**SQL and NoSQL databases**

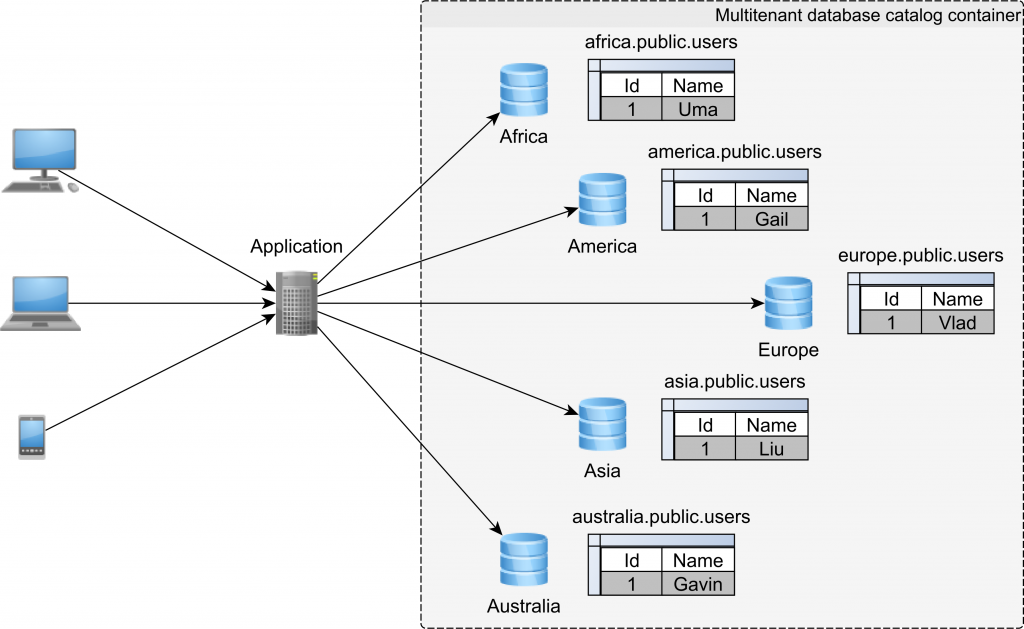
* Database evolution is moving from the line of ACID compliant transactional model to more flexible, more scalable (via distributed deployment) and more operational speed
* RDBMS features
  + Support for ACID transactions
  + Facilitation of Referential integrity
  + Fault-tolerance and accuracy
  + Structured data storage
* NoSQL is an umbrella term for any database that stores data in a way *other than* RDBMS's rigidly-typed, schema-immutable, relational tables. These can be built around
  + Key-value stores (e.g Redis)
  + Document oriented database (store their data in JSON/XML format) (e.g. Mongo)
  + Distributed databases (several trade-offs needed to be considered) (e.g. Cassandra)
  + Wide column stores
  + Time-series databases (e.g. Influx)
  + Text search (e.g. ElasticSearch)
* Key differences between SQL and NoSQL databases are:

|  |  |
| --- | --- |
| **NoSQL** | **SQL/RDBMS** |
| Non-relational | Relational |
| Many types are not table-based, if they are, schemas are not usually fixed | Fixed schema, table-based |
| Horizontally scalable | Vertically scalable |
| If distributed, follows CAP theorem (user can adjust the tradeoff between consistency, availability, and partition-tolerance) | Generally not distributed. Transactions adhere strictly to ACID (Atomic, Consistent, Isolated, Durable) properties |

* NoSQL is preferred for the following:
  + scale (including flexibility of scale)
  + flexibility (including of schema and type constraints)
  + latencyandperformance (including throughput)
  + Secondary considerations may include the size of the community (and the support and documentation you're likely to find) or the level of industry adoption (as a measure of popularity, and thus reliability and fitness for purpose) of a given solution
* RDBMS databases remain the preferred choice in industries and applications involving transactional data, where ACID-based durability is essential: user authentication and access-rights management, healthcare, banking, resource allocation and inventory management, and shopping carts, to name but a few
* RDBMS databases come with the long-established promises of isolation, security, and referential integrity, and there's no seeing those requirements go away any time soon.
* Applications involving high-volume, fast-moving *analytics* data from sensors, applications, or complementary systems, a NoSQL solution with different features or attributes might be a better fit
* NoSQL is an evolution from RDBMS databases to better serve specific needs. The "best" in any case is what suits *your* needs; remember to assess your existing infrastructure and team to find areas where gaps can become strengths

