DATABASES AND STORAGE

What is a Database?

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a Database Management System (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database.

What is DBMS?

A database typically requires a comprehensive database software program known as a Database Management System (DBMS). A DBMS serves as an interface between the database and its end-users or programs, allowing users to retrieve, update, and manage how the information is organized and optimized. A DBMS also facilitates oversight and control of databases, enabling a variety of administrative operations such as performance monitoring, tuning, and backup and recovery.

Components

Here are some common components found across different databases:

Schema

The role of a schema is to define the shape of a data structure, and specify what kinds of data can go where. Schemas can be strictly enforced across the entire database, loosely enforced on part of the database, or they might not exist at all.

Table

Each table contains various columns just like in a spreadsheet. A table can have as meager as two columns and upwards of a hundred or more columns, depending upon the kind of information being put in the table.

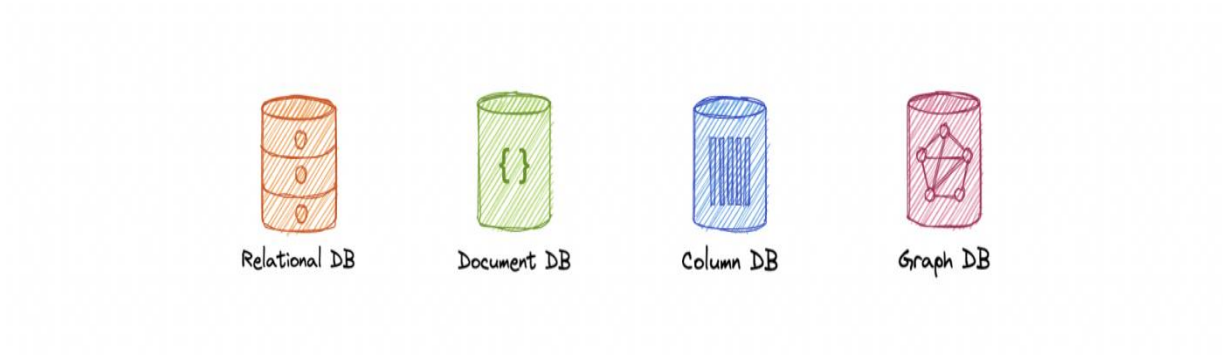
Column

A column contains a set of data values of a particular type, one value for each row of the database. A column may contain text values, numbers, enums, timestamps, etc.

Row

Data in a table is recorded in rows. There can be thousands or millions of rows in a table having any particular information.

Types



Below are different types of databases:

1. SQL
2. NoSQL

- Document
- Key-value
- Graph
- Time series
- Wide column
- Multi-model

Challenges

Some common challenges faced while running databases at scale:

1. **Absorbing significant increases in data volume:** The explosion of data coming in from sensors, connected machines, and dozens of other sources.
2. **Ensuring data security:** Data breaches are happening everywhere these days, it's more important than ever to ensure that data is secure but also easily accessible to users.
3. **Keeping up with demand:** Companies need real-time access to their data to support timely decision-making and to take advantage of new opportunities.
4. **Managing and maintaining the database and infrastructure:** As databases become more complex and data volumes grow, companies are faced with the expense of hiring additional talent to manage their databases.
5. **Removing limits on scalability:** A business needs to grow if it's going to survive, and its data management must grow along with it. But it's very difficult to predict how much capacity the company will need, particularly with on-premises databases.
6. **Ensuring data residency, data sovereignty, or latency requirements:** Some organizations have use cases that are better suited to run on-premises. In those cases, engineered systems that are pre-configured and pre-optimized for running the database are ideal.

Relational database management system (RDBMS)

A relational database like SQL is a collection of data items organized in tables.

- A SQL (or relational) database is a collection of data items with pre-defined relationships between them. These items are organized as a set of tables with columns and rows.

- Tables are used to hold information about the objects to be represented in the database.
- Each column in a table holds a certain kind of data and a field stores the actual value of an attribute.
- The rows in the table represent a collection of related values of one object or entity.
- Each row in a table could be marked with a unique identifier called a primary key, and rows among multiple tables can be made related using foreign keys.
- This data can be accessed in many different ways without re-organizing the database tables themselves.

SQL databases usually follow the ACID consistency model.

ACID is a set of properties of relational database transactions.

- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

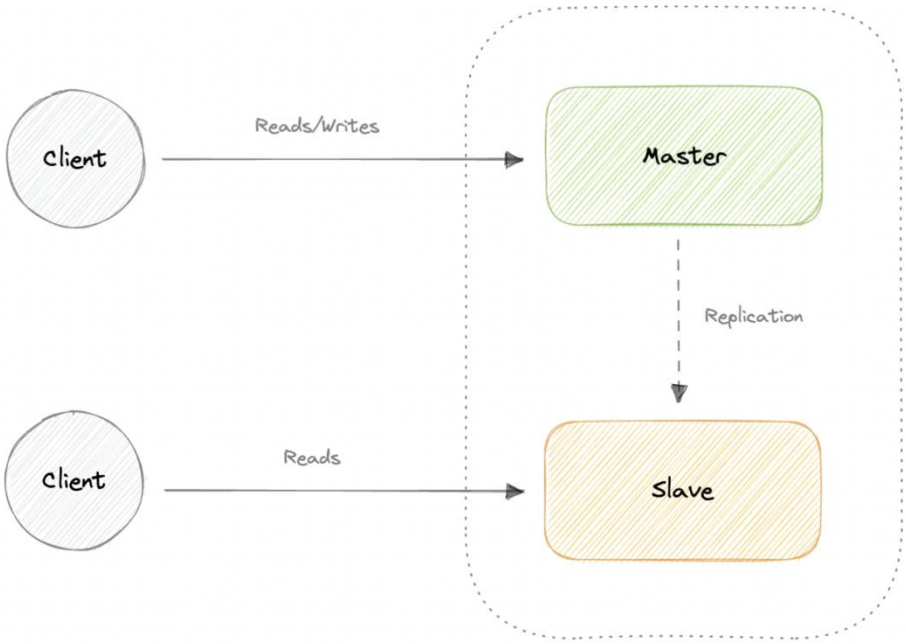
There are many techniques to scale a relational database: **master-slave replication, master-master replication, federation, sharding, denormalization,** and **SQL tuning.**

A. Database Replication

Replication is a process that involves sharing information to ensure consistency between redundant resources such as multiple databases, to improve reliability, fault-tolerance, or accessibility.

