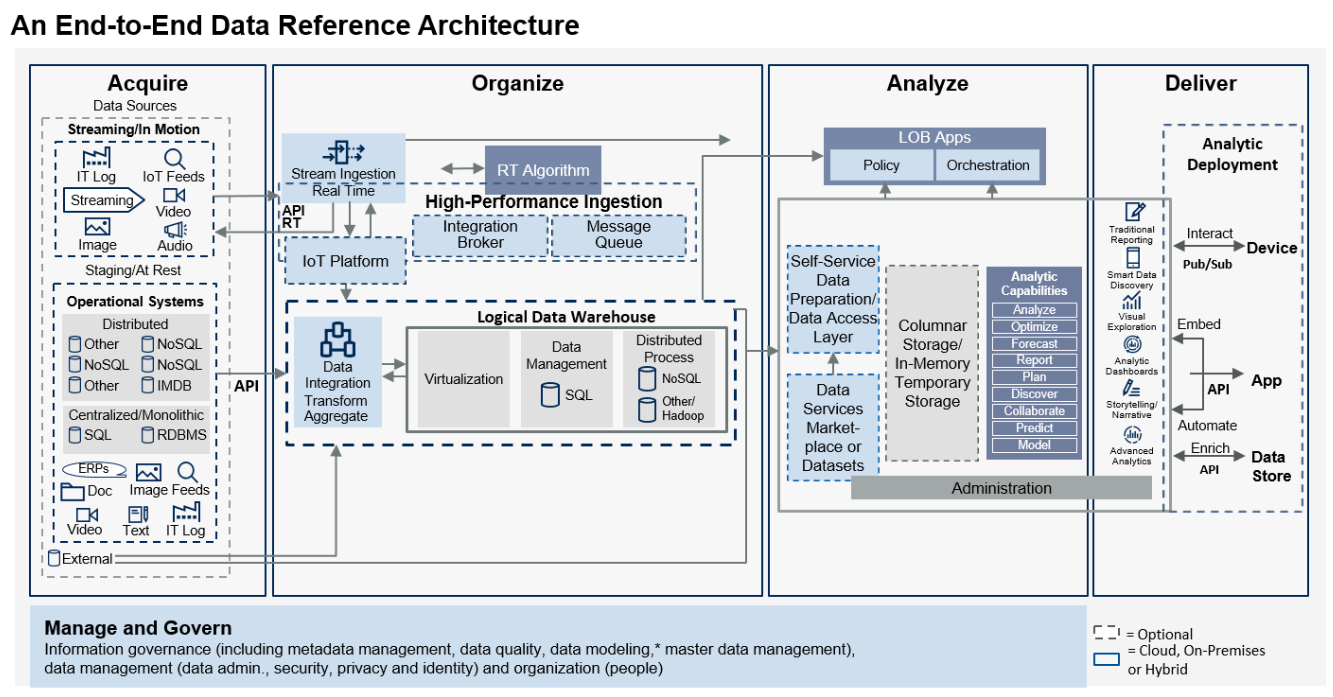
Data Modeling to Support End-to-End Data Architectures

Data models are foundational for business initiatives such as technology innovation, data quality, data lineage, gathering business information requirements and data governance. Technology paradigms can distort human perceptions of information. When misapplied such paradigms lead to incomplete, brittle or downright erroneous views of an organization’s information requirements. Often necessary is the following:

1. Create user-approved data models (Just not too much details )
2. Achieve enterprise wide consensus about the meaning of data by consulting with multiple subject-matter experts from as many departments and user bases as necessary.
3. Separate the solution design (not vendor specific ) from problem/use case description.

**Analysis**

Data modeling is a family of techniques used to describe the kinds of information that are important to an enterprise both technical and nontechnical. As shown here, the figure calls out the inclusion of data modeling as a foundational technique whose scope spans the entire architecture.



This document provides a contemporary update on many of the ways that data modeling can help technical professionals to create data architectures that automate business operations and provide actionable information to business analysts who depend on accurate, timely information.

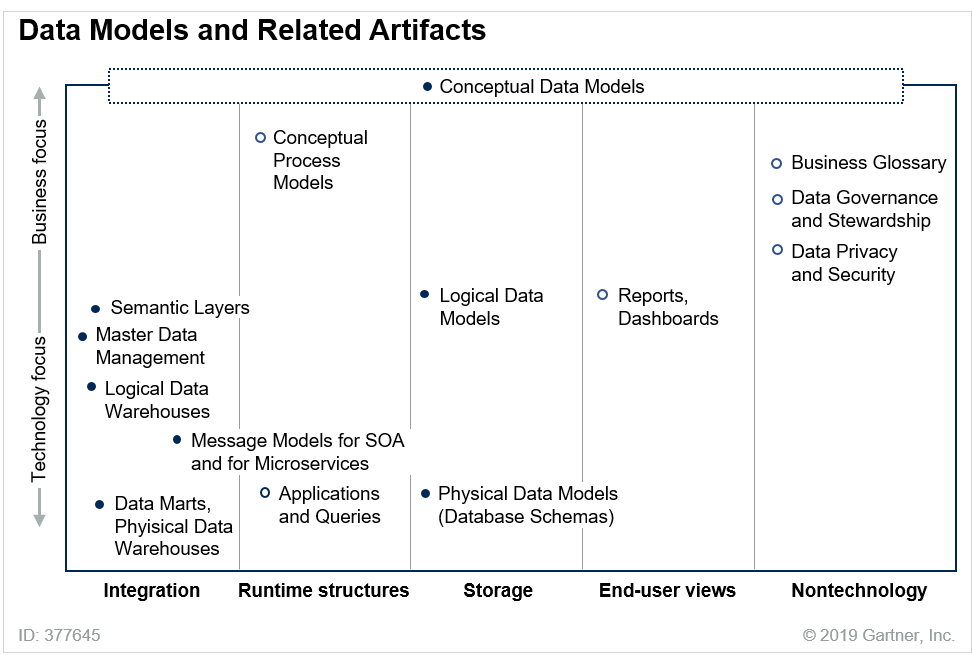
* Use cases for data modeling
* Strengths of data modeling
* Weaknesses of data modeling
* Guidance on using data modeling within a modern information architecture

Use Cases for Data Modeling

* Conceptual data models: These express business data requirements independent of any technological solutions created to satisfy those requirements.
* Nontechnological artifacts and business programs: Data models are essential inputs to nontechnological artifacts, such as business process models and business glossaries. They also provide a foundational view of data necessary for business programs, such as data governance, quality, privacy and security.
* Storage schemas: These include models for designing databases, both relational and nonrelational.
* Runtime artifacts: Runtime artifacts requiring data models include object models (most commonly modeled as UML class diagrams). They also include message models for interservice data transmission in service-oriented architectures (SOAs) and microservices architectures (typically modeled as serialized objects expressed in XML schemas or JavaScript Object Notation [JSON]).
* Data-integration artifacts: The data-integration artifacts requiring data models include:

1. Various styles of data warehouses, including relational, multidimensional and data vault.
2. Various data-virtualization approaches, including logical data warehouses and semantic layers atop data lakes.
3. Data-movement solutions based on extract, transform and load (ETL) and extract, load and transform (ELT) approaches

* End-user representations of data: These models exist to serve dashboards and reports directly. They define structures that are directly queried by reporting and visualization tools.



The figure shows three important ways to classify these artifacts:

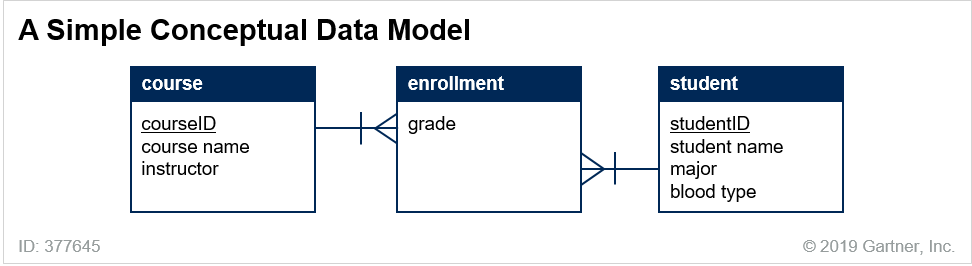
* The horizontal axis arranges five categories listed above.
* The vertical axis classifies artifacts as business-focused or technology-focused.
* The interior of the circles (filled-in or blank) indicates whether the artifact is a data model (filled-in circles) or an ancillary artifact that is closely related to a data model (blank circles).

#### **Conceptual Data Models**

A conceptual data model expresses the kinds of data an organization values. That is, it asserts what categories exist, the traits describing those categories, the ways those categories interrelate, and the ways to distinguish category members from each other. During discussions among data modelers, those parts of a conceptual model are called **entities**, **attributes**, **relationships** and **identifiers**, respectively.

A conceptual data model expresses the kinds of data an organization values.

For example Figure below shows a conceptual data model[1](https://www.gartner.com/document/3901169?ref=TrackRecommendedEmail&refval=1555684733333#dv_1_the_conceptual) with three entities, (course, enrollment and student), two relationships (the lines emerging from the enrollment entity), and eight attributes (three for course, one for enrollment and four for student). In addition, for each entity the model shows how to distinguish the members of that category from each other. That is, the model shows an identifier for each entity. The identifier, of course, is courseID, indicated with underlining. Likewise, the identifier of student is studentID. The identifier of enrollment is the combination of course and student, indicated by the perpendicular bars on the relationships connecting enrollment to course and student showing a many to many relationship.



Keep in mind that this conceptual models scope of an individual model can span multiple applications or user communities, its not the final design but a consensus of a design that lacks Technical Details.

++++++++

##### **Process and Workflow**

Ultimately, the point of the conceptual data model is to establish contextual consensus among a user community about what entities, attributes and relationships exist, and what business vocabulary should be used to refer to them. Once that business vocabulary has been established and institutionalized, those words are now available to many downstream consumers, both technical and nontechnical. Among the nontechnical consumers are business professionals who design process.

The conceptual data model should be a focal point of any effort to design or revise business processes. Process designers can describe their processes with explicit references to the named parts of the data model. In the following list, the names from a conceptual data model appear in italics:

* Step 1 reads data about a customer, including the customer’s per-order purchase limit.
* Step 2 creates data describing a new purchase order for that customer.
* Step 3 reads information about the price of each product the customer wants to buy.
* Step 4 calculates the total price of the order and compares it to the customer’s per-order purchase limit.

