PERCONA

EXPERT ANALYSIS

BY MARCOS ALBE, SUPPORT ENGINEER, PERCONA

Beyond Relational Databases:

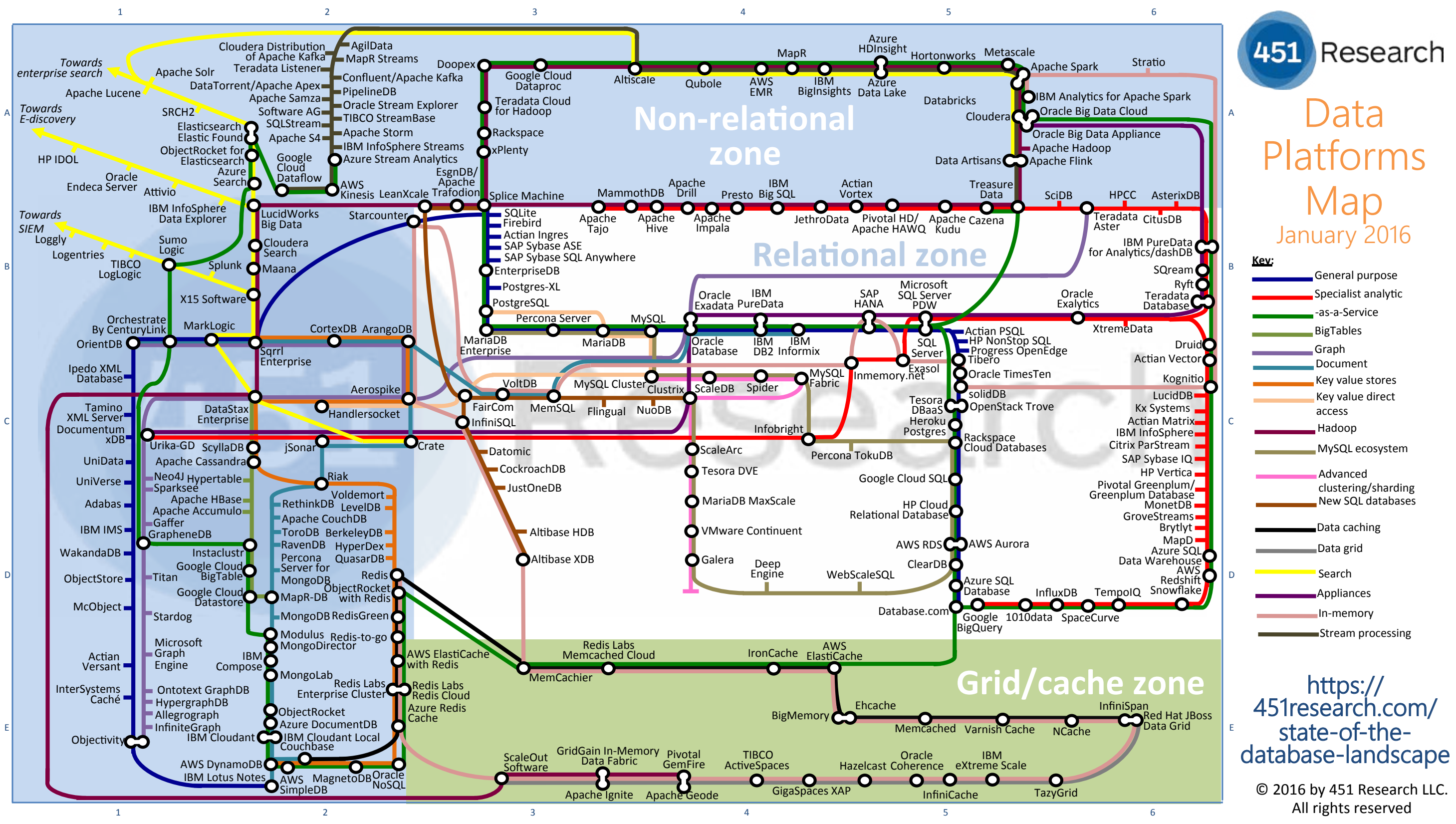
A Focus on Redis, MongoDB, and ClickHouse

Many of us use and love relational databases... until we try and use them for purposes which aren't their strong point. Queues, caches, catalogs, unstructured data, counters, and many other use cases, can be solved with relational databases, but are better served by alternative options.

In this expert analysis, we examine the goals, pros and cons, and the good and bad use cases of the most popular alternatives on the market, and look into some modern open source implementations.

Beyond Relational Databases

Developers frequently choose the backend store for the applications they produce. Amidst dozens of options, buzzwords, industry preferences, and vendor offers, it's not always easy to make the right choice... Even with a map!



Relational databases are generally the go-to option as they are flexible and well-known. But, when used for the wrong purpose, they can end up being a bottleneck down the road.