1. Master-Slave Replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.



Advantages

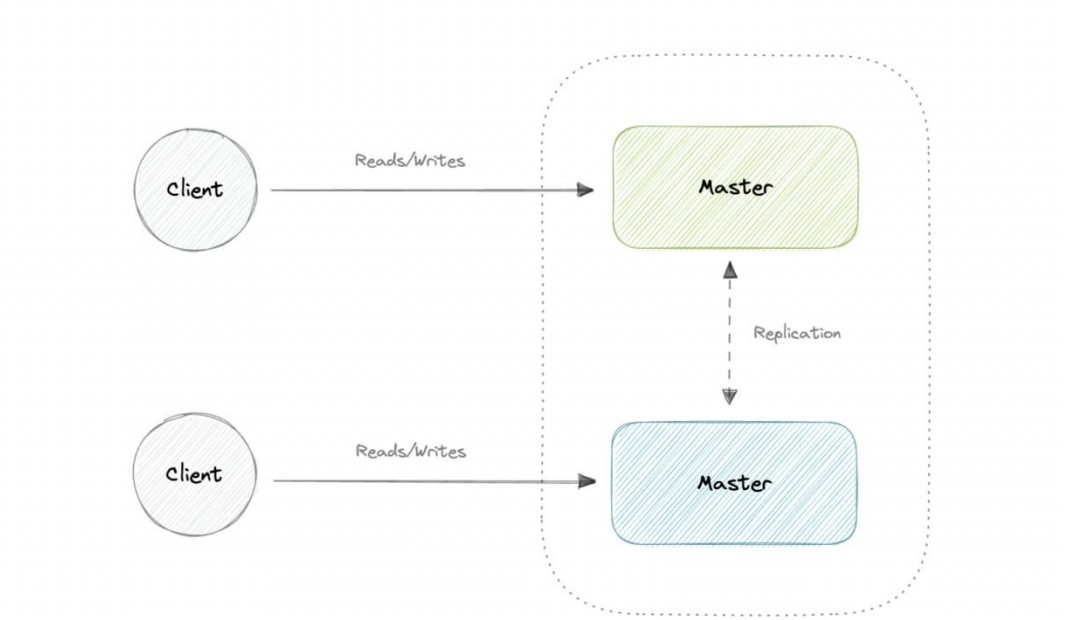
1. Backups of the entire database of relatively no impact on the master.
2. Applications can read from the slave(s) without impacting the master.
3. Slaves can be taken offline and synced back to the master without any downtime.

Disadvantages

1. Replication adds more hardware and additional complexity.
2. Downtime and possibly loss of data when a master fails.
3. All writes also have to be made to the master in a master-slave architecture.
4. The more read slaves, the more we have to replicate, which will increase replication lag.

2. Master-Master Replication

Both masters serve reads/writes and coordinate with each other. If either master goes down, the system can continue to operate with both reads and writes.



Advantages

1. Applications can read from both masters.
2. Distributes write load across both master nodes.
3. Simple, automatic, and quick failover.

Disadvantages

1. Not as simple as master-slave to configure and deploy.
2. Either loosely consistent or have increased write latency due to synchronization.
3. Conflict resolution comes into play as more write nodes are added and as latency increases.

Disadvantages: replication

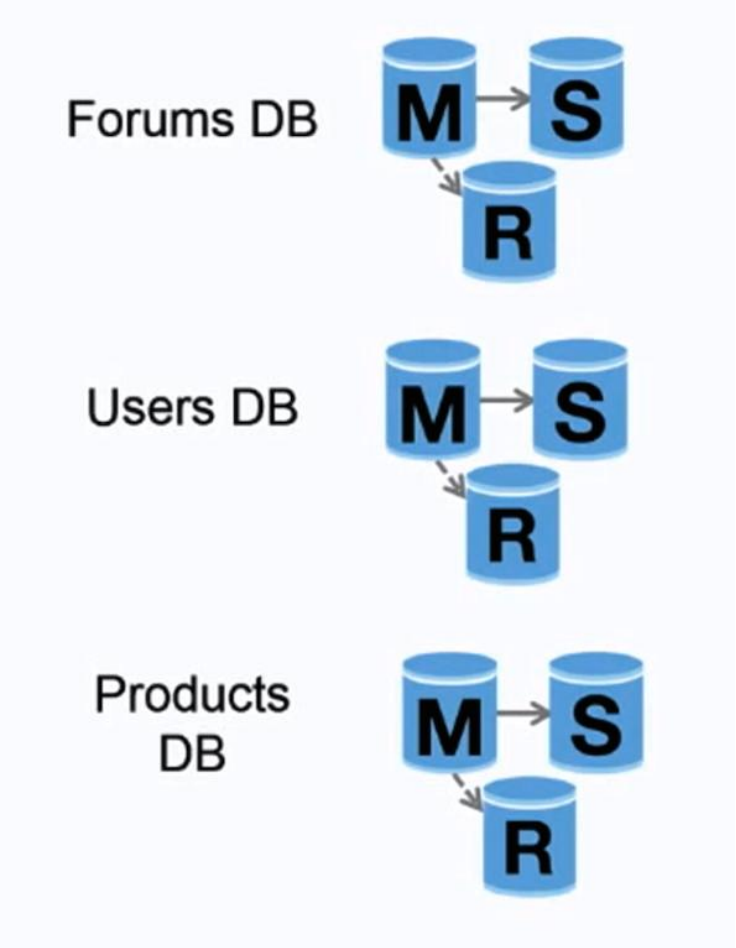
1. There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
2. Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
3. The more read slaves, the more you have to replicate, which leads to greater replication lag.
4. On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread.
5. Replication adds more hardware and additional complexity.

Source(s) and further reading: replication

1. Scalability, availability, stability, patterns

2. Multi-master replication

B. Federation



Federation (or functional partitioning) splits up databases by function.

For example, instead of a single, monolithic database, you could have three databases: forums, users, and products, resulting in less read and write traffic to each database and therefore less replication lag.

Smaller databases result in more data that can fit in memory, which in turn results in more cache hits due to improved cache locality. With no single central master serializing writes you can write in parallel, increasing throughput.

Disadvantages: federation

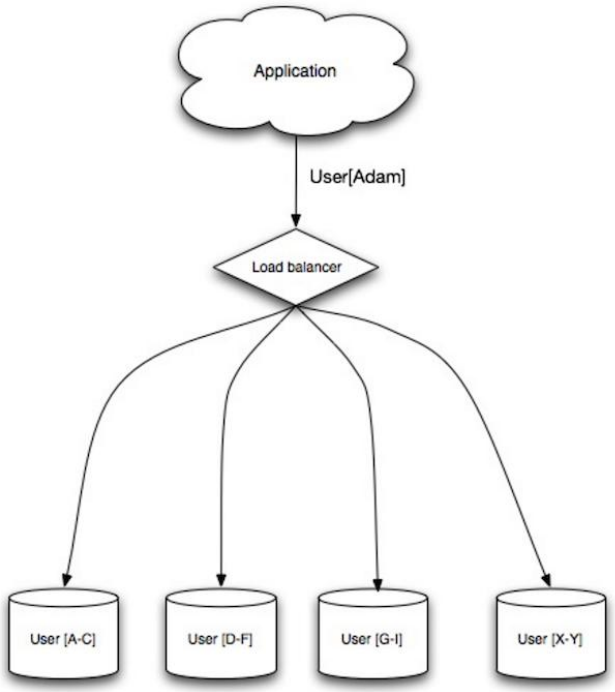
1. Federation is not effective if your schema requires huge functions or tables.
2. You'll need to update your application logic to determine which database to read and write.

3. Joining data from two databases is more complex with a server link.
4. Federation adds more hardware and additional complexity.

Sources and further reading: federation

1. Scaling up to your first 10 million users

C. Sharding



Sharding distributes data across different databases such that each database can only manage a subset of the data. Taking a users database as an example, as the number of users increases, more shards are added to the cluster.

