need to take a closer look at how we access and interact with this data.

CHAPTER 5

Agility in Models Requires Agility in Access

In 2011, Martin Fowler wrote about what he called polyglot persistence, "where any decent sized enterprise will have a variety of different data storage technologies for different kinds of data."[1] Even single systems, he envisioned, might use multiple backend data stores to take advantage of multiple NoSQL (and SQL) technologies:

> This polyglot effect will be apparent even within a single application. A complex enterprise application uses different kinds of data, and already usually integrates information from different sources. Increasingly we'll see such applications manage their own data using different technologies depending on how the data is used.

The conventional understanding is that to achieve polyglot persistence, you store each discrete data type in its own discrete technology. But the definition of polyglot is "the ability to speak many languages," not "the ability to integrate many components." This is where enterprises get into trouble. They tend to want to take the conventional route of using multiple stores for multiple data types.

On the surface, it might appear to make sense to store each data model in its own distinct management system, but integrating this data with data from other systems increases the complexity of managing any unified view of the data exponentially. Fowler correctly noted that each system will have its own interface, requiring new

1 Martin Fowler, "PolyglotPersistence", MartinFowler.com, November 16, 2011.

skills to be learned, and that different systems will have different scalability requirements:

> This will come at a cost in complexity. Each data storage mechanism introduces a new interface to be learned. Furthermore, data storage is usually a performance bottleneck, so you have to understand a lot about how the technology works to get decent speed. Using the right persistence technology will make this easier, but the challenge won't go away.

A considerable amount of coding would be required to implement and enable a clean, unified interface for data storage, retrieval, query, and search across multiple systems. The assumption made here is that orchestration would occur via web services and the application tier.

Polyglot persistence represents a desirable goal, not a problem. But by storing disparate sets of data in their own unique silos, whenever we want to combine and integrate those datasets into a unified view, we find we've actually helped proliferate the problem of silos, extract, transform, and load (ETL), data movement and transformation. We've also increased our requirements for system-to-system connectivity and custom pipelines for orchestration.

But in a multi-model database, as we stated, we can store data in different formats and different models and query across them composably. If the database stores XML, XPath, XQuery, and XSLT should be available for search and query and create, read, update, and delete (CRUD) operations, as those are the native languages for XML. If the database stores JSON, JavaScript should be available for information management, as well, because it is the natural language for JSON. If the database allows us to store Resource Description Framework (RDF) triples, a SPARQL interface should be present. If the multi-model database allows you to create relational projections or views of data, SQL should be available as a query language. Agility in models requires agility in the data access.

Depending on our use case, we'll want to use the appropriate language for the appropriate formats. Additionally, a REST interface should be available, because most modern development systems interface with data stores through web services. An out-of-the-box and extensible REST interface can abstract the underlying lower-level languages while providing the same functionality. Client APIs in common languages such as Java and .NET can also provide the same functionality. All these interfaces should be composable. Using one interface should not disallow the use of another. And this is how it is in a multi-model database—a unified repository for multiple data models persisted in multiple data formats accessible through a variety of language interfaces to a single system using a single API!

Composable Design

The Unix philosophy emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and repurposed by developers other than its original creators. The Unix philosophy favors *composable* as opposed to *monolithic* design. In a nutshell, a function in Unix tends to do one thing, do it very well, and you can use and combine functions together with other functions to create new outputs. In a multi-model system, we favor composable over monolithic design, as well. Indexes and APIs and functions aim to do one thing very well. We should be able to use and combine them in new ways to create the desired inputs and outputs for our applications.

> Be mindful of multiproduct, multi-model databases or systems claiming to be multi-model but are limited or not entirely multi-model. You'll be able to determine this because combining APIs and indexes might not work for all combinations, and certain combinations might even mean we lose other advertised capabilities of the system (or systems) as a result!

Schema-on-Write Versus Schema-on-Read

For decades now, the database world has been oriented toward the *schema-on-write* approach. First you define your schema, then you write your data, then you read your data, and it comes back in the schema you defined at the outset. This approach is so deeply ingrained in our thinking that many people would ask, "how else would you do it?" The answer is *schema-on-read*.

Schema-on-read follows a different sequence: just load the data *as is* and apply your own lens to the data when you read it back out. So instead of requiring a schema first, before doing anything with your data, a multi-model systems can use the latent schema already with the data and update this existing schema later as desired or needed.

In a multi-model database, we're going to model entities and relationships. The data will be stored as documents and triples. We saw how we wrap the entity in an envelope to add standardization of attribute names, values, and structure, as well as information related to metadata, lineage, and provenance. The shape of the data we persist is the schema we write. It is the schema persisted within the system; and in multi-model, multiple schemas can exist at once, and we can query across them all.

In the relational world, when it comes to data integration, we know we must define a schema, before we do anything else, that addresses the concerns of all stakeholders and constituencies in advance, and then we can load the data to fit that schema. Retrieving data from a relational system and reconstituting it and repurposing it essentially becomes a mapping exercise followed by some transformation, and all performed at the application tier.

But now that we have our multi-model schema, and even before we've identified entities and relationships, we can begin to query that data and provide different lenses on the same set of data. With schema-on-read, we're not tied to a predetermined structure, so we can present the data back in a schema that is most relevant to the task at hand. This is simple to do in multi-model because we have the appropriate language interfaces at the database layer to transform the data as we request it. It's not uncommon to keep one shape of the envelope data in multi-model but provide different lenses on that data for multiple consumers through languages such as JavaScript, XSLT, or SPARQL, as depicted in Figure 5-1.

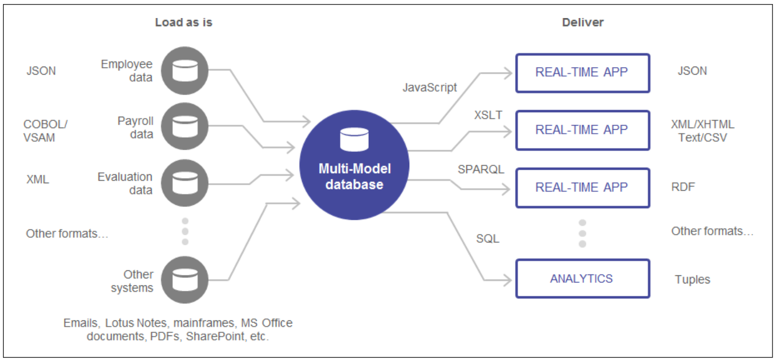


Figure 5-1. Load as is, harmonize, and deliver multiple lenses for multiple consumers on a single persisted model

We also can use SQL in a schema-on-read fashion. In multi-model, we have entities and relationships, just as we had in our source relational systems. If we want to create a similar relational view of that data, we can. We accomplish this through indexing in the database to materialize or project views. If we can create a tabular, relational view of data, we should be able to query the data in the standard natural for its format. SQL is that standard language. In today's multi-model database, SQL queries are read-only, in that they read from views of data materialized in indices. In a multi-model database, SQL becomes another lens through which to view the same data.

And this is another benefit of multi-model. Being able to provide different lenses for different consumers of data from a single source allows us to deliver data consistently, reducing data movement and copies because we don't need to copy data to a standalone silo for a specific application to ensure data access and delivery. We don't need to save the data in a format specific to a business unit or any particular use case. Applications become consumers from a unified, centralized repository. In this way, multi-model databases are great for creating integrated "data hubs" that span many data sources in a single repository.

But there is more to reducing data movement and copies than simply not copying it. Putting it all in one place will not help if the data is not indexed, accessible, and able to be processed. Table 5-1 lists the core technologies needed for each data type to be considered truly "supported" rather than simply copied and stored.

