Chapter 7

Operational Databases

In This Chapter

- Taking a look at the relational database
- Examining nonrelational and key-value pair databases
- Exploring document and columnar databases
- Getting to know graph and spatial databases
- Pursuing the polyglot

Big data is becoming an important element in the way organizations are leveraging high-volume data at the right speed to solve specific data problems. However, big data does not live in isolation. To be effective, companies often need to be able to combine the results of big data analysis with the data that exists within the business. In other words, you can't think about big data in isolation from operational data sources. There are a variety of important operational data services. In this chapter, we provide an explanation of what these sources are so that you can understand how the data inevitably will be used in conjunction with big data solutions.

One of the most important services provided by operational databases (also called *data stores*) is persistence. Persistence guarantees that the data stored in a database won't be changed without permissions and that it will available as long as it is important to the business. What good is a database if it cannot be trusted to protect the data you put in it? Given this most important requirement, you must then think about what kind of data you want to persist, how can you access and update it, and how can you use it to make business decisions. At this most fundamental level, the choice of your database engines is critical to your overall success with your big data implementation.

The forefather of persistent data stores is the relational database management system, or RDBMS. In its infancy, the computing industry used what are now considered primitive techniques for data persistence. In essence, these are the systems of record and are foundational to how companies store data about everything from customer transactions to the details of the operating the business. Even though the underlying technology has been around for quite some time, many of these systems are in operation today because the businesses they support are highly dependent on the data. To replace them would be akin to changing the engines of an airplane on a transoceanic flight. You may recall the “flat files” or “network” data stores that were prevalent before 1980 or so. Although these mechanisms were useful, they were very difficult to master and always required system programmers to write custom programs to manipulate the data. The relational model is still in wide usage today and has an important role to play in the evolution of big data.

Relational databases are built on one or more relations and are represented by tables. These tables are defined by their columns, and the data is stored in the rows. The primary key is often the first column in the table. The consistency of the database and much of its value are achieved by “normalizing” the data. As the name implies, normalized data has been converted from native format into a shared, agreed upon format. For example in one database you might have “telephone” as XXX-XXX-XXXX while in another it might be XXXXXXXXXX. To achieve a consistent view of the information, the field will need to be normalized to one form or the other. Five levels of standards exist for normalization. The choice of normal form is often relegated to the database designer and is mostly invisible to the end users. The collection of tables, keys, elements, and so on is referred to as the database *schema.*

Over the years, the structured query language (SQL) has evolved in lock step with RDBMS technology and is the most widely used mechanism for creating, querying, maintaining, and operating relational databases. These tasks are referred to as CRUD: Create, retrieve, update, and delete are common, related operations you can use directly on a database or through an application programming interface (API). Although originally devised for use with RDBMS, the popularity of SQL has also made it prevalent among nonrelational databases, as we cover later in this chapter.

How the relational database evolved

Throughout the history of the relational database, many specialty database technologies appeared specifically to address shortcomings in early RDBMS products. We witnessed the emergence of object databases, content databases, data warehouses, data marts, and others. For companies that needed these new capabilities, they created independent solutions and integrated these new solutions with the existing RDBMS applications. This was tedious, clumsy, and costly. Over time, RDBMSs embraced these new technologies and embedded them in their core product offerings, eliminating the necessity to include additional, now redundant, solutions. We suspect this will occur with big data as well. Given the fundamental differences between big data and traditional data solutions, the encapsulation of big data technologies into RDBMSs will take a few years. In contrast, we are already beginning to see the big data technologies embrace SQL and other traditional RDBMS features as peers to MapReduce.

RDBMSs Are Important in a Big Data Environment

In companies both small and large, most of their important operational information is probably stored in RDBMSs. Many companies have different RDBMSs for different areas of their business. Transactional data might be stored in one vendor's database, while customer information could be stored in another. Knowing what data is stored and where it is stored are critical building blocks in your big data implementation. It is not likely you will use RDBMSs for the core of the implementation, but you will need to rely on the data stored in RDBMSs to create the highest level of value to the business with big data. Although many different commercial relational databases are available from companies like Oracle, IBM, and Microsoft, you need to understand an open source relational database called PostgreSQL.