Similar to the advantages of federation, sharding results in less read and write traffic, less replication, and more cache hits. Index size is also reduced, which generally improves performance with faster queries. If one shard goes down, the other shards are still operational, although you'll want to add some form of replication to avoid data loss. Like federation, there is no single central master serializing writes, allowing you to write in parallel with increased throughput.

Common ways to shard a table of users is either through the user's last name initial or the user's geographic location.

Disadvantages: sharding

1. You'll need to update your application logic to work with shards, which could result in complex SQL queries.
2. Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
3. Rebalancing adds additional complexity. A sharding function based on consistent hashing can reduce the amount of transferred data.
4. Joining data from multiple shards is more complex.
5. Sharding adds more hardware and additional complexity.

Sources and further reading: sharding

1. The coming of the shard
2. Shard database architecture
3. Consistent hashing

D. SQL tuning

SQL tuning is a broad topic and many books have been written as reference.

It's important to **benchmark** and **profile** to simulate and uncover bottlenecks.

1. **Benchmark** - Simulate high-load situations with tools such as ab.
2. **Profile** - Enable tools such as the slow query log to help track performance issues.

Benchmarking and profiling might point you to the following optimizations.

Tighten up the schema

1. MySQL dumps to disk in contiguous blocks for fast access.
2. Use CHAR instead of VARCHAR for fixed-length fields.
3. CHAR effectively allows for fast, random access, whereas with VARCHAR, you must find the end of a string before moving onto the next one.
4. Use TEXT for large blocks of text such as blog posts. TEXT also allows for boolean searches. Using a TEXT field results in storing a pointer on disk that is used to locate the text block.
5. Use INT for larger numbers up to 2^32 or 4 billion.
6. Use DECIMAL for currency to avoid floating point representation errors.
7. Avoid storing large BLOBS, store the location of where to get the object instead.
8. VARCHAR(255) is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.
9. Set the NOT NULL constraint where applicable to improve search performance.

Use good indices

1. Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices.
2. Indices are usually represented as self-balancing B-tree that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
3. Placing an index can keep the data in memory, requiring more space.
4. Writes could also be slower since the index also needs to be updated.
5. When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

Avoid expensive joins

Denormalize where performance demands it.

Partition tables

Break up a table by putting hot spots in a separate table to help keep it in memory.

Tune the query cache

In some cases, the query cache could lead to performance issues.

Sources and further reading: SQL tuning

1. Tips for optimizing MySQL queries
2. Is there a good reason i see VARCHAR(255) used so often?
3. How do null values affect performance?
4. Slow query log

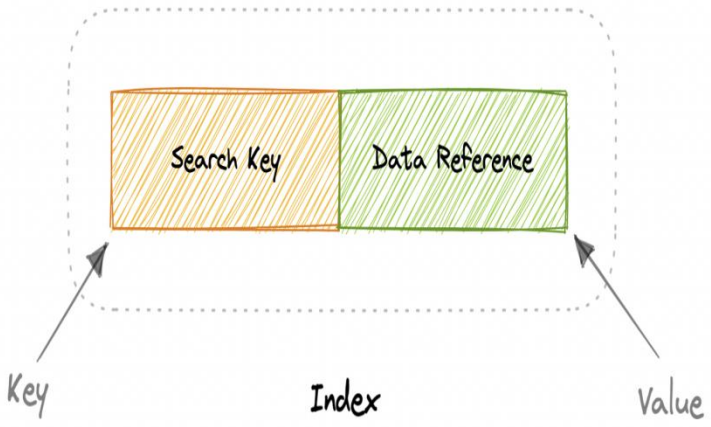
Synchronous vs. Asynchronous replication

The primary difference between synchronous and asynchronous replication is how the data is written to the replica. In synchronous replication, data is written to primary storage and the replica simultaneously. As such, the primary copy and the replica should always remain synchronized.

In contrast, asynchronous replication copies the data to the replica after the data is already written to the primary storage. Although the replication process may occur in near-real-time, it is more common for replication to occur on a scheduled basis and it is more cost-effective.

Indexes

Indexes are well known when it comes to databases, they are used to improve the speed of data retrieval operations on the data store. An index makes the trade-offs of increased storage overhead, and slower writes (since we not only have to write the data but also have to update the index) for the benefit of faster reads. Indexes are used to quickly locate data without having to examine every row in a database table. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access to ordered records.

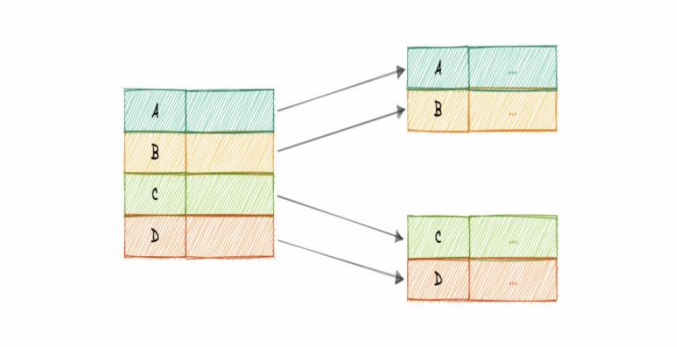


An index is a data structure that can be perceived as a table of contents that points us to the location where actual data lives. So when we create an index on a column of a table, we store that column and a pointer to the whole row in the index. Indexes are also used to create different views of the same data. For large data sets, this is an excellent way to specify different filters or sorting schemes without resorting to creating multiple additional copies of the data.

One quality that database indexes can have is that they can be **dense** or **sparse**. Each of these index qualities comes with its own trade-offs. Let's look at how each index type would work:

Dense Index

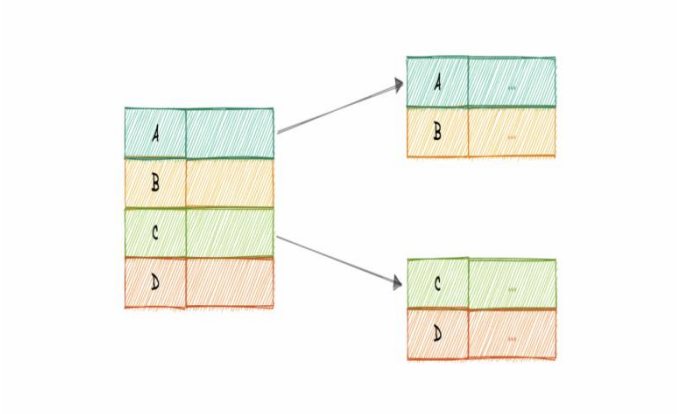
In a dense index, an index record is created for every row of the table. Records can be located directly as each record of the index holds the search key value and the pointer to the actual record.



Dense indexes require more maintenance than sparse indexes at write-time. Since every row must have an entry, the database must maintain the index on inserts, updates, and deletes. Having an entry for every row also means that dense indexes will require more memory. The benefit of a dense index is that values can be quickly found with just a binary search. Dense indexes also do not impose any ordering requirements on the data.

Sparse Index

In a sparse index, index records are created only for some of the records.



Sparse indexes require less maintenance than dense indexes at write-time since they only contain a subset of the values. This lighter maintenance burden means that inserts, updates, and deletes will be faster. Having fewer entries also means that the index will use less memory. Finding data is slower since a scan across the page typically follows the binary search. Sparse indexes are also optional when working with ordered data.

Normalization and Denormalization

Terms

Before we go any further, let's look at some commonly used terms in normalization and denormalization.