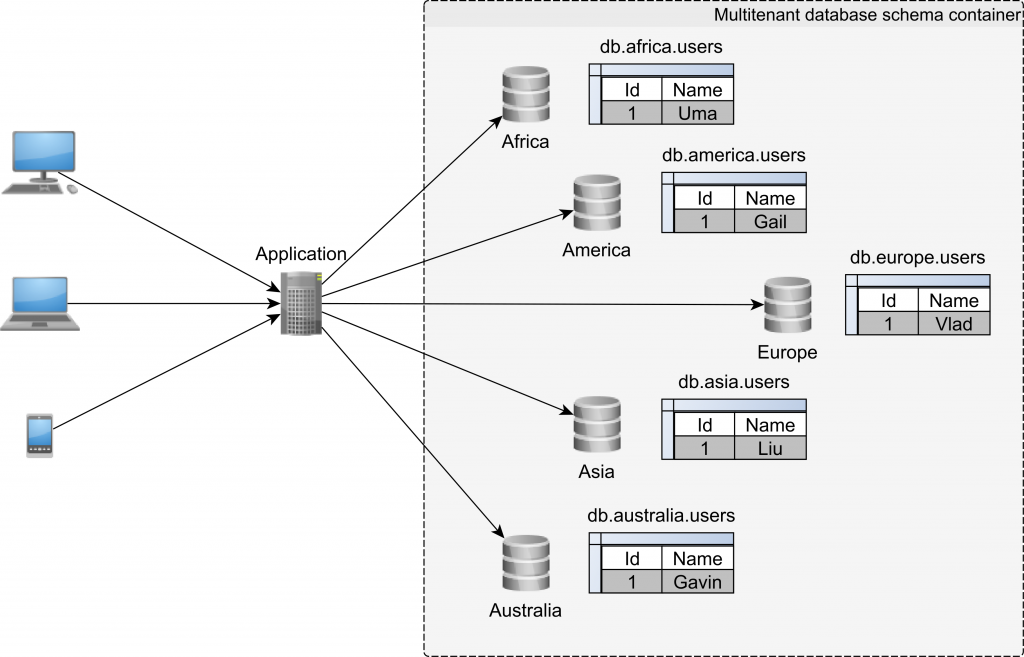
**Multi-tenancy:**

* Multi-tenancy is an architectural pattern that allows you to isolate customers even if they are using the same hardware or software components. It has become more attractive after the universal adaption of cloud computing
* Different levels of multi-tenancy in Relational DB:
  + Catalog-based: Each customer is granted access to his own catalog



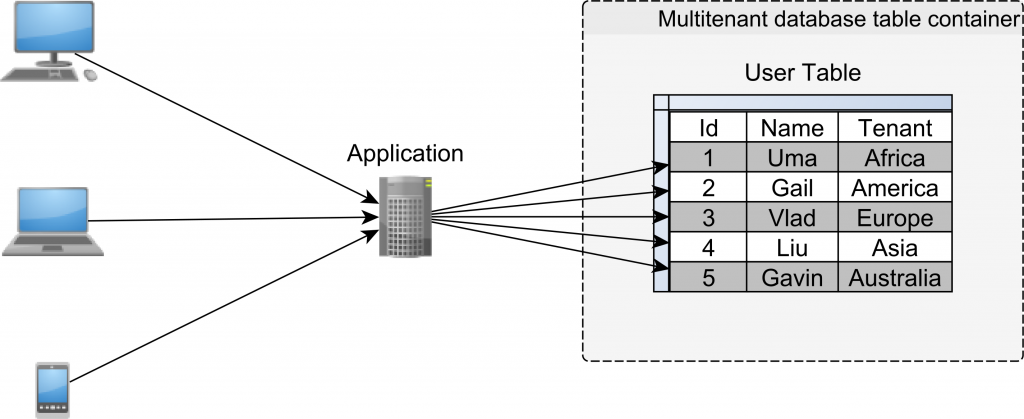
User isolation is easy to achieve. But requires more work at the ops side: monitoring, replication, backups etc.