Table 5-1. Multi-model data format and API checklist

Format	Programming languages and standards
JSON	JavaScript
RDF	SPARQL
Relational	SQL
Text	No real standard. Look for integrated search.
XML	XPath, XSLT, XQuery, XSD Validation

For example, putting JSON data into a Binary Large OBject (BLOB) in a relational database does not make it accessible in any real sense. Adding three different data types to a Hadoop-based data lake (see "Data Lake" on page 87) does not enable that data to be indexed, transformed, and exposed by different lenses, or easily processed.

This is why we emphasize the benefit of multi-model databases, rather than simply heterogeneous storage.

In addition to adhering to standards-based languages, you also should look for standards-based interfaces, such as JavaScript, Java, and REST APIs (see Table 5-1).

In our multi-model database discussion thus far, we have discussed agility in data models and agility in access and delivery through APIs, interfaces, and indexes. We see the opportunities for new ways of working with data and how we can take advantage of multi-model databases to begin addressing data integration problems. If we're going to integrate a considerable amount of data, we now need to examine scalability requirements for multi-model. Additionally, if we're going to run our business on a multi-model database, ACID transactions, security, and other enterprise features we require are of paramount importance to us, too. In Chapter 6, we continue our multi-model journey by taking a closer look at how we scale multi-model.

CHAPTER 6

Scalability and Enterprise Considerations

Scalability

There are two basic approaches to scaling a database to large datasets. One is to “scale up” by increasing the power of the server on which the database runs, and the other is to “scale out” by adding additional servers that somehow work together to increase overall capacity.

NoSQL databases scale horizontally—or scale out—and multi-model should preserve this capability. A scale-out database is partitioned across multiple nodes, typically running on commodity hardware. It’s a divide-and-conquer-approach to data management. Data is distributed across many nodes, but queried as one unit, using one set of indexes, one integrated query, and one query optimizer.

Think of it this way: a standard deck of cards has 52 cards in it. If I hand that deck to my friend Joey Scaleup and ask him to find me the two of hearts, in the worst-case scenario, Joey must flip through all 52 cards before finding the card I asked for. But if I take that same deck and hand a quarter of it each to my friends Chris, Mary, Wayne, and Colleen, in the worst case, one of them has to rifle through only 13 cards before finding the same card. I get the result in a quarter of the time. The benefit of distributing that deck is similar to how scale-out systems work. Each person holding a quarter of the deck has no idea what cards the other person is holding (shared-nothing architecture). A coordinator broadcasts the request, and each person executes the task for the cards they hold. If I want to get the result quicker, I can scale out by dividing the deck evenly among more friends. If I scale up, my only option is to find a person who can rifle through cards faster than Joey.

Relational systems traditionally scale vertically, or scale up. This approach requires adding more resources to a single system. Adding CPU, memory, and storage to a single system can quickly become cost prohibitive. In relational, we now also see engineered systems, in which proprietary software operates on proprietary hardware to help overcome the performance issues that occur when you can't split the resolution to a problem into smaller tasks.

Both approaches have pros and cons, but if we're going beyond relational, a multi-model system will need to scale out.

These NoSQL technologies all provide some way to partition your data such that the data will be distributed across a cluster of commodity servers. This is usually decided based on attributes within the data. Partitioning therefore requires some upfront understanding about your data and what kinds of questions you'll be asking, such that all of the results aren't clumped together within the cluster (which would create performance hotspots) and are instead distributed evenly across all nodes. In our card example, if we don't distribute the cards evenly, Chris could end up with all the cards and doing all the work for us, while our other three friends sit idle. Chris becomes fatigued from doing all the work, and we're not seeing the benefits of dividing and conquering our data. Similar rules apply for distributing data among multiple nodes.

So, with relational and some NoSQL systems, we decide how to partition data across nodes based on some attributes within the data, and these attributes are decided upon by what questions we think we'll want to ask of the data and how we think we'll want to index it. This is done to ensure data is distributed equally across partitions and to take advantage of the limited indexes provided by the underlying database system. However, if the questions we want to ask change, repartitioning might be required to maintain performance and to change or update our indexes. At scale, repartitioning can come at great cost with respect to time, effort, and even interruption of service. Because of this, extensive data modeling prior to loading any data can be required for NoSQL systems, too. It's also possible to just get the design wrong and have to rebuild your database whenever new data or queries are introduced. These are all common problems when you need to deal with schema-first in data integration.

Data Distribution in Multi-Model

Multi-model databases separate data ingestion from data organization. Being schema-agnostic, there is no upfront schema requirement to make a start with our data. No schema needs to be defined prior to loading the data, and no particular key or attribute is required to determine how data is partitioned across a cluster. Data can be loaded *as is* and will be distributed evenly across all partitions by default, and all search and query features and capabilities will continue to work together composably and at scale. As data volumes grow, we can add new partitions to the existing database, and data can be rebalanced automatically to maintain database performance (see Figure 6-1).

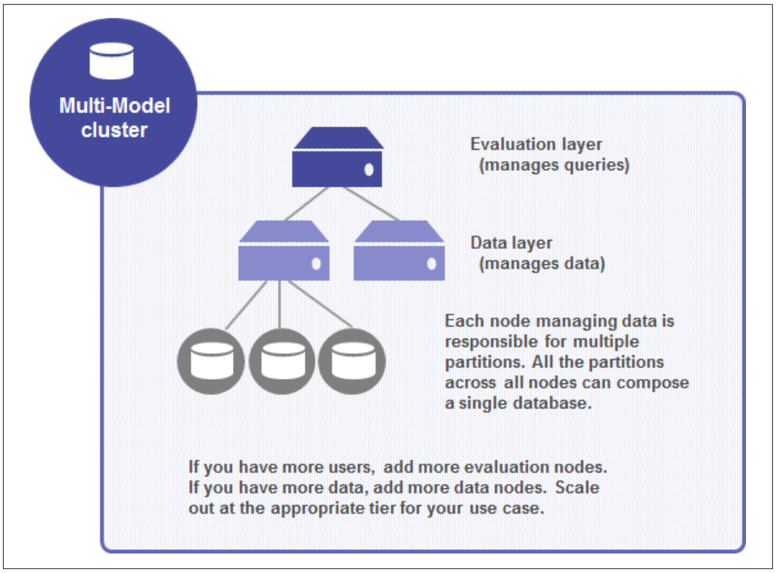

Multi-Model cluster
Evaluation layer (manages queries)
Data layer (manages data)
Each node managing data is responsible for multiple partitions. All the partitions across all nodes can compose a single database.
If you have more users, add more evaluation nodes. If you have more data, add more data nodes. Scale out at the appropriate tier for your use case.

Figure 6-1. Scale out cluster architecture

NOTE

An issue with some NoSQL systems is that when you scale out, certain indexes and features of the database, such as APIs, queries, and indexes, will no longer continue to work composably or might fail to work at all. This just isn't acceptable for any database. Luckily this isn't the case with multi-model systems.

Scaling Performance

Conceptually, a multi-model database has two major areas of concern: managing physical storage of the data and managing the queries. In a clustered system for which managing the queries is more complex and needs to be spread out among many computers, a layer needs to operate as the brain for the system that will broadcast queries and aggregate results. At MarkLogic, we call this broadcaster/aggregator the *evaluation layer*. Recall that a multi-model database uses a shared-nothing architecture, which is a distributed computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system. More specifically, none of the nodes share memory or storage.