In certain cases, scrutinizing the data used by processes can reveal shortcomings in the models (e.g., when a process step refers to a kind of data this is not present on the conceptual data model).

Some schools of thought prefer to design process models before data models, or to design both simultaneously. This can work in some cases, provided the project includes an influential and disciplined data professional. In any case, producing a high-fidelity data model — one that hews closely to users’ perceptions of their data — is an uphill slog. That’s because conceptual data models should reflect a processing-independent expression of the data. (Rare indeed is the corporate data that exists to serve a single business process. Data models that describe data only in the context of specific processes give a wrongly circumscribed view of that data. More significantly, such data models work at cross purposes with attempts to honor principles of software engineering like application-data independence.)

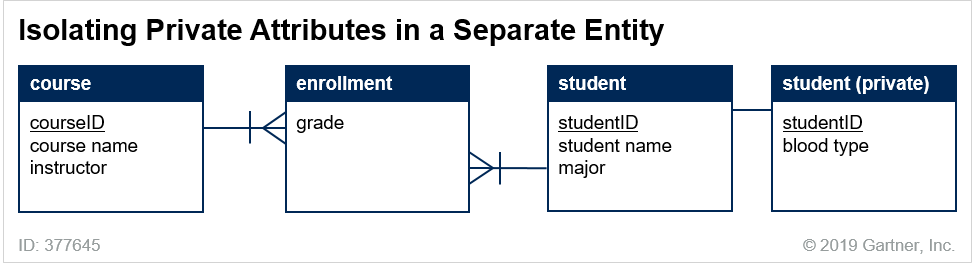
##### **Business Rules**

The process steps shown in the previous section referred to parts of a conceptual data model by name. More generally, any business rule (a constraint, process rule, validation rule, access-authorization rule, etc.) can refer to specific parts of a conceptual data model by name. Once they have been formally expressed and endorsed by the user community, the parts of a conceptual data model become the operands of the logical expressions used to articulate business rules.

##### **Business Data Programs**

Programs like data governance, data quality, data privacy and data security are properly the province of business, realized with the help and support of IT. Such programs interact with the conceptual data models in several ways.

1. First, the high-fidelity data models provide a foundational view of the data that will be subjected to these programs.
2. Second, programs — especially the governance program — should include policies that define the standards, policies and practices that apply when data models are created, deployed and subsequently revised.
3. Third, the needs of the programs often impose requirements on the data models themselves. For example, data security and privacy requirements can be reflected on data models as seen below for student.



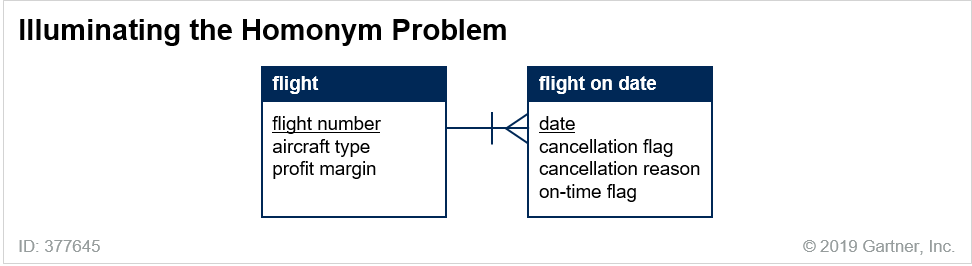
Business Glossaries

A business glossary is a list of a business’s concepts and terminology along with the definitions of the terms. Some organizations use business glossary software to manage these terms; others just use generic tools such as wiki pages. Take the following statement:

Flight 888 was canceled for thunderstorms, which is unfortunate because it has our highest profit margins.

One concept is a category of things that are subject to cancellation from thunderstorms or other weather events. The other concept is the category of things that have profit margins, this express what is known as the Homonym Problem.

One concept is a category of things that are subject to cancellation from thunderstorms or other weather events. The other concept is the category of things that have profit margins.



Data Persistence

A conceptual data model expresses a business problem — the data that an organization needs to remember. To describe the solution — the actual storage of that valuable data — technology models for database schemas are needed.

Logical and Physical Modeling

Because solutions are built with specific technologies (e.g., relational, object-oriented, graph databases, etc.), technology data models must be built with technology-specific building blocks. The first step toward designing a technology solution is the logical data model.

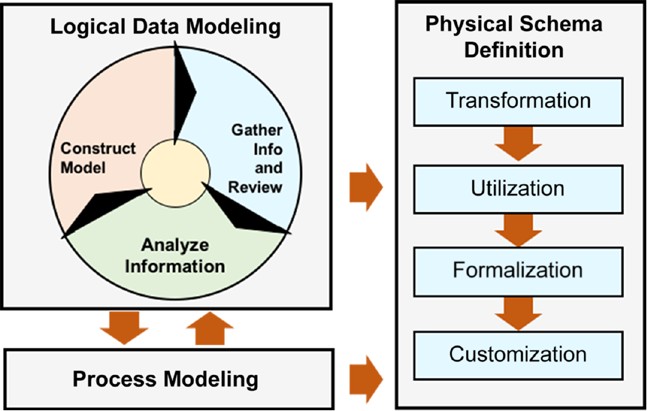
**Logical Models**

* During logical modeling, data professionals decide how best to use the available technological building blocks to satisfy the demands set forth on the conceptual data model. Table 1 shows how some of these demands might be met with the building blocks of a few technologies.

| Conceptual Construct | Relational Construct | XML Construct | Object-Oriented Construct |
| --- | --- | --- | --- |
| Entity | Table | Element | Class |
| Attribute | Column | Attribute | Member (excluding methods) |
| Relationship | Foreign key | XREF, element nesting | Association, aggregation, composition, etc. |
| Identifier | Primary key | Location within XML file | GUID |

Logical Data Modeling

The *Logical Data Modeling* (LDM) phase consists of an iterative approach focused on identifying entities and then determining the attributes and relationships supporting those entities.

Here is a overview to Logical to Physical data modeling.

The first phase, *Usage-Driven Database Design*: *Logical Data Modeling* (U3D:LDM), follows the entity-relationship approach to understand and document the data the database will store and the applications will use.

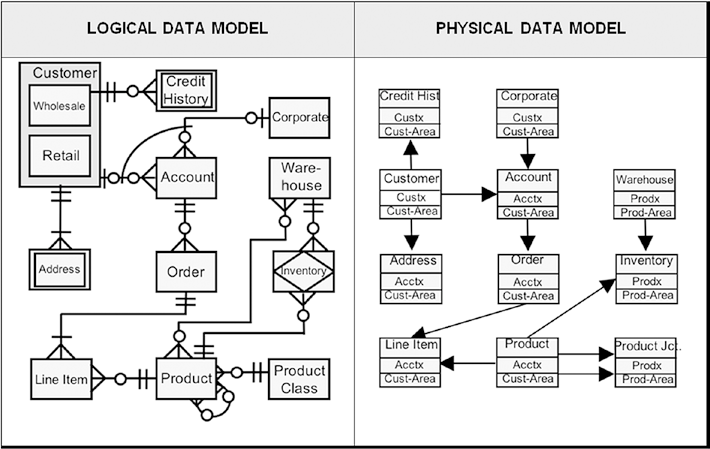
The second phase, *Usage-Driven Database Design*: *Physical Schema Definition* (U3D:PSD), converts the logical data model into a fully functional database schema.

The *Process Modeling* step in The logical data modeler and the logical process modeler should be in constant communication, sharing relevant information. The Process Modeling step is also a source of information for the Physical Schema Definition phase.

**Physical Models**

A logical model constitutes a commitment to a technology category (e.g., relational or object-oriented). A physical model constitutes a further commitment to a specific product within that category (e.g., MySQL or SQL Server). On a physical data model, you make product-specific decisions about how to implement the solution. The separation of logical and physical models is necessary because within a technology category, products vary in the features they offer. For example, not every relational database management system (RDBMS) offers the same roster of index types. It is when designing physical models that you choose nitty-gritty technological details such as indexing techniques.

The distinction between logical and physical modeling will remain important, especially as new database implementation paradigms arise. However, modern in-memory techniques are making some forms of physical data modeling simpler. That’s because in-memory databases can eliminate the need for indexes, aggregates and materialized views — some of the very constructs that are traditionally designed during physical modeling.



The first step, *Transformation*, turns the logical data model into a physical data model by converting the logical objects (entity, attribute, and relationship) into the physical database objects (record, data field, and linkage).

The second step, *Utilization*, takes the processes defined in Logical Process Modeling and Physical Process Modeling and merges them with the physical data model. The result is a modified or *rationalized physical data model* that represents how the applications will use the database. This is the step where the database design missing link—the inability to adequately and efficiently merge data and process into a single effective database design—is eliminated.

The third step, *Formalization*, identifies the data architecture, database management

system, and version that will be used and, combining them with the rationalized physical data model, creates a working database schema.

The fourth and last step, *Customization*, analyzes and improves the performance of

the database schema using a number of hardware and software techniques.

U3D is data architecture independent (hierarchical, network, inverted, relational, object-oriented, NoSQL, etc.), system software independent (z/OS, UNIX, Windows, Linux, OS X, etc.), and hardware independent (mainframe, server, PC). It works with Oracle, SQL Server, Cassandra, IMS, DB2…and even flat files.

*In relational, the relationships between entities is express in Cardinality* as the maximum number of occurrences of an entity type, usually expressed as one or many, that can be related to an occurrence of another entity type. There are four cardinality states.

* *One-to-one (1:1)*: An occurrence of entity A can relate, at most, to one occurrence of entity B, and an occurrence of entity B can

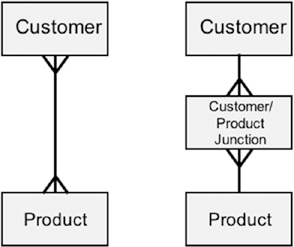
relate, at most, to one occurrence of entity A. For example, a husband can have only one wife, and a wife only one husband.