* + Schema-based: Each customer uses his own database schema



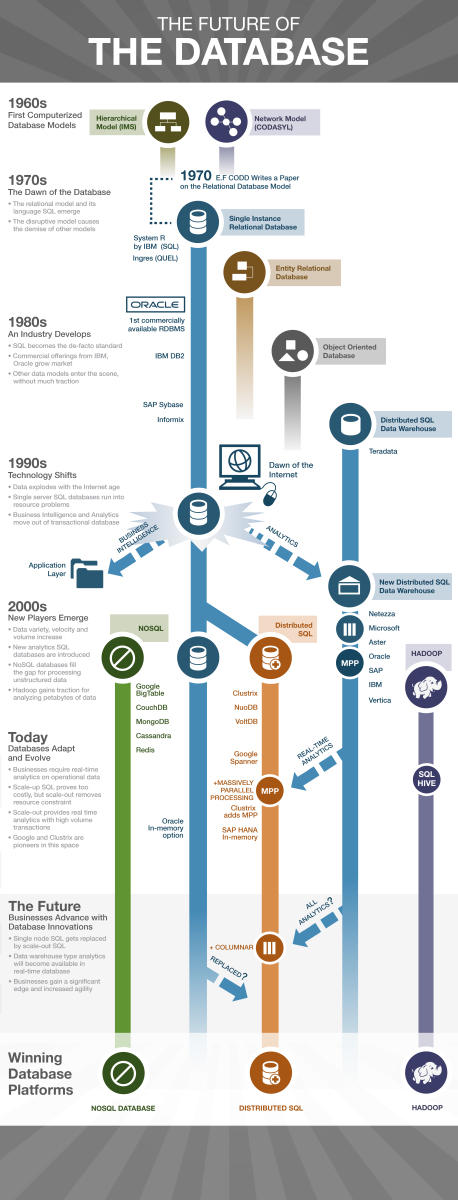
However, if schemas are co-located on the same hardware, one tenant which runs a resource-intensive job might incur latency spikes in other tenants. Therefore, although data is isolated, sharing resources might make it difficult to honor the [Service-Level Agreement](https://en.wikipedia.org/wiki/Service-level_agreement).

* + Table-based: Tenant identifier column is added to all tables



While on the Ops side, this strategy requires no additional work, the data access layer needs extra logic to make sure that each customer is allowed to see only its data and to prevent data leaking from one tenant to the other. Also, since multiple customers are stored together, tables and indexes might grow larger, putting pressure on SQL statement performance

**DB evolution trends**



**Mongo DB**

* Major features are:
  + **Rich Object model**

Supports rich and expressive object model. Objects can have properties and can be nested in one another (for multiple levels). Can easily represent any object structure in your domain. You can also index the property of any object at any level of the hierarchy

* + **Secondary Indices**

Indices speed up queries significantly but also slow down writes. Secondary indices make it easy to index any property of an object stored in Mongo DB even it is nested.

* + **Replication and high availability**

Supports a "single master" model. This means you have a master node and a number of slave nodes. In case the master goes down, one of the slaves is elected as master. This process happens automatically but it usually takes time, before the 3.2 release, 10-40 seconds were taken but after the release of Mongo DB 3.2 and later, failures are detected faster and a new leader elected in under 2-10 seconds.

The trade-off for multi-master is that the reads are slower and scale less effectively because the client must read from multiple nodes to ensure consistency. During the time of new leader election, your replica set is down and cannot take writes

* + **Native aggregation**

Mongo DB has a built-in Aggregation framework to run an ETL (**Extract, Transform** and **Load**) pipeline to transform the data stored in the database. This is great for small to medium jobs but as your data processing needs become more complicated the aggregation framework becomes difficult to debug

* + **Schema-less models**

Mongo DB, allows you to not enforce any schema on your documents. While this was the default in prior versions, in the newer version you have the option to enforce a schema for your documents. Each document in Mongo DB can have a different structure and it is up to your application to interpret the data.

While this is not relevant to most applications, in some cases the extra flexibility is important. Schema-less models mean that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data

* Advantages of Mongo DB over RDBMS:
  + Mongo DB is schema-less. It is a document database in which one collection holds different documents. But in RDBMS, you need to first design your tables, data structure, relations, and only then can you start coding.
  + Mongo DB is horizontally scalable i.e. we can add more servers (sharding) but RDBMS is only vertically scalable i.e. increasing RAM
  + Mongo DB emphasizes on the CAP theorem (Consistency, Availability, and Partition tolerance) but RDBMS emphasizes ACID properties (Atomicity, Consistency, Isolation, and Durability)
  + Mongo DB is best suitable for hierarchical data storage, but RDBMS is not
  + Mongo DB supports JSON query language along with SQL but RDBMS supports SQL query language only
  + Mongo DB is easy to set up, configure, and run in comparison to the RDBMS. It's Java client is also very easy to use
  + Mongo DB is almost 100 times faster than traditional database system like RDBMS, which is slower in comparison with the NoSQL databases
  + There is no support for complex joins in Mongo DB, but RDBMS supports complex joins, which can be difficult to understand and take too much time to execute
  + Mongo DB uses internal memory for storing working sets resulting in faster access time
  + Mongo DB supports deep query-ability i.e. we can perform dynamic queries on documents using the document-based query language that's nearly as powerful as SQL
  + In Mongo DB, Conversion/mapping of application objects to database objects is not needed