A Field Guide to the World of Modern Data Stores

Prepare yourself to navigate the world of NoSQL

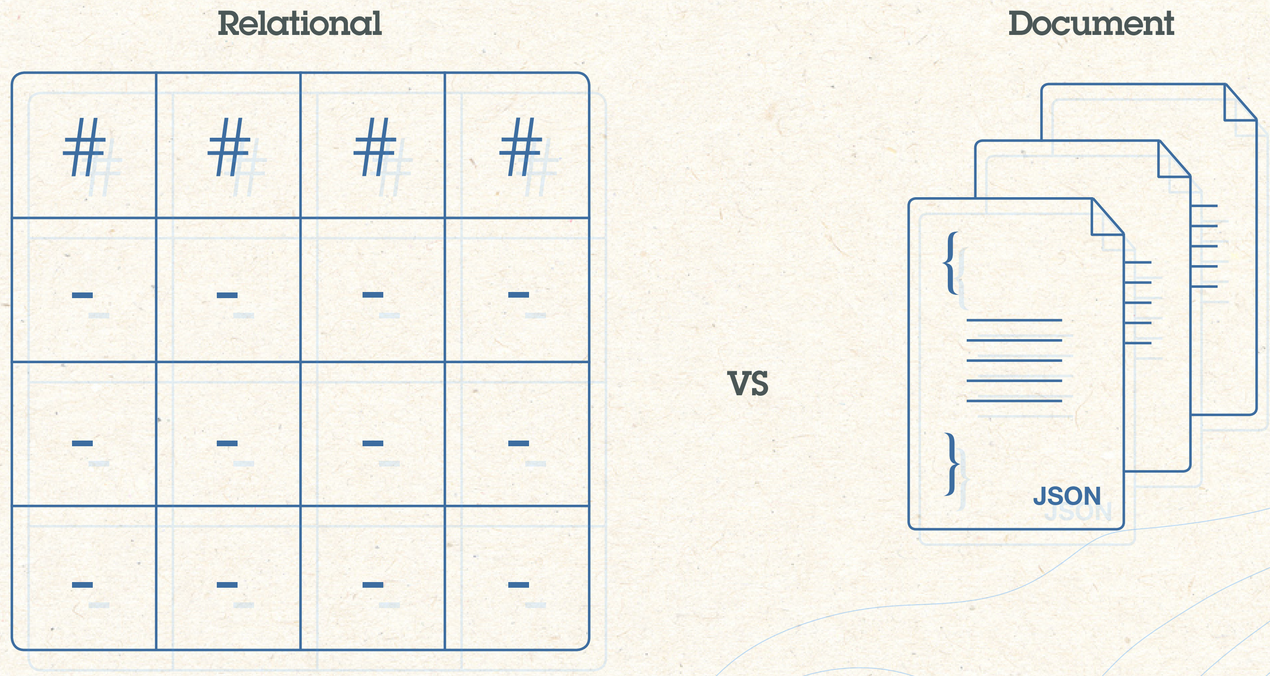
IBM

Choose Your Database Wisely...

There are many types of databases and data analysis tools to choose from when building your application. Should you use a relational database? How about a key-value store? Maybe a document database? Is a graph database the right fit? What about polyglot persistence and the need for advanced analytics?

If you feel a bit overwhelmed, don't worry. This guide lays out the various database options and analytic solutions available to meet your app's unique needs.

You'll see how data can move across databases and development languages, so you can work in your favorite environment without the friction and productivity loss of the past.



Brief History of Databases

It's a New, Non-relational World

Developers face two major questions when setting up a database: How do I set up indices for fast query and retrieval? And how useful (intuitive, expressive, optimizable, standardized) is my query language?

Starting in the 1970s – and for a period of several decades after – the answer to these questions was nearly always a relational database. These databases – characterized by schemas for data in tables, organized by rows and columns, following the principles of relational algebra – were the sole choice for developers. Persistence, consistency, integration, transactions, reporting – relational databases had it all.

Instead of running relational databases on a single, powerful server, many developers experimented with partitioning (or "sharding") their relational database management systems (RDBMS), and running them as a distributed system (in a "cluster" of servers). But developers found that relational databases suffered from data concurrency problems between primary write server nodes and read-only replicas, had trouble with access speeds, and couldn't efficiently conduct join operations between tables residing on different nodes. In this new environment, horizontal scaling became more important than vertical scaling, and eventual consistency became an acceptable alternative to immediate consistency.

SQL Heritage

In 1970, IBM's Edgar F. Codd released "A Relational Model of Data for Large Shared Data Banks," a paper which proposed a database structure built on the relational data model.

A few years later, two of Codd's IBM colleagues, Donald D. Chamberlin and Raymond F. Boyce, developed "SEQUEL" (Structured English Query Language) – later dubbed "SQL" due to an existing trademark.

Together, these concepts would form the foundation for the relational database. It's upon this historical foundation that IBM is now contributing to, and building services upon, NoSQL data stores with roots in open source.

#NotOnlySQL

In the mid-2000s, two emerging web giants proposed new models for storing and retrieving data. Google's 2006 "A Distributed Storage System for Structured Data" made the case for BigTable, its column-oriented database system. One year later, Amazon pitched its Dynamo database, offering high availability for web apps with a distributed key-value storage system.

These papers laid the groundwork for databases using new query languages and more flexible schemas that would provide greater scalability, speed and developer ease of use, and in 2009, a meet-up in San Francisco gave a single identity to these various databases springing up. The term "NoSQL" actually started as a hashtag for that meet-up, but it later grew to encompass all databases that don't use the SQL query language ("#NotOnlySQL" would have been a more accurate, albeit not as succinct, hashtag).

This group of non-relational databases (including key-value, column family, document and graph stores) offers developers a more familiar, pleasant experience when it comes to persisting app objects, in large part because they more naturally represent data structures used in software development.