In a multi-model database, the query processor determines which queries you can run and what languages you can use (query models). The storage manager/data layer dictates how the data types are stored as well as indexed and is responsible for using those indexes to resolve queries that reference them. Again, in a distributed system, the query processor needs to be able to talk to many machines. The query processor can reside in both the evaluation and data layers. Any one node in the system has no awareness of what the other nodes are managing. In Figure 6-1, for the two data manager nodes, the data manager node on the left only knows about its three partitions and the data it is managing. The left node has no idea how many other nodes are in the system or how those are operating or what data they are managing. This is what is meant by "shared-nothing." The advantages of this versus a central system that controls the cluster include eliminating any single point of failure, allowing self-healing capabilities, and providing an advantage by offering nondisruptive upgrades.

With today's modern systems, each node in a cluster can operate as both an evaluator and a data manager. Cluster topologies depend on use case and environment. But if you separate the evaluation and data layers, this should be configurable within the multi-model system and not require separate software packages. The difference really is in supporting hardware. Evaluators need less storage because they aren't managing data, but require more memory and CPU because they are broadcasting queries and aggregating results. Data managers required more storage to manage the data, but not necessarily as much memory or CPU.

A certain number of nodes will be required in any multi-model database to define "a cluster." In MarkLogic, three nodes are required. Then, you have the choice to scale out at the appropriate tier based on use case demand. If you have more users, add more nodes to your evaluation layer. If you have more data, which is often the case, add more nodes to your data layer. When you add nodes to the data layer, along with introducing its new available partitions to the database, the multi-model systems should support some rebalancing so that data distribution across all nodes remains uniform and consistent.

Clustering provides four key advantages:

- Use of commodity servers, which can be acquired at reasonable prices
- Incrementally add (or remove) new servers as needed (as data or users grow)
- Maximize cache locality by having different servers optimized for different roles and managing different parts of the data
- Failover capabilities to handle server failures.

If your multi-model database is a single software product, performance should scale near linearly. It's simpler to do capacity planning because all nodes will be operating by a uniform set of rules for a single system. In this way, you can plan in a consistent manner for upcoming storage, CPU, memory, and I/O requirements. However, if your multi-model system is multiproduct and requires additional components and plumbing to accommodate additional models such as search or semantic triples, scaling becomes much more difficult. Each system will have its own individual requirements for clustering, storage, memory, and CPU. These will all need to be accounted for, and scaling out will require more planning. It won't be as simple as just adding more nodes at the appropriate tier.

Looking at Query Capabilities of Specific Technologies

In MarkLogic, the query layer superimposes a coherent, structured tree model on top of the raw data while leaving the raw data intact. With that tree model, you get two things: a rich and well-known set of ideas and query languages that are precisely geared to talk to XML and JSON data; for example, JavaScript, XPath, XQuery, XSLT, and SPARQL. The act of wrapping it means that all of these well-understood and well-supported, standardized ways to structure and query data are now at your fingertips and can be applied to tame an unruly mass of multi-model data. This gives you a principled, reproducible way to think about the data, index it, and query it by using JSON, XML, and Resource Description Framework (RDF).

NOTE

Some systems call themselves multi-model when they are really multi-query. They are only storing one data model, but will allow you to access that data model through multiple query languages. To be multi-model, you must be able to query multiple models by using their native query language.

Managing the Physical Data

The storage manager/data layer dictates how the data types are stored as well as indexed and is responsible for using those indexes to resolve queries that reference them. It is important that the data-layer abstraction allows you to manage the way data moves in the system. In MarkLogic, the data layer is mostly centered on drive I/O, caching, journals, locking, and transactions.

MarkLogic allows the splitting of these layers to scale out either the application services or the data management.

Indexes and indexing are important parts of the data layer because they contribute to the performance of a multi-model database. We have already discussed how a shared-nothing architecture contributes to performance. The optimal construction and management of indexes completes the picture of how multi-model databases provide excellent query performance across multiple models of data.

Indexing

Let's remember why we are collecting, storing, and managing data in the first place. Ultimately, we are seeking knowledge or insight, which we access through queries. Ideally, we would like it to be easy to create those queries and for them to be answered quickly, but speed (both latency and throughput, to be exact) depends on how the data is organized. This is where indexes come in. In short: good indexes, fast queries; poor indexes, slow queries.

There are two ways to find things in a library: you could look at every book on every shelf, or you can go to a card index. Of course, scanning every book on every shelf is not very practical, so why would we want to do that in our database?

In any database system—SQL, NoSQL, NewSQL, multi-model—data is indexed and organized.

In relational database management systems (RDBMS), the DBMS needs to do a full scan of each record to perform a query unless a usable index has been defined. Suppose that you have a database of NASA scientists and astronauts, and you want to write the following query: "I want to see all astronauts who went on Apollo missions." If you have 10,000 names in your database, it will need to do a full table scan of those 10,000. You could then create an index that lets you find a particular astronaut (and the row associated with the astronaut, which presumably contains the name) very quickly.

And if you add an XML layer onto a RDBMS, you need a special index that knows about structure (paths) as well as values. This is problematic because your options limited to the following:

Define upfront which paths you want to index
: This requires some knowledge of the data and the types of queries you'll be issuing against it. This strategy will only help performance for the paths defined and proves ultimately inflexible.

Index all possible paths
: You do this blindly so that you can query any possible path in the future. But this causes an explosion of the index, especially for rich documents such as XML with inline markup, which adds to the overall footprint of the system and can create other performance issues.

In relational and other databases, you index data to speed up queries. Building indexes is difficult work. That's true of commercial databases, of open source databases like MariaDB and PostgreSQL, and of most NoSQL solutions that do indexing. (The ones that don't do indexing solve a very different problem and probably shouldn't be confused with databases.)

A key part of a multi-model database, however, is its chameleon-like ability to work with different models of data on the data's own terms. A multi-model database sidesteps the traditional problems of indexing relational data by indexing model data naturally. The multi-model option allows you to index an efficient universal representation of that data. This makes the data more uniform with a known structure, thereby speeding up indexing and search.

For example, MarkLogic does this by indexing all elements (XML) and properties (JSON) and all parent-child relationships for these in a flexible and manageable compressed-tree representation. At the end of the day, XML and JSON are both abstractions of a tree-like data structure. Although there's a mismatch between those two formats that we won't go into here, they are both abstractions of a tree, and MarkLogic indexes both similarly so that indexing across both document types provides efficient and quick lookups against both document types. And assuming documents of similar size and content, the index for one data format is not necessarily larger than the other because the index is a unified, composable structure, split across nodes in the shared-nothing architecture.

ACID Transactions

As stated earlier, ACID is an acronym that stands for the four properties of a database that ensure that the database processes transactions reliably. A database with ACID transactions ensures that two concurrent updates will not overwrite, or partially-overwrite, one another, that a completed update will be stored durably even if there is a system failure, and that consistent views of data are seen by queries, even when data changes during the query.

ACID properties are particularly important for a database that must not lose data, or serves a safety, health, or financially impactful purpose. Some systems are "good enough" platforms, such as Facebook, Craigslist, or Google's ad-serving platform, which all can lose a small amount of data in unusual circumstances or give incomplete or incorrect results without undue harm.

