Choose the right data store

11/20/2019 • 10 minutes to read • 👤👤👤👤👤 +8

In this article

Modern business systems manage increasingly large volumes of data. Data may be ingested from external services, generated by the system itself, or created by users. These data sets may have extremely varied characteristics and processing requirements. Businesses use data to assess trends, trigger business processes, audit their operations, analyze customer behavior, and many other things.

This heterogeneity means that a single data store is usually not the best approach. Instead, it's often better to store different types of data in different data stores, each focused toward a specific workload or usage pattern. The term *polyglot persistence* is used to describe solutions that use a mix of data store technologies.

Selecting the right data store for your requirements is a key design decision. There are literally hundreds of implementations to choose from among SQL and NoSQL databases. Data stores are often categorized by how they structure data and the types of operations they support. This article describes several of the most common storage models. Note that a particular data store technology may support multiple storage models. For example, a relational database management systems (RDBMS) may also support key/value or graph storage. In fact, there is a general trend for so-called *multi-model* support, where a single database system supports several models. But it's still useful to understand the different models at a high level.

Not all data stores in a given category provide the same feature-set. Most data stores provide server-side functionality to query and process data. Sometimes this functionality is built into the data storage engine. In other cases, the data storage and processing capabilities are separated, and there may be several options for processing

and analysis. Data stores also support different programmatic and management interfaces.

Generally, you should start by considering which storage model is best suited for your requirements. Then consider a particular data store within that category, based on factors such as feature set, cost, and ease of management.

Relational database management systems

Relational databases organize data as a series of two-dimensional tables with rows and columns. Each table has its own columns, and every row in a table has the same set of columns. This model is mathematically based, and most vendors provide a dialect of the Structured Query Language (SQL) for retrieving and managing data. An RDBMS typically implements a transactionally consistent mechanism that conforms to the ACID (Atomic, Consistent, Isolated, Durable) model for updating information.

An RDBMS typically supports a schema-on-write model, where the data structure is defined ahead of time, and all read or write operations must use the schema. This is in contrast to most NoSQL data stores, particularly key/value types, where the schema-on-read model assumes that the client will be imposing its own interpretive schema on data coming out of the database, and is agnostic to the data format being written.

An RDBMS is very useful when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state. However, the underlying structures do not lend themselves to scaling out by distributing storage and processing across machines. Also, information stored in an RDBMS, must be put into a relational structure by following the normalization process. While this process is well understood, it can lead to inefficiencies, because of the need to disassemble logical entities into rows in separate tables, and then reassemble the data when running queries.

Relevant Azure services:

- Azure SQL Database
- Azure Database for MySQL
- Azure Database for PostgreSQL
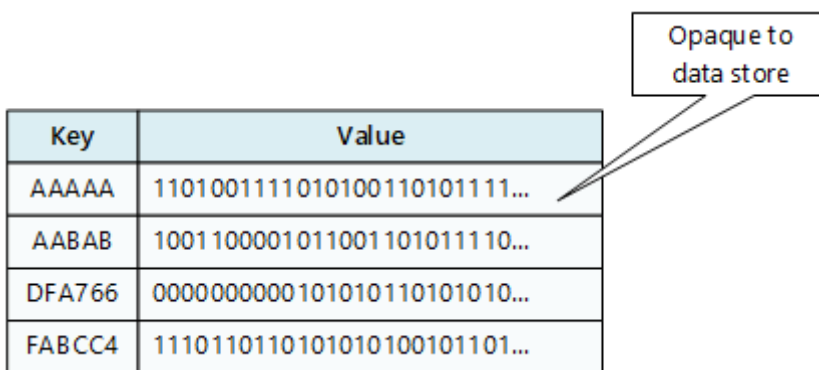- Azure Database for MariaDB

Key/value stores

A key/value store is essentially a large hash table. You associate each data value with a unique key, and the key/value store uses this key to store the data by using an

appropriate hashing function. The hashing function is selected to provide an even distribution of hashed keys across the data storage.

Most key/value stores only support simple query, insert, and delete operations. To modify a value (either partially or completely), an application must overwrite the existing data for the entire value. In most implementations, reading or writing a single value is an atomic operation. If the value is large, writing may take some time.

An application can store arbitrary data as a set of values, although some key/value stores impose limits on the maximum size of values. The stored values are opaque to the storage system software. Any schema information must be provided and interpreted by the application. Essentially, values are blobs and the key/value store simply retrieves or stores the value by key.

Key	Value
AAAAA	1101001111010100110101111...
AABAB	1001100001011001101011110...
DFA766	0000000000101010110101010...
FABCC4	1110110110101010100101101...

Opaque to data store

Key/value stores are highly optimized for applications performing simple lookups, but are less suitable for systems that need to query data across different key/value stores. Key/value stores are also not optimized for scenarios where querying by value is important, rather than performing lookups based only on keys. For example, with a relational database, you can find a record by using a WHERE clause, but key/values stores usually do not have this type of lookup capability for values.