* *One-to-many (1:N)*: One occurrence of entity A can relate to many occurrences of entity B, but an occurrence of B can relate to

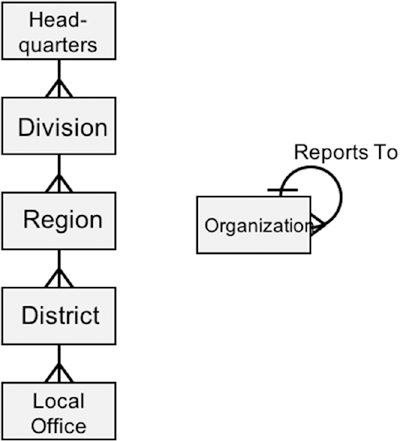
only one occurrence of A. For example, a mother can have many children, but a child can have only one mother.

* *Many-to-many (M:N)*: An occurrence of entity A can relate to multiple occurrences of entity B, while an occurrence of entity B can relate to many occurrences of entity A. For example, an uncle

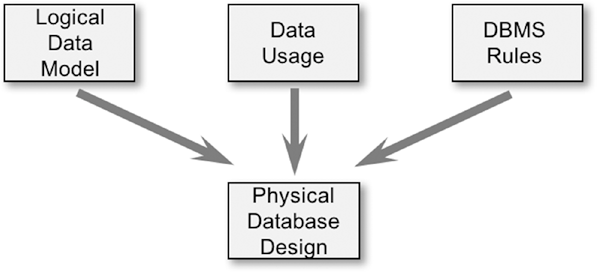
can have many nephews while a nephew can have many uncles.

* *Many-to-one (M:1)*: Because relationships are bidirectional, a many-to-one relationship is the inverse of a one-to-many relationship.

* *Self referencing (S-S): for example Organization reports to , if levels are missing, this presents a modeling problem.*



The physical data model is not DBMS architecture specific (relational, network, hierarchical, object, inverted, etc.), product specific (Oracle, IMS, SQL Server, Model 204, Cassandra, etc.), or release specific (Oracle 12, SQL Server 2014, DB2 10.5, etc.). In truth, a physical data model is a rather skimpy view of stored data. While it does deal with records and data fields, there is no capability to express such concepts as access methods or physical storage components. These must wait until later in the phase.



**Forward and Reverse Engineering**

**Forward engineering** is the process of translating a data model into another model that contains more implementation detail. Forward engineering of data models can take many forms, including (but not limited to):

* Translating a conceptual data model into a logical data model for relational technology
* Translating a conceptual data model into a logical data model for dimensional technology
* Translating a conceptual data model into a logical data model for XML technology
* Translating a relational logical data model into a physical RDBMS schema
* Translating a NoSQL logical model into an Apache Cassandra schema

Many data-modeling tools can perform some forward-engineering translations automatically. Certain target technologies are well-covered; virtually all data-modeling tools can forward engineer models to database schemas for Oracle Database, Microsoft SQL Server and MySQL. Tool coverage for other technologies is more hit-and-miss. Not all modeling tools, for example, can forward engineer data models into schemas for Cassandra or Neo4j.

**Reverse engineering** is the opposite process: It translates a data model that contains implementation detail into a logical model. Reverse engineering is useful in several contexts:

* When seeking to understand the workings of database solution whose initial design/requirements documents are no longer available (or of a solution that was implemented without formal design artifacts).
* When seeking to convert a deployed system from one DBMS technology to another. In very broad strokes, the three-step process is:

1. Reverse engineer the database structure of the existing system into a logical data model.
2. Revise or cleanup the logical data model as appropriate.
3. Forward engineer the improved logical data model into a database structure for the new target DBMS technology.

Database Technologies and Techniques

Database schema design is well-described in documentation from DBMS vendors, aftermarket documentation and computer science textbooks. This section does not duplicate that information. It does present a few suggestions for how data modelers ought to approach schema design.

**Relational Systems**

When a database schema is in a normal form, it ensures that data will be stored without redundancy. A hierarchy of normal forms exists, first normal form, second normal form, third and several beyond. The higher the normal form, the more redundancies have been eliminated from the schema.

* Normal forms are typically achieved through the process of normalization, but that is not the only technique.
* Normalization can work only if realistic data is available. This is always the case when designing data warehouses, because data exists in the upstream transactional and operational systems. But normalization is not always a viable technique during “greenfield” development of transactional systems, because the realistic data might not become available until after the system is deployed, perhaps weeks or months later. In such situations, conceptual data modeling becomes even more indispensable.

Relational systems are defined by the automony of their data contingent on the below, ACID is for transactions that insert, update, or delete database records. It guarantees the integrity of data.:

ACID stood for the following:

* *Atomicity*: Every part of a transaction must be executed before the transaction can be considered complete.
* *Consistency*: Any change to the database must be consistent with all validation rules.
* *Isolation*: Every transaction must be completed as though it were the only transaction, regardless of how many transactions there are and in what sequence they are executed. Isolation deals with

the notion of currency control.

* *Durability*: Once a transaction is committed, it stays committed. Failures from a loss of power to a computer, communications disruptions, or crashes of any type do not affect a completed

transaction.

*Normalization* is a process of reducing the structure of the model to a state such that data in any given record is totally dependent on the primary key of that record. This restriction ensures that if, for example, some data items are deleted, then all associated data items are also deleted, while all nonassociated data items are not.

To be in zero normal form (0NF):

* + 1. Every record must have a relational model–defined primary key.

To be in first normal form (1NF):

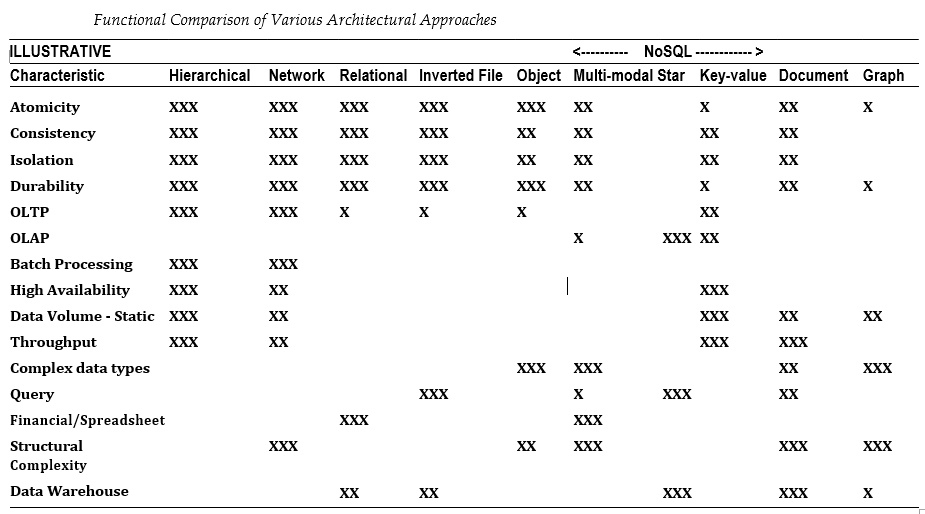
1. The record must be in zero normal form.
2. All multivalue data items (Codd calls them *repeating groups*) must be removed from the record.

To be in second normal form (2NF):

1. The record must be in first normal form.
2. Every nonkey data item must be fully functionally dependent on the primary key (no partial functional dependencies).

To be in third normal form (3NF):

1. The record must be in second normal form.
2. There can be no transitive functional dependencies.

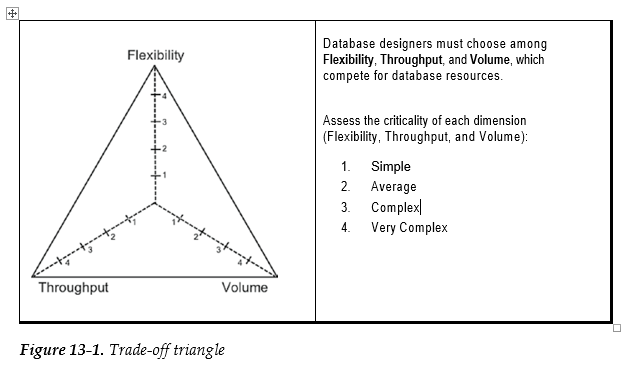


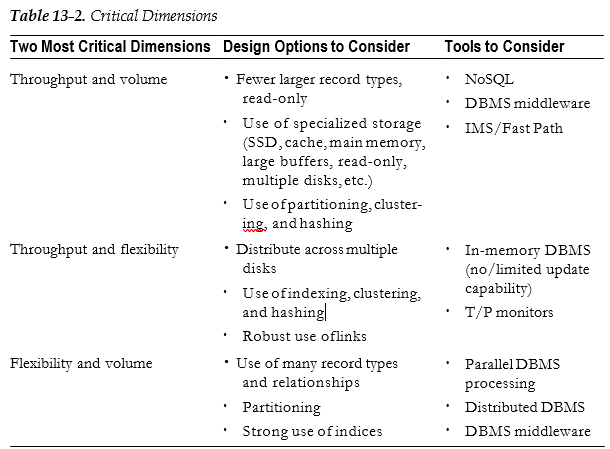
Trade-offs are everywhere, including database design. A good DBMS schema involves trade-offs related to three competing performance dimensions—flexibility, throughput, and volume.

* + *Flexibility* is the ability of the database system to support a broad range of known and unknown services and to easily adapt to business and technology changes.
  + *Throughput* is how quickly the database system can perform its function either in terms of response time for online applications or runtime for batch programs.
  + *Volume* is the number of objects/actions the database system can accommodate, such as the number of record types, or occurrences, it can support or the number of concurrent online

transactions it can handle.

These three dimensions can be easily represented as a triangle





### Denormalization

Denormalization is another favorite of NoSQL database systems, which like to cram a lot of data into a single record occurrence. They also like to add group data and repeating groups back into the parent record. If few customers have more than one address, then it might make sense to place the primary address in the customer record.

The purpose of normalization is to protect the database from ill-conceived inserts, updates, and deletes. However, if the database is read-only, then normalization is not needed. Data warehouses, which tend to be large and read-only, are prime candidates for denormalization.