PostgreSQL relational database

PostgreSQL (`www.postgresql.org`) is the most widely used open source relational database. It was originally developed at the University of California at Berkeley and has been under active development as an open source project for more than 15 years. Several factors contribute to the popularity of PostgreSQL. As an RDBMS with support for the SQL standard, it does all the things expected in a database product, plus its longevity and wide usage have made it "battle tested." It is also available on just about every variety of operating system, from PCs to mainframes.

Providing the basics and doing so reliably are only part of the story. PostgreSQL also supports many features only found in expensive proprietary RDBMSs, including the following:

- Capability to directly handle "objects" within the relational schema
- Foreign keys (referencing keys from one table in another)
- Triggers (events used to automatically start a stored procedure)
- Complex queries (subqueries and joins across discrete tables)
- Transactional integrity
- Multiversion concurrency control

The real power of PostgreSQL is its extensibility. Users and database programmers can add new capabilities without affecting the fundamental operation or reliability of the database. Possible extensions include

- Data types
- Operators
- Functions
- Indexing methods
- Procedural languages

This high level of customization makes PostgreSQL desirable when rigid, proprietary products won't get the job done. It is infinitely extensible.

Finally, the PostgreSQL license permits modification and distribution in any form, open or closed source. Any modifications can be kept private or shared with the community as you wish.

Although relational databases (including PostgreSQL) play a key role in the big data "enterprise," you also have some alternative approaches.

Nonrelational Databases

Nonrelational databases do not rely on the table/key model endemic to RDBMSs. A number of nonrelational database technologies are covered throughout this chapter, each with its own set of unique capabilities focused on specific problems outside the scope of traditional RDBMSs. In short, specialty data in the big data world requires specialty persistence and data manipulation techniques. Although these new styles of databases offer some answers to your big data challenges, they are not an express ticket to the finish line.

One emerging, popular class of nonrelational database is called not only SQL (NoSQL). Originally the originators envisioned databases that did not require the relational model and SQL. As these products were introduced into the market, the definition softened a bit and now they are thought of as "not only SQL," again bowing to the ubiquity of SQL. The other class is databases that do not support the relational model, but rely on SQL as a primary means of manipulating the data within. Even though relational and nonrelational databases have similar fundamentals, how the fundamentals are accomplished creates the differentiation. Nonrelational database technologies have the following characteristics in common:

- **Scalability:** In this instance, we are referring to the capability to write data across multiple data stores simultaneously without regard to physical limitations of the underlying infrastructure. Another important dimension is seamlessness. The databases must be able to expand and contract in response to data flows and do so invisibly to the end users.
- **Data and Query model:** Instead of the row, column, key structure, nonrelational databases use specialty frameworks to store data with a requisite set of specialty query APIs to intelligently access the data.
- **Persistence design:** Persistence is still a critical element in nonrelational databases. Due to the high velocity, variety, and volume of big data, these databases use difference mechanisms for persisting the data. The highest performance option is "in memory," where the entire database is kept in the very fast memory system of your servers.
- **Interface diversity:** Although most of these technologies support RESTful APIs as their "go to" interface, they also offer a wide variety of connection mechanisms for programmers and database managers, including analysis tools and reporting/visualization.
- **Eventual Consistency:** While RDBMS uses ACID (Atomicity, Consistency, Isolation, Durability) as a mechanism for ensuring the consistency of data, non-relational DBMS use BASE. BASE stands for Basically Available, Soft state, and Eventual Consistency. Of these, eventual consistency is most important because it is responsible for conflict resolution when data is in motion between nodes in a distributed implementation. The data state is maintained by the software and the access model relies on basic availability.

Next we examine some of the most popular styles and the open source implementations of nonrelational databases.

Key-Value Pair Databases

By far, the simplest of the NoSQL databases are those employing the key-value pair (KVP) model. KVP databases do not require a schema (like RDBMSs) and offer great flexibility and scalability. KVP databases do not offer ACID (Atomicity, Consistency, Isolation, Durability) capability, and require implementers to think about data placement, replication, and fault tolerance as they are not expressly controlled by the technology itself. KVP databases are not typed. As a result, most of the data is stored as strings. Table 7-1 lists some sample key-value pairs.

Table 7-1 Sample Key-Value Pairs

Key	*Value*
Color	Blue
Libation	Beer
Hero	Soldier

This is a very simplified set of keys and values. In a big data implementation, many individuals will have differing ideas about colors, libations, and heroes, as presented in Table 7-2.