Keys

- **Primary key:** Column or group of columns that can be used to uniquely identify every row of the table.
- **Composite key:** A primary key made up of multiple columns.
- **Super key:** Set of all keys that can uniquely identify all the rows present in a table.
- **Candidate key:** Attributes that identify rows uniquely in a table.
- **Foreign key:** It is a reference to a primary key of another table.
- **Alternate key:** Keys that are not primary keys are known as alternate keys.
- **Surrogate key:** A system-generated value that uniquely identifies each entry in a table when no other column was able to hold properties of a primary key.

Dependencies

- **Partial dependency:** Occurs when the primary key determines some other attributes.
- **Functional dependency:** It is a relationship that exists between two attributes, typically between the primary key and non-key attribute within a table.
- **Transitive functional dependency:** Occurs when some non-key attribute determines some other attribute.

Anomalies

Database anomaly happens when there is a flaw in the database due to incorrect planning or storing everything in a flat database. This is generally addressed by the process of normalization.

There are three types of database anomalies:

1. **Insertion anomaly:** Occurs when we are not able to insert certain attributes in the database without the presence of other attributes.
2. **Update anomaly:** Occurs in case of data redundancy and partial update. In other words, a correct update of the database needs other actions such as addition, deletion, or both.
3. **Deletion anomaly:** Occurs where deletion of some data requires deletion of other data.

Example

Let's consider the following table which is not normalized:

ID	Name	Role	Team
1	Peter	Software Engineer	A
2	Brian	DevOps Engineer	B
3	Hailey	Product Manager	C
4	Hailey	Product Manager	C
5	Steve	Frontend Engineer	D

Let's imagine, we hired a new person "John" but they might not be assigned a team immediately. This will cause an insertion anomaly as the team attribute is not yet present.

Next, let's say Hailey from Team C got promoted, to reflect that change in the database, we will need to update 2 rows to maintain consistency which can cause an update anomaly.

Finally, we would like to remove Team B but to do that we will also need to remove additional information such as name and role, this is an example of a deletion anomaly.

Normalization

Normalization is the process of organizing data in a database. This includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make the database more flexible by eliminating redundancy and inconsistent dependency.

Why do we need normalization?

The goal of normalization is to eliminate redundant data and ensure data is consistent. A fully normalized database allows its structure to be extended to accommodate new types of data without changing the existing structure too much. As a result, applications interacting with the database are minimally affected.

Normal forms

Normal forms are a series of guidelines to ensure that the database is normalized. Let's discuss some essential normal forms:

1NF

For a table to be in the first normal form (1NF), it should follow the following rules:

1. Repeating groups are not permitted.
2. Identify each set of related data with a primary key.
3. Set of related data should have a separate table.
4. Mixing data types in the same column is not permitted.

2NF

For a table to be in the second normal form (2NF), it should follow the following rules:

1. Satisfies the first normal form (1NF).
2. Should not have any partial dependency.

3NF

For a table to be in the third normal form (3NF), it should follow the following rules:

1. Satisfies the second normal form (2NF).
2. Transitive functional dependencies are not permitted.

BCNF

Boyce-Codd normal form (or BCNF) is a slightly stronger version of the third normal form (3NF) used to address certain types of anomalies not dealt with by 3NF as originally defined. Sometimes it is also known as the 3.5 normal form (3.5NF).

For a table to be in the Boyce-Codd normal form (BCNF), it should follow the following rules:

1. Satisfied the third normal form (3NF).
2. For every functional dependency X → Y, X should be the super key.

There are more normal forms such as 4NF, 5NF, and 6NF but we won't discuss them here. Check out this amazing video that goes into detail.

In a relational database, a relation is often described as "normalized" if it meets the third normal form. Most 3NF relations are free of insertion, update, and deletion anomalies.

As with many formal rules and specifications, real-world scenarios do not always allow for perfect compliance. If you decide to violate one of the first three rules of normalization, make sure that your application anticipates any problems that could occur, such as redundant data and inconsistent dependencies.

Advantages

Here are some advantages of normalization:

1. Reduces data redundancy.
2. Better data design.
3. Increases data consistency.
4. Enforces referential integrity.

Disadvantages

Let's look at some disadvantages of normalization:

1. Data design is complex.
2. Slower performance.
3. Maintenance overhead.
4. Require more joins.

Denormalization

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database.

It attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins.

Once data becomes distributed with techniques such as federation and sharding, managing joins across the network further increases complexity. Denormalization might circumvent the need for such complex joins.

Note: Denormalization does not mean reversing normalization.

Advantages

1. Let's look at some advantages of denormalization:
2. Retrieving data is faster.
3. Writing queries is easier.
4. Reduction in number of tables.
5. Convenient to manage.

Disadvantages

1. Below are some disadvantages of denormalization:
2. Expensive inserts and updates.
3. Increases complexity of database design.
4. Increases data redundancy.
5. More chances of data inconsistency.

ACID and BASE consistency models

Let's discuss the ACID and BASE consistency models.

ACID

The term ACID stands for Atomicity, Consistency, Isolation, and Durability. ACID properties are used for maintaining data integrity during transaction processing.

In order to maintain consistency before and after a transaction relational databases follow ACID properties. Let us understand these terms:

Atomic

- All operations in a transaction succeed or every operation is rolled back.

Consistent

- On the completion of a transaction, the database is structurally sound.

Isolated

- Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

Durable

- Once the transaction has been completed and the writes and updates have been written to the disk, it will remain in the system even if a system failure occurs.

BASE

With the increasing amount of data and high availability requirements, the approach to database design has also changed dramatically. To increase the ability to scale and at the same time be highly available, we move the logic from the database to separate servers.

In this way, the database becomes more independent and focused on the actual process of storing data.

In the NoSQL database world, ACID transactions are less common as some databases have loosened the requirements for immediate consistency, data freshness, and accuracy in order to gain other benefits, like scale and resilience.

BASE properties are much looser than ACID guarantees, but there isn't a direct one-for-one mapping between the two consistency models. Let us understand these terms:

Basic Availability

- The database appears to work most of the time.

Soft-state

- Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

Eventual consistency

- The data might not be consistent immediately but eventually, it becomes consistent. Reads in the system are still possible even though they may not give the correct response due to inconsistency.

ACID vs BASE Trade-offs

There's no right answer to whether our application needs an ACID or a BASE consistency model. Both the models have been designed to satisfy different requirements. While choosing a database we need to keep the properties of both the models and the requirements of our application in mind.

Given BASE's loose consistency, developers need to be more knowledgeable and rigorous about consistent data if they choose a BASE store for their application. It's essential to be familiar with the BASE behavior of the chosen database and work within those constraints.

On the other hand, planning around BASE limitations can sometimes be a major disadvantage when compared to the simplicity of ACID transactions. A fully ACID database is the perfect fit for use cases where data reliability and consistency are essential.

NoSQL

NoSQL is a collection of data items represented in a **key-value store, document store, wide column store,** or a **graph database**. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor eventual consistency.

BASE is often used to describe the properties of NoSQL databases. In comparison with the CAP Theorem, BASE chooses availability over consistency.