In fact, the Google F1 team claims extensive experience with *eventual consistency* approaches, which are not ACID, and writes as follows:

> *We also have a lot of experience with eventual consistency systems at Google. In all such systems, we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date. We think this is an unacceptable burden to place on developers and that consistency problems should be solved at the database level.*[1]

ACID transactions are commonplace in relational databases, and contrary to what you might have been told, ACID capabilities also exist in NoSQL and multi-model databases. They're important because ACID is the only way for a database to guarantee that users are looking at consistent and correct data. If the database does not support ACID, you will need to write application code anytime consistent and correct data is required. ACID transactions allow application developers to focus on solving business problems without needing to be experts in distributed systems and databases. If the database doesn't support ACID and consistent and correct data is required, and there are also requirements for failover and high availability, your developers will be spending significant time implementing compensations for the inadequacies of the system(s) being used. This is energy and effort that could be better spent getting applications into production and delivering results.

VoltDB, an in-memory RDBMS, has provided great questions to ask yourself if you're considering a system that doesn't support ACID.

- *What happens if two operations want to mutate the same data; which wins? What happens to the loser?*
- *How long does it take a replica to reflect changes made to a primary?*
- *What happens if an operation succeeds on a primary copy of state, but fails on a secondary copy?*
- *Between the time I read this data and the time I'm going to write data based on that read, has the original data changed?*

1 Google, Inc., "F1: A Distributed SQL Database That Scales".

- *If my operation has 10 steps, and step 7 fails, do I need to manually undo steps 1 through 6?*
- *What happens if two operations want to mutate the same data at the same time but one fails on a replica on step 7 of 10, and also two other things go wrong?*

Here are a couple other questions that are specific to multi-model:

- What happens if you need to update different parts of the same entity that are stored in different models? That is, updating a document and the triples that represent its relationships to other documents.
- If I update one or more documents, when will my full-text queries be able to see the changes made by that update?

The argument against ACID from many NoSQL providers is that it costs too much in performance and availability. Although there is some performance overhead for transaction management, modern systems are often able to provide ACID transactions at minimal overhead. We expect transactions to be present in new database products or added to entry-level products in many cases.

Users should be aware that many enterprise capabilities implicitly depend on an ACID transaction capability. For example, a consistent database backup is typically not possible if the system has no notion of an "isolated, atomic" update. The same is true of *high availability* (HA): do you require the ability to continue to use the system with a complete and accurate view of the data even if a partition fails? If you require partition failover within a cluster for HA, how will you guarantee a partition replica will serve the correct data if you can't guarantee the master partition has the correct data? What about *disaster recovery* (DR)? If you require failover for the cluster, how will you synchronize a consistent view of data in cluster A with the same view in cluster B? What about real-time requirements? Do you need to be able to query an accurate and up-to-date view of the data that's been committed to the system? Or can you wait for data to eventually persist?

ACID is a critical component to database success. And yes, multi-model databases support ACID transactions, HA, and DR; and they can provide real-time analysis of data that's been loaded consistently and correctly.

Systems that provide a polyglot persistence façade over a collection of separate, internal databases are generally not ACID compliant, because the various subproducts will commit and store their updates independently.

CAP Theorem

In a nutshell, the *CAP theorem*, also named *Brewer's theorem* after computer scientist Eric Brewer, states that for any distributed computer, you can only ever have two out of three of the following guarantees:

Consistency
: This means that each client always has the same view of the data.

Availability
: This means that all clients can always read and write.

Partition tolerance
: This means that the system works well across physical network partitions.

According to the CAP theorem, if a partition happens, a system must either be classified as giving up availability or consistency. For some systems, such as one serving up advertisements, consistency is not critical—as long as some ads are available, the system works, and a CA system is appropriate. For others—such as a financial system—an incomplete result is unacceptable, so a CP system is required.

Any distributed system should be classified as CP. So, if a host (partition) goes down in a cluster, this is bad, right? As now, we've lost that partition, if it's no longer available in a distributed database, what happens to the data for those hosts? Will there be an incomplete view of the data to any subsequent queries as a result? Actually, your data and your view of that data can remain intact in the following circumstances:

- If your system supports partition failover, you still have a complete view of your data available on the remaining hosts because a partition on one host will be replicated to another host.
- You'll still have a complete view of your data because if you lose the host and its partitions, the replica partitions in the cluster

will automatically switch to active. This is how multi-model systems enable HA.

The issue here is that the definition of "availability" by CAP is defined too narrowly to be realistic. Brewer spoke to the widespread abuse of CAP theorem in his 2012 update, "CAP Twelve Years Later: How the 'Rules' Have Changed".

Availability in the (proven version of the) CAP theorem is defined as having every node on both sides of a partition able to respond to a request. This is an unrealistic requirement. The desired state is that the data as a whole remain available, and it can. As a counterexample to CAP, if MarkLogic experiences a partition, only the side with a quorum (more than 50 percent of hosts) will continue to respond to requests and be available. Those who've used MarkLogic failover options feel that MarkLogic HA is still good, though, because it sacrifices "CAP theorem availability"; but in the case of a partition failure, still remains completely functional in the traditional, useful, HA sense.

Security

Whenever we create systems that store and retrieve data, it is important to protect the data from unauthorized use, disclosure, modification, or destruction. Ensuring that users have the proper authority to see the data, load new data, or update existing data is an important aspect of application development. Do all users need the same level of access to the data and to the functions provided by your applications? Are there subsets of users who need access to privileged functions? Are some documents restricted to certain classes of users? The answers to questions like these help provide the basis for the security requirements for any application.

Yes, NoSQL and multi-model systems can be secure. The same level of enterprise grade security found in an Oracle, Microsoft, or IBM database can be found in NoSQL and multi-model databases.

Frequently this is not the case, so be careful when choosing your NoSQL or multi-model database!

In a single-product, multi-model system, built-in security can be used to apply different security models to a variety of data via a single API. However, if you have a multiproduct multi-model system, you really need to ask yourself how that system is being secured. Different products come with different security capabilities and models.

Similar to having to compensate in our application layer for a lack of ACID capabilities, we might want to look for a single-product multi-model database if we find ourselves plumbing together different security models. With the proliferation of data and database systems to support it, we're seeing an increase in threats to databases everywhere. Security in a database shouldn't be an afterthought or bolt-on. Security really needs to be baked in at a foundational level of the database. Enterprise multi-model NoSQL systems can come with powerful and flexible tools for security, allowing us to manage our data securely without sacrificing performance or agility.

Common Criteria Certification

The Common Criteria for Information Technology Security Evaluation is an international standard for security by which vendors demonstrate their commitment and ability to provide security to their customers. The Common Criteria is the most widely recognized security certification for IT products in the world—including databases. There are 25 countries, including the United States, Canada, India, Japan, Australia, Malaysia, and many countries in the EU, that recognize the certification.

Getting the certification is a rigorous process. The certifying authority runs through a battery of tests against a system's database security target to ensure that it attains the expected results. It also tests across various areas such as authorization, authentication, security vulnerabilities (e.g., cross-site request forgery). The product does not pass the certification process unless it succeeds in each area. So, if a product vendor says it does authentication on a cluster when it's set up in a certain way, the authority ensures that it works like the vendor's documentation says it does.

MarkLogic was the first NoSQL database to receive a Common Criteria certification with MarkLogic 4, and to date (with MarkLogic 8) is still the only NoSQL database with this distinction. Other standards supported by MarkLogic are important as well, such as FIPS

140. However, these are more narrowly scoped than Common Criteria, which is widely recognized and is the standard against which all major database vendors, such as Oracle, Microsoft, and IBM, continue to certify, some of them even doing a certification every year. In total, there are only six database vendors that have earned a common criteria certification.

CHAPTER 7

Multi-Model Database Integration Patterns

We've taken a close look at the capabilities a multi-model database can bring us in providing agility in data management, securely, and at scale. After people begin to see the benefits of a multi-model database as it relates to their particular data integration challenge, the next often-asked question is, where does a multi-model database fit in my architecture?