Table 7-2 Big Data Key-Value Pairs

Key	*Value*
FacebookUser12345_Color	Red
TwitterUser67890_Color	Brownish
FoursquareUser45678_Libation	"White wine"
Google+User24356_Libation	"Dry martini with a twist"
LinkedInUser87654_Hero	"Top sales performer"

As the number of users increases, keeping track of precise keys and related values can be challenging. If you need to keep track of the opinions of millions of users, the number of key-value pairs associated with them can increase exponentially. If you do not want to constrain choices for the values, the generic string representation of KVP provides flexibility and readability.

You might need some additional help organizing data in a key-value database. Most offer the capability to aggregate keys (and their related values) into a collection. Collections can consist of any number of key-value pairs and do not require exclusive control of the individual KVP elements.

Riak key-value database

One widely used open source key-value pair database is called Riak (`http://wiki.basho.com`). It is developed and supported by a company called Basho Technologies (`www.basho.com`) and is made available under the Apache Software License v2.0.

Riak is a very fast and scalable implementation of a key-value database. It supports a high-volume environment with fast-changing data because it is lightweight. Riak is particularly effective at real-time analysis of trading in financial services. It uses "buckets" as an organizing mechanism for collections of keys and values. Riak implementations are clusters of physical or virtual nodes arranged in a peer-to-peer fashion. No master node exists, so the cluster is resilient and highly scalable. All data and operations are distributed across the cluster. Riak clusters have an interesting performance profile. Larger clusters (with more nodes) perform better and faster than clusters with fewer nodes. Communication in the cluster is implemented via a special protocol called Gossip. Gossip stores status information about the cluster and shares information about buckets.

Riak has many features and is part of an ecosystem consisting of the following:

- **Parallel processing:** Using MapReduce, Riak supports a capability to decompose and recompose queries across the cluster for real-time analysis and computation.
- **Links and link walking:** Riak can be constructed to mimic a graph database using links. A link can be thought of as a one-way connection between key-value pairs. Walking (following) the links will provide a map of relationships between key-value pairs.
- **Search:** Riak Search has a fault-tolerant, distributed full-text searching capability. Buckets can be indexed for rapid resolution of value to keys.
- **Secondary indexes:** Developers can tag values with one or more key field values. The application can then query the index and return a list of matching keys. This can be very useful in big data implementations because the operation is atomic and will support real-time behaviors.

Riak implementations are best suited for

- User data for social networks, communities, or gaming
- High-volume, media-rich data gathering and storage
- Caching layers for connecting RDBMS and NoSQL databases
- Mobile applications requiring flexibility and dependability

Document Databases

You find two kinds of document databases. One is often described as a repository for full document-style content (Word files, complete web pages, and so on). The other is a database for storing document components for permanent storage as a static entity or for dynamic assembly of the parts of a document. The structure of the documents and their parts is provided by JavaScript Object Notation (JSON) and/or Binary JSON (BSON). Document databases are most useful when you have to produce a lot of reports and they need to be dynamically assembled from elements that change frequently. A good example is document fulfillment in healthcare, where content composition will vary based on member profile (age, residency, income level), healthcare plan, and government program eligibility. For big data implementations, both styles are important, so you should understand the details of each.

At its core, JSON is a data-interchange format, based on a subset of the JavaScript programming language. Although part of a programming language, it is textual in nature and very easy to read and write. It also has the advantage of being easy for computers to handle. Two basic structures exist in JSON, and they are supported by many, if not all, modern programming languages. The first basic structure is a collection of name/value pairs, and they are represented programmatically as objects, records, keyed lists, and so on. The second basic structure is an ordered list of values, and they are represented programmatically as arrays, lists, or sequences. BSON is a binary serialization of JSON structures designed to increase performance and scalability.

Document databases are becoming a gold standard for big data adoption, so we examine two of the most popular implementations.

MongoDB

MongoDB (`www.mongodb.com`) is the project name for the “hu(mongo)us database” system. It is maintained by a company called 10gen as open source and is freely available under the GNU AGPL v3.0 license. Commercial licenses with full support are available from 10gen (`www.10gen.com`).

MongoDB is growing in popularity and may be a good choice for the data store supporting your big data implementation. MongoDB is composed of databases containing “collections.” A collection is composed of “documents,” and each document is composed of fields. Just as in relational databases, you can index a collection. Doing so increases the performance of data lookup. Unlike other databases, however, MongoDB returns something called a “cursor,” which serves as a pointer to the data. This is a very useful capability because it offers the option of counting or classifying the data without extracting it. Natively, MongoDB supports BSON, the binary implementation of JSON documents.