### Get Rid of ACID

ACID (Chapter [8](http://dx.doi.org/10.1007/978-1-4842-2722-0_8)) is expensive. It requires that all database insert, update, and delete transactions follow at least most, if not all, of the following steps:

1. The data to be changed (and sometimes even the data stored around it on the same database page or file) must be locked so that others cannot access them while the change is occurring.
2. An image of the existing data (before image) is written (involving one or more disk writes) to a journal file before the data are changed, and another image of the data (after image) is saved (one or more disk writes) to a journal file after the change.
3. All the transaction steps taken are recorded to a separate log file (one or more disk writes).
4. All the writes are *flushed* to ensure that all the changes are physically on the disk and not stored in some buffer awaiting transfer to disk.

Updating a single database record occurrence could involve more than a dozen physical disk writes (not logical writes) before the actual record occurrence update is completed. In terms of resource utilization, a database update could require 10 or more times the resources of a simple database read.

Eliminating or reducing one or more of these steps can significantly speed up a database transaction and, if all goes well, nothing is lost. This is how many of the NoSQL systems obtain their speed. By not getting into locking, journaling, and logging the update, the speed of a transaction can be increased tenfold.

If you can live with the proclivities and vagaries of the non-ACID world, then you can, if your DBMS allows, turn off the ACID functions to improve performance at the cost of guaranteed data integrity.

NOSQL

NoSQL products are categorized by some authors as schema-less, meaning that there is no formal schema like you might find with a traditional DBMS. Although the statement is technically not true, it does capture an important characteristic of NoSQL systems. You could describe NOSQL as a series of stand-alone subschemas. This observation is driven by two common NoSQL features—its usage-driven nature and its single record type structure.

First, NoSQL systems place significantly more emphasis on data usage than traditional data management systems.

Second, a goal of a NoSQL database design is to have each usage scenario supported by a single NoSQL record type. Denormalization, specifically cramming all the user- required data into a single NoSQL record, is what gives NoSQL its speed . A single NoSQL record might contain multiple occurrences of multiple entities. The resulting NoSQL *fat record* can then be accessed with a single I/O.

Many NoSQL database management systems have given up the ACID guarantees for exceptional performance in one or another area. However, because so many developers and DBMS purchasers know about the benefits of ACID, the NoSQL community came

up with its own acronym: BASE. (Apparently, this community likes chemistry puns.) Yes, those super-fast or super-big DBMSs that fall short on the ACID standard can now possibly claim that they support the BASE model. What does BASE stand for? Why **b**asic **a**vailability, **s**oft state, and **e**ventual consistency!

* *Basic availability*: Data requests are not guaranteed for completeness or consistency.
* *Soft state*: The state of the system and its data are unknown, although it will probably be determinable in some future time.
* *Eventual consistency*: The system is, or will be, consistent but cannot be guaranteed to be consistent at any specific time.

Want to ensure that your data are valid? BASE systems will eventually figure it out, if you can wait. For all others, stick with ACID.

**Nonrelational Systems**

The various flavors of NoSQL database and their user cases are well-described elsewhere and will not be repeated here. (See [“Framework for Assessing NoSQL Databases.”](https://www.gartner.com/document/code/278868?ref=grbody&refval=3901169)) However, a few notes about how data modelers should think about NoSQL databases are in order.

First, (and as attested earlier), NoSQL schema design is not a substitute for conceptual data modeling.

Second, data modelers involved in a NoSQL project should focus on *why* a particular NoSQL solution has been chosen. The reasons fall into several categories:

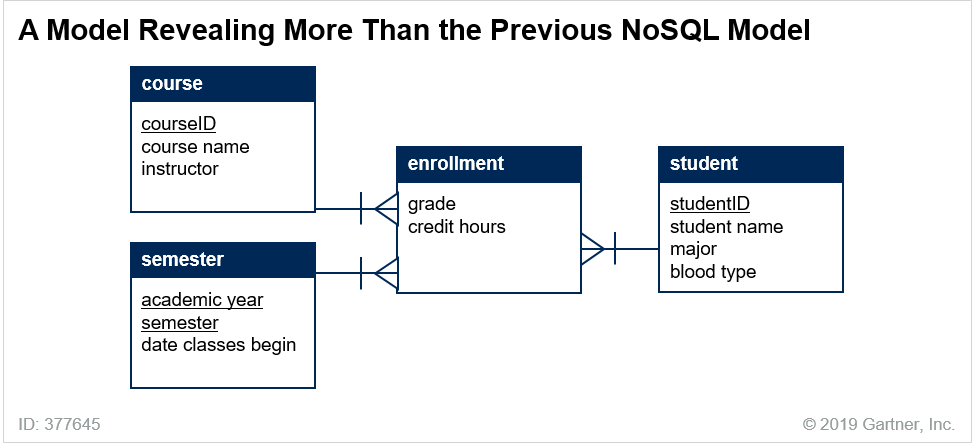
* The chosen NoSQL metamodel has an affinity with the to-be-managed data. For example, documents belong in document databases. Likewise, some data has an inherent affinity with the nodes-and-edges metamodel of graph databases. Such data belongs in graph databases.
* The data is highly structured, but the data model is highly flexible. In such cases, a database technique that combines the data and the metadata into a single file — RDF-based triple-stores are examples — is recommended.
* A NoSQL DBMS is preferred *despite* a lack of affinity with the to-be-managed data. The motivation could be performance advantages, or it could be because the organization wants to leverage programmer expertise in NoSQL technologies.

The situations described in the last bullet are not necessarily problematic, but they can become so if conceptual data modeling is not performed. Figures 6 and 7 illustrate.

Figure 6 shows how a database designer might conceptualize a database that is optimized for processing student transcripts. (Each transcript is a per-student report showing all the courses a student takes, grouped by semester.) A database structure based on Figure 6 would arrange the data so that all data for a transcript would be stored together; it could be retrieved and processed as a unit. This would yield performance advantages, especially in a multinode cluster.



* A student can take the same course multiple times, but not in the same semester. This fact is revealed by the identifier of the *enrollment* entity: {course, student, semester}. There is nothing about the NoSQL visualization that enforces that restriction.
* The instructor of a course does not vary. There is nothing in the NoSQL visualization suggesting as much. In fact, the structure of the NoSQL visualization might lead database designers to assume otherwise.



If the NoSQL approach was chosen because the specific NoSQL metamodel has a conceptual affinity with the data to be managed, data modelers have some room to relax. It’s when the NoSQL database model will distort the data phenomenon that the need for a conceptual data model becomes inescapable.



Four types of NoSQL databases are Document-oriented, Key-Value Pairs, Column- oriented and Graph.

**Storage for Microservices Architectures**

<https://www.gartner.com/document/3901169?ref=TrackRecommendedEmail&refval=1555684733333>

Microservices architecture is an approach for building distributed applications that support agile and scalable delivery. Within a microservices architecture, each service uses its own storage layer. (That is, services do not share a database.) Such an approach imposes a special burden on data designers: all data that is needed by a service must be available to that service. (For more information, see [“Working With Data in a Microservices Architecture”](https://www.gartner.com/document/code/370409?ref=grbody&refval=3901169)).

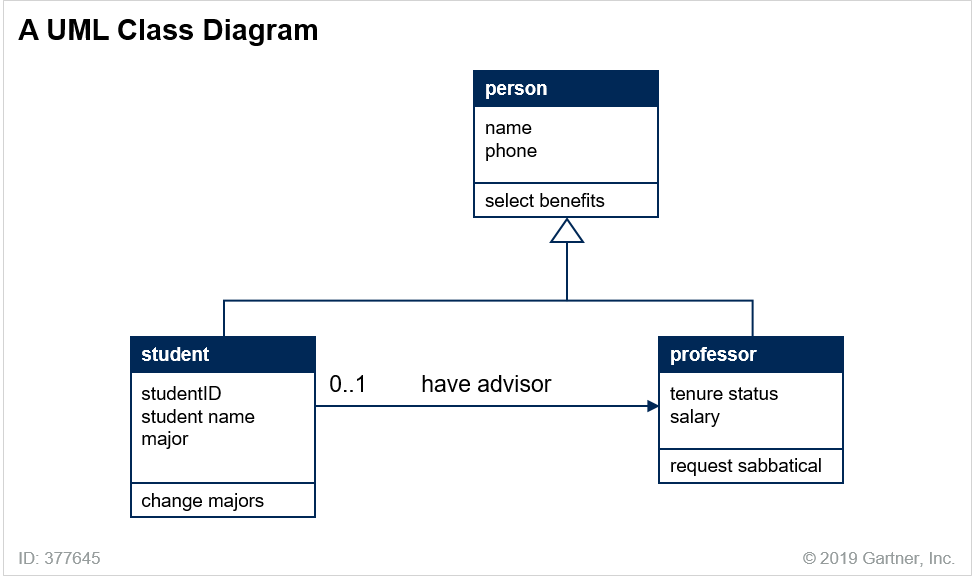
Because data modeling is difficult, some businesses rely on off-the-shelf data models, Off-the-shelf data models that are targeted at specific industries have a close cousin: industry-standard vocabularies. An industry-standard vocabulary establishes and names structures that can accommodate the information common to an industry or use case. Some typical industry-standard vocabularies are: Darwin Information Typing Architecture (DITA), eXtensible Business Reporting Language (XBRL), and Health Level Seven (HL7).

RunTime

Most discussions of data modeling pertain to the design of persistent data — the storage layer of the application stack. However, a processing layer is also critical to this discussion. Data models should be used to design the implementation details of runtime data structures and to design message models used for inter service communication.

Runtime structures are typically modeled with Unified Modeling Language (UML) class diagrams, a primary design notation of software engineers. This is not surprising, because UML is a technique designed to help software engineers design and analyze object-oriented software systems. The design of runtime data structures is not typically performed by a data modeler, but by a software engineer. The primary goal is to produce structures that support the runtime operations on the data.

The “U” in UML stands for “unified,” not “universal.” UML is not universally applicable, so it’s not appropriate for conceptual data modeling. UML class diagrams introduce considerable notational complexity that is of unclear value for conceptual data modelers. Figure below shows a UML diagram. Notice that the diagram includes process requirements: activities called *select benefits, change majors* and *request sabbatical*.



Misguided, however, is using UML to perform conceptual data modeling, which would be derailed by UML’s notational complexity, its inclusion of processing concerns, and its overarching allegiance to the technological requirements of object-oriented systems.