The answer is, *it depends*. We say that jokingly, but with some truth. There's often a notion that a multi-model database, or any new database being introduced into an environment, will come in to replace some or all existing systems and perhaps do the job of existing systems, either better or with improved performance. Although this can happen, and system replacement might be a goal of IT or the business, when introducing a multi-model database into your architecture, this often just isn't the case. What happens most often is rationalization and improvement of an enterprise's integration strategy as a whole, as opposed to incrementally improving a single system.

Other systems exist for very good reasons. The problem with those systems is that they are silos. Given time, and data integration, a multi-model database can replace some existing systems, but that's not how data integration in these environments begins. More likely, the technologies disrupted by a multi-model database will be those supporting data movement, extract, transform, and load (ETL), system-to-system connectivity, data workflow transformations, mappings, and encodings.

The prime use case for a multi-model database is data integration. As such, data models in these environments are already defined. The multi-model database will be integrating data from existing systems and data models. Sure, you can set up a multi-model database for new use cases, and development of data models upon it might be completely new, but it's rare that you're starting with new data model development with regard to the data models you'll be working with. Those use cases do exist. But for our purposes, we're going to focus on data integration because that's where you'll find 80 percent of the use cases for multi-model databases generally start.

An illustration is always helpful. Be it healthcare, financial services, insurance, manufacturing, or any other industry, companies have had the same sets of tools at their disposal for data management, and these tools more or less connect in similar ways. As a result, without talking to one another, companies across different industries have created variations of the same beast for managing their data. A common, simplified enterprise data pipeline looks similar to what is shown in Figure 7-1.

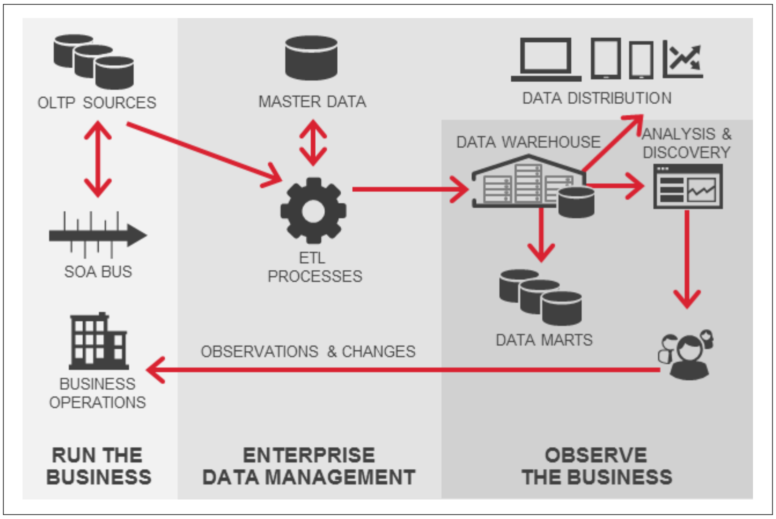


Figure 7-1. A simplified enterprise data pipeline

There is often a distinction between run-the-business and observe-the-business functions. Enterprise data management functions exist to manage transformation, master data, and ensure proper distribution. Data is copied and transformed to meet various needs.

Front-office online transaction processing (OLTP) systems book an order, or enter a policy, or commit a trade. This transactional data comes from some series of external sources back to HQ, and enters a Rube-Goldbergian colossus of data movement and transformation to ensure data quality, encode data properly, enrich data with additional data, and do some analysis for alerting purposes. Data hops from system to system, and it might have order numbers and names normalized by one system, encodings for product names applied at another, latitudes and longitudes applied at the next, and so on and so forth. The data mutates with each hop until it finally enters some central repository or set of repositories. For older, larger, well-established companies, these likely include mainframes. Because of the age and cryptic legacy interfaces of mainframes, it's not uncommon to send data to a staging area that will feed both the ultimate legacy target system, and a data warehouse. Or the data warehouse might be fed from the ultimate source system directly. Feeding this warehouse requires what? You guessed it! Data movement and transformation (aka ETL). Let's dissect the various components in this pipeline:

ETL
: These are the previously mentioned extract-transform-load processes. From the perspective of data integration, they are the most costly and time-consuming processes. They are also typically the most brittle part of the architecture because they are the only place where the modeling inflexibility associated with relational database management systems (RDBMS) gets addressed.

Master data management (MDM)
: Master definitions of important business entities become necessary as a result of data and business silos. Ironically, the accepted wisdom in most cases is to create yet another data silo in the hope that one more RDBMS data store will somehow succeed to serve the needs of the entire enterprise. The reality, however, is that the same ETL dependencies result in yet another system that can't keep up with the pace of business change. Those dependencies also often make data quality even more complex to manage.

Data distribution

Whether an enterprise is in the data distribution business (e.g., publishers) or simply has internal stakeholders who rely on the timely distribution of data (any enterprise), delivering quality data in a timely fashion is critical. When the data distribution process depends on a brittle data architecture, data quality and time-to-delivery challenges are negatively affected by sometimes even the smallest business change.

Data warehouses

These are the systems that are designed to support cross-line-of-business discovery and analysis. However, due to modeling and ETL dependencies, the approach is very much an after-the-fact exercise that invariably lags—often significantly—the most recent state of the business. We refer to the functions performed by data warehouses as observe-the-business functions, since their job is to report on the state of the business as opposed to doing something about it.

Data marts

A reaction to the slow-moving pace and lack of completeness of enterprise data warehouses. Here we copy similar data, at lesser scale, to a silo for a particular business unit that might combine a subset of integrated data warehouse data with some of its own data that might not exist in the data warehouse. Or in some cases, for the sake of "expediency," data marts might bypass the warehouse completely and use the same attributes but call them something different in their schema. In either case, the creation of these additional silos adds yet more complexity to overall enterprise data architecture.

Service-oriented architecture (SOA)

The run-the-business functions have integration needs as well; however, these are more real-time and transactional in nature. As a result, the strategy has been to focus mostly on the coarse-grained functions between systems and leave the data persistence operations to the silos themselves. This has resulted in a data integration strategy that is function-focused but not data-focused, putting integration in the application layer, not with the database.

Impact of analysis on operations

The net result of the preceding components found in the enterprise pipeline has been an ever-increasing distance between discovery and operations, creating data integration choke-points. Each of those red arrows in Figure 7-1 reduces data quality and also takes time, because of data having to move and be transformed and copied through the pipeline.

ETL: this three-letter acronym and its depiction in Figure 7-1 might look simple, but we know it is much more complex, and ETL requirements are always much worse than you think. Figure 7-2 provides you with an idea of this complexity.