MongoDB is also an ecosystem consisting of the following elements:

- High-availability and replication services for scaling across local and wide-area networks.
- A grid-based file system (GridFS), enabling the storage of large objects by dividing them among multiple documents.
- MapReduce to support analytics and aggregation of different collections/documents.
- A sharding service that distributes a single database across a cluster of servers in a single or in multiple data centers. The service is driven by a shard key. The shard key is used to distribute documents intelligently across multiple instances.
- A querying service that supports ad hoc queries, distributed queries, and full-text search.

Effective MongoDB implementations include

- High-volume content management
- Social networking
- Archiving
- Real-time analytics

CouchDB

Another very popular nonrelational database is CouchDB (`http://couchdb.apache.org`). Like MongoDB, CouchDB is open source. It is maintained by the Apache Software Foundation (`www.apache.org`) and is made available under the Apache License v2.0. Unlike MongoDB, CouchDB was designed to mimic the web in all respects. For example, CouchDB is resilient to network dropouts and will continue to operate beautifully in areas where network connectivity is spotty. It is also at home on a smartphone or in a data center. This all comes with a few trade-offs. Because of the underlying web mimicry, CouchDB is high latency resulting in a preference for local data storage. Although capable of working in a non-distributed manner, CouchDB is not well suited to smaller implementations. You must determine whether these trade-offs can be ignored as you begin your big data implementation.

CouchDB databases are composed of documents consisting of fields and attachments as well as a "description" of the document in the form of metadata that is automatically maintained by the system. The underlying technology features all ACID capabilities that you are familiar with from the RDBMS world. The advantage in CouchDB over relational is that the data is packaged and ready for manipulation or storage rather than scattered across rows and tables.

CouchDB is also an ecosystem with the following capabilities:

- **Compaction:** The databases are compressed to eliminate wasted space when a certain level of emptiness is reached. This helps performance and efficiency for persistence.
- **View model:** A mechanism for filtering, organizing, and reporting on data utilizing a set of definitions that are stored as documents in the database. You find a one-to-many relationship of databases to views, so you can create many different ways of representing the data you have "sliced and diced."
- **Replication and distributed services:** Document storage is designed to provide bidirectional replication. Partial replicas can be maintained to support criteria-based distribution or migration to devices with limited connectivity. Native replication is peer based, but you can implement Master/Slave, Master/Master, and other types of replication modalities.

Effective CouchDB implementations include

- High-volume content management
- Scaling from smartphone to data center
- Applications with limited or slow network connectivity

Columnar Databases

Relational databases are *row oriented,* as the data in each row of a table is stored together. In a columnar, or column-oriented database, the data is stored *across* rows. Although this may seem like a trivial distinction, it is the most important underlying characteristic of columnar databases. It is very easy to add columns, and they may be added row by row, offering great flexibility, performance, and scalability. When you have volume and variety of data, you might want to use a columnar database. It is very adaptable; you simply continue to add columns.

HBase columnar database

One of the most popular columnar databases is HBase (`http://hbase.apache.org`). It, too, is a project in the Apache Software Foundation distributed under the Apache Software License v2.0. HBase uses the Hadoop file system and MapReduce engine for its core data storage needs. For more on MapReduce, refer to Chapter 8; for more on Hadoop, check out Chapter 9.

The design of HBase is modeled on Google's BigTable (an efficient form of storing nonrelational data). Therefore, implementations of HBase are highly scalable, sparse, distributed, persistent multidimensional sorted maps. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes. When your big data implementation requires random, real-time read/write data access, HBase is a very good solution. It is often used to store results for later analytical processing.

Important characteristics of HBase include the following:

- **Consistency:** Although not an "ACID" implementation, HBase offers strongly consistent reads and writes and is not based on an eventually consistent model. This means you can use it for high-speed requirements as long as you do not need the "extra features" offered by RDBMS like full transaction support or typed columns.
- **Sharding:** Because the data is distributed by the supporting file system, HBase offers transparent, automatic splitting and redistribution of its content.
- **High availability:** Through the implementation of region servers, HBase supports LAN and WAN failover and recovery. At the core, there is a master server responsible for monitoring the region servers and all metadata for the cluster.
- **Client API:** HBase offers programmatic access through a Java API.
- **Support for IT operations:** Implementers can expose performance and other metrics through a set of built-in web pages.