##### **Message Models**

Service-oriented architecture (SOA) and microservices architecture (MSA) are characterized by loosely coupled, narrowly scoped component services that can be developed, deployed and operated independently. Services exchange information at runtime through message models, typically expressed as XML schemas or JSON objects. These message models should be understood as data models, with the following caveats:

* They represent transient (i.e., nonpersistent) manifestations of data.
* Message models received by a service are typically designed to contain all the data that a service would need to perform its function. Thus, customary data-modeling goals such as high normal forms and application/data independence are immaterial to the design of message models.
* The designers of message models often approach the problem as analogous to the design of input and output parameter sequences for the service.

The storage layer of an MSA imposes special design sensitivities on data modelers, mentioned previously and repeated here for emphasis: Within an MSA, each service uses its own storage layer. (That is, services do not share a database.)

#### **Data Integration**

Several data integration techniques require data-modeling expertise, either at design time or runtime. Given that one of the thorniest problems of data integration is mismatched data models in to-be-integrated systems, data modeling expertise will be needed to choose approaches for resolving these differences. This will be a design-time activity.

Some data-integration problems are addressed at runtime. For example, metadata discovery tools can comb through existing data resources, inferring data models based on the contents and structure of data files. The expertise of data modelers will be needed to interpret the recommendations of these tools.

Beyond these contributions, some other forms of data integration will use data modeling expertise more significantly. The next sections elaborate.

##### **Data Warehousing**

The following definition is probably familiar to you:

A data warehouse (DW) is a storage architecture designed to hold a comprehensive, historical record of data used by an organization and to make that data available for easy analysis and reporting.

DW storage structures can be relational, or they can include multidimensional cubes, star schemas and snowflake schemas. Typical data warehouses include data from multiple upstream systems. Such data movements are designed as queries, which means their design requires the scrutiny and understanding of the data models from those source systems. Thus problems that fall into three broad classes: (a) inconsistent data values, (b) inconsistent data models, and (c) inconsistent metamodels. The data used by an organization can originate in transactional/operational systems of record, or in internal or external systems of engagement. Some of these systems might include semistructured or unstructured data.

Data warehousing arose because individual database solutions could not simultaneously accommodate write-intensive transactional workloads and read-intensive analytical workloads. (Any attempt to tune database parameters for one workload would severely compromise the performance of the other.) The architectural separation of these workloads into separate systems — online transaction processing (OLTP) and online analytical processing (OLAP) — offered greater opportunity for performance enhancements than merely adjusting parameters.

+++++++++++++++

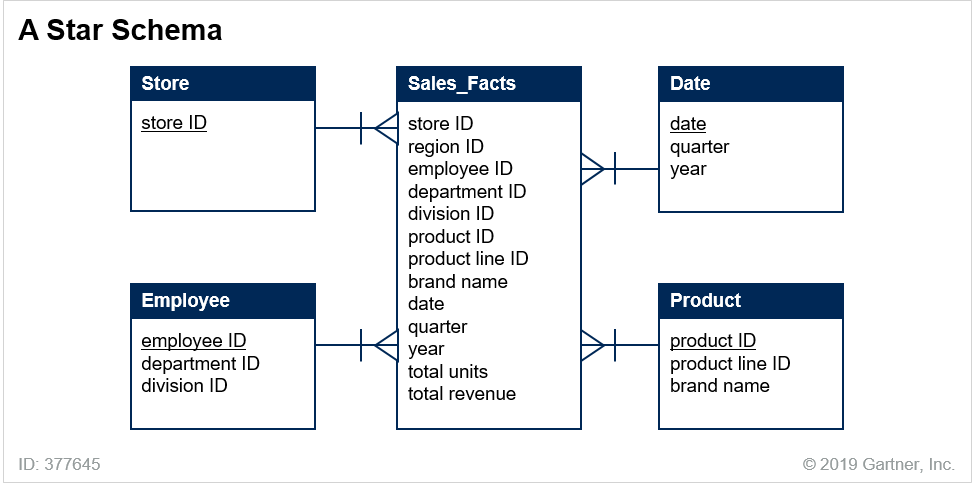
There are three approaches to the design of physical data warehouses. They are discussed in the following sections.

###### Dimensional Models

Some data warehouses use dimensional modeling techniques, which differentiate fact tables from dimension tables. When designing dimensional warehouses, data modelers create “star schemas,” where a single fact table is at the center and various dimension tables radiate out from the fact table. Figure 9 shows a star schema for a dimensional model. The fact table is “Sales\_Facts” and there are four dimension tables.

Figure 9. A Star Schema

Source: Gartner (February 2019)



The basics of dimensional design are straightforward; the distinction between facts and dimensions is easy to grasp. Facts are things that you count or measure, like total units and total revenue in Figure 9. Dimensions are ways that you can group collections of facts into summary values, like product line and division. The model in Figure 9 supports all of the following queries:

* Which employee has sold the most units? Which store has sold the most units?
* Which employee has sold the most units last year? Last quarter?
* Which department generated the most revenue last year?

But complications arise: Dimensional modelers must choose an approach for coping with changes to dimension values. For example, when an employee changes departments, how should the data warehouse acknowledge that? This problem, known as the “slowly changing dimensions” problem, presents data modelers with the following choices:

* Simply overwrite the previous department value for the employee. The data warehouse will always show only the employee’s current department and division. This approach is rarely used because it distorts the historical reports about how well departments performed in the past.
* Retain the employee’s original department and the current department. This halfway approach also distorts historical reports.
* Use time stamps or effective dates to keep track of exactly when the employees join and leave specific departments and divisions. This is the most frequently used method. Some databases support this style of modeling natively.

###### **Relational Warehouses**

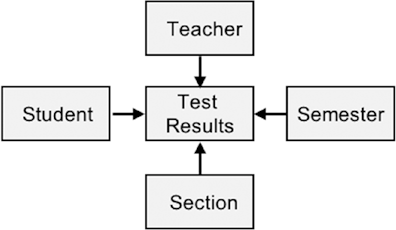
The goal of dimensional warehousing is to create a circumscribed warehouse that supports analytical queries for a specific set of facts. In contrast, the goal of relational warehousing is to maintain a corporate data model that maintains a single source of truth for the enterprise. These expressive ambitions are higher than those of a dimensional warehouse, and they have historically come at a price. Relational warehouses are harder to design, and in the past became too complex to support analytical queries directly without either special hardware (i.e., “data warehouse appliances”), or data marts. Data marts — downstream subsets of the data in the data warehouse — are designed to support small sets of analytical queries specific to an analytical use case. Data marts have traditionally been implemented as physical databases, but these days they are often virtual, implemented simply as views atop the underlying data warehouse.

The decision to create a relational warehouse has significant consequences for data modelers, including:

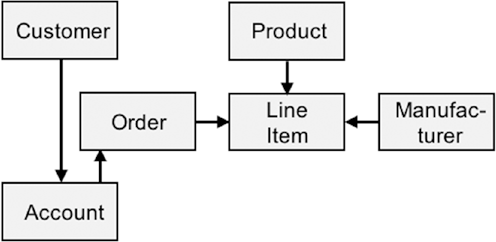
* Creating the data model for a relational data warehouse requires significant investment from data modelers and from business subject matter experts. Initial setup and delivery will take more time than initial setup and delivery of a dimensional model.
* Modifying the data warehouse — say, while integrating a previously excluded transactional system into the data warehouse — is complicated and can be disruptive to the data model of the existing warehouse.
* More ETL queries will be needed, because the relational warehouse is more likely to require downstream data marts than a multidimensional warehouse.

Data Warehouse Architecture

If you look at successful decision support systems (there are a few), you will notice that the data warehouse database design looks nothing like its operational cousin. The most common data architecture is the star schema—a single fact record surrounded by multiple dimension records forming a star pattern (Figure 14-1). The fact record type sits in the middle of the schema at the many end of a number of one-to-many relationships. The dimension records are used to query the fact record.

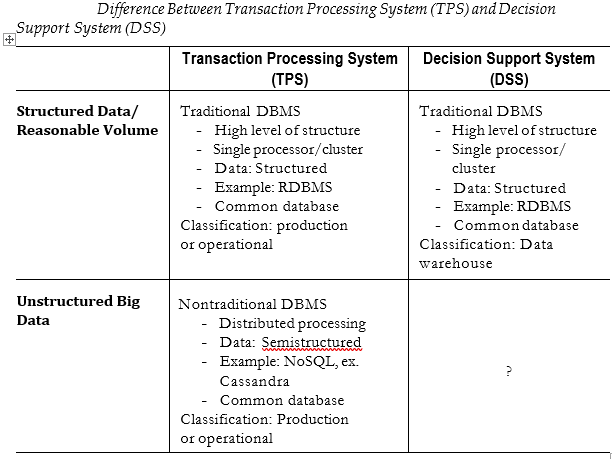


The Rationalized physical data model in Figure 14-3 is not a star but a snowflake (because the Customer and Account records are linked to the Order dimension), although “spiderweb” would be a more accurate though less majestic name.



Big Data is almost always unstructured data. P292

NoSQL systems were specifically designed, according to their vendors, to store unstructured data.



###### **Data Vaults**

Despite their differences, dimensional and relational data-warehousing techniques both strive to create carefully curated, defect-free data in the warehouse. By contrast, the data vault technique shifts the burden for curated, analysis-ready data elsewhere. The physical structures of a data vault warehouse provide only the following:

* Long-term historical storage of data from multiple operational/transactional systems.
* A data-modeling strategy that easily accommodates the addition of new data, even if that data comes from previously excluded source systems.