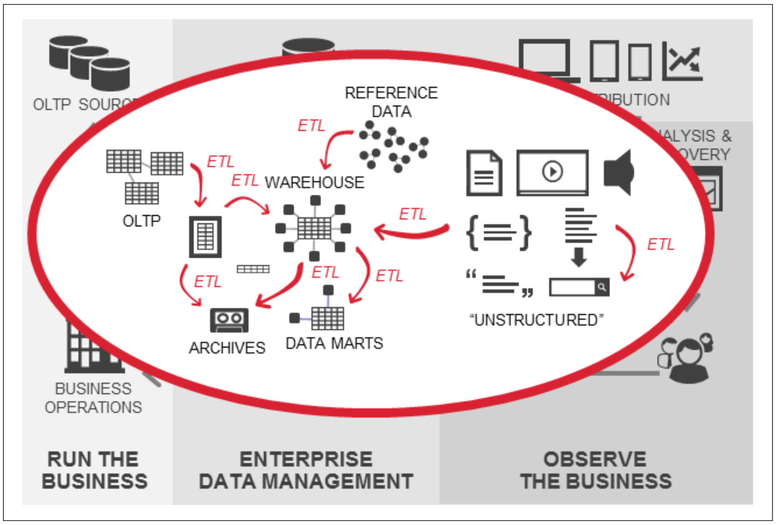


Figure 7-2. On any whiteboard, ETL is much more complex than it is given credit for

And here is where a tremendous amount of effort is spent (See Figure 1-2 in Chapter 1). Business is not static, though. New source systems are added (either by acquisition and/or new business requirements) and new ways to use and query data are developed, thus data management problems grow as new application-specific silos and new data marts are stood up. With this activity comes an ever-increasing gap between analysis and operations. And in this swamp of data movement, transformation and silos are where multi-model databases often begin.

Data movement, ETL, and associated tools are the first parts of an enterprise architecture to be rationalized with the incorporation of a multi-model database. The architecture will begin to simplify as data movement is reduced. I think this is an important point. In my experience, after people get their data into the multi-model database, they soon forget about the complexity of the landscape they previously maintained because they begin to focus on the data and all the exciting new ways they can collect, aggregate, and deliver information to consumers.

In the existing architecture, delays in synchronizing the transformation pipelines and schemas for supporting apps can cause the business to be delayed weeks in getting answers to the questions it wants to ask of its data. Also, with this complexity comes more opportunities for risk and error. Data can become stuck anywhere along the pipeline, which causes more headaches when trying to capture a complete picture of what the data looks like. Reconciliation procedures come with their own set of challenges and schedules.

With multi-model databases, source systems for ingest can change, and none of the data will be dropped on the floor. You query against the data you know about and continue to harmonize after you notice that the data has changed. Multi-model databases with alerting can detect a change in the shape of the records being ingested and then prompt you to do some analysis and incorporate any new attributes. Data will be loaded *as is*, harmonized, and delivered to downstream systems. Multi-model database solutions for data integration often follow a pattern that very much looks like a data hub, as illustrated in Figure 7-3.

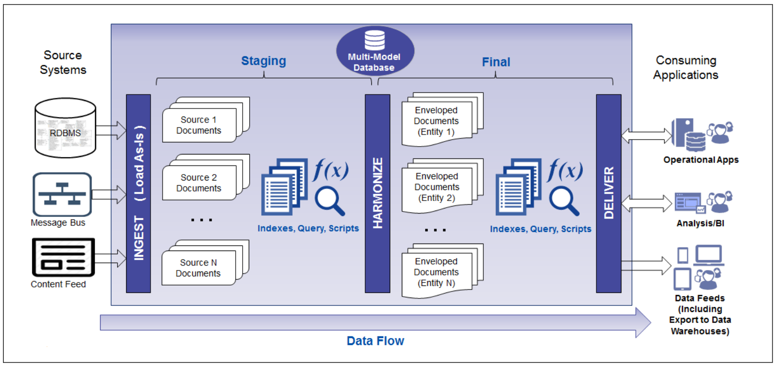


Figure 7-3. Operational data hub pattern

It *depends* comes into play again with regard to the size of the data integration problem to be solved and the availability of those in the organization to begin implementing a multi-model database solution while also managing other projects. None of this stuff is like flipping a switch. You don't purchase a multi-model database, move your data over, flip a switch, and all applications start using the hub. In reality, a phased approach will be enacted. Here again, we see the benefit of a multi-model database's ability to scale out as demand increases come into play.

Enterprise Data Warehouse

Likely there are some integration patterns already in place where you are introducing a multi-model solution. At a 30,000-foot level, where multi-model fits within these patterns will look something similar to Figure 7-4.

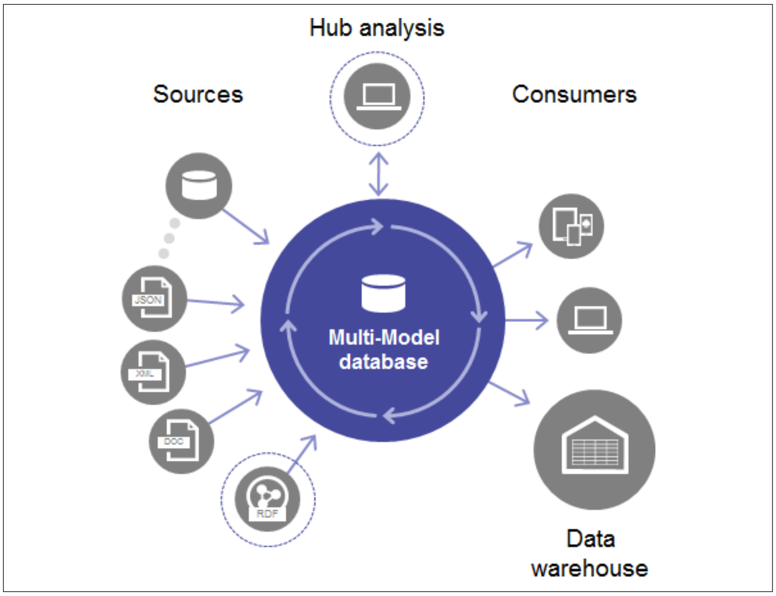


Figure 7-4. Multi-model working with an enterprise data warehouse

As an augmentation to an *enterprise data warehouse* (EDW), a multi-model system will be the rapid aggregator and harmonizer of data to feed to the EDW. Here ETL and data movement are reduced. An EDW is often found with batch-oriented workloads. It is commonly used for analysis only, contains only structured data, and is

ETL- and model-dependent. After the data is in the warehouse, analysis is reactive and query-based.

The multi-model database alongside an EDW allows you to store all your data after loading it *as is*. The EDW becomes a consumer (one of many) to the multi-model database. Real-time interactive queries will be possible against the multi-model database. The multi-model database provides a bridge between analysis and operations, allowing for two-way analysis, cross-line-of-business operations, and proactive alerting when identifying new information arriving in the system.

SOA

If you have a SOA infrastructure, it usually has the following characteristics:

- Function-focused
- Emphasis on data movement
- SLA-dependent on downstream systems
- Ephemeral information exchange
- Least-common-denominator data interaction

When augmented with a multi-model database (see Figure 7-5), transformations can be removed from the application layer. Mutations to data previously occurring within services now can be captured to enhance a SOA to include the multi-model data hub benefits of having:

- A data-centric service architecture (both data- *and* function-focused)
- Emphasis on data *harmonization*
- The ability to proxy for offline systems/services as appropriate
- *Durable* information interchange and management
- An interchange architecture that throws nothing away in the data lifecycle, enhancing data provenance