A single key/value store can be extremely scalable, as the data store can easily distribute data across multiple nodes on separate machines.

Relevant Azure services:

- Cosmos DB
- Azure Cache for Redis

Document databases

A document database is conceptually similar to a key/value store, except that it stores a collection of named fields and data (known as documents), each of which could be simple scalar items or compound elements such as lists and child collections. The data in the fields of a document can be encoded in a variety of ways, including XML, YAML,

JSON, BSON, or even stored as plain text. Unlike key/value stores, the fields in documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document may contain information that would be spread across several relational tables in an RDBMS.

A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. Applications can store different data in documents as business requirements change.

Key	Document

1001	{
	   "CustomerID": 99,
	   "OrderItems": [
	     { "ProductID": 2010,
	       "Quantity": 2,
	       "Cost": 520
	     },
	     { "ProductID": 4365,
	       "Quantity": 1,
	       "Cost": 18
	     }],
	     "OrderDate": "04/01/2017"
	}

1002	{
	   "CustomerID": 220,
	   "OrderItems": [
	     { "ProductID": 1285,
	       "Quantity": 1,
	       "Cost": 120
	     }],
	     "OrderDate": "05/08/2017"
	}

The application can retrieve documents by using the document key. This is a unique identifier for the document, which is often hashed, to help distribute data evenly. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as the key. The application can also query documents based on the value of one or more fields. Some document databases support indexing to facilitate fast lookup of documents based on one or more indexed fields.

Many document databases support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Read and write operations over multiple fields in a single document are usually atomic.

Relevant Azure service: Cosmos DB

Graph databases

A graph database stores two types of information, nodes and edges. You can think of nodes as entities. Edges which specify the relationships between nodes. Both nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

The purpose of a graph database is to allow an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.
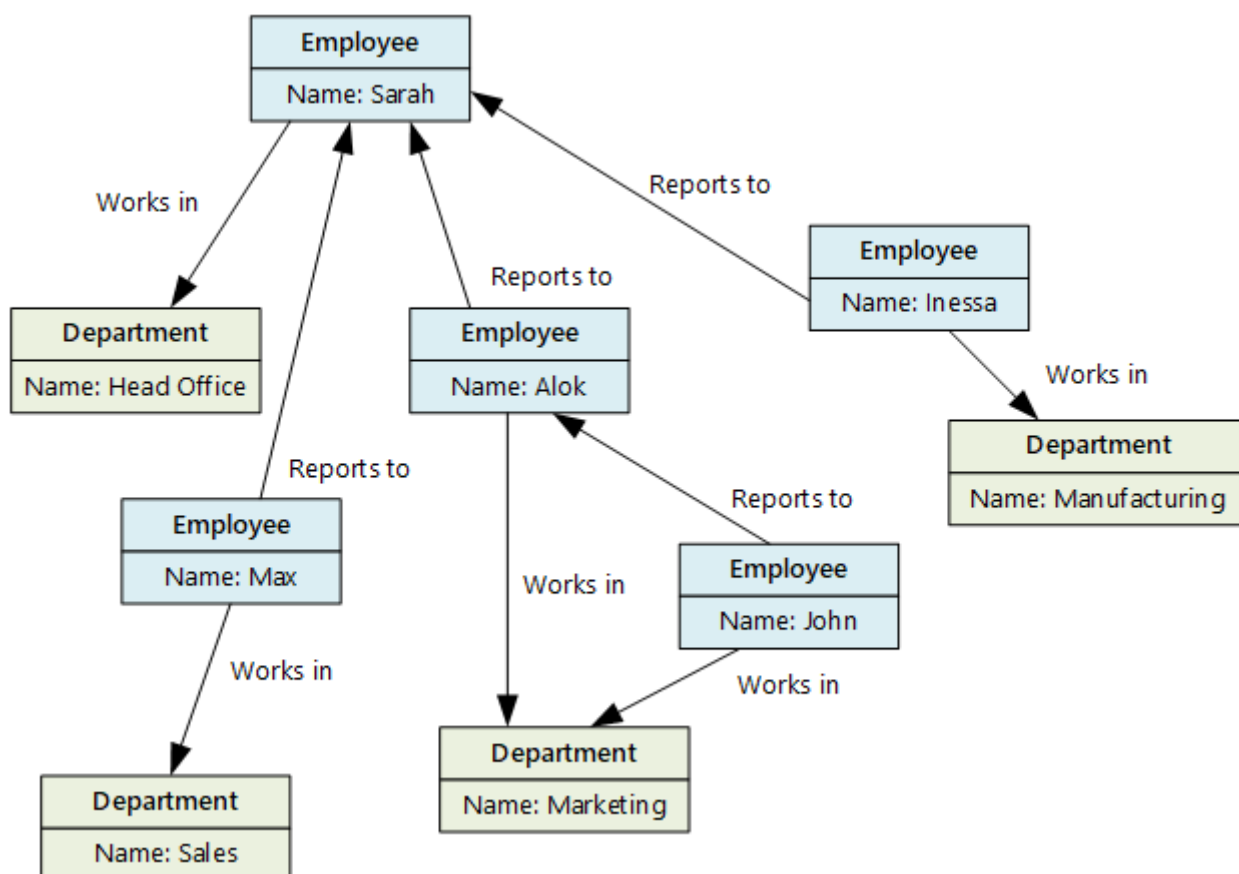
This structure makes it straightforward to perform queries such as "Find all employees who report directly or indirectly to Sarah" or "Who works in the same department as John?" For large graphs with lots of entities and relationships, you can perform very complex analyses very quickly. Many graph databases provide a query language that you can use to traverse a network of relationships efficiently.