- **Basically available** - the system guarantees availability.
- **Soft state** - the state of the system may change over time, even without input.
- **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

In addition to choosing between SQL or NoSQL, it is helpful to understand which type of NoSQL database best fits your use cases. We'll review **key-value stores, document stores, wide column stores**, and **graph databases** in the next section.

Key-value store

Abstraction: hash table

A key-value store generally allows for O(1) reads and writes and is often backed by memory or SSD. Data stores can maintain keys in lexicographic order, allowing efficient retrieval of key ranges. Key-value stores can allow for storing of metadata with a value.

Key-value stores provide high performance and are often used for simple data models or for rapidly-changing data, such as an in-memory cache layer. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

A key-value store is the basis for more complex systems such as a document store, and in some cases, a graph database.

Sources and further reading: key-value store

1. Key-value database
2. Disadvantages of key-value stores
3. Redis architecture
4. Memcached architecture

Document store

Abstraction: key-value store with documents stored as values

A document store is centered around documents (XML, JSON, binary, etc), where a document stores all information for a given object. Document stores provide APIs or a query language to query based on the internal structure of the document itself. Note, many key-value stores include features for working with a value's metadata, blurring the lines between these two storage types.

Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories. Although documents can be organized or grouped together, documents may have fields that are completely different from each other.

Some document stores like MongoDB and CouchDB also provide a SQL-like language to perform complex queries. DynamoDB supports both key-values and documents.

Document stores provide high flexibility and are often used for working with occasionally changing data.

Sources and further reading: document store

1. Document-oriented database
2. MongoDB architecture
3. CouchDB architecture
4. Elasticsearch architecture

Wide column store

Abstraction: nested map ColumnFamily <RowKey, Columns <ColKey, Value, Timestamp>>

A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table). Super column families further group column families. You can access each column independently with a row key, and columns with the same row key form a row. Each value contains a timestamp for versioning and for conflict resolution.

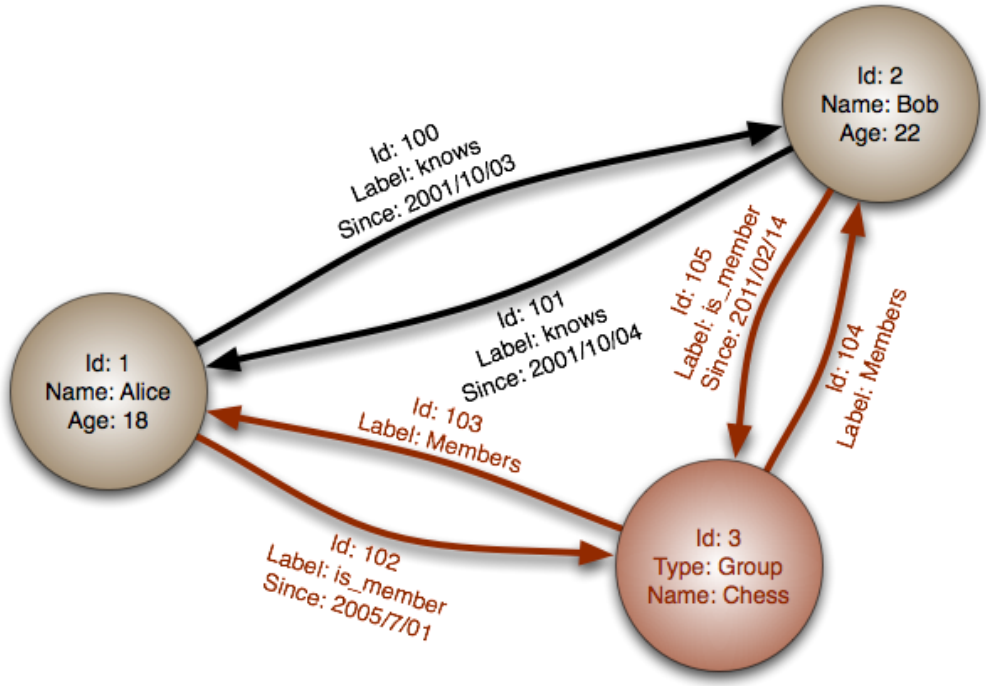
Google introduced Bigtable as the first wide column store, which influenced the open-source HBase often-used in the Hadoop ecosystem, and Cassandra from Facebook. Stores such as BigTable, HBase, and Cassandra maintain keys in lexicographic order, allowing efficient retrieval of selective key ranges.

Wide column stores offer high availability and high scalability. They are often used for very large data sets.

Sources and further reading: wide column store

1. SQL & NoSQL, a brief history
2. Bigtable architecture
3. HBase architecture
4. Cassandra architecture

Graph database



Abstraction: graph

In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

Graphs databases offer high performance for data models with complex relationships, such as a social network. They are relatively new and are not yet widely-used; it might be more difficult to find development tools and resources. Many graphs can only be accessed with REST API's.

Sources and further reading: graph

1. Graph database
2. Neo4j
3. FlockDB

Sources and further reading: NoSQL

1. Explanation of base terminology
2. NoSQL databases a survey and decision guidance
3. Scalability
4. Introduction to NoSQL
5. NoSQL patterns

SQL or NoSQL

Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.

Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

Reasons for **NoSQL**:

1. Semi-structured data
2. Dynamic or flexible schema
3. Non-relational data
4. No need for complex joins
5. Store many TB (or PB) of data

6. Very data intensive workload
7. Very high throughput for IOPS

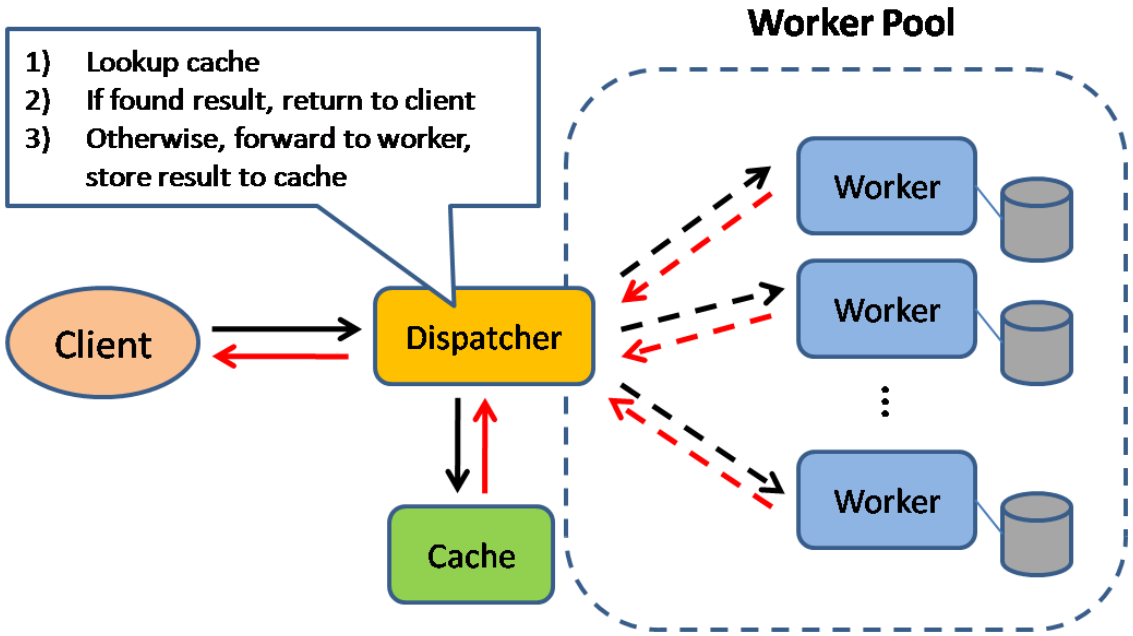
Sample data well-suited for NoSQL:

1. Rapid ingest of clickstream and log data.
2. Leader-board or scoring data
3. Temporary data, such as a shopping cart
4. Frequently accessed ('hot') tables
5. Metadata/lookup tables

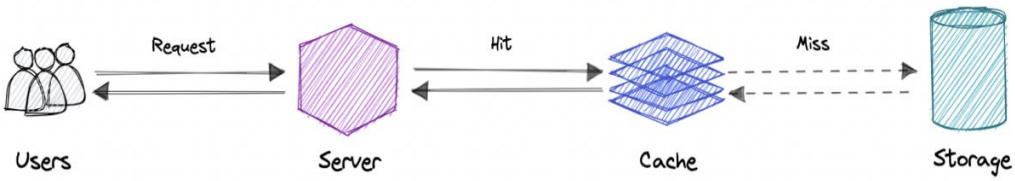
Sources and further reading: SQL or NoSQL

- Scaling up to your first 10 million users
- SQL vs NoSQL differences

Caching



"There are only two hard things in Computer Science: cache invalidation and naming things." - Phil Karlton



A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer. Trading off capacity for speed, a cache typically stores a subset of data transiently, in contrast to databases whose data is usually complete and durable.

Caches take advantage of the locality of reference principle "recently requested data is likely to be requested again".

Caching and Memory

Like a computer's memory, a cache is a compact, fast-performing memory that stores data in a hierarchy of levels, starting at level one, and progressing from there sequentially. They are labeled as L1, L2, L3, and so on. A cache also gets written if requested, such as when there has been an update and new content needs to be saved to the cache, replacing the older content that was saved.

No matter whether the cache is read or written, it's done one block at a time. Each block also has a tag that includes the location where the data was stored in the cache. When data is requested from the cache, a search occurs through the tags to find the specific content that's needed in level one (L1) of the memory. If the correct data isn't found, more searches are conducted in L2.

If the data isn't found there, searches are continued in L3, then L4, and so on until it has been found, then, it's read and loaded. If the data isn't found in the cache at all, then it's written into it for quick retrieval the next time.

Cache hit and Cache miss

Cache hit

A cache hit describes the situation where content is successfully served from the cache. The tags are searched in the memory rapidly, and when the data is found and read, it's considered a cache hit.

Cold, Warm, and Hot Caches

A cache hit can also be described as cold, warm, or hot. In each of these, the speed at which the data is read is described.

A hot cache is an instance where data was read from the memory at the fastest possible rate. This happens when the data is retrieved from L1.

A cold cache is the slowest possible rate for data to be read, though, it's still successful so it's still considered a cache hit. The data is just found lower in the memory hierarchy such as in L3, or lower.

A warm cache is used to describe data that's found in L2 or L3. It's not as fast as a hot cache, but it's still faster than a cold cache. Generally, calling a cache warm is used to express that it's slower and closer to a cold cache than a hot one.

Cache miss

A cache miss refers to the instance when the memory is searched, and the data isn't found. When this happens, the content is transferred and written into the cache.

Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.

Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

Client caching

Caches can be located on the client side (OS or browser), server-side, or in a distinct cache layer.

CDN caching

CDN’s are considered a type of cache.

Web server caching

Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.

Database caching

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

Application caching

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM.

Redis has the following additional features:

- Persistence option
- Built-in data structures such as sorted sets and lists

There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:

- Row level
- Query-level
- Fully-formed serializable objects
- Fully-rendered HTML

Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.

Caching at the database query level

Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

- Hard to delete a cached result with complex queries
- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

Caching at the object level

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structures:

- Remove the object from cache if its underlying data has changed
- Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

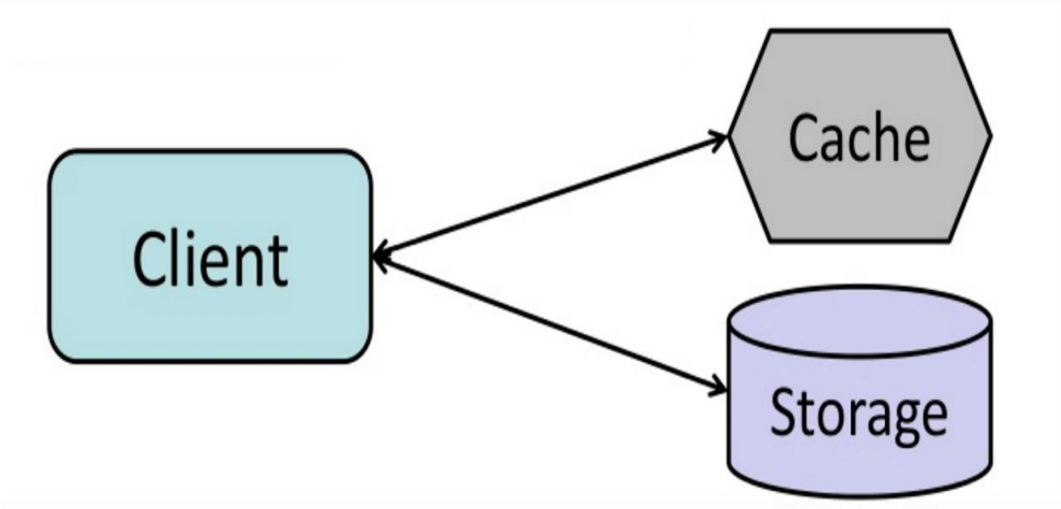
Suggestions of what to cache:

- User sessions
- Fully rendered web pages
- Activity streams
- User graph data

When to update the cache

Since you can only store a limited amount of data in cache, you'll need to determine which cache update strategy works best for your use case.

Cache-aside



The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

```
def get_user(self, user_id):
user = cache.get("user.{0}", user_id)
if user is None:
user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
if user is not None:
key = "user.{0}".format(user_id)
cache.set(key, json.dumps(user))
return user
```

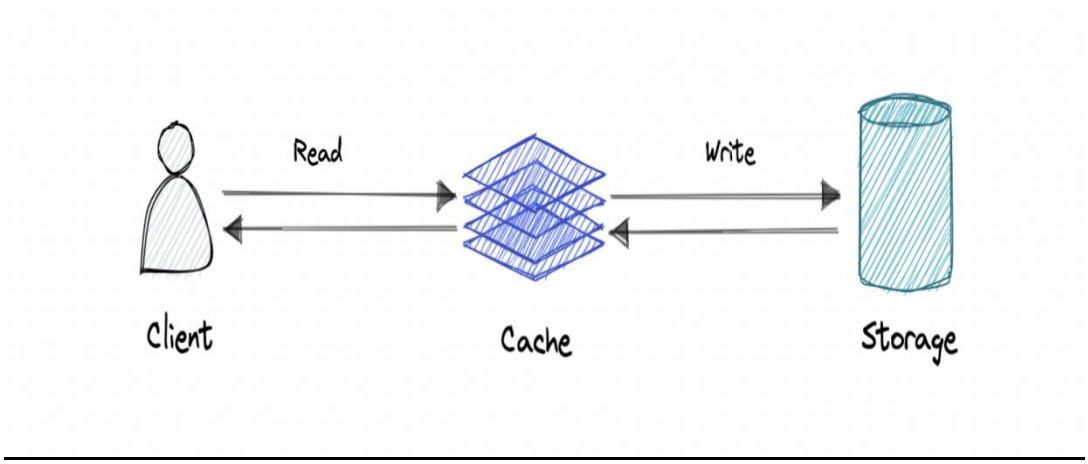
Memcached is generally used in this manner.