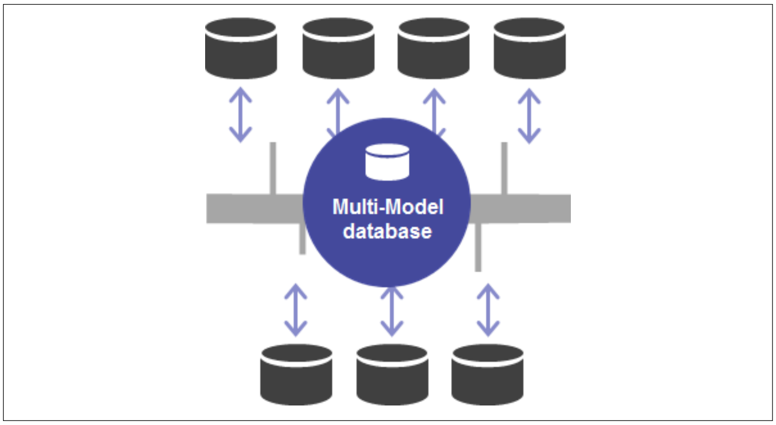


Figure 7-5. Multi-model working within a SOA environment

Data Lake

As many organizations are finding, copying data to Hadoop Distributed File System (HDFS) does not magically make Hadoop useful. Mapping data that arrives in HDFS to business entities with meaning makes it useful. With Hadoop came more attention on scale and economies of scale. It brought with it the promise of addressing a variety of structured and unstructured data and expanded what was possible with analysis and observe-the-business functions. However, even though anyone can load anything *as is* to a filesystem, the gaps that come with Hadoop require a level of effort to implement logic to make up for its shortcomings, and with this comes great complexity. Hadoop can be lacking in enterprise features such as security and operational maturity. It exists primarily in the analytical domain, focusing on observe-the-business type problems and leaving run-the-business functions to legacy technologies.

As a modular data warehouse, a Hadoop distribution is a collection of many open source projects, which are fit-for-purpose technologies. There are differing qualities of service and maturity across projects, and significant expertise and effort is required to configure and integrate these projects. As a result, there is a lot of churn, and it is possible to implement something similar to our simplified enterprise data pipeline in Hadoop (see Figure 7-1), actually widening yet again the gap between analysis and operations. It is possible to actually create silos of content within HDFS, except these silos are more technical in nature, as you store multiple models for specific technical representations of the same data for various applications: a model for Hive, a model for SOLR, a model for Spark, and so on.

A data lake on its own usually has the following characteristics:

- Batch-oriented
- Analysis only
- Saves everything and processes with brute force
- Simplified security model
- Limited or no context
- Multi-layered ecosystem that *encourages* technical silos

It's not uncommon to find people embracing multi-model databases to augment their data lakes (see Figure 7-6) to simplify the environment so that they can either slosh information in *as is* from HDFS, or use HDFS as a tier of storage of the multi-model database itself. Both cases deliver value more rapidly within a system with a more complete feature set. Augmenting a data lake with a multi-model database has the following benefits:

- Makes the entire architecture real-time capable
- Provides two-way (i.e., read and write) analysis
- Brings agility and "three Vs" capability to run-the-business operational functions
- Save *and index* everything for sub-second processing
- Mature and fine-grained security model
- Advanced semantics capability for rich context
- Reduction or *elimination of* technical silos

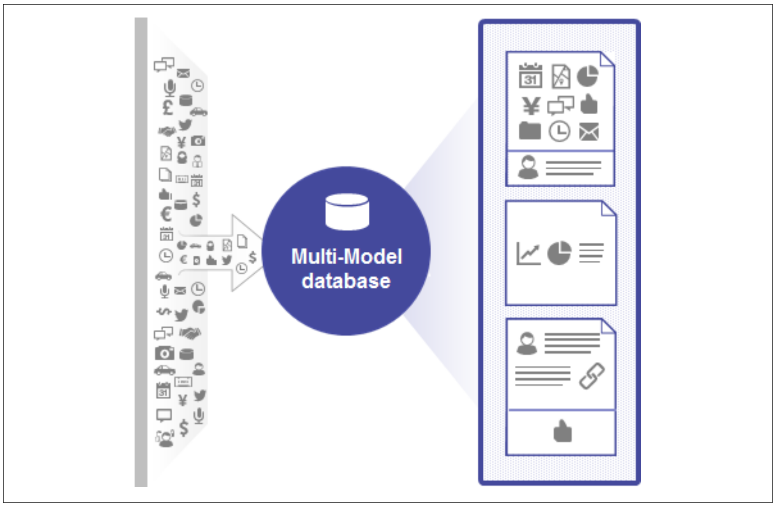


Figure 7-6. Multi-model working with a data lake

As mentioned previously, HDFS is often used in conjunction with a multi-model database for archive purposes. However, it also can be used as a source of enrichment.

For instance, the Internet of Things (IoT) presents new sources of data for data integration that we're beginning to see enter into multi-model systems. Sensor data on its own might not be very valuable (who cares if every five minutes a CO_2 sensor reports, "All OK here!"), and so landing the mass of IoT data quickly to a low-cost filesystem like HDFS can make sense. But, if we examine the IoT data in aggregate and over time, we can mine IoT data from HDFS and bring in aggregations or enrichment to combine with the data from other sources we've integrated in multi-model. In this way, if we combine the aggregate driving habits for a particular car and match them with the structure and unstructured policy information for an insurance customer and combine this with weather data and maps, we can develop applications that deliver actionable intelligence: "It looks like you're on a road with a road out ahead, and you're not driving a four-wheel drive vehicle. Here's an alternate route."

Microservices

Microservices refer to an architectural style that provides an approach to developing a single application as a collection of independent services. A single microservice is an autonomous service that performs one function well and runs in its own process communicating via lightweight mechanisms, often an HTTP resource API and RESTful interfaces. Services in this framework are modeled around business domains, instead of technology layers, so as to avoid many of the problems found in a traditional tiered architecture. Services are also independently deployable, and you can automate that deployment. A minimum amount of centralized management is required for microservices, which might independently utilize different data storage technologies and be written in different programming languages to support them.[1]

Microservices are becoming increasingly attractive in enterprise for primarily two reasons:

- Microservices reduce or eliminate the dependency on the traditional, monolithic enterprise application.
- Microservices serve an agile, fast-paced development cycle and, because they are at once flexible and focused, they can serve the needs of stakeholders across an organization.

The microservice architecture deconstructs the monolith into a suite of modular, composable services that allow for independent replacement and upgradeability.[2] However, like the SOA of the last decade, if there is a focus only on functions and not data, data silos can proliferate even more rapidly. That is why a single-product multi-model database fits well in this environment, because it encapsulates many dependencies within a single deployable unit, such as the following:

- Database
- Search
- Semantics
- HTTP server with REST API
- Client APIs (Java, JavaScript)
- Scalability

1 James Lewis and Martin Fowler, "Microservices: A definition of this new architectural term", MartinFowler.com, March 25, 2014.

2 See Sam Newman's book on the subject, *Building Microservices* (O'Reilly).

- HA/DR
- Security

However, the key benefit here is not so much around technology stack simplification; rather, it is more around ensuring that the service architecture is *data-focused* to ensure data harmonization within the services architecture, as opposed to proliferating data isolation (see Figure 7-7).

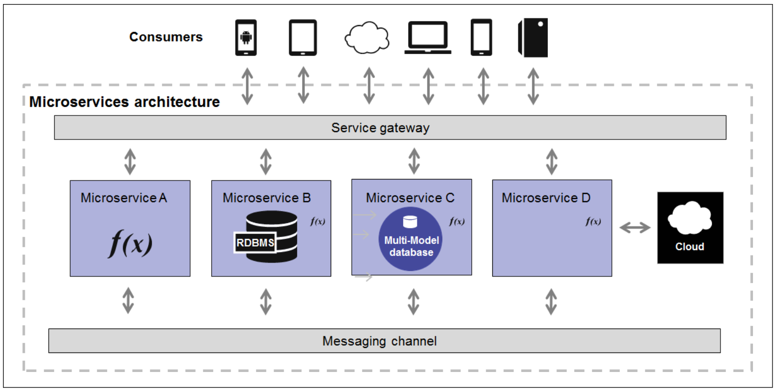


Figure 7-7. Multi-model and microservices (a multi-model database can operate on-premises or in the cloud)

Rethinking MDM

Now, MDM on its own isn't a pattern, per se, with regard to architecture. But, we do notice a pattern and similarities with how many MDM projects operate, are managed, and tend to fail. They often attempt to integrate data by using relational tools and fall victim to a workflow pattern that looks very similar to our development example for integrating data sources in Figure 1-2 in Chapter 1.

But what if MDM projects were business-outcome driven? Typically, MDM project progress is often measured in terms of technical milestones. But a couple of years later, the end result still doesn't look like the outcome people actually want. A multi-model database supports an agile approach to mastering data that can be geared exclusively toward business outcomes (see Figure 7-8). A multi-model system can handle partially done MDM, whereas an RDBMS can't. This means changing business goals during an MDM project that's already in progress is not a problem for a multi-model database. We reap all the benefits of being able to work with data in a fashion similar to Figure 3-9 in Chapter 3.

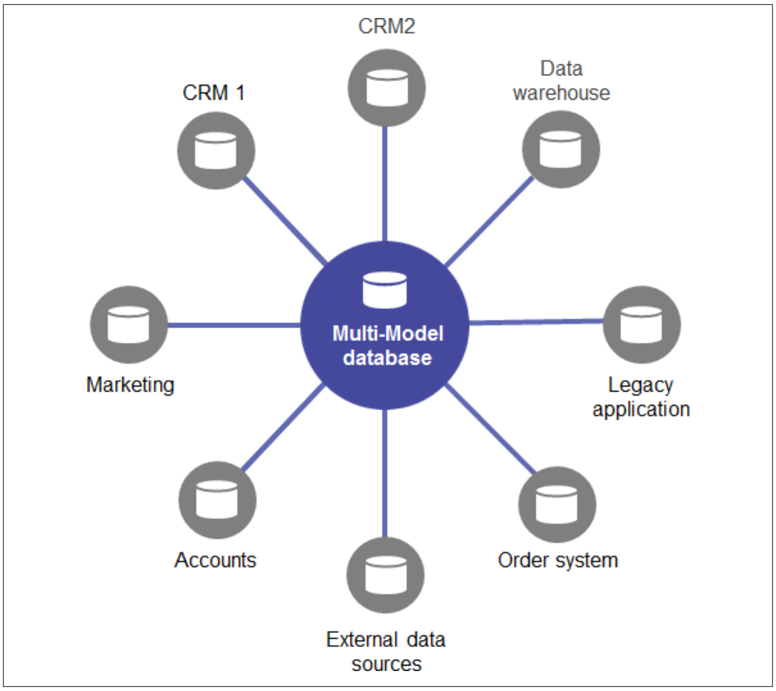


Figure 7-8. Multi-model for MDM

Benefits of using a multi-model database to support MDM projects include the following:

- Achieving progress incrementally tied to business drivers and events
- Measuring progress in weeks and months, not years
- Saving "all of the breadcrumbs" to provide a clearer view of provenance
- Increasing data quality—minimizing the need for fuzzy matches

CHAPTER 8
Summary

The rapid growth of data, including the digitization of human communication, has created a proliferation of data silos throughout enterprises. Trying to see across these silos—creating a 360-degree view—has been an arduous task, if not a losing battle, as companies spend untold millions trying to buy tools that help them parse data in the traditional, relational way.

The challenge is integrating data from silos:

- ETL and schema-first systems are the enemy of progress and getting things done.
- We are going to use many models (relational, mainframe, text, graph, document, key-value).
- We are going to use multiple formats (JSON, XML, text, binary).
- Much of our data actually comes structured from relational tables, but the same entity type can be modeled in different ways across multiple different silos.
- The natural approach has been for our people to code their way out of the problem of many models and polyglot persistence with many technical silos.
- The next step is to move the complexity into multi-model database management systems (DBMS) products that load *as is*.

- This means new products, new evaluation criteria, and new (higher) expectations for DBMS as we move forward and evolve.
- A significant *unlearning of biases and assumptions* is required.
- We will be introducing new products into existing architectures.
- Change management will be affected because when we do the work and how quickly we accomplish it will change.

In addition to the data, the context of data is not necessarily in the database. Today, it might be stored in Microsoft SharePoint, a Microsoft Excel spreadsheet, in an expert's head, or an entity relationship diagram printed out a few months ago and hung on a DBA's office wall—everywhere except for the database where the data is stored. Making sense of the data within one database is difficult. Across data silos it can be impossible. This makes getting and reconciling metadata and reference data a brittle and expensive process.

The drive to shorten development cycles, meet business needs, and produce an agile, data-centric environment has created a need for a flexible DBMS that can store, manage, and query the right data model for the right business function. A multi-model database allows us to capture data's context and store it in the database along with the data, to provide auditability and enhance how we operate with our unified data in the future.

A *true* multi-model DBMS provides the following:

- Native storage of multiple structures (structure-aware)
- The ability to load data *as is* (no schema required prior to loading data)
- Ability to index multiple structures (different indexes)
- Multiple methods of querying those different structures (different APIs and query languages)
- Composable indexes and APIs (use features together without compromise)
- Proven enterprise capabilities (ACID, scalability, HA/DR, failover, security)
- Ability to run on-premises or in the cloud

- All in a single software product designed specifically to address multi-model data management.

With all these capabilities of a true multi-model DBMS at our disposal, we can do the following:

- Rapidly deploy operational data hubs to integrate our data silos
- Load only the data we require or want *as is*, with no upfront, schema-first requirement
- Employ the envelope pattern to keep our source data in a shape aligned closely to its native data model
- Harmonize source data with standardizations of field names, attributes, and structure
- Add additional rich information to our data envelopes such as lineage, provenance, triples, and any other metadata we might want to store
- Deliver the data we persist in multiple formats to multiple consumers with governed transform rules
- Integrate our silos easily into existing architectures, disrupting ETL and data movement at first and potentially EOL'ing other systems as we progress, all to the benefit of simplifying our infrastructure environments
- Integrate our silos in about a quarter of the time that traditional methods take

Whatever we practice, we become professional at it. Over a long period of time, many have become experts at working with relational systems. From developers and DBAs all the way up through the groups and individuals in the business organization, the impacts on how we integrate (or fail to integrate) data from relational systems and other silo'd data sources are felt. We can't solve the problem with the same thinking and tools that caused the problem in the first place. To achieve a unified view of our data, we'll need to employ techniques that we never have before. Fortunately for us, we are not alone. There are many already down this path, transforming their data pipeline architectures and their business organizations to enjoy rapid and tremendous success with multi-model databases. We can learn from the new, updated practices of their data-integration techniques using multi-model database management systems to begin our own journey.

About the Authors

Pete Aven is a principal technologist at MarkLogic Corporation, where he assists organizations in understanding and implementing MarkLogic Enterprise NoSQL solutions. Pete has more than 15 years' experience in software engineering and system architecture with an emphasis on delivering large-scale, data-driven applications. He has helped solve data integration challenges for companies in industries such as publishing, healthcare, insurance, and manufacturing. Pete holds a bachelor's degree in linguistics and computer science from UCLA.

Diane Burley is the Chief Content Strategist for MarkLogic Corporation, a Silicon Valley–based multi-model database platform provider that enables organizations to quickly integrate data from silos to impact both the top and bottom lines. At MarkLogic, she is responsible for the overall content strategies, developing the frameworks, processes, procedures, and technologies that impact multichannel delivery of content and reports across all departments.