Relevant Azure service: Cosmos DB

Column-family databases

A column-family database organizes data into rows and columns. In its simplest form, a column-family database can appear very similar to a relational database, at least conceptually. The real power of a column-family database lies in its denormalized approach to structuring sparse data.

You can think of a column-family database as holding tabular data with rows and columns, but the columns are divided into groups known as *column families*. Each column family holds a set of columns that are logically related together and are typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. Within a column family, new columns can be added dynamically, and rows can be sparse (that is, a row doesn't need to have a value for every column).

The following diagram shows an example with two column families, `Identity` and `Contact Info`. The data for a single entity has the same row key in each column-family. This structure, where the rows for any given object in a column family can vary dynamically, is an important benefit of the column-family approach, making this form of data store highly suited for storing structured, volatile data.

CustomerID	Column Family: Identity
001	First name: Mu Bae Last name: Min
002	First name: Francisco Last name: Vila Nova Suffix: Jr.
003	First name: Lena Last name: Adamcyz Title: Dr.

CustomerID	Column Family: Contact Info
001	Phone number: 555-0100 Email: someone@example.com
002	Email: vilanova@contoso.com
003	Phone number: 555-0120

Unlike a key/value store or a document database, most column-family databases store data in key order, rather than by computing a hash. Many implementations allow you to create indexes over specific columns in a column-family. Indexes let you retrieve data by columns value, rather than row key.

Read and write operations for a row are usually atomic with a single column-family, although some implementations provide atomicity across the entire row, spanning multiple column-families.

Relevant Azure service: HBase in HDInsight

Data analytics

Data analytics stores provide massively parallel solutions for ingesting, storing, and analyzing data. This data is distributed across multiple servers using a share-nothing architecture to maximize scalability and minimize dependencies. The data is unlikely to be static, so these stores must be able to handle large quantities of information, arriving in a variety of formats from multiple streams, while continuing to process new queries.

Relevant Azure services:

- Azure Synapse Analytics
- Azure Data Lake
- Azure Data Explorer

Search Engine Databases

A search engine database supports the ability to search for information held in external data stores and services. A search engine database can be used to index massive volumes of data and provide near real-time access to these indexes. Although search engine databases are commonly thought of as being synonymous with the web, many large-scale systems use them to provide structured and ad-hoc search capabilities on top of their own databases.

The key characteristics of a search engine database are the ability to store and index information very quickly, and provide fast response times for search requests. Indexes can be multi-dimensional and may support free-text searches across large volumes of text data. Indexing can be performed using a pull model, triggered by the search engine database, or using a push model, initiated by external application code.

Searching can be exact or fuzzy. A fuzzy search finds documents that match a set of terms and calculates how closely they match. Some search engines also support linguistic analysis that can return matches based on synonyms, genre expansions (for example, matching `dogs` to `pets`), and stemming (matching words with the same root).

Relevant Azure service: Azure Search

Time Series Databases

Time series data is a set of values organized by time, and a time series database is a database that is optimized for this type of data. Time series databases must support a very high number of writes, as they typically collect large amounts of data in real time from a large number of sources. Updates are rare, and deletes are often done as bulk

operations. Although the records written to a time-series database are generally small, there are often a large number of records, and total data size can grow rapidly.

Time series databases are good for storing telemetry data. Scenarios include IoT sensors or application/system counters.

Relevant Azure service: Time Series Insights

Object storage

Object storage is optimized for storing and retrieving large binary objects (images, files, video and audio streams, large application data objects and documents, virtual machine disk images). Objects in these store types are composed of the stored data, some metadata, and a unique ID for accessing the object. Object stores enables the management of extremely large amounts of unstructured data.

Relevant Azure service: Blob Storage

Shared files

Sometimes, using simple flat files can be the most effective means of storing and retrieving information. Using file shares enables files to be accessed across a network. Given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for performing basic, low-level operations such as simple read and write requests.

Relevant Azure service: File Storage

Is this page helpful?

👍 Yes   👎 No