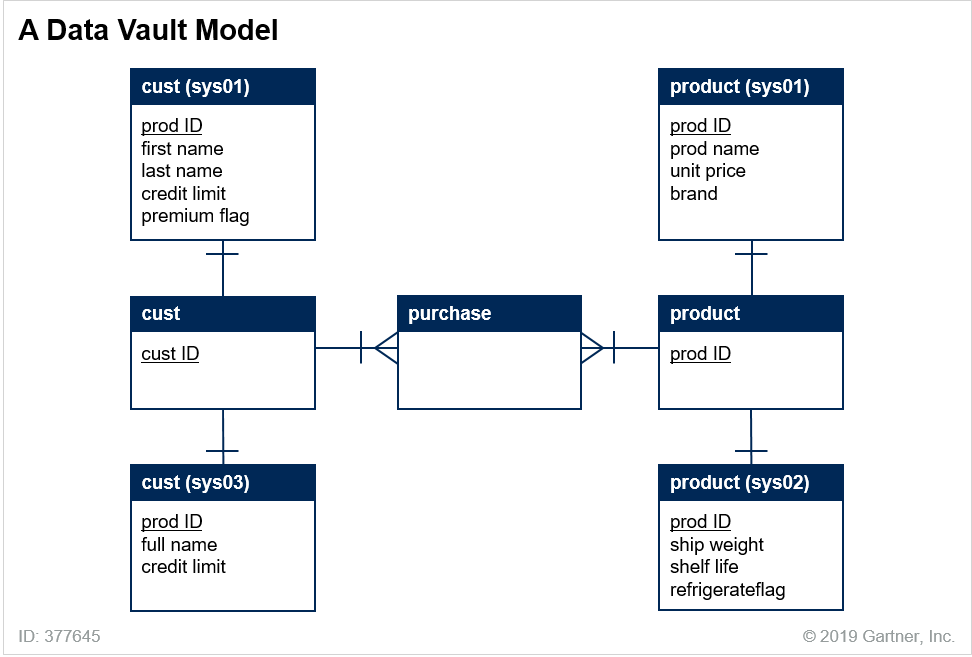
Noticeably absent from the preceding goals is any attempt to use the data vault repository for cleansed or integrated data. Instead, the data value relies on downstream architectural components for presenting analysis-ready data.

For the physical data vault repository itself, data modelers employ a strategy that is reminiscent of what relational database designers know as vertical partitioning. In vertical partitioning, schema designers separate data columns based on different storage requirements, often motivated by security needs. For example, the data model in Figure 4 is a modification of the data model in Figure 2. That modification constitutes an application of vertical partitioning.

Analogously, data vault designers create separate partitions for the descriptive data — in relational vocabulary, for the nonidentifying columns — of each table of each source system. In this case, the partitions are not separated to honor concerns about privacy. Rather, the partitions correspond to the source systems for the data. Figure 10 illustrates.

Figure 10. A Data Vault Model

Source: Gartner (February 2019)



Notice the following from Figure 10:

* Three significant entities — customer, product, and purchase — are interrelated with relationships that indicate some enterprisewide realities, namely, that customers purchase products, that each customer can purchase many products and each product can be purchased by many customers.
* Descriptive attributes about the significant entities are stored in ancillary entities.
* Each (significant entity, system) pair can have an ancillary entity dedicated to storing those descriptive attributes recognized for that entity by that system.
* No effort has been made to reconcile data-model differences across systems. For example, two entities in the data vault store customer names, but one entity (from sys01) stores first-plus-last names, and another (sys03) stores full names.
* No effort has been made to reconcile data-value differences across systems. For example, the credit limit attribute is stored in two tables in the data vault. For a given customer, the values in these tables might or might not conflict.

**Despite these shortcomings, the data vault technique offers an advantage. With this approach to modeling, new systems can be integrated into the data-value model with absolutely no disruption to the existing model. If** a fourth system is to be included and that system includes descriptive attributes for customer, a new entity will be added to the data vault. That entity — to be named “cust(sys04)”— can be created and populated with no effect whatsoever on the existing entities in the data vault.

Of course, because it reflects no cleansing, reconciliation or aggregation of the incoming data, the data vault cannot serve any analytical query load directly from the physical structures of the data vault. Data vaults are merely centralized historical repositories of data from various operational systems. To be part of a data-warehousing solution, data vaults must be supplemented by downstream data marts, by presentation layers, or by logical data warehouses, described in the next section.

Logical Data Warehousing

The logical data warehouse (LDW) technique recognizes that it is no longer possible for a single system to store all data and execute all types of analytical processing. Therefore, the LDW extends the original enterprise data warehouse (EDW) by integrating multiple analytical systems. These systems are typically the EDW, data lake, some data marts and perhaps an operational data store (ODS). The LDW often includes a data virtualization layer that makes these disparate systems appear to the analytical consumers as one system. “Everything happens with database views.”.

In a logical data warehouse, view definitions are arranged in layers. **The bottom layer of views** — the virtual base layer — harvests data from the various sources. These can be operational systems, traditional data warehouses, historical repositories of transactional data, document databases, data lakes or just about any form of data storage.

**The top layer of views** — the “data consumption layer” — presents aggregated collections of data that are formatted to correspond exactly to the needs of analytical clients .

##### **Data Lakes**

Gartner defines a data lake as follows:

A data lake is a collection of storage instances of various data assets additional to the originating data sources. These assets are stored in a near-exact, or even exact, copy of the source format. The purpose of a data lake is to present an unrefined view of data to only the most highly skilled analysts. This will help them explore their data refinement and analysis techniques independent of any of the system-of-record compromises that may exist in a traditional analytic data store (such as a data mart or data warehouse).

###### Schema-on-Read Solutions

Data lakes are often the environment for ingestion of high-velocity streaming data. In many situations, the incoming data arrives so quickly that it must be stored as-is, without regard for its quality. The strategy in such situations is to enforce the schema later, perhaps just before the data is used for some analytical purpose. This approach is called “schema-on-read.”

<https://www.gartner.com/document/3901169?ref=TrackRecommendedEmail&refval=1555684733333>

**Types of NoSQL Databases**

There are four types of NoSQL databases: Document-oriented, Key-Value Pairs, Wide Column

### (or Column Family) and Graph (Abramova, Bernardino, & Furtado, 2014).

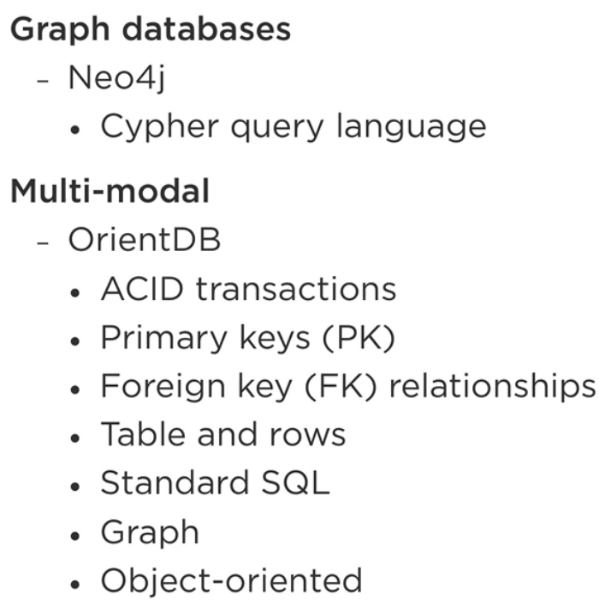
### 

### Graph

*Graph* NoSQL databases are modern versions of the network architecture with two major differences. First, they are tuned for high performance using techniques regularly found in other NoSQL products. Second, they have a DML that makes it easier to navigate the database.

Graph is a mathematical term for a structure consisting of a number of nodes. Nodes are connected to each other by edges. Unlike trees, there is no up or down. The nodes are, of course, records, and the edges are lines or links. Graph systems are often hybrids combining other architectures into a single implementation, with key-value being a favorite. Graph systems get their speed from embedded pointers linking the various notes.

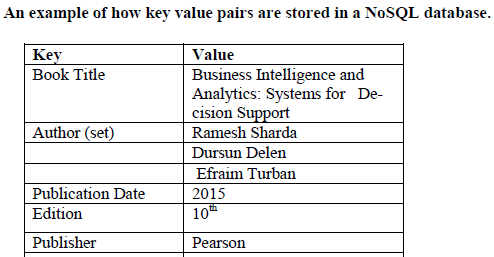
Neo4j, from Neo Technology, Inc., is an example of an ACID-compliant graph database management system.



### Document Management

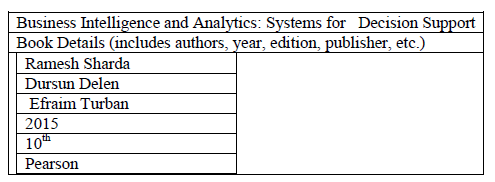
The *document management* system, usually listed as a separate NoSQL model, is often a subset of the key-value approach. The key is the document name or description, and the value is the underlying document. Each value occurrence contains not only the document but the description (metadata, data type) of the document. the data is denormalized, semi-structured and stored hierarchically.

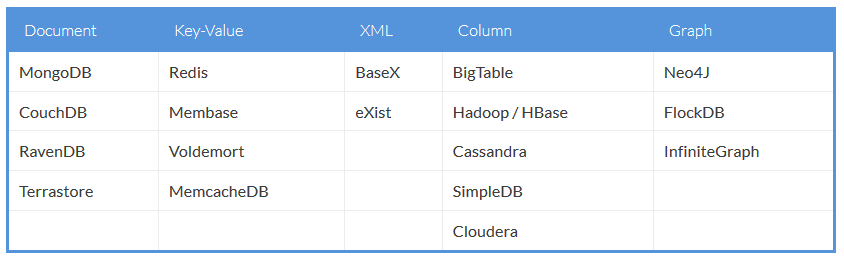
MongoDB from MongoDB Inc. is a good example of a document DBMS.