To help you make the right choice for your business we have focused on the three most popular database alternatives: Key-value, Document, and Columnar, and have given an overview of situations when relational databases might be a poor choice. In this paper, we will consider alternatives to relational databases, and examine Key-Value databases in more details.

Why Not Relational Databases?

MySQL, PostgreSQL, and all other relational databases work well for general purposes. They can represent any data model and this flexibility is one of the reasons that they are very popular. You can represent the necessary data structures required for a social graph, for an OLAP cube, for a work queue, for an Entity-Key-Value model, and many more. And these will work... but the relational model is not always the best fit for structures such as these.

Relational databases don't shine here due to the way the data is represented, because of performance overhead, or for other reasons that are inherent to the relational paradigm. The bottomline is that they will fail to scale for many workloads that are commonplace in the industry.

Another reason for relational databases popularity is SQL. This follows a well-defined standard, so a single expert can apply the same knowledge through many systems. But, much as I love SQL, I recognize that it is sometimes hard to grasp for developers. It is not hard to end up with JOIN bombs, and the rigidity of the schema is a pain-point most of us are likely to have suffered during early development stages. In some environments, employing a query language more naturally integrated with the programming language, and with a flexible schema, would be highly valued. These are two key features of **Document stores**.

Another feature that helped popularize MySQL is that it provides ACID properties and referential integrity, which guarantees strong consistency for our data. But, ACID and referential integrity are concepts that imply paying a performance price for. This is because all the locking mechanisms, and the intrinsic waits for IO that are needed to provide such guarantees, can become extremely costly. This is one of the main reasons that relational databases fail to scale for some workloads. So, in cases where data can be rebuilt, or is known to have few relationships, an in-memory **Key-Value database** might be a better fit.

Finally, we know for a fact that MySQL can store 70TB of data, or more in a single instance. We even have a customer with 34TB in a single table! So, yes: it can handle these volumes of data... as long as you don't really need to read all of it! In which case, I question why you stored all that data in the first place? The problem with relational databases is the way that they store data on disk, because to read a single column you must either index it (paying the associated maintenance costs), or read all of the columns. To effectively process these volumes of data we must store it with a physical format that allows us to be more selective of what we read from the disk. This is the main idea behind **Columnar databases**.

So, the same reasons that make relational databases great for a wide variety of cases, are the reasons that make them a very bad fit for certain use cases. It's not a matter of going "all relational", or "all NoSQL"! Instead, the idea is to embrace the concept of 'Polyglot Persistence' and enjoy the benefits that each database has to offer.

So, let's dive into the first possible alternative to relational databases.

Key-Value Databases

Key-value stores came onto the scene in a big way in 2003. Their main purpose at the time was to provide a stateless object cache in front of bigger slower relational databases.

Most key-value stores are basically a persistent (and glorified) hash table, which makes their access times incredibly and consistently fast. While this type of storage is not usually adequate for applications of high complexity, it is exactly that simplicity that makes such systems attractive in many other circumstances. Personally I believe that they are not a replacement for relational databases, but rather a complementary technology.

Key-value is a fundamental data representation, or data structure. It is very simple, consisting of (key-> value) pairs. So, if you want to retrieve or modify the value associated with a key, all you need is the key itself. This also implies that if you don't have the key, you can't really find the values.

You need to think of the keys here like PRIMARY keys, in that they must be unique and cannot be null. As a general rule there are no limitations on contents or key length. But, with larger keys we can expect a degradation in performance.

Key-value stores allow storage of arbitrary data, and they have no inherent knowledge of what format the data is stored in. To the key-value store, the data is a grey box, just like any old BLOB in a relational database. Because no metadata is stored, and because optional values are not represented by placeholders as in most RDBMs, key-value stores often use far less memory to handle the same data. This can lead to large performance gains in certain workloads.

Key-value databases carry the NoSQL pennant and each of them use a custom set of commands and simple protocols to communicate with clients. The number of operations used with key-value pairs is small, there are only four of them: Create, Read, Update and Delete, all done via key access. You can see how this matches up very nicely with HTTP Rest APIs, and also with the operations that you need to build a good caching layer.

Another really attractive thing about key-value stores is the relative simplicity to horizontally shard them. This is partly because you do not have to think about possible joins or ranges spanning multiple shards, as the main use case is single-key store and retrieval. This sharding is most commonly done through constant hashing algorithms, so when nodes are added or removed, the number of elements that must be relocated to/from each existing node won't cause a massive operation that will stall the servers.

Redis

A popular example of key-value is Redis. As you would expect from a key-value store: it is fast, VERY fast, and it can hold hundreds of millions of keys in memory, and handle tens of thousands of requests per second without much stress.