HBase implementations are best suited for

- High-volume, incremental data gathering and processing
- Real-time information exchange (for example, messaging)
- Frequently changing content serving

Graph Databases

The fundamental structure for graph databases is called "node-relationship." This structure is most useful when you must deal with highly interconnected data. Nodes and relationships support *properties,* a key-value pair where the data is stored. These databases are navigated by following the relationships. This kind of storage and navigation is not possible in RDBMSs due to the rigid table structures and the inability to follow connections between the data wherever they might lead us. A graph database might be used to manage geographic data for oil exploration or to model and optimize a telecommunications provider's networks.

Neo4J graph database

One of the most widely used graph databases is Neo4J (`www.neo4j.org`). It is an open source project licensed under the GNU public license v3.0. A supported, commercial version is provided by Neo Technology under the GNU AGPL v3.0 and commercial licensing. Neo4J is an ACID transaction database offering high availability through clustering. It is a trustworthy and scalable database that is easy to model because of the node-relationship properties' fundamental structure and how naturally it maps to our own human relationships. It does not require a schema, nor does it require data typing, so it is inherently very flexible.

With this flexibility comes a few limitations. Nodes cannot reference themselves directly. For example, you (as a node) cannot also be *your own* father or mother (as relationships), but you can be *a* father or mother. There might be real world cases where self-reference is required. If so, a graph database is not the best solution since the rules about self-reference are strictly enforced. While the replication capability is very good, Neo4J can only replicate entire graphs, placing a limit on the overall size of the graph (approximately 34 billion of nodes and 34 billion relationships).

Important characteristics of Neo4J include the following:

- **Integration with other databases:** Neo4J supports transaction management with rollback to allow seamless interoperability with nongraphing data stores.
- **Synchronization services:** Neo4J supports event-driven behaviors via an event bus, periodic synchronization using itself, or an RDBMS as the master, and traditional batch synchronization.
- **Resiliency:** Neo4J supports cold (that is, when database is not running) and hot (when it is running) backups, as well as a high-availability clustering mode. Standard alerts are available for integration with existing operations management systems.
- **Query language:** Neo4J supports a declarative language called Cypher, designed specifically to query graphs and their components. Cypher commands are loosely based on SQL syntax and are targeted at ad hoc queries of the graph data.

Neo4J implementations are best suited for

- Social networking
- Classification of biological or medical domains
- Creating dynamic communities of practice or interest

Spatial Databases

Whether you know it or not, you may interact with spatial data every day. If you use a smartphone or Global Positioning System (GPS) for directions to a particular place, or if you ask a search engine for the locations of seafood restaurants near a physical address or landmark, you are using applications relying on spatial data. Spatial data itself is standardized through the efforts of the Open Geospatial Consortium (OGC; `www.opengeospatial.org`), which establishes OpenGIS (Geographic Information System) and a number of other standards for spatial data.

This is important because spatial databases are implementations of the OGC standards, and your company might have specific needs met (or not met) by the standards. A spatial database becomes important when organizations begin to leverage several different dimensions of data to help make a decision. For example, a meteorologist doing research might want to store and evaluate data related to a hurricane, including temperature, wind speed, and humidity, and model those results in three dimensions.

In their simplest form, spatial databases store data about 2-dimensional, 2.5-dimensional, and 3-dimensional objects. You are probably familiar with 2D and 3D objects as we interact with them all the time. A 2D object has length and width. A 3D object adds depth to the length and width. A page from this book is a 2D object, while the entire book is a 3D object. What about 2.5D? 2.5D objects are a special type of spatial data. They are 2D objects with elevation as the extra "half" dimension. Most 2.5D spatial databases contain mapping information and are often referred to as Geographic Information Systems (GISs).

The atomic elements of spatial databases are lines, points, and polygons. They can be combined in any fashion to represent any object constrained by 2, 2.5, or 3 dimensions. Due to the special nature of spatial data objects, designers created indexing mechanisms (spatial indices) designed to support ad hoc queries and visual representations of the contents of the database. For example, a spatial index would answer the query "What is the distance between one point and another point?" or "Does a specific line intersect with a particular set of polygons?" If this seems like a huge problem, that's because it is. Spatial data may well represent the biggest big data challenge of all.