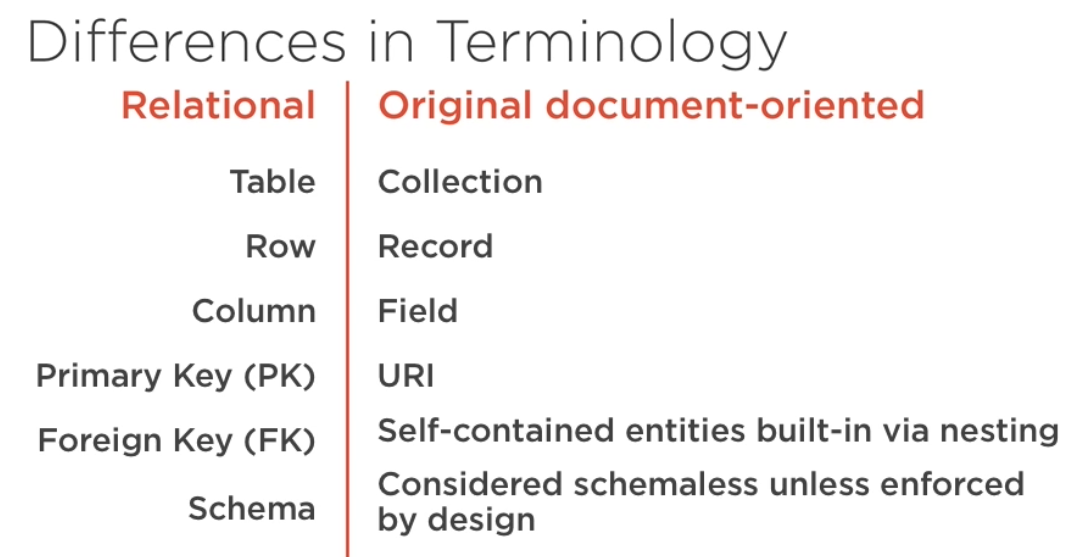


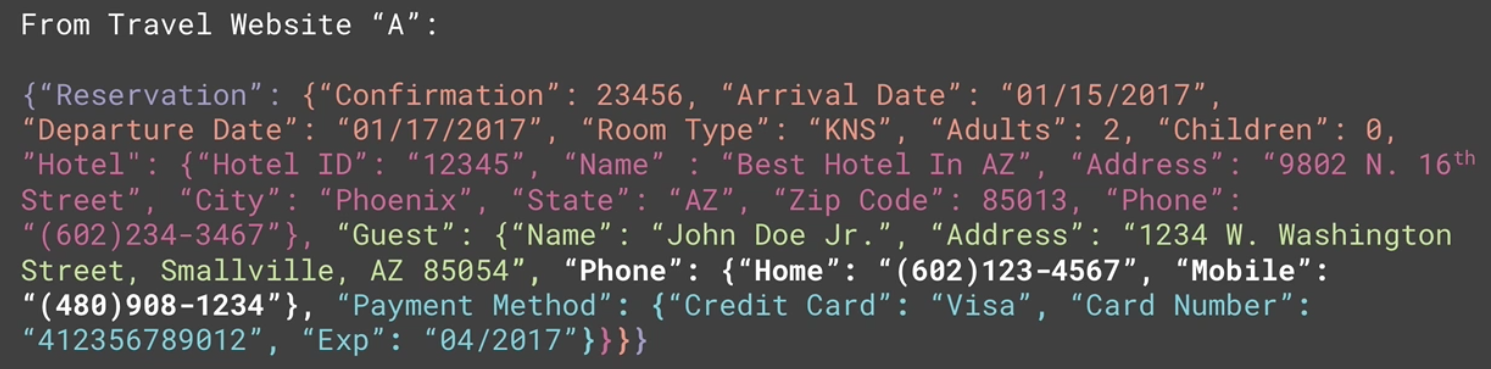
Wide column

Stored the above data as below in highly flexible column formats so They can support complex data types, unstructured text and graphics (e.g. jpeg, gif, bmp, etc.).