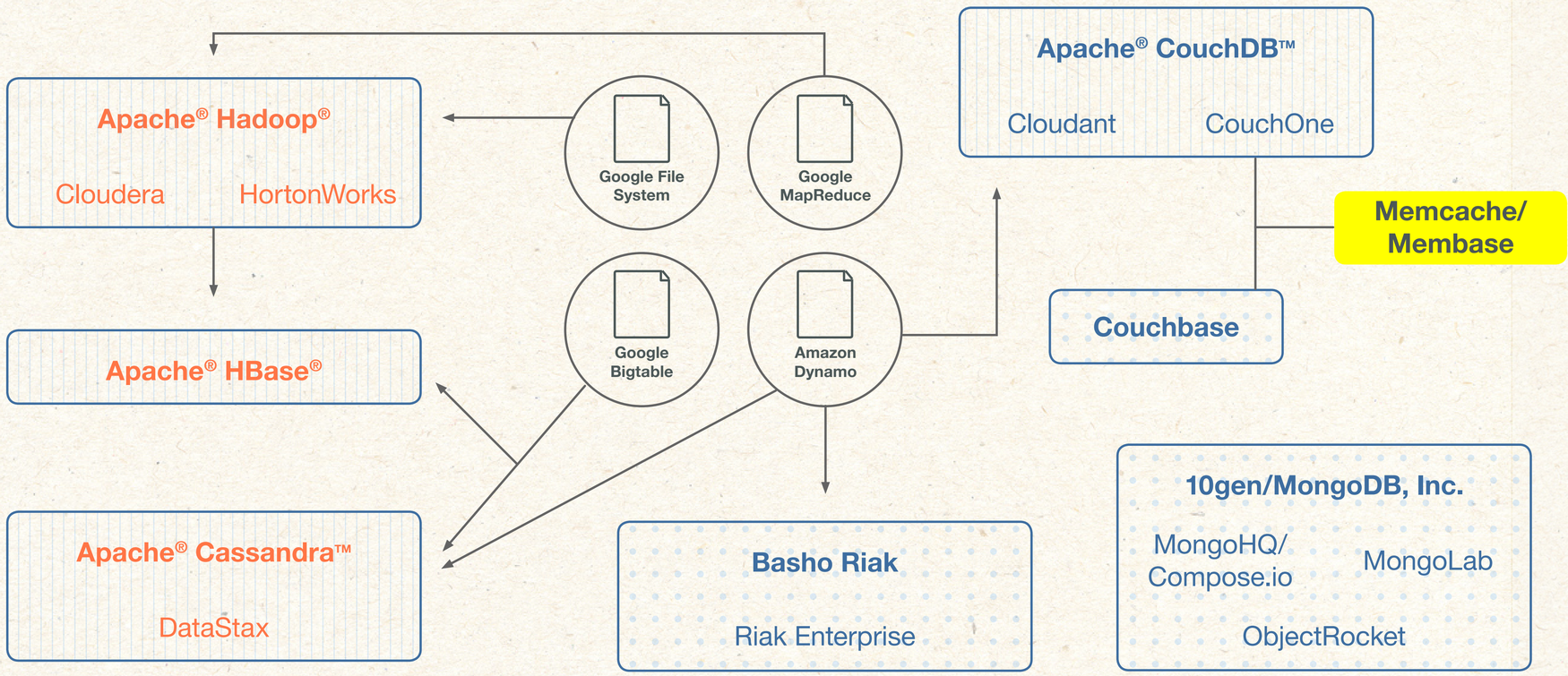
The Rise of NoSQL

The success of NoSQL stands in contrast with an earlier challenger to RDMS dominance, the object database. Software projects of the pre-web era tended to integrate data rather than services, which prevented object databases from ever fully taking off among developers.

But in the post-Dot-Com Boom world, with the introduction of web APIs, the right NoSQL databases can be paired with corresponding microservices – often in multi-database architectures that leverage the most logical DB for each part of the application.

NoSQL Family Tree

Understanding the Architecture that Runs Tomorrow's Web



Open Source Roots

Another factor instrumental in the rise in popularity of non-relational databases was their strong roots in the open source community. Databases born in open source already have a base of users – users who introduced these tools into their working environments.

As the usage of open source databases grew, companies began providing their own commercial versions, outfitted with additional support and services (e.g. IBM® Cloudant® for Apache CouchDB, and Datastax for Apache Cassandra). They also contributed to these existing communities, by funding related research projects, donating technology to open source ecosystems, and educating the next generation of developers and data scientists.

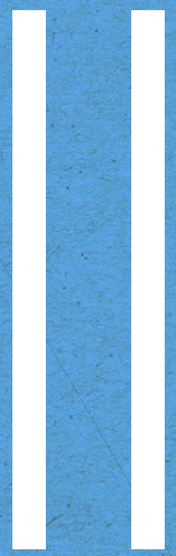
Enter Database-as-a-Service

With a name to organize under, broad communities supporting them, and the rise of the cloud providing a new means to deliver database services, the decade beginning in 2010 kicked off with developers turning to polyglot persistence – an application stack environment with several database types working together – and Database-as-a-Service (DBaaS).

Today, there are a variety of DBaaS options that developers can use in conjunction with one another to meet the unique needs of their app, as they operate in cloud- and internet-based environments. In this new realm, many deploy relational and non-relational together, which provides flexibility, scalability, availability, lower operational cost and specialized capabilities.

“Today, there are a variety of DBaaS options that developers can use in conjunction with one another to meet the unique needs of their app...”

Key-Value Stores



The O.G. NoSQL

As we know, all good things start somewhere, and dating back to the Indexed Sequential Access Method (ISAM) days of the 1970s, key-value is actually the oldest NoSQL model. It represents the most basic type of non-relational database where each item in the database is stored as an attribute name - or key - together with its value. Another popular key-value database, Berkeley DB, originated at the UC-Berkeley Computer Systems Research Group in the early 1990s, and is still in wide use. However, it was Amazon's Dynamo 2007 whitepaper that really sparked the use of key-value stores in distributed systems.

One significant takeaway from the whitepaper was that Dynamo enables high availability. When several copies of a database are spread across many nodes, the process by which they coordinate their response is known as "quorum." With a Dynamo-style quorum (like you'd find in DynamoDB, IBM Cloudant or Riak), the process can be tuned to provide looser or stricter levels of consistency. Typically, a relaxed consistency model means that if part of a cluster goes down, the data can still be accessed and consistency can be coordinated when all nodes are back online. For many applications, it's often better to be able to access stale data than having no access at all.

Schema-less Storage

Key-value stores also allow the application developer to store schema-less data. Key-value databases were born from the need to support rapid scaling data, and are commonly used in scenarios requiring a constant stream of small reads and writes, such as user preference or profile stores, product recommendations and real-time digital advertising.

Key-value stores are extremely useful for applications that only query data by a single key. Performance and scalability, combined with the simplicity of data access patterns, are the main attractions of these types of databases.

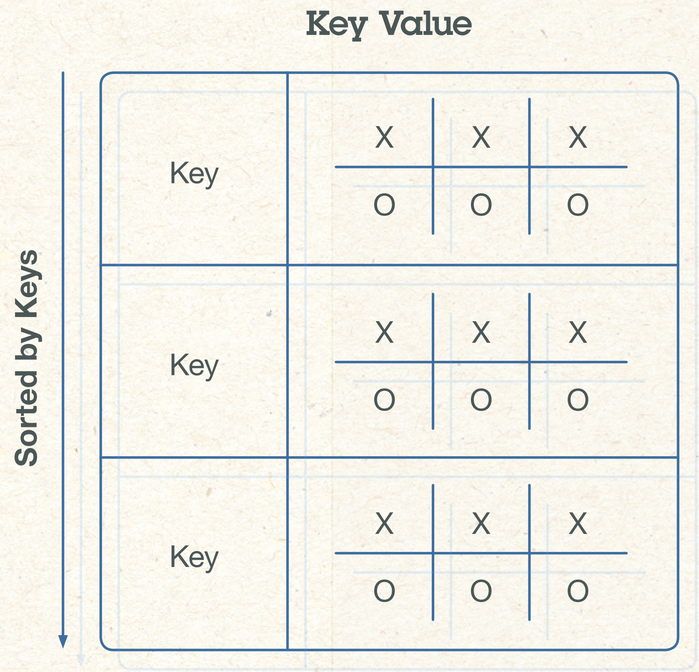


Figure 1

Get, Put, Delete

In the key-value structure, the key is usually a simple string of characters, and the value is a series of uninterrupted bytes that are opaque to the database. The data itself is usually some primitive data type (string, integer, array) or a more complex object that an application needs to persist and access directly. This replaces the rigidity of relational schemas with a more flexible data model that allows developers to easily modify fields and object structures as their applications evolve.

In general, key-value stores have no query language. They simply provide a way to store, retrieve and update data using simple get, put and delete commands. The path to retrieve data is a direct request to the object, whether it is in memory or on disk. The simplicity of this model makes a key-value store fast, easy to use, scalable, portable and flexible.

The [Key-]Value of Simplicity

A key-value store consists of a set of tables with keys paired to objects (values), like a hash table in computer programming. There's no comparison, aggregation or sorting of records. And because values are opaque, it removes the need to index the data to improve performance. You cannot, however, filter or control what's returned from a request based on the value.

Key-value stores scale out by implementing sharding, replication and auto recovery. They scale up by maintaining the database in RAM and minimizing the effects of ACID guarantees (a guarantee that committed transactions persist somewhere) by avoiding locks, latches and low-overhead server calls.