Subsequent reads of data added to cache are fast. Cache-aside is also referred to as lazy loading. Only requested data is cached, which avoids filling up the cache with data that isn't requested.

Disadvantages cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

Write-through cache



The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

- Application adds/updates entry in cache
- Cache synchronously writes entry to data store
- Return

Application code:

```
set_user(12345, {"foo":"bar"})
```

Cache code:

```
def set_user(user_id, values):
  user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
cache.set(user_id, user)
```

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.

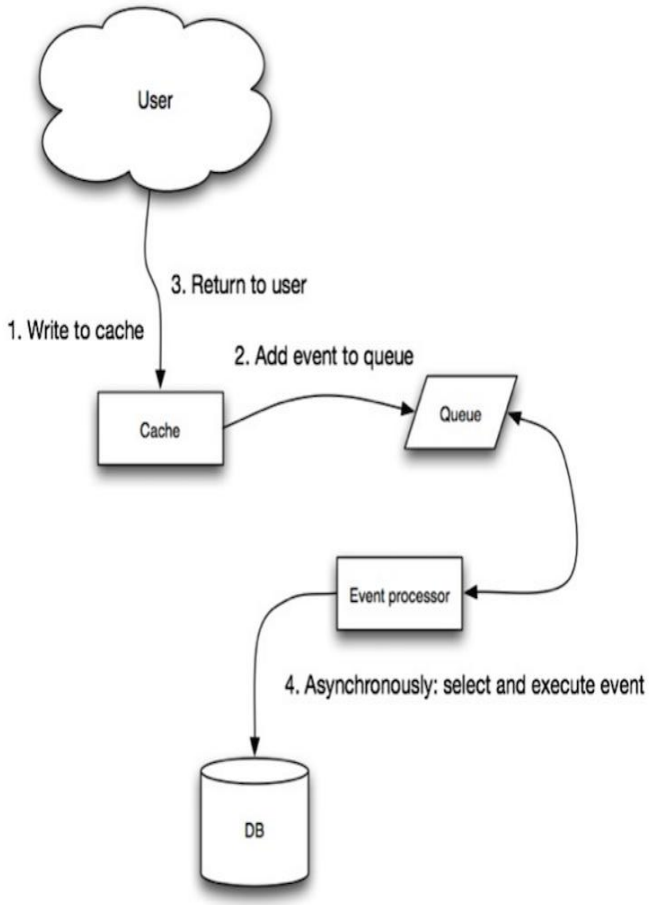
Data is written into the cache and the corresponding database simultaneously.

Pro: Fast retrieval, complete data consistency between cache and storage.

Disadvantages: write through

- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
- Most data written might never be read, which can be minimized with a TTL.
- Higher latency for write operations.

Write-behind (write-back)



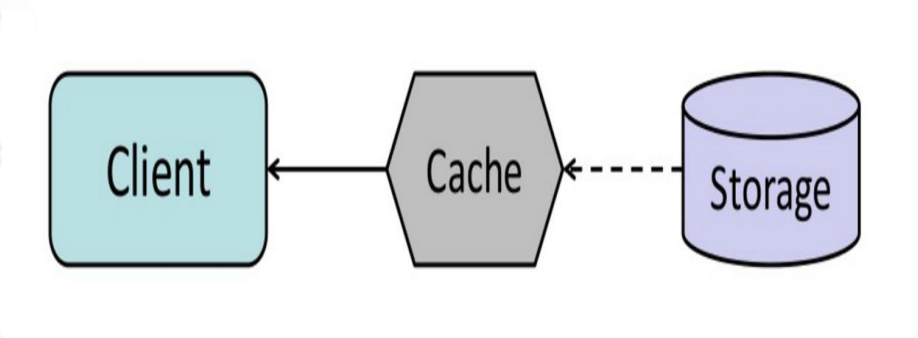
In write-behind, the application does the following:

- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance

Disadvantages: write-behind

- There could be data loss if the cache goes down prior to its contents hitting the data store.
- It is more complex to implement write-behind than it is to implement cache-aside or write-through.

Refresh-ahead



You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

Disadvantages: refresh-ahead

- Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

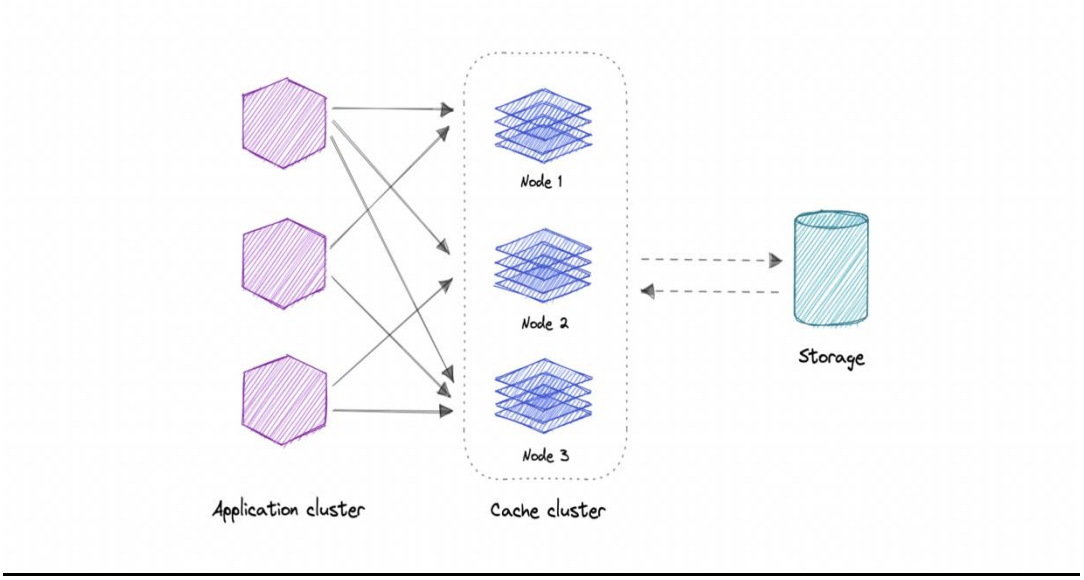
Eviction policies

Following are some of the most common cache eviction policies:

- **First In First Out (FIFO)**: The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
- **Last In First Out (LIFO)**: The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
- **Least Recently Used (LRU)**: Discards the least recently used items first.
- **Most Recently Used (MRU)**: Discards, in contrast to LRU, the most recently used items first.
- **Least Frequently Used (LFU)**: Counts how often an item is needed. Those that are used least often are discarded first.