PostGIS/OpenGEO Suite

PostGIS (`www.postgis.org`) is an open source project maintained by Refractions Research (`www.refractions.net`) and is licensed under the GNU General Public License (GPL). PostGIS is also supplied as part of the OpenGeo Suite community edition and is offered and supported by OpenGeo (`www.opengeo.org`) under an enterprise license.

PostGIS is a little different than some of the other databases discussed in this chapter. It is a specialized, layered implementation running on the workhorse RDBMS PostgreSQL. This approach offers the best of both worlds. You get all the benefits of an SQL RDBMS (such as transactional integrity and ACID) and support for the specialized operations needed for spatial applications (reprojection, geodetic support, geometry conversion, and so on).

Although the database itself is very important, you will also require other pieces of technology to address spatial application requirements. Fortunately, PostGIS is part of an ecosystem of components designed to work together to address these needs. In addition to PostGIS, the OpenGEO Suite consists of the following:

- **GeoServer:** Implemented in Java, the GeoServer can publish spatial information from several of the major sources of spatial data on the web. It can integrate with Google Earth and also has an excellent web-based administrative front end.
- **OpenLayers:** A library for JavaScript that is useful for displaying maps and other representations of spatial data in a web browser. It can manipulate images from most of the mapping sources on the web, including Bing Maps, Google Maps, Yahoo! Maps, OpenStreetMap, and so on.
- **GeoExt:** Designed to make the map information from OpenLayers readily available to the web application developer. GeoExt widgets can be used to create editing, viewing, styling, and other interactive web experiences.
- **GeoWebCache:** After you have the data in a server and can display it in a browser, you need to find a way to make it fast. GeoWebCache is the accelerator. It caches chunks of image data (called tiles) and makes them available for rapid delivery to the display device.

While many of the uses of spatial data involve maps and locations, spatial data has many other contemporary and future applications, including

- Precise 3D modeling of the human body, buildings, the atmosphere, and so on
- Gathering and analysis of data from sensor networks
- Integration with historical data to examine 3D space/objects over time

Polyglot Persistence

The official definition of *polyglot* is "someone who speaks or writes several languages." The term is borrowed in this context and redefined as a set of applications that use several core database technologies, and this is the most likely outcome of your big data implementation planning. It is going to be difficult to choose one persistence style no matter how narrow your approach to big data might be. A polyglot persistence database is used when it is necessary to solve a complex problem by breaking that problem into segments and applying different database models. It is then necessary to aggregate the results into a hybrid data storage and analysis solution. A number of factors affect this decision:

- You are already using polyglot persistence in your existing workplace. If your enterprise or organization is large, you are probably using multiple RDBMSs, data warehouses, data marts, flat files, content management servers, and so on. This hybrid environment is common, and you need to understand it so that you can make the right decisions about integration, analytics, timeliness of data, data visibility, and so on. You need to understand all of that because you need to figure out how it is going to fit into your big data implementation.
- The most ideal of environments, where you have only one persistence technology, is probably not suited to big data problem solving. At the very least, you will need to introduce another style of database and other supporting technologies for your new implementation.
- Depending on the variety and velocity of your big data gathering, you may need to consider different databases to support one implementation. You should also consider your requirements for transactional integrity. Do you need to support ACID compliance or will BASE compliance be sufficient?

As an example, suppose that you need to identify all the customers for your consumer hard goods product who have purchased in the last 12 months and have commented on social websites about their experience — AND whether they have had any support cases (when, how many, how resolved), where they acquired the product, how it was delivered (and was the delivery routing cost efficient with respect to energy consumption?), what they paid, how they paid, whether they have been to the company website, how many times, what they did on the site, and so on. Then suppose that you want to offer them a promotional discount to their smartphone when they are entering one of your (or one of your partners') retail stores.

This is a big data challenge at its best. Multiple sources of data with very different structures need to be collected and analyzed so that you can get the answers to these questions. Then you need determine whether the customers qualify for the promotion and, in real time, push them a coupon offering them something new and interesting.

This type of problem cannot be solved easily or cost-effectively with one type of database technology. Even though some of the basic information is transactional and probably in an RDBMS, the other information is nonrelational and will require at least two types of persistence engines (spatial and graph). You now have polyglot persistence.