Cut Down on Queries

Key-value stores are generally good solutions if you have a simple application with only one kind of object, and you only need to look up objects based on one attribute. The simplicity of key-value stores also may make them the easiest to use.

Say you're running a search engine, and lots of people are searching for news around a particular topic. Rather than the same queries hitting the database over and over again, the queries and their corresponding result sets can be cached in a key-value store, where the query string is the "key" and the "value" is a list of relevant news articles. That way, the database is better able to respond to new queries and stash results in the cache. Thus, the key-value store acts as a cache, reducing read-load on the RDBMS server.

How it Works

Some key-value databases, such as Redis, support familiar data structures, including lists, sets, hashes and sorted sets. They enable users to read and write values by using a key:

- **Get (key), returns the value associated with the provided key**
- **Put (key, value), associates a value with the key**
- **Multi-get (key1, key2, etc.), returns the list of values associated with the list of keys**
- **Delete (key), removes the entry**

Document Stores

What's Up, Doc?

The term "document" can be confusing. A document store does not have much to do with "documents" in the usual sense - it does not refer to a letter, book or article. In this case, a document refers to a data record that is self-describing in regards to the data elements it contains.

A document is an object and keys (strings) can have values of recognizable types, including numbers, booleans, strings, as well as nested arrays and dictionaries. Documents can also contain BLOBs, called attachments. You can add, change or remove attachments in a very similar way as you would add, change or remove key-value data in a doc.

Ask and Ye Shall Receive

At its core, a document database can be considered a key-value store with one major exception: Instead of persisting opaque values, a document database requires the data to be stored in a format that the database can understand (i.e. JSON, XML, etc.).

Some document databases require a driver to convert the binary representation of the on-disk data into these formats. Others store data natively in formats like JSON. With these databases, in which no driver is required, the only way to talk to the database is by using its HTTP API. Once the data is in the right format, you can enable queries on the data's values.

The overall concerns document databases address are simplicity and scalability, as well as fast iteration in development. Say you need to add a new field to your object to run a new feature in your app. With a doc store, there's no schema to update – just add the new field and go. No messing with ORMs or worrying about breaking someone else's feature.

Document Example: User Profile

```
{
"ID": 1,
   "First":
"Mike",
   "Last";
"Broberg",
   "Zip": "02135",
   "City": "Bos",
   "State": "MA" }
```

=

All data in a single document

User Info

KEY	First	Last	ZiP_id
1	Mike	Broberg	2
2	Bob	Loblaw	2
3	Tony	Bologna	2
4	Ricky	Ricardo	3

+

Address Info

ZiP_id	CITY	STATE	ZIP
1	Trenton	NJ	08608
2	Bos	MA	02135
3	KC	MO	64101
4	Billings	MT	59101

RDBMS Has Issues with Change

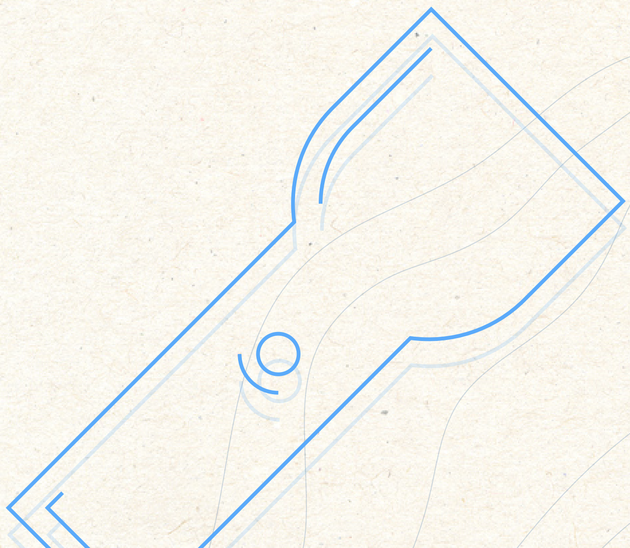
Document databases offer an alternative to relational databases, not a replacement. When developing an application with a relational database, all the app's data elements are first mapped to an abstract entity-relationship model, which defines the data, relationships and structure that will then subsequently be used to create relational table and column definitions.

This means that if (and when) the application data model changes, each of the corresponding table and column definitions need to be adjusted accordingly. Typically, these types of changes require the application developer to request the database administrator to aid in updating the schema.

Retrieve a Doc, Take a Ride

If you were to suspend all disbelief and think of a car stored in an RDBMS, it would get broken apart into its components, each then stored in a unique table for that component. (Here, the component is the data's unit of aggregation.) If you would like to take the entire car for a drive, the parts would need to be retrieved from the tables via JOINs and then put together into the combined result the user wants. If you want to make changes to the car, then you would most certainly need the help of a mechanic.

In this scenario, a document database, in which information is stored and accessed in a more aggregated form, is probably a better option. In a document store, an entire car would be stored as the unit of aggregation, and can be retrieved as a single unit (document), without the need to do all the JOINs like you'd have to do with an RDBMS.



Graph Databases

IV

Data Storage That Thinks Like You Do

A graph database is another example of technology created to fix a problem that relational databases simply weren't designed to handle.

The modern graph database, pioneered by Neo4j inventor Emil Eifrem, is a data storage and processing engine that makes the persistence and exploration of data and relationships more efficient. In graph theory, structures are composed of vertices and edges (data and connections), or what would later be called "data relationships." Graphs behave similar to how people think – in specific relationships between discrete units of data.

A way to understand graph databases is to imagine yourself in a room with a team of seven people. You're about to tackle a new project. What's the first thing you do? You start jotting down ideas on a whiteboard; within a few moments you find yourselves circling some concepts and connecting them with lines. You are modeling a graph.

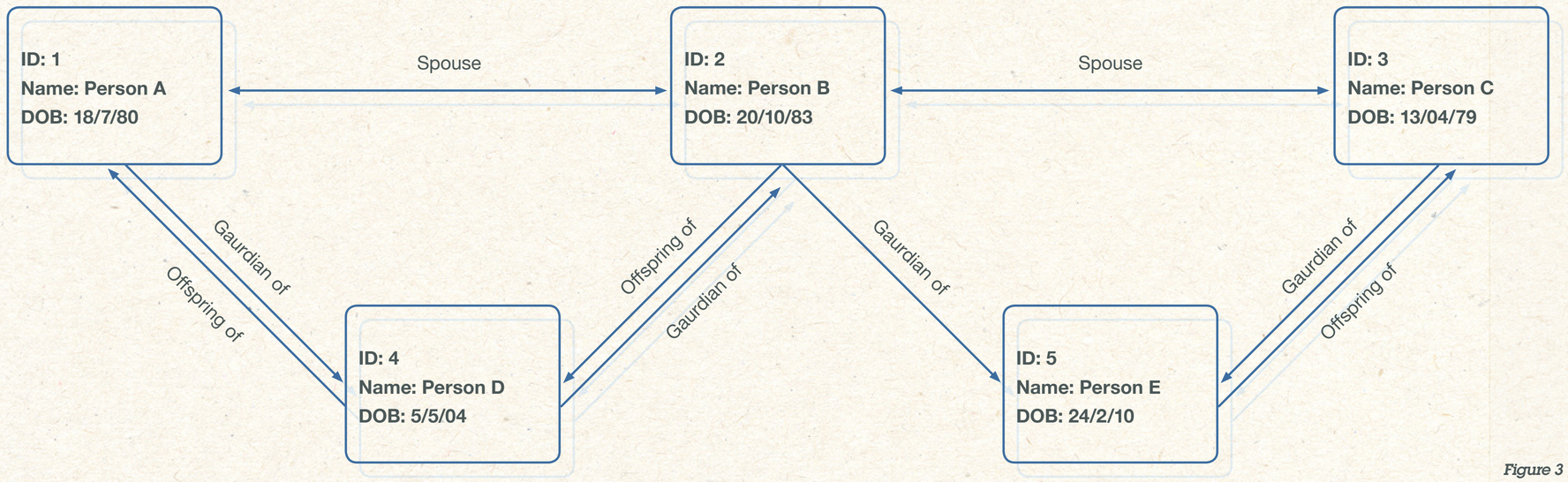


Figure 3

It's All About Relationships

Traditional database management systems store data in tabular form – rows and columns – but the information we deal with on a daily basis – including our knowledge of family members, coworkers and bank accounts – exists in relationships.

Why shoehorn data into tabs and rows if it only slows the work and complicates or restricts the analysis? By storing data relationships directly as a graph – made up of nodes or vertices, connected via relationships or edges to form a mesh of information – you reduce complexity and eliminate the extra work involved in transforming the data from the model to storage.

Querying graphs is all about traversal. Graph queries answer questions like, "Can I find a path through the graph that starts here and ends there?" and "What can I learn from the connections along that path?"

Think of a map, in which the nodes are street intersections and the edges are the roads connecting them. Each section of road probably has a name, a type, a length, a speed limit, etc. What do you learn from piecing together that path?

When you query a graph database, you get all the nodes (data points) that have a particular property and are related to other nodes. Both nodes and edges can store additional properties such as key-value pairs. This is important because all of the data in your organization is only important in how it relates to other data points. The value of the data rests in data relationships; graph databases let you get at that value more quickly and easily. Graph visualization of data relationships leads to a fuller picture of what's happening, so that users have better insight.

Catching Thieves in the Act

People often first associate graph data stores with social networks, but they're also ideal for many critical business functions. For example, fraud detection applications rely on crunching data and analyzing it for patterns that fall outside of normal statistical distributions. For relational databases, the time it takes to process this data often means catching thieves after the fact — not in the act. But graph databases can more efficiently explore data for these kinds of link analysis applications, so fraud detection can take place while crime is happening, and fraud can be stopped in real-time.

What's more, since graph databases are actually OLTP (transactional) as well as OLAP (analytical) databases, the fraud detection engine can be embedded within the transactional application without expected hits to performance. That's because graphs happen to be extremely fast in how they complete transactions (it has to do with storing the data and connections together, and the indexing of queried properties).

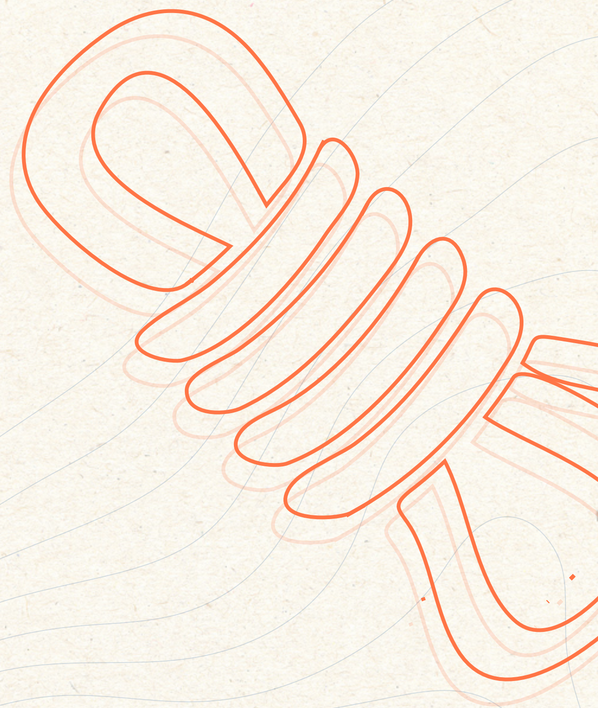
Friendly Suggestions

Another key use for graph databases is to serve up "rich" in-app recommendations, drawing from both current session and historical data, along with any personalization and sentiment analysis that may already exist about the user. This combination used to be tricky to pull off because it involved utilizing multiple different data stores and usually different data types.

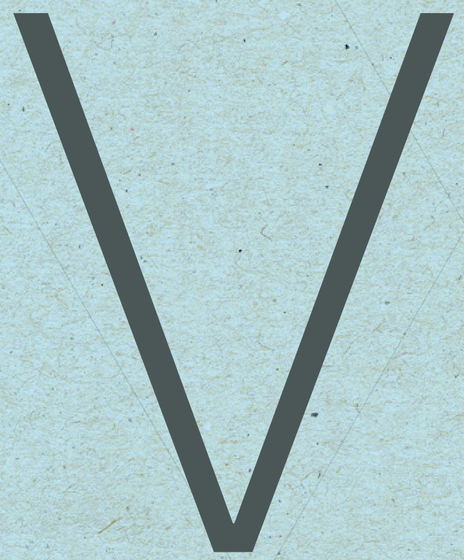
It's Important to Remember...

Remember that recommendations don't only have to be for consumer products (e.g. "since you bought this shirt, look at these matching slacks").

Recommendations can also serve up business-to-business data such as routing, network switches or inventory information. Similarly, the implications can be applied across industries including healthcare, security, telecommunications and many more.



Column Family Stores



All in the (Column) Family

Column family stores – like Apache Cassandra and Apache HBase – share a similar architectural structure with key-value stores, but are otherwise quite different from other non-relational databases.

"Column family" refers to the sets of columns that are the units of access control in this type of data store. Column family stores utilize hash maps (essentially a list of key-value pairs). They are organized into cells in corresponding columns. A record is a grouping of these columns. It's possible for columns to be "sparse," which means there is no schema forcing each record to have a corresponding entry in every column.

Column family stores enable very quick data access using a row key, column name and cell timestamp (see Figure 4).

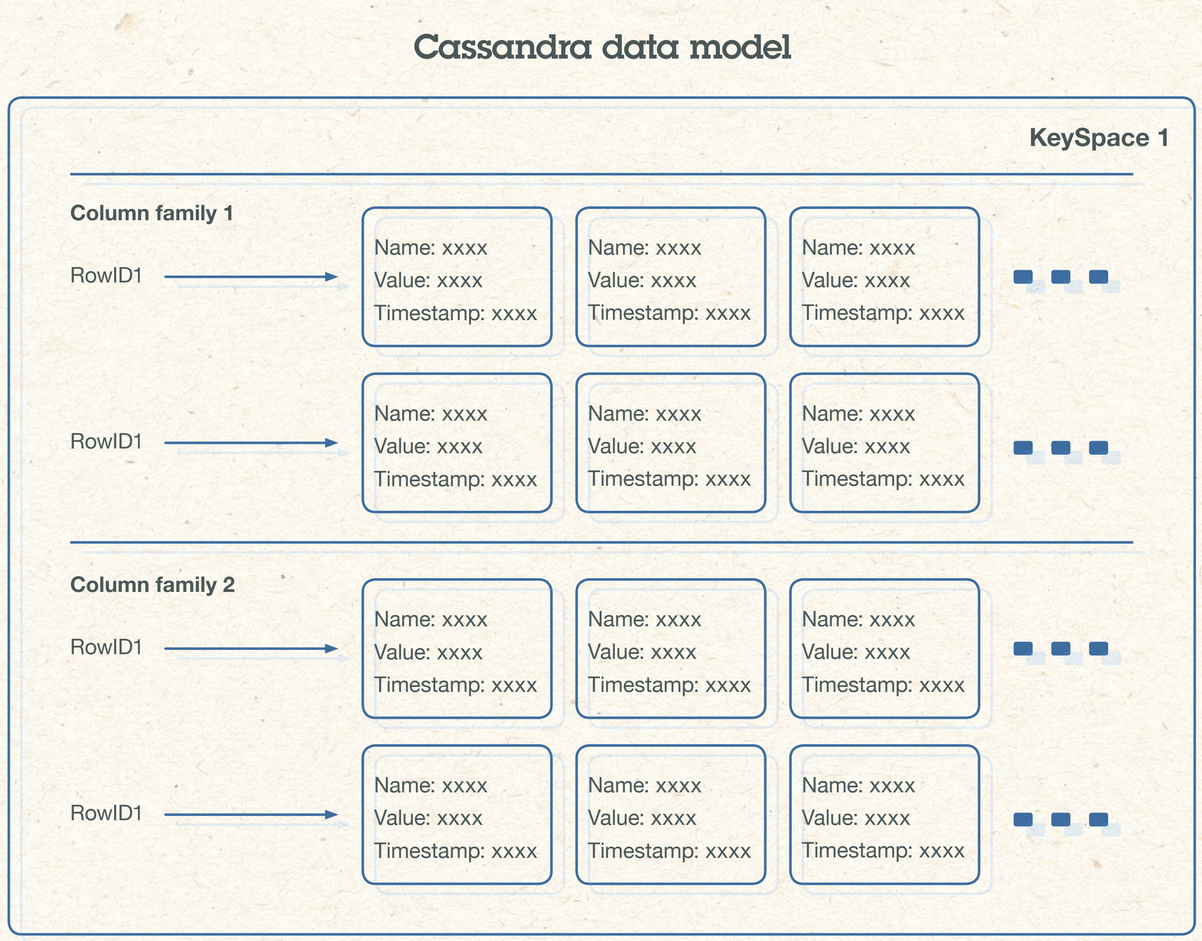


Figure 4

What You Get

Some defining characteristics of column family databases include:

- **High scalability and high availability**
- **No single point of failure**
- **High write throughput and good read throughput**
- **Flexible schemas**
- **Tunable consistency**
- **Support for replication**

The Column Family Tree

A quick clarification is in order: Column family stores and columnar databases (also called “column-oriented” databases) are not the same. They might share similar names and a few structural elements, but they’re two independent concepts.

As veteran DBAs will notice, columnar databases are actually a flavor of relational databases, but with data organized in columns, not rows, for querying. Meanwhile, column family data tables are stored in columns.

Column family stores are also not SQL-based. And being in columnar databases, the columns aren’t homogenous across rows.

Column Family vs. RDBMS

When arriving at a column family database, like Apache Cassandra, you will notice some major differences. Here are some things to look out for:

- **No support for ACID transactions** – there is no guarantee that batch operations will be carried out in an atomic fashion.
- **No support for JOINS** – if you need to join two column families, you must do it programmatically. However, because they store so much data in single columns, column family stores often circumvent the need for JOINs.
- **Keys must be unique** – if the same key has been used twice, it will overwrite data.
- **Secondary search indexes may not be supported** – you must instead build indexes using sort orders and slices.

Analytics and NoSQL

VI

Schema vs. Schema-less

Modern applications accumulate loads of data, much of it semi-structured. But, how do you gain insights from it?

The volume, velocity and variety (three V's) of semi-structured data – like in JSON documents – make it difficult to analyze. Many of the related challenges boil down to the fundamental dichotomy of schema vs. schema-less data structures. The "schema-less" nature of non-relational databases makes them difficult to apply traditional analytics and business intelligence (BI) tools.

To understand this, let's take a quick trip back in time.

Drilling into DB History

Once upon a time, most data was stored on a number of files that would repeat that data. For example, let's imagine a hardware store a few decades ago. This store is cutting edge for its time, and has implemented a bookkeeping system on a PC in the back office.

Each time a customer returned to the store to replace a drill bit, our hardware store's PC bookkeeping system would store their information in a table like this (see Figure 5):

Customer Sales Table

Sale_Date	Product_Name	Product_Price	Cust_Name	Cust_Addr	Cust_Zip
11/23/1987	Masonry Drill Bit Set (5 pc)	$15.00	George Carpenter	123 Main St	12345
11/25/1987	Wood Twist Drill Bit	$2.00	George Carpenter	345 Center St	12345
11/25/1987	Wood Twist Drill Bit	$2.00	George Carpenter	123 Main St	12345

Figure 5

Who is Georg3?!

There were two major pains to this method:

1. The data took up a lot of space in a world where storage was expensive (consider that today's smartphones have vastly more storage capacity than the first PCs). So, the repetition of data in these files could prove very costly.

2. Updating data was flawed. For example, say the storeowner wanted to get a list of customers for a holiday mailing list. If there were any errors in the spelling of a name with multiple instances – for example 'Georg3' instead of 'George' – a query of the sales table could create multiple entries for the same customer, resulting in that customer being mailed redundant Holiday cards. For a large retailer, the same scenario could add up to significant extra costs and lost customers.

Enter RDBMS

The original answer to these pain points was the relational database. With a relational database, customers could be assigned a Customer ID and would be separated into their own table. Here, their records could be updated and edited in one place. Then, using a Foreign Key, you could quickly determine the identity of the customer with a query that joined these tables. We would also do the same for the product data. Let's look at what that new model looks with Figure 6.

Sale Table

Sale Date	ProductID	CustomerID
11/23/1987	1	1
11/25/1987	2	2
11/25/1987	2	1

Customer Table

CustomerID	Cust Name	Cust Addr	Cust Zip
1	George Carpenter	123 Main St	12345
2	Jane Crafty	345 Center St	12345

Product Table

ProductID	Product name	Product price
1	Masonry Drill Bit Set (5 pc)	$15.00
2	Wood Twist Drill Bit	$2.00

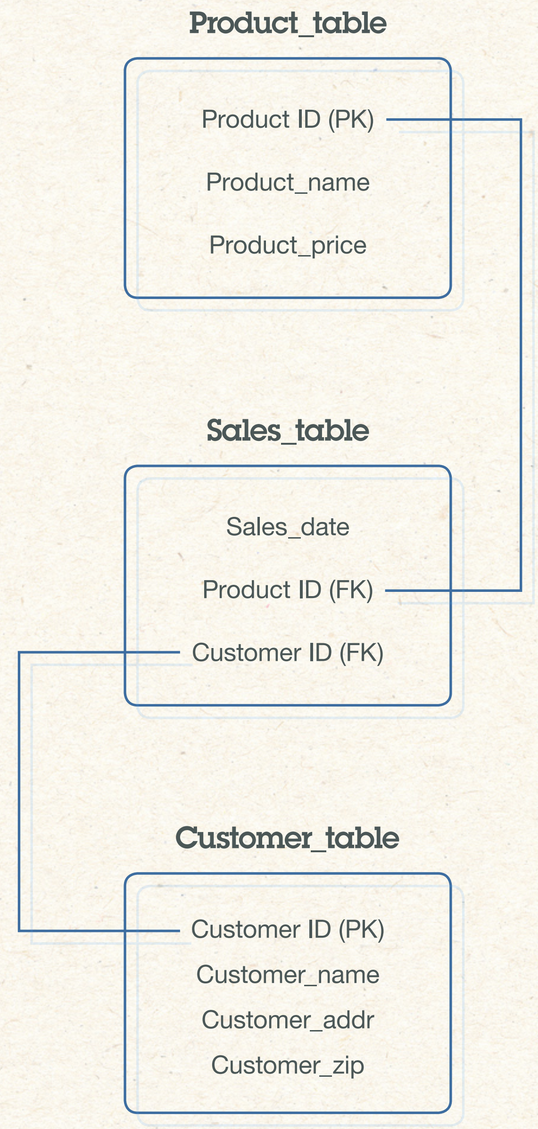
Figure 6

A Schema is Born

The new schema (represented here), with its rigid rules, won't allow you to enter in data that doesn't exist. For example, when entering a Foreign Key in the sales table, you can't put a record as a sale that has no Customer ID associated with it. The system breaks.

This is very helpful if the success of your app depends on not having junk data or not losing data in a sale. You now have much more control over the entities in your application. Adding an attribute to your Products table is considerably more simplistic over the cumbersome, time-consuming and error-prone method of inserting an extra field across a large number of records.

Going back to our hardware store example, what if the storeowner wanted to add the color of the product to that sales table? In our new schema world, we just need to add it to the Product table and assign the color. All legacy sales queries now have this new data available - you just have to run a report again with the new field specified.



But NoSQL is Schema-less…

Schemas are great for reporting and analytics. You know where things are. The schema is your map and you can pick which attributes you are going to analyze. In our previous example, it would be very easy to get the sum and count of sales by product, by zip code and by individual customers. And with the new product color field we added, we could do some digging on what colors sell the most and when.

But getting back to NoSQL, non-relational data stores lack a rigid schema. So, how can we explore data in the ways described above when it is stored in a non-relational format, as in JSON documents?

So How About Them Analytics?

The schema-less nature of non-relational databases is one of the key reasons for their popularity. The rigid schemas in relational databases – while great at maintaining data accuracy – can also bring web and mobile applications to a screeching halt, along with their millions of users. That's why non-relational databases have exploded in popularity; the structure of these databases is more flexible to additions and subtractions of attributes, which is great for powering scalable apps with global reach.

Again, the problem comes in when you want to start getting insights from data in a non-relational database. In most cases, analytics applications are going to be looking for a relational schema – but they won't find that in a JSON store, for example.

“In most cases, analytics applications are going to be looking for a relational schema – but they won't find that in a JSON store, for example.”

NoSQL Analytics Drivers

There are a few key reasons to think about analytics and NoSQL. Years ago, all you needed to get funding for a great app was the idea itself and a business plan. Now, with the massive amounts of data we have on hand, investors require quantitative proof for every aspect of your startup business before they commit.

Some metrics you can get fairly easily: number of app downloads, signups for a beta, number of page views, etc. But there are many more metrics around web events and user engagement that investors will ask about. How many users sign up but never return? How active are users? These kinds of numbers aren't just important to get funding. Ultimately, this is the data that should be driving most decisions for your software business.

Meet the Operational Data Store

Let's continue with the startup metaphor. As you start to grow and expand, you will want to utilize multiple data stores collecting information about your business. You may use a SaaS billing solution, for example, and you'll want to bring the data from that system into a central dashboard where you can compare sales and marketing trends. Or, you may want to explore user sentiments in a popular Reddit post about your app.

Unless you want to create a dashboard that makes API calls to every single SaaS system and operational database you're using, you need to think about how to get your data into a format that you can easily see historically across your business. To do so, you can bring data from your multiplicity of cloud-based data sources into a single operational data store (ODS).

Schema Discovery for Dummies

Once your data is centralized – in a JSON database like CouchDB, IBM Cloudant or MongoDB, for example – it can be "piped" to a variety of data stores and analytics tools optimized for visualizing and exploring data.

Of course, the JSON data will still need to be put into a schema for analysis. This can be done in multiple ways, but one simple approach is to leverage a schema discovery process (SDP) – like the one used in the integration between IBM Cloudant and IBM dashDB™, a managed columnar data warehouse. Cloudant's SDP will determine which schema in dashDB a given JSON document will fit into. It will also identify nested structures in your JSON and will create additional tables to hold that data appropriately, creating a key using the document ID native to your JSON document.

Embeddable BI

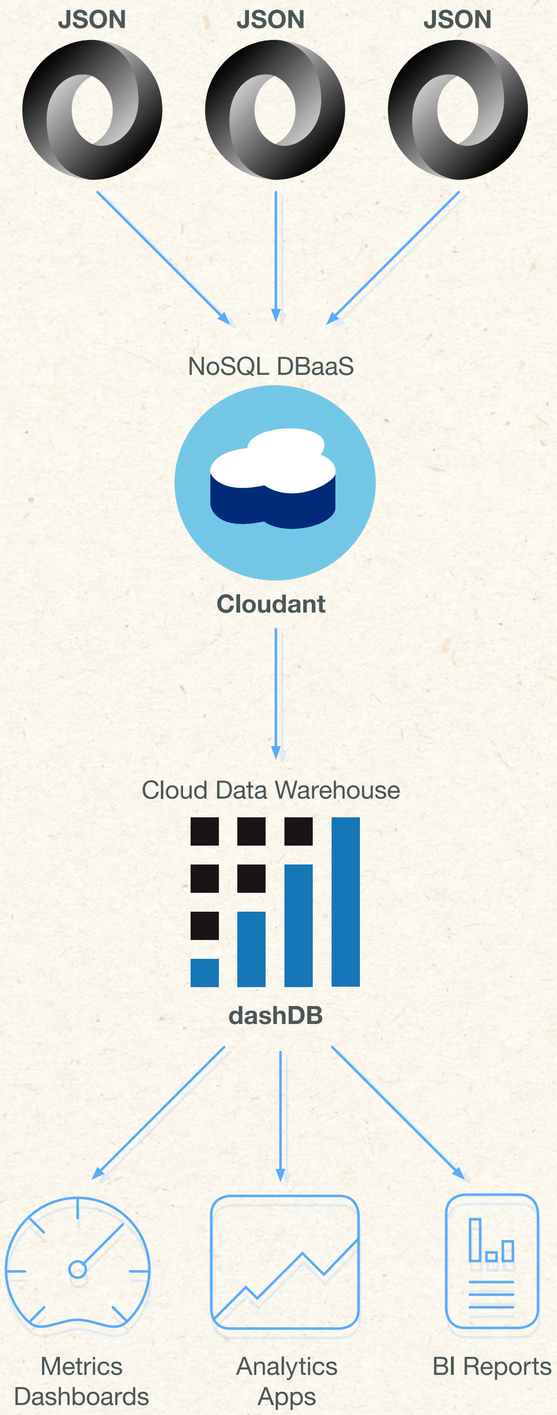
Once JSON data has been given a schema and is stored in a relational data warehouse, like dashDB, it is ripe for visualization, analytics and exploration using whatever SQL-compatible tools you like. This offers many opportunities for "embedded BI" – which means using metrics alongside the operational processes that your app is recording.

For example, if you have an app with the ability to enter data to perform daily workflows, embedded BI could enable a trended graph showing your user their history or any other helpful information collected in your data warehouse. Data that is coming from your data warehouse isn't just showing up on management reports, but informing your users in real-time about their usage.

Bridging NoSQL to SQL

The rise of non-relational databases has created great flexibility for app developers in terms of writing data and leveraging a format that is made for web technologies. When you look at the history of how the database has evolved, it can't go without notice that we are now using a data structure that is similar to how we started out - schema-less - and it solves the problems created by normalizing data in relational architectures.

With the decreasing costs of data storage and the inception of cloud platforms, the trend of data denormalization has been a vital force in accelerating innovation. By building a bridge from NoSQL to SQL, developers can easily move their application data, discover its schema, and connect on an integrated platform built for analytical advancement. (To learn more about doing this yourself, refer to the Simple Data Pipe sample application in chapter 8).

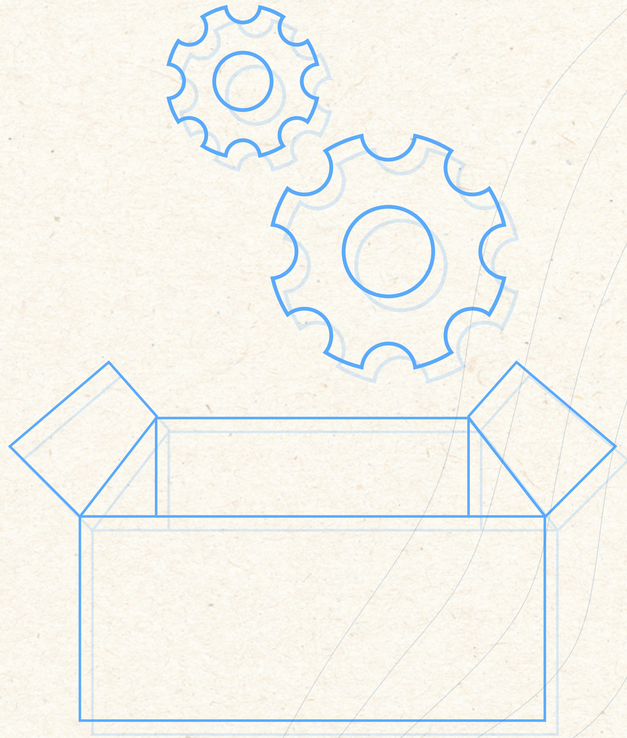


Open for Data

VII

Why Open Source?

Developing within an open source software (OSS) environment might at first seem confusing. You might be asking “Why would I give away my expertly crafted code into an open forum, rather than develop an idea fully, and possibly procure an IT patent on it?” While it’s true that there can be advantages to securing patents on original designs and other types of IT property, there are also many advantages to treading the OSS path.



The Benefits

The speed of software development is usually accelerated when at least a few experts are working to solve a challenge. Embedded within this development process are ongoing peer reviews and code testings that do not have to wait until formal development cycles end and testing cycles begin. The community of coding peers tends to find bugs and fix issues incredibly quick - much faster than was traditionally the case. And speaking of community, the participation and commitment of coders around the world helps to remove “brain trust” constraints and challenges around finding the right people.

The Challenges

While creating products built on open source has advantages, it also comes with a few challenges. By design, open source projects are built with a particular mindset. Multi-tenancy is a main pillar of many cloud-based services, but it wasn’t a primary focus when certain open source projects (which they rely on) were built. Additionally, many modern cloud, web and mobile offerings are always on, highly available and resilient, with guaranteed service level agreements (SLAs). How can those SLAs be fulfilled when products are built on open source technologies?

Here’s another difference in the OSS mindset versus traditional enterprise assumptions: By design, operational questions are not the first things developers think of when building an open source project. But operations and management concerns are a major component of enterprise IT decisions about environment, infrastructure and architecture. Marrying those two worlds does take some finesse — or at least a certain amount of project maturity.

Open Managed Platforms

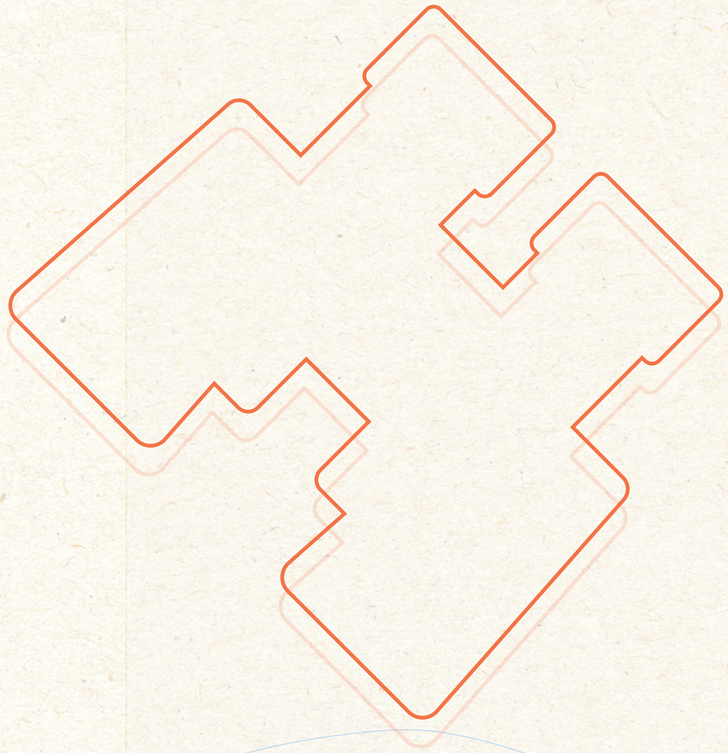
When open source technologies – including data stores – are offered up as a managed service, application developers can focus on building great applications, not obsessing about the infrastructure. They can pick the right database tool(s) for their project and install the cutting-edge features users now demand (for example, full-text search, real-time recommendations and geo-spatial functionality, all within a transactional application) without the headaches that normally come from managing such an application stack.

Polyglot Persistence

In so-called polyglot persistence environments, an application stack comprises several database types working together. It's often a good idea, especially with a legacy application, to keep the existing data store, especially if it is performing well, and integrate with new technology to enhance functionality (instead of going the "rip and replace" route). An example would be integrating a graph database into an existing data layer to build out a recommendation engine or add fraud detection to an app.

IT Likes It, Too

To the extent that data services are offered as a platform, integrations between technologies can become simpler, and provisioning can take place at the click of a mouse. What's more, enterprise IT managers also stand to gain increased visibility and control of their environments when using Platform-as-a-Service (PaaS) - it's no longer a situation where a one-off download of a rogue technology onto an individual laptop for experimentation ends up in production. Rather, it's a systematic provisioning of managed databases as services, with enterprise requirements enabled from the start.



Getting Started

VIII

Your Cloud DB Toolkit

Hopefully this guide has shown you how diverse and dynamic your "cloud database toolkit" really is. Getting a grasp of this rich cloud DB landscape is challenging, but it doesn't have to be scary and overwhelming. This section contains some relatively straightforward sample applications that you can build using IBM Cloud Data Services, entirely free of charge.

To learn more about IBM cloud databases for developers, you can head over to **http://ibm.biz/trytoolkit**

	Query language/style						Analytics query		Data format		
Database Type	REST API	ACID	SQL	Map/ Reduce	Java Script	Full text/ regex	R	Spark	JSON	XML	RDF
Key-value store											
Document											
Relational											
Column											
Graph											

Supported at some level

Not Supported (or a poor fit)

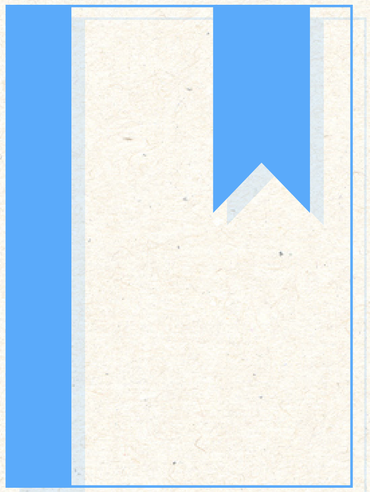
Simple Chrome Extensions

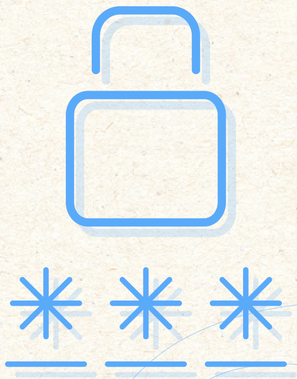
For something simple to get started, you can build two Google Chrome extensions – essentially, small web apps built from HTML, CSS and JavaScript that are downloaded and installed in a Chrome browser. The first is linkshare, a simple bookmarking tool, and the second is volt, a simple password database.

To enable offline functionality, these two sample Chrome extensions use PouchDB, an in-browser JSON document store which can sync with remote PouchDB, CouchDB or IBM Cloudant, a managed JSON database.

Learn More

Find on Github





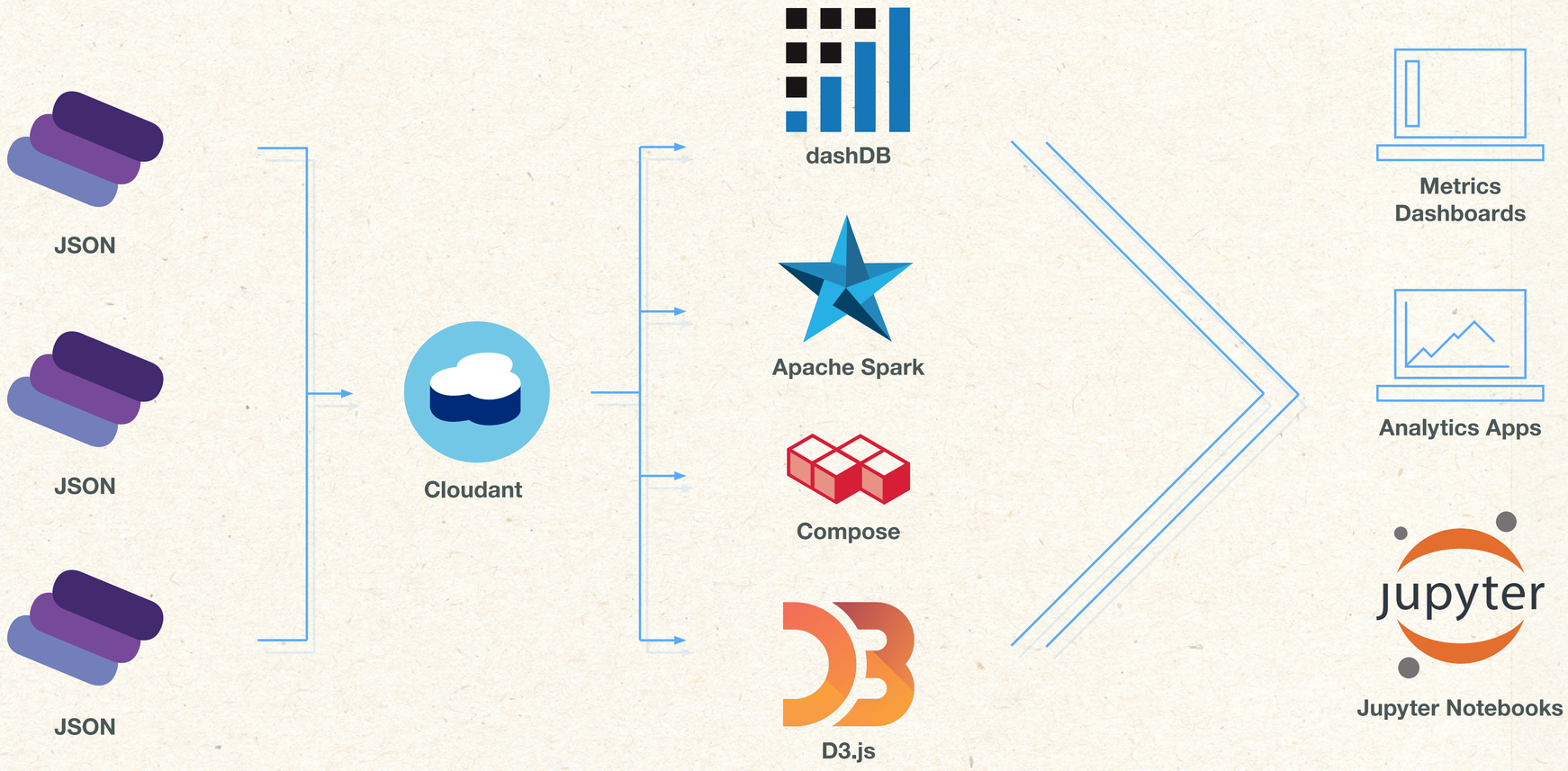
Simple Data Pipe

You can deploy the Simple Data Pipe application to connect to data behind multiple web APIs and land it in a single staging ground in its native form. There, you can analyze and explore the data with your analytic and BI tools of choice.

This app uses IBM Cloudant as the operational data store, and it provides prebuilt connections to many popular data sources, like SalesForce, Stripe and Reddit. It also easily moves data to IBM dashDB — a columnar database — for data warehousing, or to IBM Analytics for Apache® Spark™ for advanced analytics processing.

Learn More

Find on Github



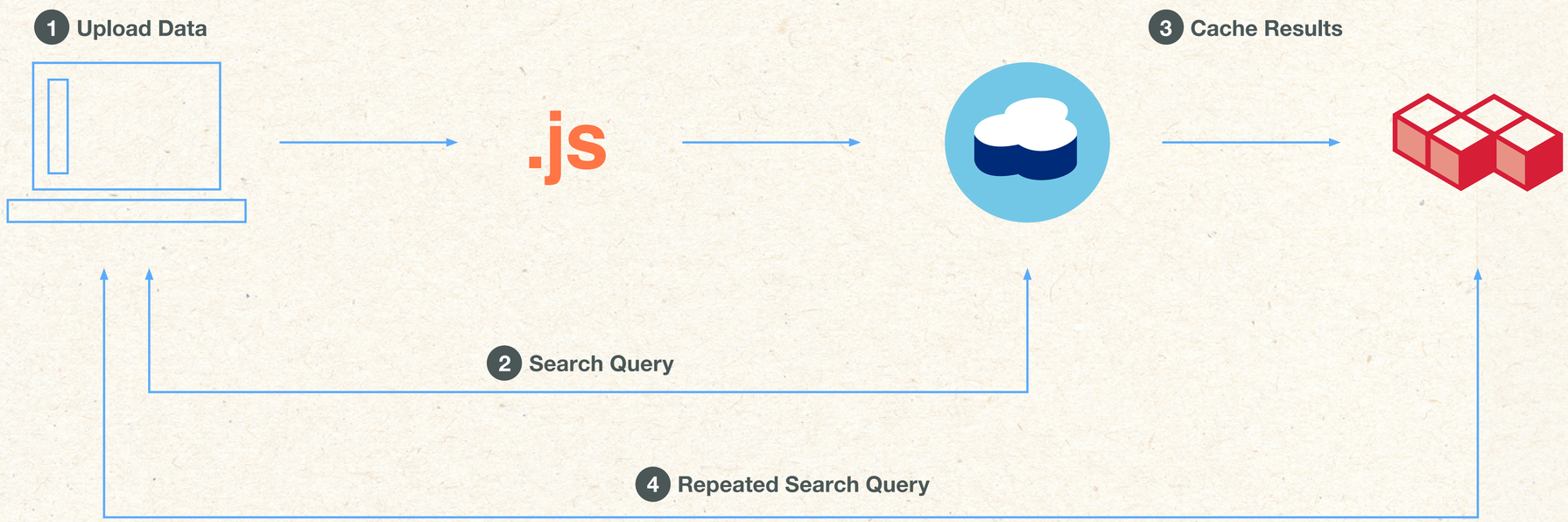
Simple Search Service

The Simple Search Service is a Node.JS application that will allow you to turn your spreadsheet or SQL dump into a full-fledged faceted search engine.

This app imports data into Cloudant in JSON format and uses a RESTful API to cache search results for faster retrieval. It can be further scaled by adding multiple nodes and a centralized cache using Redis by IBM Compose, a managed key-value database.

Learn More

Find on Github



About IBM Cloud Data Services

IBM Cloud Data Services provides developers, data science professionals and analytic architects with a broad portfolio of composable, integrated data services covering content, data and analytics. The open, self-service offers from Cloud Data Services speed up time to market, improve uptime and deliver higher value, backed by IBM technology leadership.

For information about how IBM Cloud Data Services is changing the way services are created for and delivered to developers, follow us on Twitter at **@IBMcloudant** and visit **ibm.biz/clouddataservices.**