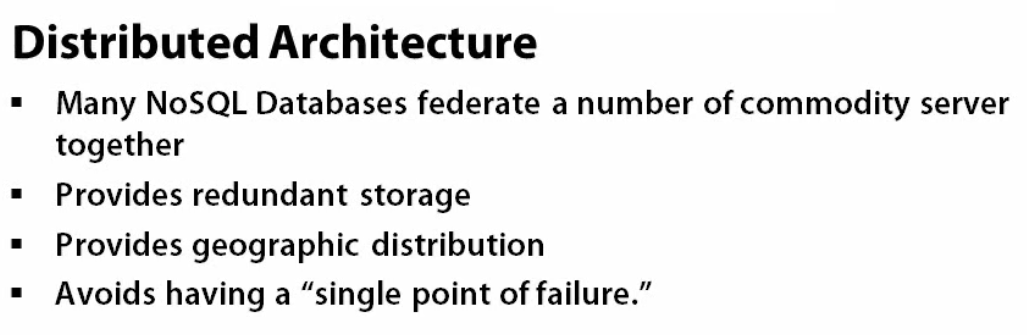


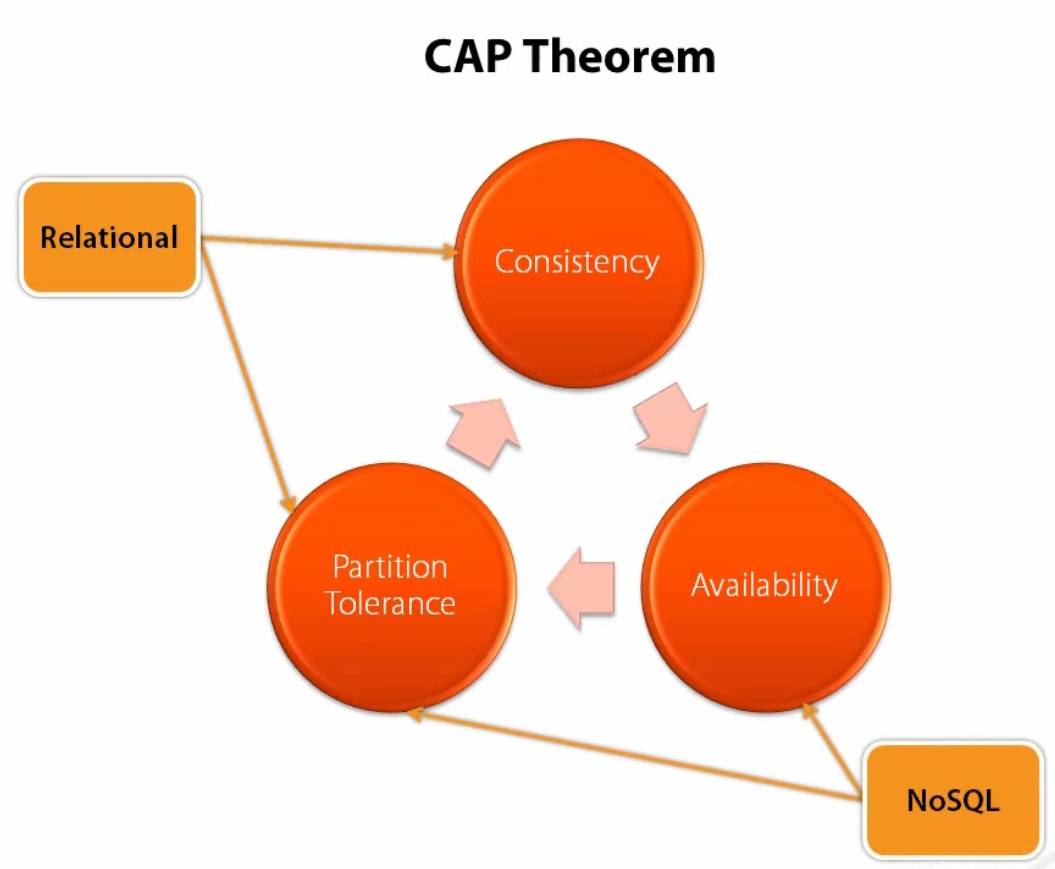


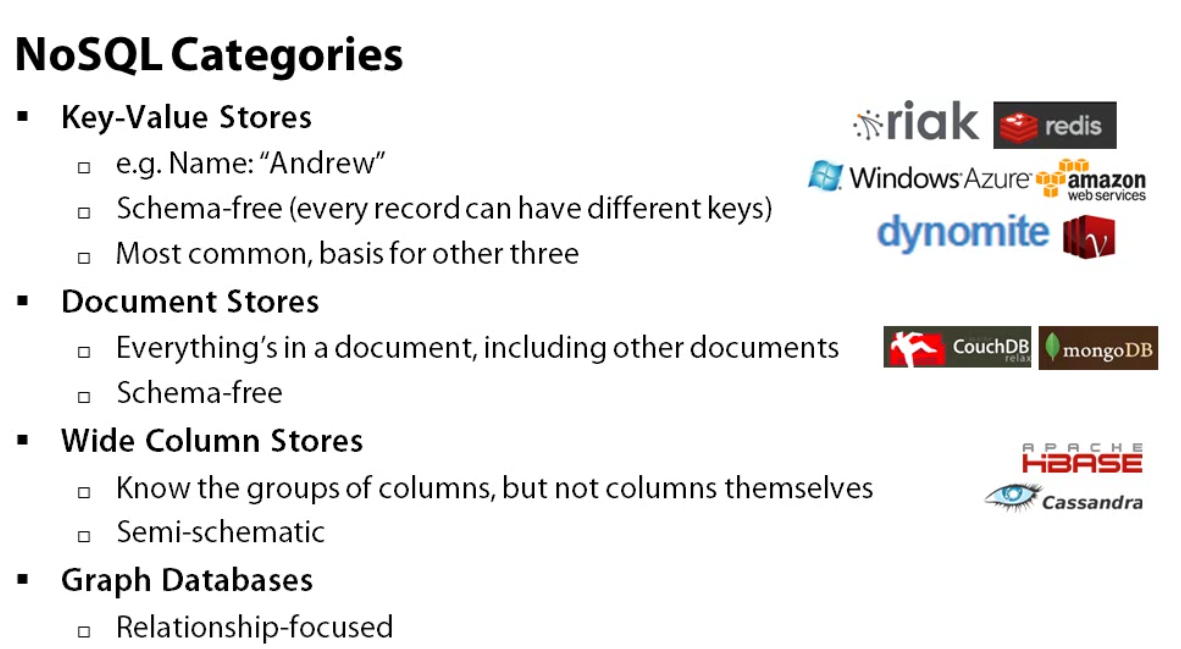


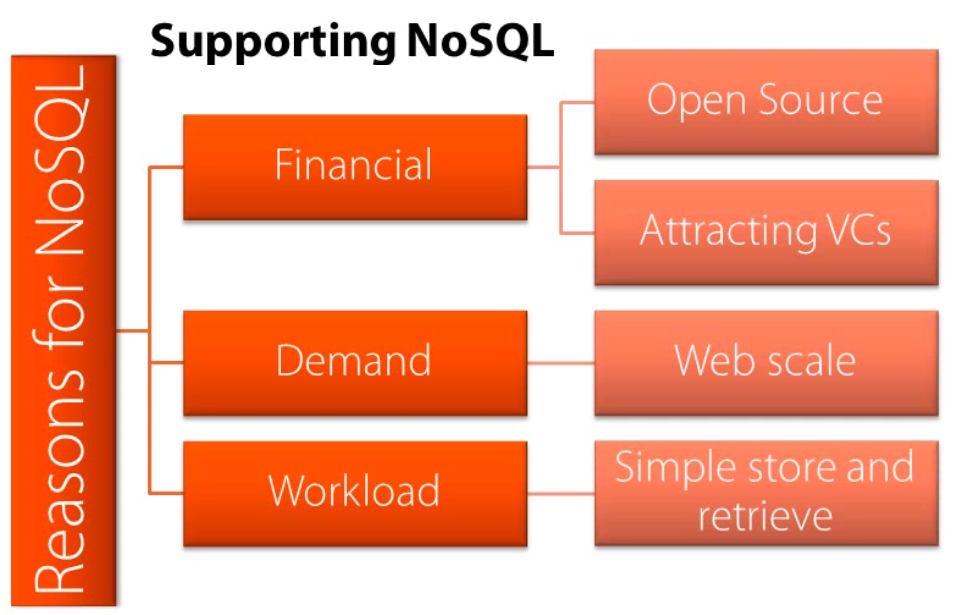


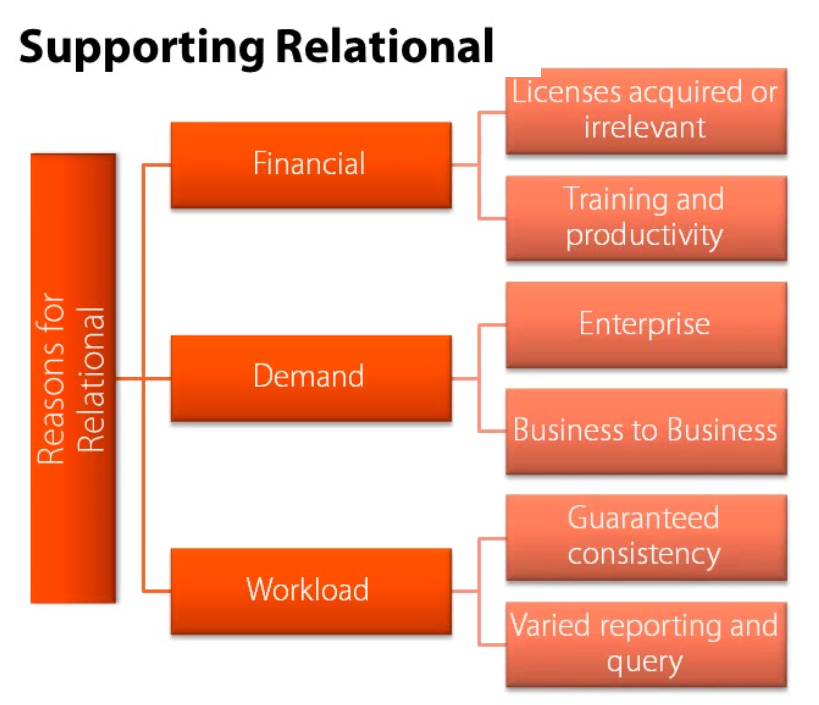


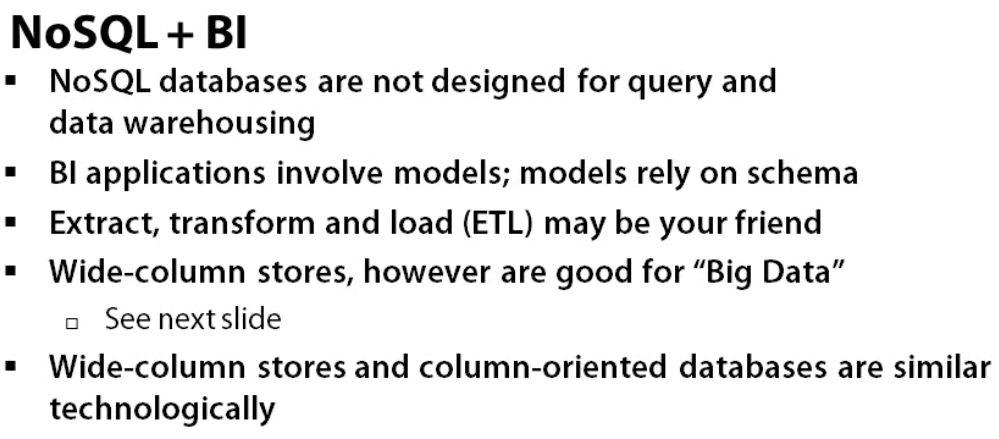


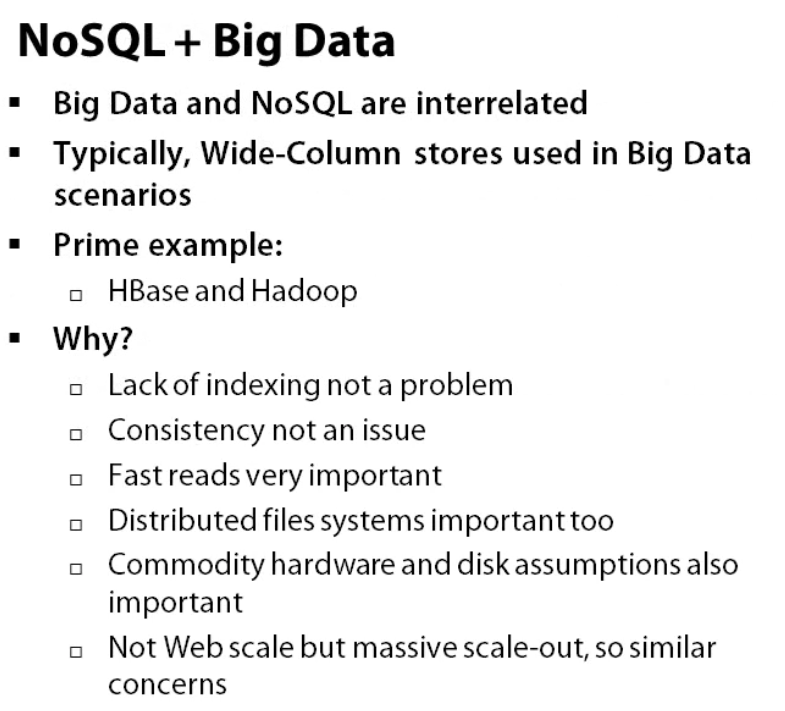


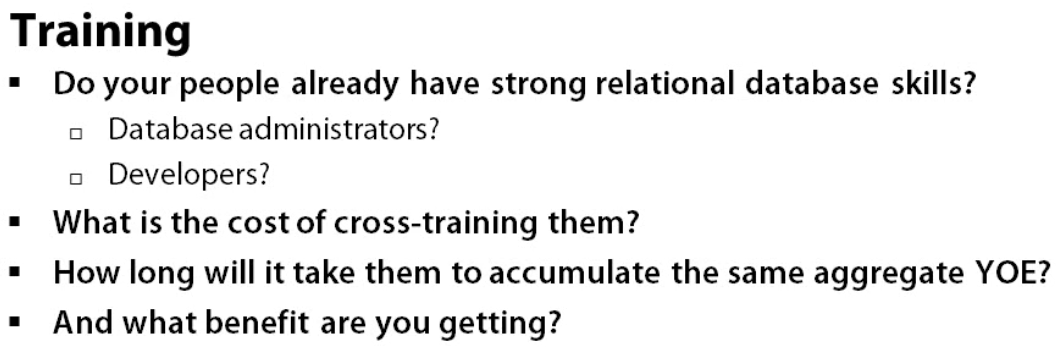


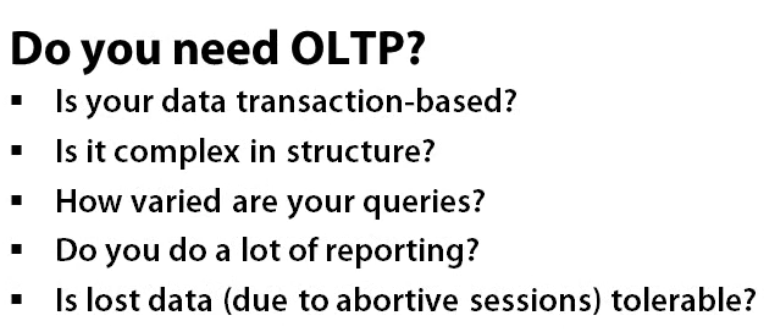












Hadoop

The core of Hadoop, both figuratively and literally, is two products,

1) Hadoop Distributed File System (HDFS) and

2) MapReduce, Moves applications to the nodes where data need to be processed.

HDFS allows data to be stored in a large number of nodes, collectively called a *cluster*. A single file might be distributed and replicated across multiple nodes within the cluster. The distribution—one file broken into multiple parts (called *blocks*)—allows parallel processing of a single file, while replication—storing multiple copies of the same data (blocks) in multiple locations—provides high availability and backup. MapReduce is the application framework that oversees how files are processed by assigning application programs to various nodes. MapReduce copies and sends computer code to servers where there are data waiting to be processed vs how conventional data processing works (data is pushed to databases).

Considerable work and execution time is required to set up a MapReduce/ HDFS cluster-wide process. This effort makes sense if the data file is very large. It makes no sense if the data file is small.

Hadoop family also includes another product—Hive. Hive is a

SQL-like product that can front-end both MapReduce and HDFS. Hive can also create its own relational-like database stored in its own workspace. The Hive users think they are querying a relational database, but the data they are using might be in HDFS, in a native Hive file, or in some third-party database. Hive data are often HDFS data, they do not have a normal schema. In fact, they do not have a schema at all until runtime This approach is called *schema on read*..

Neither Cassandra nor HBase is a great candidate for DSSs. Their internals are designed for one I/O retrieval of fat records. Traditional DBMSs bind data to the schema when the data are written to the database. This is called *schema on write*.