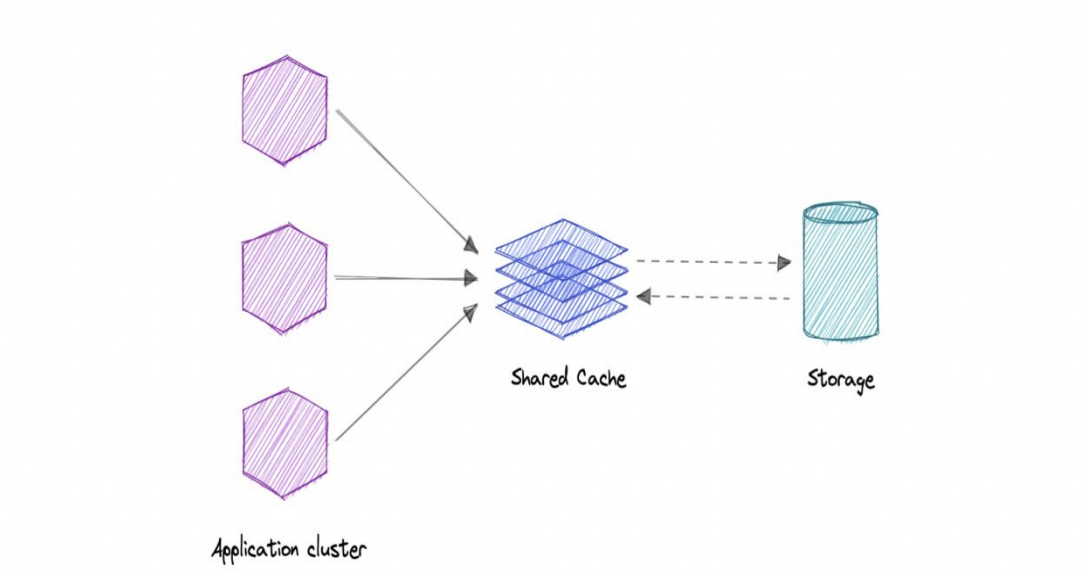
- **Random Replacement (RR)**: Randomly selects a candidate item and discards it to make space when necessary.

Distributed Cache



A distributed cache is a system that pools together the random-access memory (RAM) of multiple networked computers into a single in-memory data store used as a data cache to provide fast access to data. While most caches are traditionally in one physical server or hardware component, a distributed cache can grow beyond the memory limits of a single computer by linking together multiple computers.

Global Cache



As the name suggests, we will have a single shared cache that all the application nodes will use. When the requested data is not found in the global cache, it's the responsibility of the cache to find out the missing piece of data from the underlying data store.

Use cases

Caching can have many real-world use cases such as:

- Database Caching
- Content Delivery Network (CDN)
- Domain Name System (DNS) Caching
- API Caching

When not to use caching?

Let's also look at some scenarios where we should not use cache:

- Caching isn't helpful when it takes just as long to access the cache as it does to access the primary data store.
- Caching doesn't work as well when requests have low repetition (higher randomness), because caching performance comes from repeated memory access patterns.
- Caching isn't helpful when the data changes frequently, as the cached version gets out of sync, and the primary data store must be accessed every time.

It's important to note that a cache should not be used as permanent data storage. They are almost always implemented in volatile memory because it is faster, and thus should be considered transient.

Advantages

Below are some advantages of caching:

- Improves performance
- Reduce latency
- Reduce load on the database
- Reduce network cost
- Increase Read Throughput
- Need to maintain consistency between caches and the source of truth such as the database through cache invalidation.
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Need to make application changes such as adding Redis or memcached.

Examples

Here are some commonly used technologies for caching:

- Redis
- Memcached
- Amazon Elasticache
- Aerospike

Sources and further reading

- From cache to in-memory data grid
- Scalable system design patterns
- Introduction to architecting systems for scale
- Scalability, availability, stability, patterns
- Scalability

- AWS ElastiCache strategies
- Wikipedia

STORAGE TYPE

Storage is a mechanism that enables a system to retain data, either temporarily or permanently. This topic is mostly skipped over in the context of system design, however, it is important to have a basic understanding of some common types of storage techniques that can help us fine-tune our storage components. Let's discuss some important storage concepts:

RAID

RAID (Redundant Array of Independent Disks) is a way of storing the same data on multiple hard disks or solid-state drives (SSDs) to protect data in the case of a drive failure.

There are different RAID levels, however, and not all have the goal of providing redundancy. Let's discuss some commonly used RAID levels:

- **RAID 0**: Also known as striping, data is split evenly across all the drives in the array.
- **RAID 1**: Also known as mirroring, at least two drives contains the exact copy of a set of data. If a drive fails, others will still work.
- **RAID 5**: Striping with parity. Requires the use of at least 3 drives, striping the data across multiple drives like RAID 0, but also has a parity distributed across the drives.
- **RAID 6**: Striping with double parity. RAID 6 is like RAID 5, but the parity data are written to two drives.
- **RAID 10**: Combines striping plus mirroring from RAID 0 and RAID 1. It provides security by mirroring all data on secondary drives while using striping across each set of drives to speed up data transfers.

Comparison

Let's compare all the features of different RAID levels:

Features	RAID 0	RAID 1	RAID 5	RAID 6	RAID 10
Description	Striping	Mirroring	Striping with Parity	Striping with double parity	Striping and Mirroring
Minimum Disks	2	2	3	4	4
Read Performance	High	High	High	High	High
Write Performance	High	Medium	High	High	Medium
Cost	Low	High	Low	Low	High
Fault Tolerance	None	Single-drive failure	Single-drive failure	Two-drive failure	Up to one disk failure in each sub-array
Capacity Utilization	100%	50%	67%-94%	50%-80%	50%

Volumes

Volume is a fixed amount of storage on a disk or tape. The term volume is often used as a synonym for the storage itself, but it is possible for a single disk to contain more than one volume or a volume to span more than one disk.

File storage

File storage is a solution to store data as files and present it to its final users as a hierarchical directories structure. The main advantage is to provide a user-friendly solution to store and retrieve files. To locate a file in file storage, the complete path of the file is required. It is economical and easily structured and is usually found on hard drives, which means that they appear exactly the same for the user and on the hard drive.

Example: Amazon EFS, Azure files, Google Cloud Filestore, etc.

Block storage

Block storage divides data into blocks (chunks) and stores them as separate pieces. Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever it is most convenient.

Block storage also decouples data from user environments, allowing that data to be spread across multiple environments. This creates multiple paths to the data and allows the user to retrieve it quickly. When a user or application requests data from a block storage system, the underlying storage system reassembles the data blocks and presents the data to the user or application

Example: Amazon EBS.

Object Storage

Object storage, which is also known as object-based storage, breaks data files up into pieces called objects. It then stores those objects in a single repository, which can be spread out across multiple networked systems.

Example: Amazon S3, Azure Blob Storage, Google Cloud Storage, etc.

NAS

A NAS (Network Attached Storage) is a storage device connected to a network that allows storage and retrieval of data from a central location for authorized network users. NAS devices are flexible, meaning that as we need additional storage, we can add to what we have. It's faster, less expensive, and provides all the benefits of a public cloud on-site, giving us complete control.

HDFS

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput

access to application data and is suitable for applications that have large data sets. It has many similarities with existing distributed file systems.

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks, all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance.