Enterprise search is the practice of making content from multiple enterprise-type sources, such as databases and intranets, searchable to a defined audience. It is a type of vertical search that is **specifically designed to help employees find information within their organization**. Enterprise search systems typically index a wide range of data sources, including **File systems, Intranets, Document management systems, Emails, Databases, Collaboration tools, Knowledge bases, Customer relationship management (CRM) systems, Enterprise resource planning (ERP) systems.**
Here are some examples of how enterprise search can be used in different industries:

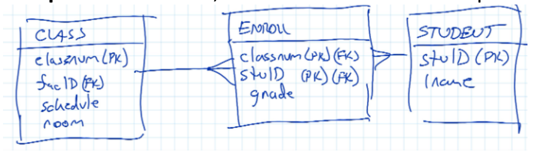
- Healthcare: Enterprise search can be used to help healthcare professionals find patient records, medical research, and other relevant information.
- Financial services: Enterprise search can be used to help financial professionals find financial data, investment reports, and other relevant information.
- Manufacturing: Enterprise search can be used to help manufacturing workers find product designs, engineering drawings, and other relevant information.
- Retail: Enterprise search can be used to help retail workers find product information, customer data, and other relevant information.
- Education: Enterprise search can be used to help students and faculty find research papers, course materials, and other relevant information.

nDCG10 scores are a measure of the ranking quality of a search engine or other information retrieval system. Between 0 and 1.

- *What type of database is this?* Databases come in a variety of genres, such as relational, key-value, columnar, document-oriented, and graph. Popular databases—including those covered in this book—can generally be grouped into one of these broad categories. You'll learn about each type and the kinds of problems for which they're best suited. We've specifically chosen databases to span these categories, including one relational database (Postgres), a key-value store (Redis), a column-oriented database (HBase), two document-oriented databases (MongoDB, CouchDB), a graph database (Neo4J), and a cloud-based database that's a difficult-to-classify hybrid (DynamoDB).
- *What was the driving force?* Databases are not created in a vacuum. They are designed to solve problems presented by real use cases. RDBMS databases arose in a world where query flexibility was more important than flexible schemas. On the other hand, column-oriented databases were built to be well suited for storing large amounts of data across several machines, while data relationships took a backseat. We'll cover use cases for each database, along with related examples.
- *How do you talk to it?* Databases often support a variety of connection options. Whenever a database has an interactive command-line interface, we'll start with that before moving on to other means. Where programming is needed, we've stuck mostly to Ruby and JavaScript, though a few other languages sneak in from time to time—such as PL/pgSQL (Postgres) and Cypher (Neo4J). At a lower level, we'll discuss protocols such as REST (CouchDB) and Thrift (HBase). In the final chapter, we present a more complex database setup tied together by a Node.js JavaScript implementation.
- *What makes it unique?* Any database will support writing data and reading it back out again. What else it does varies greatly from one database to the next. Some allow querying on arbitrary fields. Some provide indexing for rapid lookup. Some support ad hoc queries, while queries must be planned for others. Is the data schema a rigid framework enforced by the database or merely a set of guidelines to be renegotiated at will? Understanding capabilities and constraints will help you pick the right database for the job.
- *How does it perform?* How does this database function and at what cost? Does it support sharding? How about replication? Does it distribute data evenly using consistent hashing, or does it keep like data together? Is this database tuned for reading, writing, or some other operation? How much control do you have over its tuning, if any?
- *How does it scale?* Scalability is related to performance. Talking about scalability without the context of what you want to *scale to* is generally fruitless. This book will give you the background you need to ask the right questions to establish that context. While the discussion on *how* to scale each database will be intentionally light, in these pages you'll find out whether each database is geared more for horizontal scaling (MongoDB, HBase, DynamoDB), traditional vertical scaling (Postgres, Neo4J, Redis), or something in between.

Key features of NoSQL: Dynamic schema, Horizontal scalability, Document-based, key-value-based, column-based, Distributed and high availability,
Advantages of NoSQL: High Scalability, Flexibility, High availability, Scalability, Performance, Cost-effectiveness, Agility
Disadvantages: Lack of standardization, Lack of ACID compliance, Narrow focus, Open source, Lack of support for complex queries, Lack of maturity, Management challenge, GUI not available, Backup, Large document size. Flexibility and Performance.
Examples of NoSQL: **Graph Databases**: Amazon Neptune, Neo4j **Key value store:** Memcached, Redis, Coherence **Column:** Hbase, Big Table, Accumulo **Document-based:** Examples – MongoDB, CouchDB, Cloudant
When should NoSQL be used: When a huge amount of data needs to be stored and retrieved. The relationship between the data you store is not that important. The data changes over time and is not structured. Support of Constraints and Joins is not required at the database level. The data is growing continuously, and you need to scale the database regularly to handle the data.
Five models used to improve search: Relevance Model, Query understanding, Document understanding, Personalization, Ranking algorithm.
MongoDB is an excellent choice for an ever-growing class of web projects with large-scale data storage requirements but very little budget to buy big-iron hardware. Thanks to its lack of structured schema, Mongo can grow and change along with your data model. If you're in a web startup with dreams of enormity or are already large with the need to scale servers horizontally, consider MongoDB.
ObjectId: You may have noticed that the JSON output of your newly inserted town contains an _id field of type ObjectId. This is akin to SERIAL incrementing a numeric primary key in PostgreSQL. The ObjectId is always 12 bytes, composed of a timestamp, client machine ID, client process ID, and a 3-byte incremented counter.
Find: db.<col_name>.find({"_id": ObjectId("")}, {<col1>:1, <col2>:1})
var population_range = {$lt: 1000000,$gt: 10000} db.towns.find({ name : /^P/, population : population_range },{ name: 1 })
db.customers.find({$and:[{'tier_and_details.active': true},{'tier_and_details.tier': "Gold" }]},{username: 1, name: 1, email: 1})
db.customers.find({$or : [{ name: /^S/ }, { birthdate : { $gte: ISODate("2000-01-01") } }]},{ username: 1, name: 1, email: 1, birthdate: 1 })
Matching exact value: { famousFor : *'food'* }, **Matching Partial value:** { famourFor: /moma/ }, **All matching values:** { famousFor : { $all : [*'food'*, *'beer'*] } }
Lack of matching values: { famousFor : { $nin : [*'food'*, *'beer'*]}}, **Elem Match:** db.countries.find({'exports.foods' : {$elemMatch : {name : 'bacon',tasty : true }}},{_id:0, name:1 })
Update: db.customers.updateOne({_id: ObjectId("64fe3342344043214785fe37")}, {$set: {state : "MI"}}), db.customers.updateMany({'tier_and_details.active': true },{$set: {is_premium: true}})
Delete: db.customers.deleteOne({_id: ObjectId("64fe3342344043214785fe37")}), db.customers.deleteMany({'tier_and_details.tier': "Silver"})

Neo4j focuses more on the relationships between values than on the commonalities among sets of values (such as collections of documents or tables of rows). In this way, it can store highly variable data in a natural and straightforward way. In Neo4j, nodes inside of graphs act like documents because they store properties, but what makes Neo4j special is that the relationship between those nodes takes center stage.
CREATE (w:Wine {name:"Prancing Wolf", style: "ice wine", vintage: 2015}) MATCH(n) RETURN n; CREATE (p:Publication {name: "Wine Expert Monthly"})
MATCH (p:Publication {name: "Wine Expert Monthly"}),(w:Wine {name: "Prancing Wolf", vintage: 2015}) CREATE (p)-[r:reported_on]->(w)
MATCH ()-[r:short_lived_relationship]-() DELETE r -> Delete RelationShip, MATCH (e:EphemeralNode) DELETE e -> Delete Node
CREATE (Salaar:Movie {title: 'Salaar Part - 1', released: 2023, tagline:'Cease Fire'})
CREATE (Rohith:Person {name: 'Rohith Anugolu', born: 1997}) CREATE (Rohith)-[:ACTED_IN {roles:['Salaar Rajamannan']}] -> (Salaar) CREATE (Rohith)-[:DIRECTED]->(Salaar)
WITH Rohith as a MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d) RETURN a,m,d LIMIT 10;
MATCH(p:Person)-[:ACTED_IN]->(m:Movie) WHERE m.released >= 1990 AND m.released <= 1999 AND m.released - p.born = 35 RETURN p,m.released;
MATCH (p1:Person)-[:ACTED_IN]->(m1:Movie)<-[:DIRECTED]-(p2:Person), (p1)-[:ACTED_IN]->(m2:Movie)<-[:DIRECTED]-(p2)
WHERE m1 <> m2 RETURN DISTINCT p1.name AS Actor, p2.name AS Director, m1.title AS FirstMovie, m2.title AS SecondMovie
CONSTRAINT: CREATE CONSTRAINT FOR (n:Movie) REQUIRE (n.title) IS UNIQUE, **INDEX**: CREATE INDEX FOR (m:Movie) ON (m.released)
DELETE ALL NODES/RELATIONSHIPS: MATCH (n) DETACH DELETE n
Redis (REmote DIctionary Service) is a simple-to-use key-value store with a sophisticated set of commands. And when it comes to speed, Redis is hard to beat. Reads are fast and writes are even faster, handling upwards of 100,000 SET operations per second by some benchmarks. Redis can also be used as a blocking queue (or stack) and a publish-subscribe system. It features configurable expiry policies, durability levels, and replication options.
You've seen **transactions** in previous databases (Postgres and Neo4j), and Redis's **MULTI** block atomic commands are a similar concept. Wrapping two operations like SET and INCR in a single block will complete either successfully or not at all. But you will never end up with a partial operation. Let's key another shortcode to a URL and increment the count all in one transaction. We begin the transaction with the MULTI command and execute it with EXEC. When using MULTI, the commands aren't executed when we define them (like Postgres transactions). Instead, they are queued and then executed in sequence. Like ROLLBACK in SQL, you can stop a transaction with the **DISCARD** command, which will clear the transaction queue. Unlike ROLLBACK, it won't revert the database; it will simply not run the transaction at all. The effect is identical, although the underlying concept is a different mechanism (transaction rollback vs. operation cancellation).
Hashes are like nested Redis objects that can take any number of key-value pairs.
MSET user:luc:name "Luc" user:luc:password s3cret | MGET user:luc:name user:luc:password | HMSET user:luc name "Luc" password s3cret | HVALS user:luc (retrieve all) | HGET user:luc password | HINCRBY | HLEN | HGETALL (get all values for key) | HSETNX
Lists contain multiple ordered values that can act both as queues (first value in, first value out) and as stacks (last value in, first value out).
RPUSH eric:wishlist 7wks gog prag | LLEN eric:wishlist | LRANGE eric:wishlist 0 -1 | LREM eric:wishlist 0 gog | LPOP eric:wishlist | RPOP | LPUSH | RPUSH | RPOPLPUSH
Sets are unordered collections with no duplicate values and are an excellent choice for performing complex operations between two or more key values, such as unions or intersections.
SADD news nytimes.com pragprog.com | SMEMBERS news | SADD tech pragprog.com apple.com | SINTER news tech | SDIFF news tech | SUNION news tech | SUNIONSTORE websites news tech | SINTERSTORE | SDIFFSTORE | SMOVE | SCARD (count) | SREM news "nytimes.com" | SPOP
Database Normalization
In database management systems (DBMS), normal forms are a series of guidelines that help to ensure that the design of a database is efficient, organized, and free from data anomalies. There are several levels of normalization, each with its own set of guidelines, known as normal forms.
Important Points Regarding Normal Forms in DBMS
First Normal Form (1NF): This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries.
Second Normal Form (2NF): 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns.
Third Normal Form (3NF): 3NF builds on 2NF by requiring that all non-key attributes are independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table.
Boyce-Codd Normal Form (BCNF): BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key.
Fourth Normal Form (4NF): 4NF is a further refinement of BCNF that ensures that a table does not contain any multi-valued dependencies.
Fifth Normal Form (5NF): 5NF is the highest level of normalization and involves decomposing a table into smaller tables to remove data redundancy and improve data integrity.
Entity Relationship Model: The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related. The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically. The Entity Relationship Diagram explains the relationship among the entities present in the database. ER models are used to model real-world objects like a person, a car, or a company and the relation between these real-world objects. In short, the ER Diagram is the structural format of the database.
Components: Entities, Attributes and Relationships


CLASS
classnum (PK)
facID (FK)
schedule
room
ENROLL
classnum (PK) (FK)
stuID (PK) (FK)
grade
STUDENT
stuID (PK)
lname