Redis has a lot of advantages:

- It is mature
- It has an active community and large user base
- It is actively developed and maintained
- It has a friendly BSD license
- There are Redis connectors available for virtually any programming language

What Makes Redis special?

The folks at Redis.io define their product as a data-structure server, as it supports many different data structures including strings, hashes, geohashes, lists, sets, etc. For example, if your application requires a list that should be shared among all application servers, Redis not only gives you where to store the list, but also abstracts the operations you can perform on the list by providing commands such as:

- LPUSH (prepend value)
- LPOP (get and remove value)
- LREM (remove value)
- LINDEX (get element by its index)

This makes your code cleaner and guarantees atomicity of these operations in a distributed fashion.

Also, compared to other key-value stores Redis has many advanced features. These include: built-in replication, Lua scripting, LRU eviction, multi key transactions, blocking queue commands, atomic operations on the data structures, different levels of on-disk persistence and Publisher/Subscriber facilities. All of these should make programmers' lives easier.

Another huge advantage for key-value stores is that there are basically only two kinds of locks: shared and exclusive. These only operate at the key level which allows for massive concurrency (unless you have extremely hot keys).

What is it Redis Good For?

Redis can do a great job as an LRU or TTL-based full-object cache in front of MySQL. This was, and still is, an amazing use case for in-memory key value stores. Thanks to its advanced feature set (and the fact that these operations are atomic), it is very useful for things like maintaining sessions, keeping counters, work queues, creating publish/subscribe real time notifications, and so on.

Because of their lack of complexity in data handling (i.e. they don't do any data validation on the way in), writing many small pieces of data is very quick. If you only need CRUD operations on primary keys, you should seriously consider using a horizontally scalable key-value database like Redis.

Redis is good for handling large quantities of data. Petabyte? No problem. Because you can just horizontally scale and distribute the data over lots and lots of nodes (recommended up to ~1000 nodes). Of course, this requires you to have petabytes of memory. For example, to hold 1PB of data you would (roughly) require 1024 nodes, with 1TB of RAM each. This wouldn't be cheap!

When to Avoid Redis?

Redis' focus is not on long-term data persistence. You can make it very durable, but then it would lose the speed that makes it so desirable. While Redis has got advanced features, but, if you need to scan, filter, and join keys, then the job is likely done better by a relational database. And, if you need to shard and/or want to fully utilize CPU power, you need to be ready to take on some operational complexities. If you don't have a dedicated person/team for this it might become a pain point.

Also, Redis has very little security built-in. It has only a thin authentication layer (also tied to the concept of keeping things simple and lightweight), and no concept of ACLs of any kind. So untrusted access must always be done through your own custom API, that will implement whatever ACL you consider necessary.

Redis in a Nutshell

- Fast... very fast!
- Mature, large community
- Data structure server
- Advanced features
- Horizontally scalable/built-in HA (Sentinel/Cluster)
- BSD license/commercial licenses available
- Client libraries for almost every programming language

Redis Drawbacks

- Lower durability
- Limited access patterns
- Lack of security
- No secondary indexes

Redis Good Use Cases

- Lots of data and simple data access patterns
- High concurrency
- Massive small-data intake
- Session cache/full page cache
- Counters/leaderboards/single-row hotspots
- Queues
- Distributed publish/subscribe notifications

Document Databases

Inspired by the late 1990s Lotus Notes, these databases are (as their name implies) designed to manage and store so-called "documents". The main characteristics of these documents are their schema-less organization of the data. Records are not required to have a uniform structure, meaning each record in the same table can have a totally different set of attributes. To allow this flexibility in schema design, data input and output is done using a standard exchange format such as XML, JSON or BSON (Binary JSON). This allows you to richly describe documents of any type.

Document stores are a subclass of key-value (KV) stores, as at their heart they basically have pairs of keys and values, and these values "can store anything". What differentiates document databases from KV stores is that the document paradigm does impose a minimum structure on the value. For example, you can store any string you want, but it must be wrapped in a JSON object or an XML document. So the values in document databases actually contain semi-structured data. Based on this (loose) structure, these databases can build secondary indexes and unique indexes. This is really the defining feature that made them very popular: they combine the performance, flexibility and simplicity of a KV store with the ability to add secondary indexes on any field.

This flexible schema arrangement allows for quick prototyping, as developers no longer need to (immediately) worry about the underlying database structure. Production deployments of new features that require schema modifications are easier to carry on, as both forms of the schema can co-exist within a single on-line table. They can also save lots of disk and memory space, as nothing is stored for attributes that would have a NULL value in a relational database. So, data collections with lots of attributes, where only a few of them will take actual values, would have a much smaller footprint.

Probably one of the most desirable properties of a flexible schema is polymorphism. This is the capacity for the records to hold shared and non-shared attributes. For example in a catalog of sport items every record will have a sku, a price, a sport category, and each item has unique characteristics, such as size, diameter, color, material, etc. This goes very well with class inheritance polymorphism, and Object-Oriented Programming ideas.

Another benefit of the schema-less paradigm is that one can nest documents, forming complex objects that can be stored as a single entity inside a collection. As an example, here is a document describing a blogpost:

```
blogpost = {
        title: “Basis data procuratio ratio mirabilis est!”,
        date: “2020-04-20”,
        body: “Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor inci-
didunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in volupta-
te velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proi-
dent, sunt in culpa qui officia deserunt mollit anim id est laborum.”,
        author: {
                name: “Marcos Albe”,
                company: “Percona”,
                email: “marcos@example.com”,
                bio: “Generic carbonoid unit”
                country: “UY”
},
comments: [
{
        datetime: “2020-04-19 10:02:11”,
        author: {
                name: “Joe Doe”,
                email: “jd@example.com”,
                country: “US”
        },
        comment: “Hi, nice blogpost. Will you cover graph databases someday?”
},
{
        datetime: “2020-04-20 14:33:56”,
        author: {
                name: “Marcos Albe”,
                company: “Percona”,
                email: “marcos@example.com”,
                bio: “Generic carbonoid unit”
                country: “UY”
},
        comment: “Hi Joe\n\nYes, I will cover Neo4J in another blogpost soon.”
        }
  ]
}
```

You can see that everything relating to the post is contained within a single document object; The author and comment (as well as comment's authors) entities that would have been in a separate table on a relational design, are now embedded within the blogpost document. This is very convenient from a programmers point of view as they no longer need to worry about joining or fetching separate pieces of data. They only need to update the object and send it back to the database, making the data handling code very fluent. Also, notice how some attributes (company, bio) are present only in one of the author entities and not in the other, this saves space and removes the need to fill in attributes with NULLs.

All that is good, but as with everything in life, there is a tradeoff: you are creating a denormalized dataset. Notice how the author entry appears twice: once in the post metadata, and once again in one of the comments metadata. This has a few drawbacks:

1) Each entry would need to be deleted or updated separately, which implies added code to do so and added operations on the database. The risk of introducing inconsistencies is high, as an update/delete can easily be forgotten.

2) Data duplication will inflate the on-disk and in-memory dataset footprint.

3) Read and Write amplification. If you have a 10MB document (because you nested another 100 things in it) and you only update one attribute, you are still writing 10MB. If you have to read a non-indexed attribute from the document, then you are reading the whole document (see [DOCS-11416]).

Some people may suggest that you move the author entities to their own collection and then reference an ID. In my humble opinion, if you start using lookup() too often in your application, you will be better served using a relational database, which really excels at that.

Also, one has to remember that "flexible schema" does not mean "no schema." In fact, you should design your schema even more carefully than a relational database, thinking ahead on all the use cases for the data, as the access patterns will dictate what schema design patterns are better for your case. So, you can prototype easily, but you need to have a solid understanding of the tradeoffs you face as a result of the decisions taken in your schema design.

You must also bear in mind that in order to enjoy the type of indexing most people would consider useful, all your documents must have the indexed attributes, otherwise you might end up missing records when querying the collection through that (sparse) index.

MongoDB

MongoDB is the most prominent document database, with its most advanced engine, WiredTiger, now well established. Its development continues to be strong, it has a large adoption rate, an increasing user base and a great community. This all indicates that it will keep improving over time. MongoDB uses an SSPL license that caused some controversy at the time. The SSPL (Server Side Public License) is basically a GPL that forbids cloud providers to keep code changes for themselves if those changes are used to provide MongoDB-as-a-service.

In this ecosystem there is also Percona Server For MongoDB, a Free/Open-Source drop-in replacement for MongoDB Community Edition, which provides alternatives to features that are otherwise only available in the paid Enterprise version of MongoDB. Percona Server For MongoDB includes LDAP authentication, database auditing, Memory engine, hot backups and log redaction, so you can get the features you need most for free, without vendor lock-in.

What Makes it Special?

Sharding, in my opinion, is the defining characteristic of MongoDB. Easy to set up, transparently-rebalanced sharding allows you to scale-out easily and with minimal pain. With a properly designed schema and properly chosen sharding key, MongoDB really allows you "throw money at the problem" and add more servers to horizontally scale without having to touch your application. Its High Availability solution (Replica Sets) are easy to set up ,and provide automatic failover and easy re-provisioning of the replicas.

Another key feature of MongoDB is its aggregation framework, which enables business intelligence processing to happen on the same database. There are two aggregation methods: The aggregation pipeline, which provides good performance and a coherent interface and Map reduce which is slow, but offers more or less complete algorithmic freedom.

A more advanced feature is the geo-spatial functions, like geoNear which, as the name suggests, will find nearby points and is shard-aware. On-disk encryption is also available if your business needs to comply with certain regulations or if you are a bit paranoid about data leaks (hint: you should be!) Capped collections and the ability to set a TTL to expire data in a collection allows you to control data growth, as well as other nice tricks such as tags, which allow queries to be routed to specific servers, clusters,or regions.

Regarding MongoDB drivers, you can find them for every major programming language, and these connectors are aware of the replication and sharding topology so they allow us to more easily scale. Two of the supported languages have alternative asynchronous-style drivers, which allows you to send queries to the database in a non-blocking fashion. For example, you could send an update for which you don't need an immediate response and the rest of your code can continue running.

Last, but not least, MongoDB, like MySQL, has storage engine options. MMAPv1 has been removed since v4.2, but for years the Percona Memory Engine has been available alongside WiredTiger.

What is it Good For?

Many of the stronger use cases of MongoDB include products or items catalogs. This is due to polymorphism, which is a natural attribute of the flexible schema concept.

Metadata storage is something MongoDB should excel at. Its flexible schema concept allows you to store any metadata that an object might need. Things like user preferences or customization fit very well here; these are a super-set of the catalog case.

Other particularly suitable cases are those where data expires. This includes sessions, shopping carts, work queues, and any time you want to remove abandoned things automatically, and where each item shares a common set of attributes, but requires different attributes on each entry, are a great use case for the TTL feature and flexible schema. It also works well when you want to limit how much data you collect. For example,event logs where you want to set a maximum disk footprint, are a good case for MongoDB's capped collections.

Another great use case is data consolidation. Imagine a corporation that sells products in multiple verticals, or has a wide array of different products, and they want customers to perceive their brand and their service as a single entity. You need to bring together the fragmented customer data from multiple disparate sources into a single place, to enable the agents of each corporate unit to have a holistic view and be able to provide a unified service. This is much easier when you can use a flexible schema.

When to Avoid it?

If you have an entity with deeply-recursive children, MongoDB documents can become very ugly. Indexing, searching, and sorting these recursive embedded documents can be very hard.

Ad-hoc querying or large amount of views required over the same data. Remember: to access those embedded documents (like the comments in the blogpost) you need an index on the attribute that you want to gather. You can end up with too many indexes if you denormalize too much, or you might end up with too many lookup()s (a.k.a. joins) if you normalize too much. So, this problem is not easy to solve, and that is precisely what Relational Databases do so elegantly.

Applications that require multi-statement transactions might not be best served by MongoDB. Multi-statement (or multi-document) transactions were simply not possible until 4.0, and it appears their performance is still not amazing; Quoting the manual page about transactions:

> *In most cases, multi-document transaction incurs a greater performance cost over single document writes, and the availability of multi-document transactions should not be a replacement for effective schema design. For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases. That is, for many scenarios, modeling your data appropriately will minimize the need for multi-document transactions.*

So, make sure to measure response times, especially if you use shards.

Pure Developer Laziness™. While we all agree there is an "impedance mismatch between OOM and Relational databases" (as Martin Fowler called it), it is unwise to allow developers to choose a storage backend that is a bad fit, or where better alternatives exist, just because the interface is comfortable. This leads to project failures down the road, as the database fails to scale and operations become a nightmare.

MongoDB in a Nutshell

- Simple sharding and replication!
- Flexible schema (a.k.a. Schema Polymorphism)
- Excellent compression (with WiredTiger)
- On-disk encryption with the MongoDB Enterprise version (or free with Percona Server For MongoDB)
- Drivers for all major programming languages (sharding and replica aware)
- Geospatial functions
- Extensive aggregation functionality

MongoDB Bad Use Cases

Recursiveness

Multiple views of the data

Multi-document transactions

Developer happiness pill

MongoDB Good Use Cases

Catalogs

Analytics/BI

Time series.

Metadata repositories

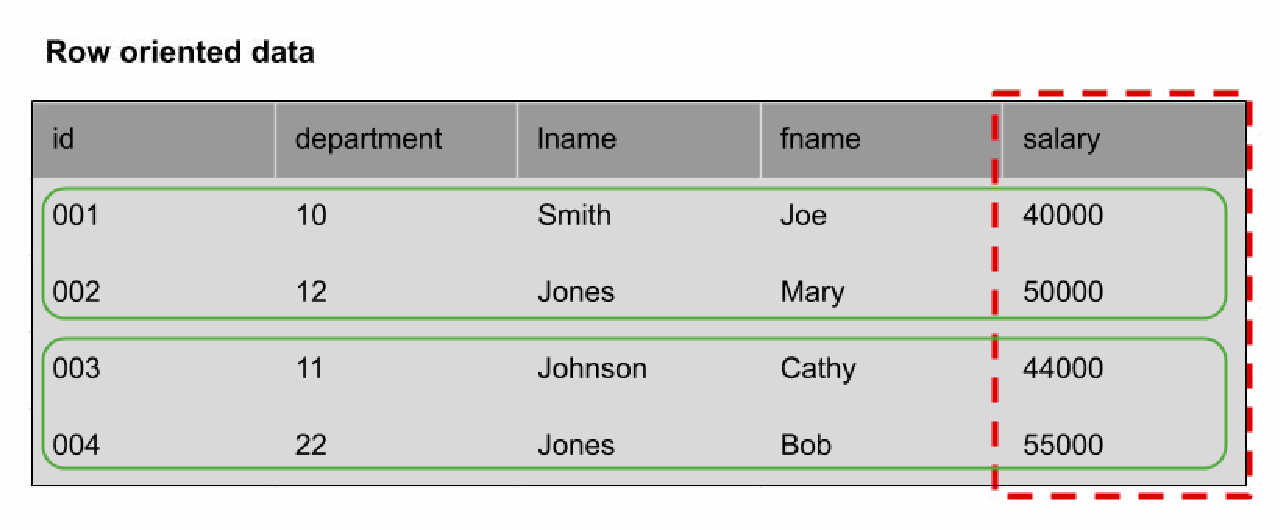
Prototype development

Data with expiration date

Columnar Databases

Columnar databases have already been around for a long time, because the features they provide have been necessary since computers started to be useful. Modern implementations came around the early 90s, with Expressway 103, which later [became Sybase IQ](#) (click the link for some nostalgia!)

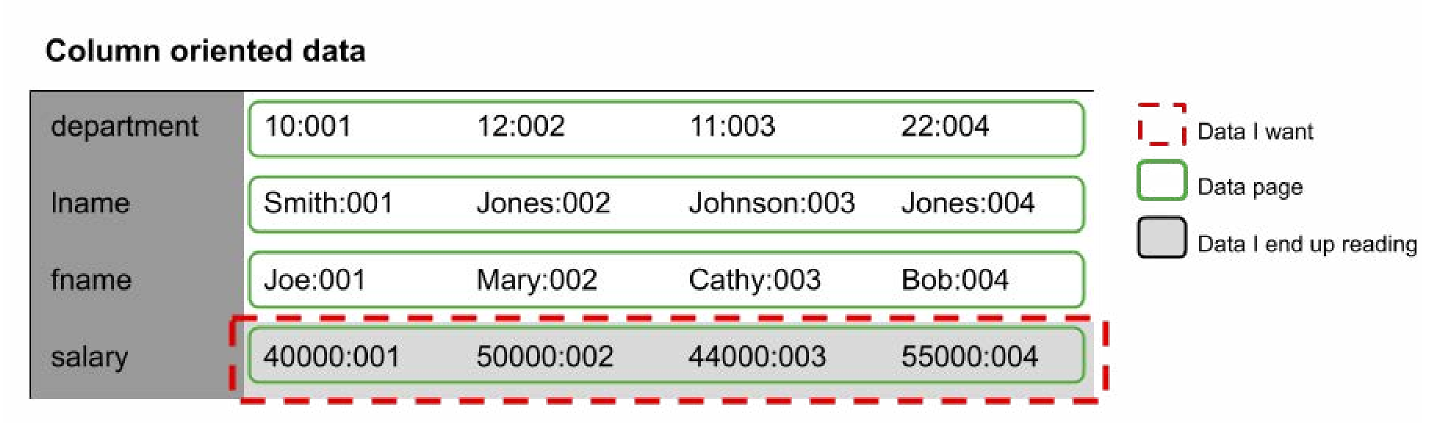
The problem that column-oriented databases try to solve is that of aggregation, transformation, and calculations on massive amounts of data. This is mainly for the purpose of data -mining, analytics and business-intelligence (commonly referred to as OLAP workloads) and more recently for AI training. The name “Columnar” comes from the on-disk data organization. A row-oriented database would usually layout data on disk as follows, with data for each row written together within data pages, with each page consisting of multiple rows. When reading a single column from a single row, we are still forced to read the full page.



Row oriented data

id	department	lname	fname	salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	55000

In this example the data we want are the salaries, but we need to read the two pages and all of the four rows to access the single column we are interested in. Column-oriented databases will group the values for each column in a data page, and keep a reference to the row that the column belongs to:



Column oriented data

department	10:001	12:002	11:003	22:004
lname	Smith:001	Jones:002	Johnson:003	Jones:004
fname	Joe:001	Mary:002	Cathy:003	Bob:004
salary	40000:001	50000:002	44000:003	55000:004

This way, if we just want to aggregate a single (or few) columns’ values, we only have to perform a fraction of the IOPS.

The attentive reader will ask; “can’t you just use a covering index?” Sure! Covering indexes (that is: an index that contains all the columns that are of interest to answer a query) can help... with a limited number of queries. For ad-hoc decision-support queries you simply cannot create all the necessary indexes. Instead, it would require you to create an index on each column, and then have a database that allows you to use multiples of these individual indexes in a single query… this is what a columnar database does in a specialized and much more efficient way!

So, let's say you have a 10G table with three columns. Column A takes 4GB, column B takes another 5GB, column C (not indexed) takes 1GB. Now, if we have to scan the values of "C" only, on a row-oriented database you must read the whole 10GB, but on a column-oriented database we can scan just that and read 1GB, with a huge 90% saving. Often more than one column is necessary to fully solve a query, but savings can still be massive, and this can give reads considerable headroom.

Of course, there is no such thing as a free lunch, and there is a downside to this storage layout. Writes become expensive. Where you used to have one single write for, say, 20 columns, now you have 20 writes, one for each column. Each write comes with an associated latency cost, and that latency multiplies itself by 20.

A possible alternative when doing this kind of OLAP work is to save intermediate summaries instead of having massive tables with the raw unaggregated data. But, this aggregation dilutes the resolution of the data and cuts short the number of possible custom reports. It is also not always possible to do effective aggregation of data. If your aggregation key has lots of unique values (i.e. its cardinality is high) then the aggregation will yield a similar number of rows to the original. This makes columnar store very desirable, as it allows you to preserve the original raw data and freely perform aggregations on it.

ClickHouse

ClickHouse was born as the data store for Yandex Metrica (the Russian equivalent of Google Analytics) and was open sourced in mid-2016 using an Apache 2 license model. Its release, along with MariaDB's ColumnStore (which was also released Q2 2016), marked the comeback of enterprise-grade columnar databases to the open source world, after InfoBright Community Edition ceased to exist.

With a strong focus on time-series and massive amounts of data (remember, it was created to handle the the stream of ad clicks from the Yandex advertising network), ClickHouse can deliver blazing performance, reaching the billion-rows-per-second realm for some specific use cases. It provides real-time access to data as it can efficiently handle batch writes, keeping data updated-to-the-minute (unlike many other data analytics solutions where data is refreshed periodically by Extract-Transform-Load processes).

What Makes it Special?

Clickhouse claims to be a "true column-oriented DBMS." This means that there isn't any "garbage" stored along the values. For instance, constant-length data types are used to avoid storing the length of the values. Although storing one or two extra bytes per row might seem trivial, when you are handling hundreds-of-billions of rows, these few bytes add up a lot of space and will influence compression performance. By storing data in the most compact way, ClickHouse can save lots of disk, memory, and CPU processing.

This is all worth noting because there are systems that can store values of individual columns separately, but can't effectively process analytical queries due to their optimization for other scenarios. Examples include; HBase, BigTable, Cassandra, and HyperTable. In these systems, you will get throughput around a hundred thousand rows per second, but not hundreds of millions of rows per second.

Another big benefit is ClickHouse's massive parallel processing capabilities; It can parallelize a query on a single node through all its CPU cores, and it can distribute the query through multiple shards (where each shard is a group of replicas that are used for fault tolerance) in a transparent way for the user. Depending on the query and data distribution, this can reach linear scalability. The linked articles show different scenarios and potential difficulties to scale.

One critical aspect of databases designed for data warehousing is their disk footprint. Columnar databases are usually tasked with handling datasets in the dozens-of-terabytes, which can make storage an operational problem and/or expensive. To help manage this growth ClickHouse provides two compression methods: LZ4 and ZSTD. LZ4 provides fast and shallow compression resulting in less CPU overhead and larger disk footprint. ZSTD will be slower and consume more CPU, but provides much better compression ratios and storage savings. Altinity has a good blog post showing compression ratios and the performance impact of compression algorithms.

ClickHouse delivers on both with two "disk-oriented' features: data compression and on-disk aggregation. Compression improves performance for purely IO bound workloads. For the type of data usually seen in OLAP and time-series databases the compression ratio can be pretty large, which can lead to big savings on storage and operational times on systems expecting terabytes of data. The on-disk GROUP BY allows you to avoid out-of-memory situations and still remain relatively efficient for extremely massive aggregations. This would be impossible, or impractical, with other columnar databases.

Aggregation is one of the most important features of a column-oriented DBMS, and so its implementation is one of the most heavily optimized parts of ClickHouse.

Another thing that makes ClickHouse special, but in an awkward way, is its SQL support; ClickHouse uses a declarative query language based on SQL that in many cases is indistinguishable from ANSI SQL. For example, JOINs are supported, sub-queries are supported in FROM, IN, JOIN clauses and in scalar subqueries. Conditions are also similar and you can use LIKE, IN, BETWEEN, etc. But, it has differences. Functions have different names, data types are different (this is the part most people find unfamiliar), it has special syntax for lookup dictionaries (more on this below), correlated subqueries are not supported, EXISTS is not supported, etc. You can check their ANSI compatibility table for more information.

Data replication and support for data integrity on replicas is built-in. ClickHouse uses asynchronous multi-master replication. After being written to any available replica, data is distributed to all the remaining replicas. The system maintains identical data on different replicas.

Indexes: MergeTree engine in ClickHouse supports primary keys, which allows extraction of data for a specific range, similar to having a partitioned table on which you can prune partitions not needed for your SELECT.

Dictionaries: These are lookup tables that allow you to avoid JOINs. Data can reside in-memory and speed up lookups.

Suitable for online queries: ClickHouse's low latency allows using the system as the back-end for a web interface, because queries can be processed without delay.

Support for approximated calculations: The system contains aggregate functions for approximated calculation of the number of various values, medians, and quantiles.

Supports running a query based on a part (sample) of data and getting an approximated result. In this case, proportionally less data is retrieved from the disk.

Supports running an aggregation for a limited number of random keys, instead of for all keys. Under certain conditions for key distribution in the data, this provides a reasonably accurate result while using fewer resources.

When to Use it?

Suitable for read-mostly or read-intensive, large data repositories. This is where Columnar storage shines with sequential reads of millions of values for a handful of columns for projects where the dataset will not fit in memory (otherwise using RDBMS is feasible for the same).

Good for full table/large range reads. If you only have to read a handful of rows and not large ranges, sticking with your RDBMS might be better. Otherwise, if you need to scan a huge volume of archived rows in your tables, consider a columnar store.

MergeTree, ClickHouse's most advanced engine, is designed for time-series...so much so that it MUST have a date column, and the whole thing was designed with this in mind, so it is certainly one of the strongest use cases. Strong enough that we have a PMM (Percona Monitoring and Management) version under development where Clickhouse will be used for Query Analytics and for long-term metrics storage).

Good for unstructured problems where "good" indexes are hard to forecast (Decision Support Systems). The self-indexing nature of columnar databases allow for ad-hoc querying without the need for indexing. This is highly desirable for analysis, where crossing of data can happen over any dimensions.

One could argue that covering indexes would do a similar job. Sure, they do and we often solve problems for customers that run OLAP reports on their OLTP servers by adding a few covering indexes. But, this has a limit, and as business demands for more reports grows, the amount of indexes would grow to the point where they would be too costly to maintain for each INSERT/DELETE/UPDATE. There would also be no ad-hoc querying when analysts come up with new ideas.

Providing ACID guarantees is not in the plans for column-oriented databases because that would add additional overheads, which we don't want in a process that is about reporting and not about transaction handling. The transaction has already happened and was "certified" and "blessed", so columnar stores do not need to check or care about data safety. Also, the lack of DELETE/UPDATE forces us to rebuild the dataset if it is not an static archive.

When to Avoid it?

Columnar databases aren't designed for "SELECT *" queries, or queries fetching most of the columns. The savings in a columnar database come from not reading data in columns that aren't necessary for calculations. That unnecessary data tends to make up most of the table in many cases, and these reads can be avoided or delayed. But, if your project requires reading most/all of the columns then staying with a row-oriented system is perhaps a better choice (and maybe you will need to live with intermediate aggregates for your data, instead of the raw data).

It is not good for transactional (OLTP) workloads. Doing INSERTs of a single row or a few rows is not really efficient, and neither would be doing UPDATE and DELETEs on a regular basis. The loading of data to ClickHouse should be done through an asynchronous process and in large batches.

Not ideal for mixed read/write. Batched INSERTS are OK, but again no more than a few writes per second for scattered records should be planned.

Bad for unstructured data where it is impossible to separate what is needed from what is unnecessary for analysis.

Conclusion

Relational databases are great for a wide variety of cases, but there are some situations, or businesses, that require something different. This doesn't necessarily mean that you need to be committed to just one database type, in many cases it might make sense to embrace a multi-database strategy and enjoy the benefits that each database has to offer.

In this document we looked at three alternatives to relational databases and covered some of the most popular alternatives on the market, use cases, and pros and cons. We hope this has opened your eyes to some of the alternatives available which might add value to your database strategy and allow you to fully utilize your data.

Percona is an unbiased open source database expert and our team is always available if you would like to discuss the best database solution available for your business.

To learn more about how Percona can help, and for pricing information, please contact us at +1-888-316-9775 (USA), +44 203 608 6727 (Europe), or email us at sales@percona.com.