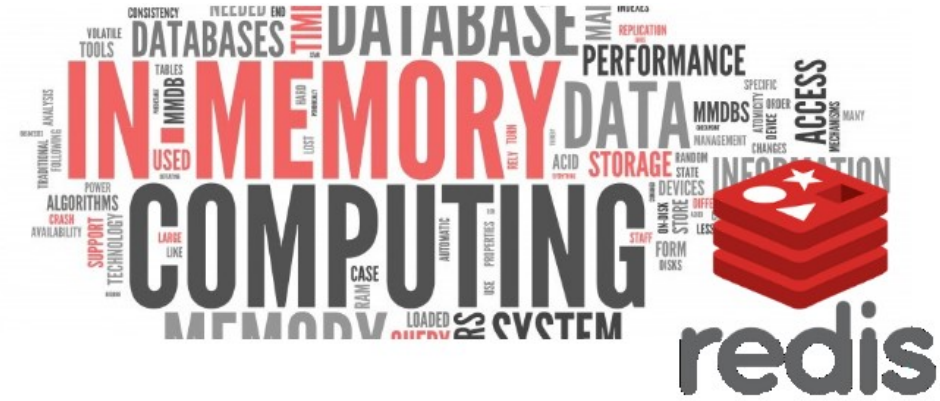
ULB UNIVERSITÉ LIBRE DE BRUXELLES



INFO-H-415 Advanced Databases

Prof. Esteban Zimanyi

In-memory Databases and Redis

Larissa Leite
Alexandr Tretyak

Brussels, December 2015

In-memory databases

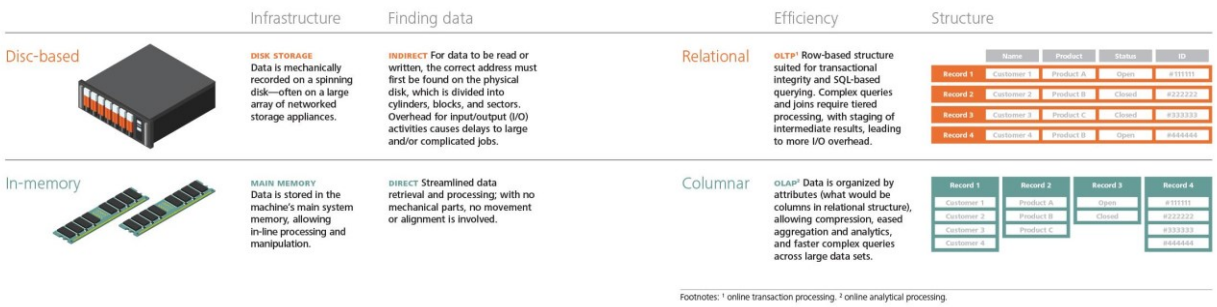
An in-memory database system is a database management system that stores data entirely in main memory, in contrast to traditional database systems, which are designed for data storage on-disk. Since working with data in memory is much faster than writing to and reading from a file system, in-memory databases systems perform very well with respect to applications' data management functions. Hence, in-memory databases are most commonly used in applications that demand very fast data access, storage and manipulation.

An important use for in-memory database systems is in real-time embedded systems. When running on real-time operating systems, they provide the responsiveness needed in applications including IP network routing, telecom switching, and industrial control. Non-embedded applications demanding outstanding performance are also an important area for in-memory database systems. For example, applications for financial markets use in-memory database to provide instant data manipulation, in order to identify and leverage market opportunities, while some multi-user web applications – such as e-commerce and social networking sites – use them to cache portions of their back-end on-disk database systems.

In the real world, IT and data management practitioners need to assess when the investments in technology, resources and new skills required to transition to an in-memory framework is really essential. The practical aspects involve weighing the need for increased database performance versus the associated costs of acquiring and deploying an in-memory platform. Even though RAM costs have decreased, systems with large-scale memory configurations will still be significantly more expensive compared to database servers that stay with disk storage only. To make an in-memory database technology purchase pay off, it is necessary to find applications with characteristics that make them a good fit, which lies partly in assessing the organisation's demand for processing increased data volumes and the business value that could be delivered as a result of reduced database response time.

In the context of a supply chain management fuelled by in-memory software, enabling real-time analysis of a variety of data streams - inventory data from warehouses and retail locations, information about items in transit on trucks or rail cars, updates on traffic and weather conditions - could help drive faster decisions on routing and distribution to ensure that goods get to where they need to be, when they need to be there. A resulting increase in sales clearly could justify the in-memory investment. Hence, when considering going for an in-memory type of solution, it is necessary to question what will be the difference of running a query or processing transactions 10, 50 or 100 times faster for a specific business domain.

Therefore, with in-memory, companies can crunch massive amounts of data, in real time, to improve relationships with their customers to generate add-on sales and to price based on demand, the marketing team wants real-time modelling of changes to sales campaigns, operations wants to adjust fulfilment and supply chain priorities on the fly, internal audit wants real-time fraud and threat detection.



Background

Creating a unified view of big data is a burdensome and onerous task. Moreover, business analysis reports typically are not run directly on operational level, but on aggregated data from a data warehouse. Operational data is transferred into the data warehouse in batch jobs, which makes flexible, ad hoc reporting on the most up-to-date data impossible. As a consequence, company leaders have to make decisions based on data which are either out of date or incomplete. This is obviously not a true ‘‘real-time’’ solution. The continuous and rapid evolution of hardware architectures seen since the introduction of the microprocessor plays an important role on this scenario. This has become especially significant in the last decade where multi-core processors and the availability of large amounts of main memory at low cost are now enabling new breakthroughs in the software industry. It has become possible to store data sets of whole companies entirely in main memory, offering performance that is orders of magnitudes faster than traditional disk-based systems. Hard disks, the only remaining mechanical device in a world of silicon, will soon only be necessary for backing up data. With in-memory computing and insert-only databases using row and column-oriented storage, transactional and analytical processing can be unified. This development has the potential to change how enterprises work and finally offer the promise of real time computing.

One important question on this context is "*How can companies take advantage of in-memory applications to improve the efficiency and profitability of their business?*". It is not very difficult to predict that this breakthrough innovation will lead to fundamentally improved business processes, better decision-making, and new performance standards for enterprise applications across industries and organisational hierarchies. In-memory technology is a catalyst for innovations, and the enabler for a level of information quality that has not been possible until the near past. In-memory enterprise data management provides the necessary equipment to excel in a future where businesses face ever-growing demands from customers, partners, and shareholders. With billions of users and a hundred times as many sensors and devices on the internet, the amount of data we are confronted with is growing exponentially. Being able to quickly extract business-relevant information not only provides unique opportunities for businesses; it is a critical differentiator in competitive markets.

With in-memory technology, companies can turn the massive amounts of data available into information to create strategic advantage in near-real time. Operational business data can be interactively analysed and queried directly by decision makers, opening up completely new scenarios, horizons and opportunities.

When considering the area of financial accounting, where data needs to be frequently aggregated for reporting on a daily, weekly, monthly, or annual basis, in-memory data management along with the necessary filtering and aggregation can happen in real time. Accounting can be done anytime and in an ad hoc manner. Financial applications not only becomes significantly faster, but they will also become less complex and easier to use. Every user of the system is able to directly analyse massive amounts of data. New data are available for analysis as soon as they are entered into the operational system, and simulations, forecasts, and what-if scenarios can be done on demand, anytime and anywhere. What took hours or days in traditional disk-based systems can happen in the blink of an eye. Users of in-memory enterprise systems tend to be more productive and responsive.

Comparison of RDBMS and In-memory technologies

In-memory technology is set to revolutionise enterprise applications both in terms of functionality and cost due to a vastly improved performance. This enables enterprise developers to create completely new applications and allow enterprise users and administrators to think in new ways about how they wish to view and store their data. The performance improvements also mean that costly workarounds, necessary in the past to ensure that data could be processed in a timely manner, are no longer necessary. Chief amongst these is the need for separate operational and analytical systems. In-memory technology allows analytics to be run on operational data, simplifying both the software and the hardware landscape, leading ultimately to lower overall cost.

Special materialised data structures, called cubes, have been created to efficiently serve analytical reports. Such cubes are based on a fixed number of dimensions along which analytical reports can define their result sets. Consequently, only a particular set of reports can be served by one cube. If other dimensions are needed, a new cube has to be created or existing ones have to be extended. In the worst case, a linear increase in the number of dimensions of a cube can result in an exponential growth of its storage requirements. Extending a cube can result in a deteriorating performance of those reports already using it. The decision to extend a cube or build a new one has to be considered carefully. In any case, a wide variety of cubes may be built during the lifetime of a system to serve reporting, thus increasing storage requirements and also maintenance efforts. Instead of working with a predefined set of reports, business users should be able to formulate ad-hoc reports. Their playground should be the entire set of data the company owns, possibly including further data from external sources. Assuming a fast in-memory database, no more pre-computed materialised data structures are needed. As soon as changes to data are committed to the database, they will be visible for reporting. The preparation and conversion steps of data if still needed for reports are done during query execution and computations take place on the fly. Computation on the fly during reporting on the basis of cubes that do not store data, but only provide the interface for reporting, solves a problem that has existed up to now and allows for performance optimisation of all analytical reports likewise.

As data volumes grew, RDBMSs were no longer able to efficiently service the requirements of all categories of enterprise applications. In particular, it became impossible for the DBMS itself to service ad-hoc queries on the entire transactional database in a timely manner. One of the reasons the DBMSs were unable to handle these ad-hoc queries is the design of the database schemas that underlie most transactional enterprise applications. OLTP schemas are highly normalised to minimise the data entry volume and to speed up

inserts, update and deletes. This high degree of normalisation is a disadvantage when it comes to retrieving data, as multiple tables may have to be joined to get all the desired information. Creating these joins and reading from multiple tables can have a severe impact on performance, as multiple reads to disk may be required. Analytical queries need to access large portions of the whole database, which results in long run times with regard to traditional solutions. Online Analytical Processing (OLAP) systems were developed to address the requirement of large enterprises to analyse their data in a timely manner. These systems relied on specialised data structures designed to optimise read performance and provide quick processing of complex analytical queries. Data must be transferred out of an enterprise's transactional system, into an analytical system and then prepared for predefined reports.

The data transfer happens in cyclic batches, in a so-called Extract, Transform, and Load (ETL) process. The required reports may contain data from a number of different source systems. This must be extracted and converted into a single format that is appropriate for transformation processing. Rules are then applied during the transformation phase to make sure that the data can be loaded into the target OLAP system. These rules perform a number of different functions, for example, removing duplicates, sorting and aggregation. Finally, the transformed data is loaded into a target schema optimised for fast report generation. This process has the severe limitation in that one is unable to do real-time analytics as the analytical queries are posed against a copy of the data in the OLAP system that does not include the latest transactions.

The main reason that current RDBMSs cannot perform the required queries fast enough is that query data must be retrieved from disk. Modern systems make extensive use of caching to store frequently accessed data in main memory but for queries that process large amounts of data, disk reads are still required. Simply accessing and reading the data from disk can take a significant amount of time. Main memory or in-memory databases have existed since the 1980s but it is only recently that Dynamic Random Access Memory (DRAM) has become inexpensive enough to make these systems a viable option for large enterprise systems. The ability of the database layer in an enterprise application to process large volumes of data quickly is fundamental to our aim of removing the need for a separate analytics systems. This will allow us to achieve our goal of providing a sub-second response time for any business query. In-memory databases based on the latest hardware can provide this functionality and they form the cornerstone of the database architecture.

Technological Improvements

Since in-memory databases utilise the server's main memory as primary storage location, the size, cost, and access speed provided by main memory components are vitally important. With the help of data compression, today's standard server systems comprise sufficiently large main memory volumes to accommodate the entire operational data of all companies. Main memory, as the primary storage location is, nevertheless, becoming increasingly attractive as a result of the decreasing cost/size ratio. The database can be directly optimised for main memory access, omitting the implementation of special algorithms to optimise disk access.

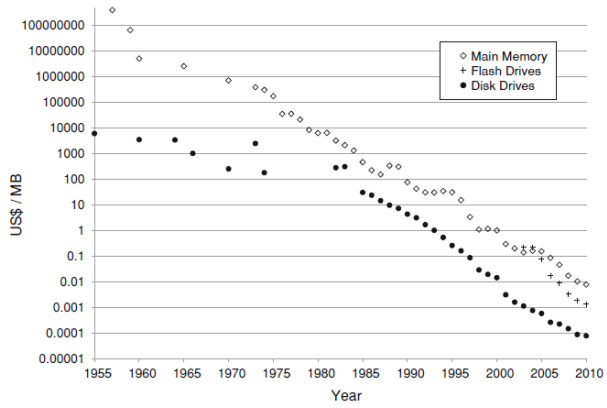


Figure 1. The storage price development

The cost/size relation for disks as well as main memory has decreased exponentially in the past according to the Fig.1. For example, the price for 1 MBof disk space dropped below US $ 0.01 in 2001, which is a rapid decrease compared to the cost of more than US $ 250 in 1970. A similar development can be observed for main memory. In addition to the attractiveness of fitting all operational business data of a company into main memory, optimising and simplifying data access accordingly, the access speed of main memory compared to that of disks is four orders of magnitude faster: a main memory reference takes 100 ns. Current disks typically provide read and write seek times about 5ms.

In reality, the performance of Central Processing Units (CPUs) doubles every 20 months on average. The brilliant achievement that computer architects have managed is not only creating faster transistors, which results in increased clock speeds, but also in an increased number of transistors per CPU per square meter, which became cheaper due to efficient production methods and decreased material consumption. This leads to higher performance for roughly the same manufacturing cost. For example, in 1971, a processor consisted of 2,300 transistors whereas in 2006 it consisted of about 1.7 billion transistors at approximately the same price [Shiva 2013]. Not only does an increased number of transistors play a role in performance gain, but also more efficient circuitry. A performance gain of up to a factor of two per core has been reached from one generation to the next, while the number of transistors remained constant.

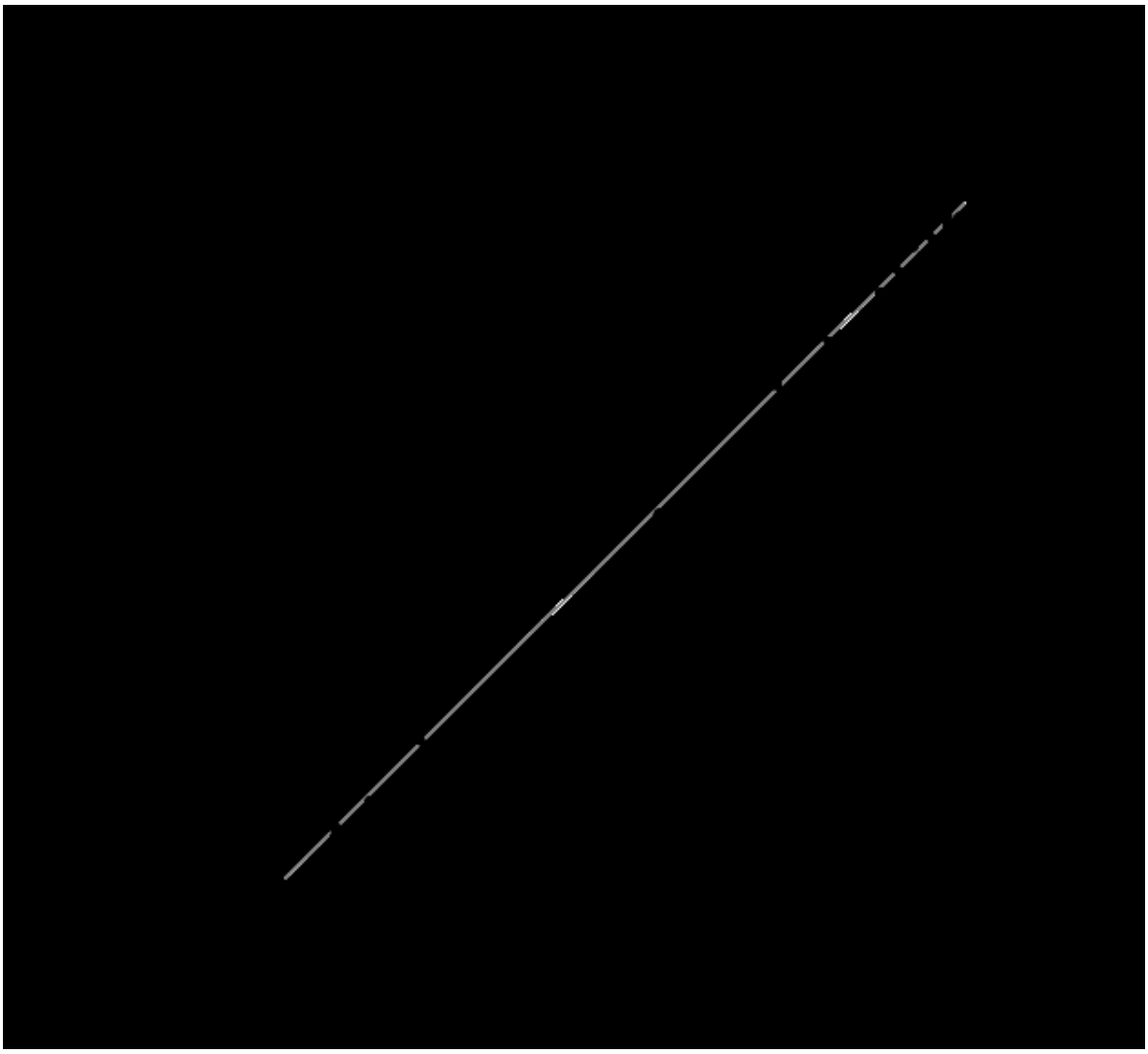


Figure 2. Plot of CPU transistor counts

The clock speed of processors had been growing exponentially for almost 30 years, but has stagnated since 2002. Power consumption, heat distribution and dissipation, and the speed of light have become the limiting factors for Moore's Law. The Front Side Bus (FSB) speed, having grown exponentially in the past, has also stagnated. In 2001, IBM introduced the first processor on one chip, which was able to compute multiple threads at the same time independently. The IBM Power 4 was built for the high-end server market and was part of IBM's Regatta Servers. Regatta was the codename for a module containing multiple chips, resulting in eight cores per module. In 2002, Intel introduced its proprietary hyper-threading technology, which optimises processor utilisation by providing thread-level parallelism on a single core. With hyper-threading technology, multiple logical processors with duplicated architectural state are created from a single physical processor. Several tasks can be executed virtually in parallel, thereby increasing processor utilisation. Yet, the tasks are not truly executed in parallel because the execution resources are still shared and only multiple instructions of different tasks that are compatible regarding resource usage can be executed in a single processing step. Hyper-threading is applicable to single-core as well as to multi-core processors. Until 2005, single-core processors dominated the home and business computer domain. For the consumer market,

multi-core processors were introduced in 2005 starting with two cores on one chip, for example, Advanced Micro Devices's (AMD) Athlon 64 X2. At its developer forum in autumn 2006, Intel presented a prototype for an 80-core processor, while IBM introduced the Cell Broadband Engine with ten cores in the same year. The IBM Cell Broadband Engine consists of two different types of processing elements, one two-core PowerPC processing element and up to eight synergistic processing elements that aim at providing parallelism at all abstraction levels. In 2008, Tilera introduced its Tile64, a multi-core processor for the high-end embedded systems market that consists of 64 cores. 3Leaf is offering a product that is based on the HyperTransport architecture with 192 cores. In the future, higher numbers of cores are anticipated on a single chip. In 2008, Tilera predicted a chip with 4,096 cores by 2017 for the embedded systems market and Sun estimated that servers are going to feature 32 and up to 128 cores by 2018.

The increased performance of the FSB, which so far has been the only interface from the CPU to main memory and all other input/output (I/O) components, that no longer keeps up with the exponential growth of processor performance. Important for the calculation of the theoretical memory throughput are clock speed (cycles per second) and bandwidth of the data bus. The increased clock speed and the use of multiple cores per machine are resulting in a widening gap between the ability of processors to digest data and the ability of the infrastructure to provide data. In-memory and column-oriented data storage enable the usage of additional processing power despite the bottleneck created by the aforementioned widening gap. High compression rates of column-oriented storage can lead to a better utilisation of bandwidth. In-memory data storage can utilise enhanced algorithms for data access, for example, prefetching. Using compressed data and algorithms that work on compressed data is standard technology and has already proven to be sufficient to compensate the data supply bottleneck for machines with a small number of cores. It is, however, failing with the addition of many more cores. Experiments with column-oriented, compressed in-memory storage and data-intensive applications showed that the FSB was well utilised, though not yet congested, in an eight-core machine. The data processing requirements of the same applications on a 24-core machine surmounted the FSB's ability to provide enough data. From these experiments it was concluded that new memory access strategies are needed for machines with even more cores to circumvent the data supply bottleneck. Processing resources are often underutilised and the growing performance gap between memory latency and processor frequency intensifies the underutilisation. Intel improved the available transfer rate, doubling the amount of data that can be transferred in one cycle or added additional independent buses on multi-processor boards. The HyperTransport protocol was introduced by AMD in 2001 to integrate the memory controller into the processor. Similar to the HyperTransport protocol, Intel introduced Quick Path Interconnect (QPI) in the second half of 2008. QPI is a point-to-point system interconnect interface for memory and multiple processing cores, which replaces the FSB. Every processor has one or multiple memory controllers with several channels to access main memory in addition to a special bus to transfer data among processors. Compared to Intel FSB in 2007 with a bandwidth of 12.8 GB/s, QPI helped to increase the available bandwidth to 25.6 GB/s in 2008. In Intel's Nehalem EP chips, each processor has three channels from the memory controller to the physical memory. In Intel's Nehalem EX chips, these channels have been expanded to four channels per processor.

On an Intel XEON 7560 (Nehalem EX) system with four processors, benchmark results have shown that a throughput of more than 72 GB/s is possible. In contrast to using the FSB the memory access time differs between local memory (adjacent slots) and remote memory that is adjacent to the other processing units. As a result of this characteristic, architectures based on the FSB are called Uniform Memory Access (UMA)

and the new architectures are called Non-Uniform Memory Access (NUMA). There is a differentiation between cache-coherent NUMA (ccNUMA) and non cache-coherent NUMA systems. Incc NUMA systems, all CPU caches share the same view to the available memory and coherency is ensured by a protocol implemented in hardware. Non cache-coherent NUMA systems require software layers to take care of conflicting memory accesses. Since most of the available standard hardware only provides ccNUMA, it is preferably to pay attention and to concentrate on this form. To exploit NUMA completely, applications have to be made aware of primarily loading data from the locally attached memory slots of a processor. Memory-bound applications might see a degradation of up to 25% of their performance if only remote memory is accessed instead of the local memory. Reasons for this degradation can be the saturation of the QPI link between processor cores to transport data from the adjacent memory slot of another core, or the influence of higher latency of a single access to a remote memory slot. The full degradation might not be experienced, as memory caches and prefetching of data mitigates the effects of local versus remote memory. Assume a job can be split into many parallel tasks. For the parallel execution of these tasks distribution of data is relevant. Optimal performance can only be reached if the executed tasks solely access local memory. If data is badly distributed and many tasks need to access remote memory, the connections between the processors can become flooded with extensive data transfer. Aside from the use for data-intensive applications, some vendors use NUMA to create alternatives for distributed systems. Through NUMA, multiple physical machines can be consolidated into one virtual machine. The difference in the commonly used term of virtual machine, where part of a physical machine is provided as a virtual machine. With NUMA, several physical machines fully contribute to the one virtual machine giving the user the impression of working with an extensively large server. With such a virtual machine, the main memory of all nodes and all CPUs can be accessed as local resources. Extensions to the operating system enable the system to efficiently scale out without any need for special remote communication that would have to be handled in the operating system or the applications. In most cases, the remote memory access is improved by the reservation of some local memory to cache portions of the remote memory. Further research will show if these solutions can outperform hand-made distributed solutions. 3Leaf, for example, is a vendor that uses specialised hardware. Other companies, for example, ScaleMP rely on pure software solutions to build virtual systems. In summary, the enhancement of the clock speed of CPU cores has tended to stagnate, while adding more cores per machine is now the reason for progress. As we have seen, increasing the number of cores does not entirely solve all existing problems, as other bottlenecks exist, for example, the gap between memory access speed and the clock speed of CPUs. Compression reduces the effects of this gap at the expense of computing cycles. NUMA as an alternative interconnection strategy for memory access through multiple cores has been developed. Increased performance through the addition of more cores and NUMA can only be utilised by adapting the software accordingly. In-memory databases in combination with column-oriented data storage are particularly well suited for multi-core architectures. Column-oriented data storage inherently provides vertical partitioning that supports operator parallelism.

Redis

Redis is an open source (BSD licensed), in-memory and key-value **data structure store**, used as database, cache and message broker. Most key-value data stores have a limited set of datatypes, but Redis is relatively rich, since it supports data structures such as strings, hashes, lists, sets, sorted sets. Redis has built-in replication, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

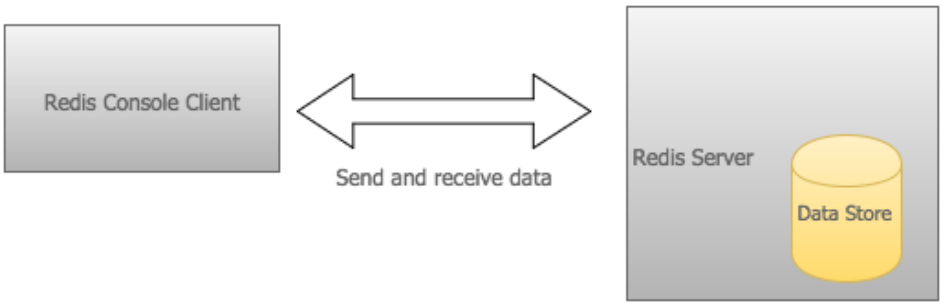
Redis manages data in the form of key and value pairs, which are stored in main memory (RAM). Since **all data needs to fit in main memory**, ideally it is preferred to store essential information which needs to be accessed, modified and inserted at a very fast rate. In summary, Redis outperforms relational databases by an order of magnitude when it comes to speed. However, that also means that the size of the Redis datastore is limited to the size of the available memory. Therefore, although it is possible to use Redis as a standalone database system, it is not very common to use it as the main primary database, mainly due to its limitation regarding storage space. Redis usually comes along with a relational database (MySQL, SQL Server, PostgreSQL, Oracle, etc) or even with other non-relational database. This means that in order to take advantage of Redis, it is not necessary to switch to Redis, but rather use it in existing environments to do things that were not possible before or to fix existing problems.

The main advantage of adding Redis to an application is the gain of performance. Additionally, when compared to other databases, Redis supports much richer data types. It allows keys and values to be strings, lists, sets of strings, sorted sets and hashes, which are more associable to the data types of the high level programming languages commonly used in software development. This eases mapping the concepts employed in development, as well as it is more intuitive than when it comes to the tables in relational databases. Moreover, it is important to note that Redis operations and transactions are atomic.

If using data structures instead of tables poses as an advantage, since developers can directly map the same data structures in their code, the way querying works in Redis could possibly be considered as a disadvantage. This is because there is no query language (only commands), no support for relational algebra, and no ad-hoc queries (as it happens when using SQL on a RDBMS). Additionally, all data accesses should be anticipated by the developer, and the proper data structures and data access paths must be designed and carefully considered before developing the application.

Architecture

Redis architecture contains two main processes: Redis client and Redis Server. They can be in the same computer or in two different computers.



Redis server is responsible for storing data in memory and it handles all kinds of management. Redis client can be Redis console client or any other programming language's Redis API.

Redis stores everything in main memory, which is volatile, and it also uses persistence on disk so the data is not lost once the Redis server is restarted (for instance, because the computer was turned off). However, this is not mandatory and depends on the application requirements.

Data Structures

Redis allows the storage of keys that map to any one of different data structure types: STRINGs, LISTs, SETs, HASHes, and ZSETs. The first four should be familiar to almost any programmer, and their implementation and semantics are similar to those same structures built in a variety of programming languages.

Structure type	What it contains	Structure read/write ability
`STRING`	Strings, integers, or floating-point values	Operate on the whole string, parts, increment/decrement the integers and floats
`LIST`	Linked list of strings	Push or pop items from both ends, trim based on offsets, read individual or multiple items, find or remove items by value
`SET`	Unordered collection of unique strings	Add, fetch, or remove individual items, check membership, intersect, union, difference, fetch random items
`HASH`	Unordered hash table of keys to values	Add, fetch, or remove individual items, fetch the whole hash
`ZSET` (sorted set)	Ordered mapping of string members to floating-point scores, ordered by score	Add, fetch, or remove individual values, fetch items based on score ranges or member value

Redis keys

Redis keys are binary safe, which means that any binary sequence can be used as a key, from a string to the content of a JPEG file. An empty string is also a valid key. Shorter keys are recommended not only memory-wise, but also because the of the key lookups, which may require several key-comparisons. However, the

keys shouldn't be very short because it's better if they are readable. For example, writing "u1000flw" as a key instead of "user:1000:followers" is not a very good choice. Additionally, sticking with a pattern for the keys is also preferable.

Redis expires: keys with limited time to live

Regardless of the value type, it is possible to set a timeout for a key, which means that it has a limited time to live. When the time to live elapses, the key is automatically destroyed, exactly as if the user called the DEL command with the key.

Altering and querying the key space

There are commands that can be used with keys of any type. For example the EXISTS command returns 1 or 0 to signal if a given key exists or not in the database, while the DEL command deletes a key and associated value, whatever the value is.

There are many key space related commands, but the above two are the essential ones together with the TYPE command, which returns the kind of value stored at the specified key.

Redis Strings

String is the simplest type of value that can be used as a Redis key. Using the SET and the GET commands are the way to set and retrieve a string value. SET will replace any existing value already stored into the key, in the case that the key already exists, even if the key is associated with a non-string value. Values can be strings (including binary data) of every kind, and it cannot be bigger than 512 MB.

Redis Lists

Redis lists are implemented using Linked Lists, which means that the operation of adding a new element in the head or in the tail of the list is performed *in constant time*. The speed of adding a new element with the LPUSH command to the head of a list with ten elements is the same as adding an element to the head of list with 10 million elements. Redis Lists are implemented with linked lists because for a database system it is essential to be able to add elements to a list of any size in a very fast way.

The LPUSH command adds a new element into a list, on the left (at the head), while the RPUSH command adds a new element into a list, on the right (at the tail). Finally, the LRANGE command extracts ranges of elements from lists:

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```

Note that LRANGE takes two indexes, the first and the last element of the range to return. Both the indexes can be negative, telling Redis to start counting from the end: so -1 is the last element, -2 is the penultimate element of the list, and so forth. Moreover, both LPUSH and RPUSH commands are *variadic commands*, meaning that you are free to push multiple elements into a list in a single call:

```
> rpush mylist 1 2 3 4 5 "foo bar"
(integer) 9
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
4) "1"
5) "2"
6) "3"
7) "4"
8) "5"
9) "foo bar"
```

An important operation defined on Redis lists is the ability to *pop elements*. Popping elements is the operation of both retrieving the element from the list, and eliminating it from the list, at the same time. Elements can be popped from left and right, similarly to how they can be pushed to both sides of the list:

```
> rpush mylist a b c
(integer) 3
> rpop mylist
"c"
> rpop mylist
"b"
> rpop mylist
"a"
```

Redis returns a NULL value to signal that there are no elements into the list.

Common use cases for lists

Lists are useful for a number of tasks, and on Redis website it is possible to observe its use in the social network domain, for example, to remember the latest updates posted by users. Twitter takes the latest tweets posted by users into Redis lists. To describe a common use case, if a home page shows the latest photos published in a photo sharing social network and the goal is to speedup access, lists can be useful, for instance in two scenarios: (i) every time a user posts a new photo, its ID is added into a list with LPUSH, and (ii) when users visit the home page, LRANGE 0 9 is used in order to get the latest 10 posted items.

Redis Hashes

Redis hashes are basically field-value pairs:

```
> hmset user:1000 username antirez birthyear 1977 verified 1
OK
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```

While hashes are handy to represent *objects*, the number of fields allowed inside a hash has no limits other than available memory. The command HMSET sets multiple fields of the hash, while HGET retrieves a single field. HMGET is similar to HGET but returns an array of values:

```
> hmget user:1000 username birthyear no-such-field
1) "antirez"
2) "1977"
3) (nil)
```

There are commands that are able to perform operations on individual fields as well, like HINCRBY:

```
> hincrby user:1000 birthyear 10
(integer) 1987
> hincrby user:1000 birthyear 10
(integer) 1997
```

Redis Sets

Redis Sets are unordered collections of strings. The SADD command adds new elements to a set. It's also possible to do a number of other operations against sets like testing if a given element already exists, performing the intersection, union or difference between multiple sets, and so forth.

Sets are good for expressing relations between objects. For instance, sets can be used to implement tags. A simple way to model this problem is to have a set for every object we want to tag. The set contains the IDs of the tags associated with the object.

In case we want to tag *news*, if our news ID 1000 is tagged with tags 1, 2, 5 and 77, we can have one set associating our tag IDs with the *news* item:

```
> sadd news:1000:tags 1 2 5 77
(integer) 4
```

However sometimes it is necessary to retrieve inverse relation as well: the list of all the news tagged with a given tag:

```
> sadd tag:1:news 1000
(integer) 1
> sadd tag:2:news 1000
(integer) 1
> sadd tag:5:news 1000
(integer) 1
> sadd tag:77:news 1000
(integer) 1
```

To get all the tags for a given object is trivial:

```
> smembers news:1000:tags
1. 5
2. 1
3. 77
4. 2
```

Note: in the example it is assume that there is another data structure, for example a Redis hash, which maps tag IDs to tag names.

Redis Sorted sets

Sorted sets are a data type which is similar to a mix between a Set and a Hash. Like sets, sorted sets are composed of unique, non-repeating string elements, so in some sense a sorted set is a set as well. However, while elements inside sets are not ordered, every element in a sorted set is associated with a floating point value, called *the score* (this is why the type is also similar to a hash, since every element is mapped to a value). Moreover, elements in a sorted sets are *taken in order* (so they are not ordered on request, order is a peculiarity of the data structure used to represent sorted sets). They are ordered according to the following rule:

- If A and B are two elements with a different score, then A > B if A.score is > B.score.
- If A and B have exactly the same score, then A > B if the A string is lexicographically greater than the B string. A and B strings can't be equal since sorted sets only have unique elements.

Considering a simple example of adding a few selected hackers names as sorted set elements, with their year of birth as "score":

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer 1)
> zadd hackers 1953 "Richard Stallman"
(integer) 1
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
```

```
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
> zadd hackers 1912 "Alan Turing"
(integer) 1
```

It is possible to observe that ZADD is similar to SADD, but takes one additional argument (placed before the element to be added) which is the score. ZADD is also variadic, so it is possible to specify multiple score-value pairs. With sorted sets it is trivial to return a list of hackers sorted by their birth year because actually *they are already sorted*.

In terms of implementation, it is important to note that sorted sets are implemented via a dual-ported data structure containing both a skip list and a hash table, so every time an element is added, Redis performs an O(log(N)) operation.

Persistence

There are three different possibilities to work with persistence using Redis: RDB, AOF and SAVE command.

RDB Mechanism

RDB makes a copy of all the data in memory and stores them in secondary, permanent storage. It consists of a very compact single-file and it performs point-in-time snapshots at specified intervals. RDB works very well for backups (daily snapshots, for instance). However, it is not the best approach in order to minimise the chance of data loss. The data created after a snapshot can be lost in case something goes wrong - a power outage, for example - before Redis performs the next snapshot.

AOF

AOF logs all the write operations received by the server, and allows the fsync policy to be defined (no fsync at all, fsync every second or every query). The AOF log is an append log only, and this mechanism allows Redis to deal quite easily when there is a power outage, or when new versions of the file needs to be created due to file size constraints. Compared to RDB, AOF files are usually bigger than the equivalent RDB files, and it can also be slower depending on the fsync policy.

SAVE Command

The SAVE command is a way to manually set the Redis server to create a RDB snapshot anytime in a given condition. For example, the server can be configured to automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed.

In order to get the best persistence result, AOF can be used together with RDB. As a matter of fact, the Redis community aim to unify AOF and RDB into a single persistent model in the future.

Replication

Redis offers master-slave replication that allows slave Redis servers to be exact copies of master servers. Redis uses asynchronous non-blocking replication both on the master and slave sides. This means that they can handle queries while performing synchronisation. Replication can be used both for scalability, in order to have multiple slaves for read-only queries, or simply for data redundancy. If you set up a slave, upon connection it sends a SYNC command. It doesn't matter if it's the first time it has connected or if it's a reconnection. The master then starts background saving, and starts to buffer all new commands received that will modify the dataset. When the background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory. The master will then send to the slave all buffered commands. This is done as a stream of commands and is in the same format of the Redis protocol itself. Slaves are able to automatically reconnect when the master-slave link goes down for some reason. If the master receives multiple concurrent slave synchronisation requests, it performs a single background save in order to serve all of them.

Scalability

The master-slave approach is also very important in terms of scalability. The simplest method to increase total read throughput available to Redis is to add read-only slave servers. Additional servers can run by connecting to a master, receive a replica of the master's data, and be kept up to date in near-real time. This enhances additional read query capacity with every new slave by running read queries against one of several slaves. With respect to maximising performance when the capacity of a single machine has been reached, it's time to shard the data across multiple machines.

As from Redis 3.0, Redis includes a built in option for sharding called Redis Cluster. Redis Cluster is a distributed implementation that provides high performance and linear scalability up to 1000 nodes. There are no proxies, asynchronous replication is used, and no merge operations are performed on values. It also aims to ensure an acceptable degree of write safety. The system tries (in a best-effort way) to retain all the writes originating from clients connected with the majority of the master nodes. Redis Cluster is able to survive partitions where the majority of the master nodes are reachable and there is at least one reachable slave for every master node that is no longer reachable.

Industry use cases

In this section, we explore some practical examples on how Redis has been useful for big companies.

Pinterest

Pinterest is a photo sharing social network, and it figures as one of the most popular applications at the moment. It was founded in 2010 and it has more than 70 million registered users. As most other social

networks, they have a follower model modelled as a graph. Unlike Facebook and Twitter, users can follow not only other users, but also boards (which can be seen as a photo album). Additionally, if a user A follows user B, A automatically follows all boards owned by B. With their enormous growth, they felt the need to rethink their follower model. They used a traditional model of storing the sharded graph on MySQL and caching it with memcached, but this solution was reaching its limits. Also, caching the graph data was hard because the cache was useful only if the entire subgraph of a user (vertex) is in cache, which can quickly result in an attempt to cache the entire graph. Moreover, caching also implies that queries "*Does user A follow user B?*" are either in the cache or not. Many times the answer was not in the cache, which required expensive lookups to the persistence store.

The corpus size of the entire Pinterest follower graph is relatively small, so loading the entire graph in memory is feasible. They use Redis to store the graph, which is sharded by user IDs, and the Redis AOF feature updates to disk every second to avoid significant data loss. Since Redis is single threaded, they run multiple instances of Redis to fully utilise the CPU cores. Each Redis instance serves one virtual shard, which allows easy splitting of shards when the instances on one machine reach capacity.

Pinterest needs to efficiently respond to point queries such as *"Does user A follow user B"*, support filtering queries used on a search page such as *"Which of these 25 users are followed by me?"*, and get all users followed by users or boards to fan out the incoming pins. In order to respond to such queries, they explore the main data structures of Redis to build their model: Set, Sorted Set and Hash.

In the end, when Pinterest migrated away from the existing sharded MySQL cluster, they saved about 30% IOps.

Flickr

Flickr is a photo organiser and sharing. When they launched their new mobile app back in 2012, their main goal was to further increase user engagement by allowing users to know what was happening on Flickr in as near-real time as possible. In order to make this possible, they developed an event system that involved three steps: (i) event generation, which happens while processing a user request, and to ensure that there was little to no impact on the response times, they do a lightweight write into a global Redis queue; (ii) event targeting, in which many workers read from the global Redis queue, and load up any additional information necessary to act on the notification (checking to see what users should be notified, whether those users have devices that are registered to receive notifications, if they have opted out of notifications of this type); and (iii) message delivery, to which they have built a separate endpoint in NodeJS, and rely on the publish subscribe Redis built-in functionality.

Flickr accomplished their initial goals when they turned to Redis to help their push notifications functionality involved near-real time response, and handling thousands of notification per second, which they assessed in their stress test: they were able to process more than 2,000 notifications per second on a single host (8 Node.js workers, each subscribing to multiple shards).

Redis *versus* other databases

Before comparing Redis directly to other database solutions, we would like to illustrate Redis' popularity and The business software site G2 Crowd published in the fall of 2015 a report ranking eight NoSQL databases to help purchasers in their selections. Redis earned the highest overall *satisfaction* and *market presence* scores. A few factors were taken into consideration, such as customer satisfaction reported by users, and with vendor market presence determined from social and public data to rank products. The Fall 2015 report was based on more than 130 reviews written by developers, database administrators and other business professionals.

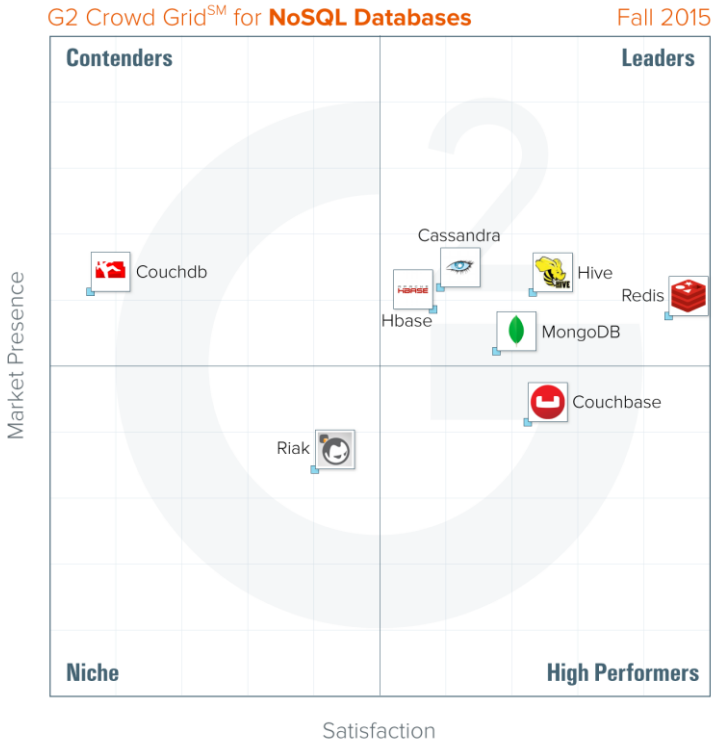


Figure 3. NoSQL Database Ranking

Also, Redis figures well in another ranking, published by The DB-Engines Ranking, which represents a list of key-value database systems ranked by their current popularity. The popularity of a system is measured considering: (i) the number of mentions of the system on websites, given by the number of results in search engines queries (Google, Google Trends and Bing); (ii) frequency of technical discussions about the system, as in related questions and the number of interested users on Stack Overflow and DBA Stack Exchange; (iii) number of job offers in which the system is mentioned on leading job search engines (Indeed and Simply Hired); (iv) relevance in social networks, mainly LinkedIn and Twitter.

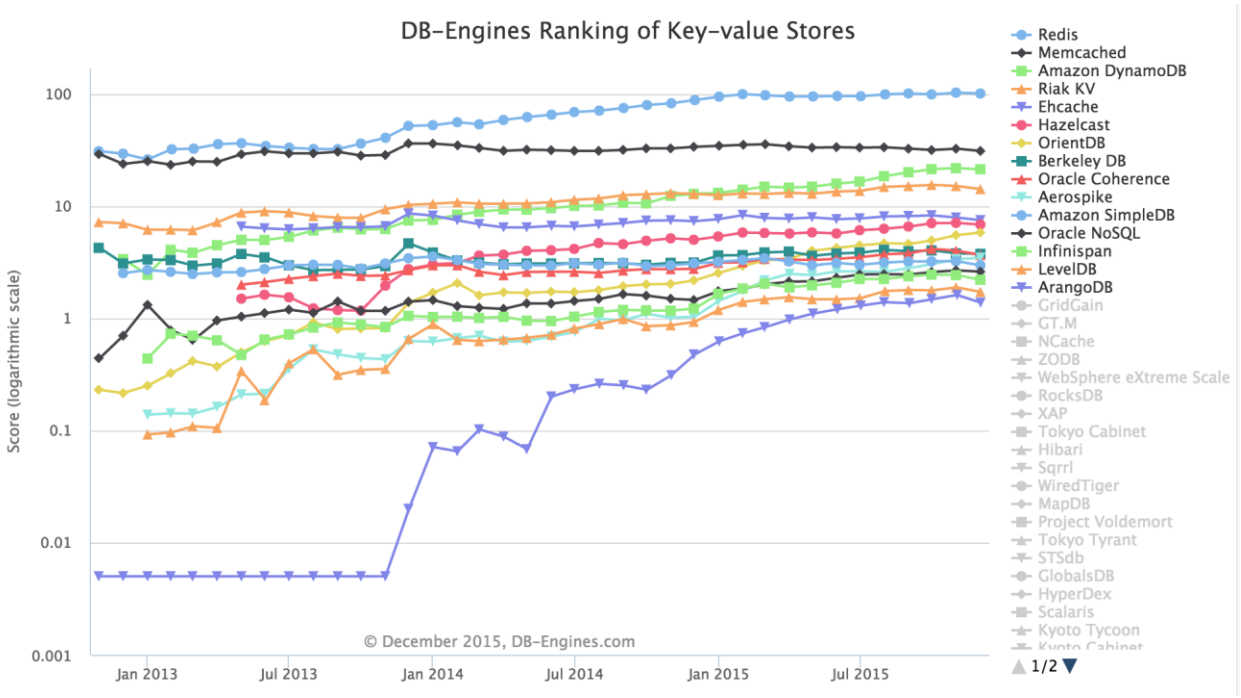


Figure 4. Key-value stores Database Ranking

Redis *vs* Memcached

Memcached vs Redis is a debate that often arises when cache comes into play, if the goal is to increase the performance of any application. Both Memcached and Redis are in-memory, key-value data stores. Hence, they keep all the data in main memory, and they perform similarly with respect to throughput and latency. Memcached was developed in 2003, whereas Redis first release was in 2009, and parts of it were built in response to lessons learned from using Memcached. Both of them are used by many companies, mainly because they are not only extremely effective, but also relatively simple. Getting started with either Memcached or Redis does not take more than a few minutes, which turns a small amount of time and effort into a huge and positive impact on performance.

Redis is newer and has more features when compared to Memcached. However, for caching small and static data Memcached can be considered a better and more efficient choice, because Memcached's internal memory management consumes comparatively less memory resources for metadata. The other case in which Memcached is preferable over Redis is for horizontal scaling, partly due to its design and simpler capabilities. Hence, Redis is usually a better choice as a result of its greater power and efficiency overall, and its more sophisticated approaches to memory management. Additionally, Redis provides greater flexibility regarding the objects that can be cached. Whereas Memcached works only with strings, and limits key names to 250 bytes and values to 1MB, Redis allows keys and values to be at most 512MB each, and it has six data types that allows better manipulation of cached data as well as more possibilities to the developer. In contrast to Memcached, Redis stores data to disk, and therefore offers support in case of failures and replication.

Redis *vs* MongoDB

MongoDB is one of the most popular NoSQL solutions. It is a cross-platform document-oriented database that relies on JSON-like documents with dynamic schemas, which makes the integration of data in certain types of applications easier and faster. Comparing MongoDB to Redis, MongoDB is more suitable for querying, whereas Redis is better on performance. However, MongoDB can cope well with much larger datasets than Redis, since all the data manipulated by Redis needs to fit in main memory. Additionally, for those who come from a relational database and SQL background, the learning curve is steeper for Redis when compared to MongoDB.

Redis *vs* SQL Server and Oracle

Overall, comparing Redis to SQL Server or Oracle, both traditional relational database solutions, is not very applicable. They operate with different purposes, and Redis does not support most of the common features inherent to DBMSs. However, both SQL Server and Oracle are offering in-memory solutions that pose as competition to Redis. Microsoft released SQL Server Hekaton in 2014 built-in with the new release of SQL Server. Hekaton is an in-Memory OLTP new database engine component, fully integrated into SQL Server. It is optimised for OLTP workloads accessing memory resident data. In-Memory OLTP allows OLTP workloads to achieve significant improvements in performance, and reduction in processing time. Tables can be declared as 'memory optimised' to enable In-Memory OLTP's capabilities. Hekaton was not meant to run an entire database in memory but rather as a new engine designed to handle specific cases while keeping the interface the same. Everything to the end-user is identical to the rest of SQL Server: SQL and stored procedures to access data, triggers and indexes, and everything is still atomic with ACID properties. As for Oracle, they also introduced an in-memory solution in 2014. Similarly to Hekaton, Oracle Database In-Memory offers the possibility to populate only performance sensitive tables or partitions into memory. Additionally, they provide a unique dual-format architecture that enables tables to be simultaneously represented in memory using traditional row format and a new in-memory column format that has proven to be useful in analytics scenarios. Oracle Database In-Memory implements state-of-the-art algorithms for in-memory scans, joins and aggregation, which enable analytics that previously took hours to complete in seconds, easing real-time business decisions. Enabling Oracle Database In-Memory is as easy as setting the size of the in-memory column store and identifying tables to bring into memory. Background processes populate data from storage into in-memory columns while the database remains fully active and accessible. No changes are required to use it with any application or tool that runs against the Oracle Database.

Practical examples using Redis

We developed two applications to illustrate the use of Redis. The first was a chat application that highlights how Redis can be extremely useful in a context where two factors are necessary: (i) fast access to real time data, and (ii) the publish/subscribe messaging pattern plays an essential role on this domain. The second was a Twitter-like simple social network that contains basically users and a timeline (which represents status messages from the users themselves and the users they follow). Additionally, users can follow other users, and on each user's timeline the statuses of the users they follow are displayed in descending chronological order.

Chat application

The chat was developed as a web application, we used NodeJS on the server side with the Socket.IO library. Socket.IO is a library that is used both on the server and client sides, to ease the communication between both parts. It has built-in events such as connect, disconnect, join, message, that are defined on the server side, so that the server can detect and react to the different types of actions made by the client application.

With respect to the publish/subscribe messaging pattern, Redis has a built-in implementation and it provides three main commands: SUBSCRIBE, UNSUBSCRIBE and PUBLISH. As accurately described on Redis' website: "*The senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterised into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology*".

Along with the pubsub pattern, we also use Redis to store the chat messages. Every time a new message is sent by an user, the server receives this message and executes the following command in order to save the incoming message: redisClient.lpush('messages', JSON.stringify(data)); This code pushes the latest message to the messages list stored in Redis. Then, when the client launches, or in case of a disconnect, users are still able to see the messages when they are back online. In our case, we just created one chat room, but with Redis it is also very simple to store messages for different chat rooms, which is more likely in a real world scenario.

When comparing to what would be possible to do in a relational database, it is very clear that without Redis some additional work or library would be necessary to implement the publish/subscribe messaging pattern. Moreover, even though a chat application can be easily modeled with a relational schema, it is much faster to retrieve messages from main memory than from disk, and this performance attribute is extremely valuable in the context of a real-time application.

The code for this application is available at: http://github.com/larissaleite/redis-examples/chat

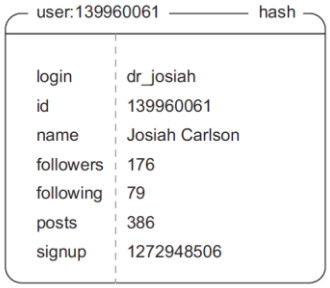
Twitter-like social network application

One of the first examples displayed on Redis' website is a Twitter-clone developed with Redis and PHP. Based on that, we developed a simplified version of a social network based on Twitter using the redis-py: a client library of Redis for Python. Our implementation was also based on the book "Redis in Action" [Carlson 2013]. The main goal of selecting this type of application was the ease to generate data to simulate a real application, when compared to the chat app. Moreover, this domain is more easily modelled in a relational database. Hence, we stored the same amount of data in SQLite in order to assess how the queries would perform when compared to Redis.

In Redis, we use the main data structures to hold the data of the social network as follows:

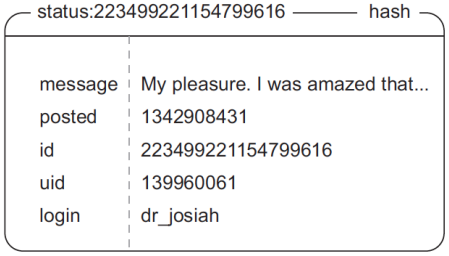
Users' information (login, how many people they follow, how many followers they have, and the amount of status messages they have posted) are stored as a HASH.

`user:<id>, {followers: 0, following: 0, posts: 0, login: <username>}`



The status messages are also stored as a HASH:

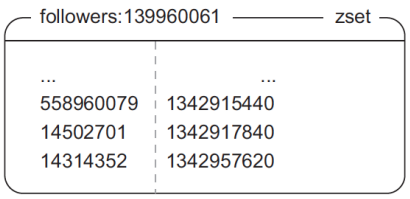
`status:<id>, {uid: <user_id>, message: <message content>}`

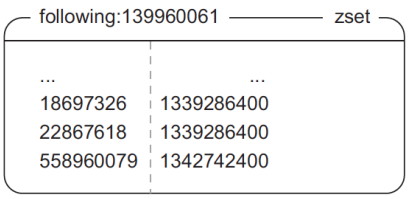


We store as a ZSET (sorted set) the lists of users that each user follows, and the users that follow them. To keep a list of followers and a list of those people that a user is following, we'll also store user ID s and timestamps in ZSET s as well, with members being user ID s, and scores being the timestamp of when the user was followed:

`followers:<user_id>, [<follower_id> <timestamp>, <follower_id> <timestamp>, <follower_id> <timestamp> …]`

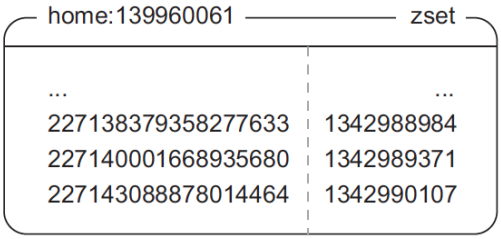
`following:<user_id>, [<following_id> <timestamp>, <following_id> <timestamp>, <following_id> <timestamp> …]`



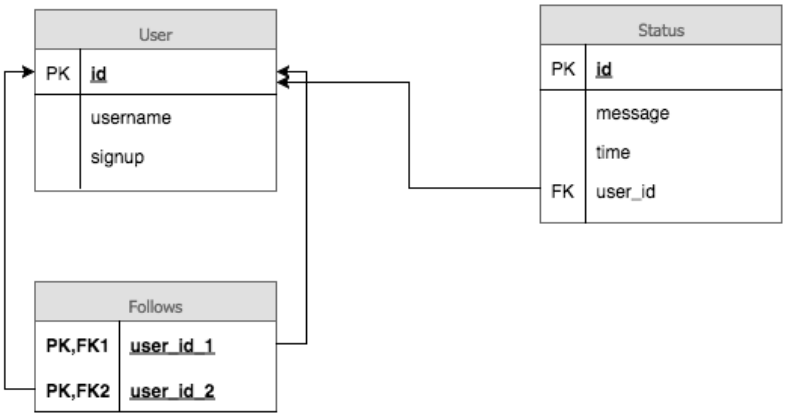


Finally, the timeline, the most important feature of the application, is also stored as a ZSET, in order to keep the statuses ordered by timestamp (called the score in Redis):

`home:<user_id>, [<status_id> <status_timestamp>, <status_id> <status_timestamp>, <status_id> <status_timestamp>, …]`



In our relational database using SQLite, we modelled the relationships as follows:



In our application we basically created five methods:

	Redis	SQLite
create_user(username)	Inserts a new user as hash	Inserts a new user to the users table
post_status(user, message)	Inserts a new status as a hash and adds the status id to the sorted set of every follower of the corresponding user	Inserts a new status to the status table, referencing the corresponding user in foreign key column
follow_user(user1, user2)	Inserts the user id of user1 to the followers sorted set of user2 and inserts the user id of user2 to the following sorted set of user1	Inserts a new line to the follows table

retrieve_timeline(user)	Retrieves the timeline of the corresponding user by selecting all the status id from the home sorted set and getting the status content from the status hash	Retrieves the timeline of the corresponding user by joining the tables user and status on the user id
retrieve_followers(user)	Retrieves basic information from all the followers of the corresponding user by getting all the followers from the followers sorted set and getting the information from each follower's hash	Retrieves basic information from all the followers of the corresponding user by selecting all the followers from the follows table and joining with the user table
retrieve_following(user)	Retrieves basic information from all the users are followed by the corresponding user, from the following sorted set and getting the information from each user's hash	Retrieves basic information from all the followers of the corresponding user by selecting all the users followed by him/her from the follows table and joining with the user table

The methods' signatures were the same for both versions of the application, using Redis and SQLite.

We generated data corresponding to registering 10000 users, each user with 200 followers and 100 status messages. In this context, we calculated the time in milliseconds to run the queries corresponding to the last three methods:

	Redis	SQLite
retrieve_timeline	0.36213	456.429
retrieve_followers	0.05584	0.17909
retrieve_following	0.04713	0.18737

These numbers show that despite Redis taking longer to write all the data to the database, the performance when retrieving such data, specially in the case of the timeline, is significantly better when compared to SQLite.

The code for this application is available at: http://github.com/larissaleite/redis-examples/twitter

Conclusion

Not so long ago, it would not have been easy to imagine enterprise-class database management systems running primarily in main memory. But since the cost of memory is orders-of-magnitude less expensive than it used to be, the decrease in prices have opened new opportunities for configuring database systems to take advantage of increased main-memory capacities.

In-memory databases provide accelerated application performance in two ways: (i) maintaining data in main memory instead of significantly slower disk-based storage minimises the data latency associated with database queries; (ii) alternative database architectures enable more efficient use of the available memory. For example, many in-memory technologies use a columnar layout in tables instead of a row-based orientation, which are more suitable for compression, and the ability to rapidly scan all column values speeds up query execution.

It's no longer just startup companies developing in-memory databases designed to support high-performance processing needs. Leading database and software vendors - IBM, Oracle, Microsoft, SAP, Teradata - are marketing database technologies that support in-memory processing, putting money behind their belief that mainstream organizations are ready to consider incorporating such software into IT systems.

In order to acquire an in-memory solution, it is important to assess the overall characteristics of the application, the organisation's demand for processing increased data volumes and the business value that could be delivered as a result of reduced database response times. It's also a good idea to take the overall characteristics of the organisation into account. In-memory databases are worth considering in organisations whose business processes can benefit from real-time data availability, simultaneous mixed-use applications, and noticeably faster reporting and analytics are good candidates for deploying in-memory databases. In most cases, the consideration of in-memory software must be aligned with IT spending priorities and corporate business objectives, including a demonstrable awareness of how key areas of corporate performance could be improved by the faster transaction processing and access to reports and ad hoc query results that in-memory processing makes possible.

Redis is one of the most popular and complete key-values in-memory database solutions. It is used by many large and well-known companies like Twitter, GitHub and StackOverflow. Redis maps keys to values that can be not only strings, but also more complex data types such as lists, sets, sorted sets and hash tables. It supports high-level, atomic, server-side operations like intersection, union, and difference between sets and sorting of lists, sets and sorted sets. Along with the rich data structures, incorporating Redis normally means a significant gain in performance.

Redis holds the whole data set into main memory and provides different levels of on-disk persistence. Since all data needs to fit in main memory, ideally it is preferred to store essential information which needs to be accessed, modified and inserted at a very fast rate.

References

- http://www.informationweek.com/big-data/big-data-analytics/in-memory-databases-do-you-need-the-speed/d/d-id/1114076
- http://oldblog.antirez.com/post/take-advantage-of-redis-adding-it-to-your-stack.html
- http://www.interworx.com/community/redis-what-you-need-to-know/
- https://dzone.com/articles/introduction-to-redis-in-memory-key-value-datastore
- http://openmymind.net/redis.pdf
- http://qnimate.com/overview-of-redis-architecture/
- http://redis.io/
- https://blog.pivotal.io/pivotal/case-studies/using-redis-at-pinterest-for-billions-of-relationships
- https://engineering.pinterest.com/blog/building-follower-model-scratch
- http://code.flickr.net/2012/12/12/highly-available-real-time-notifications/
- http://www.infoworld.com/article/2825890/application-development/why-redis-beats-memcached-for-caching.html
- https://www.quora.com/What-is-the-difference-between-MongoDB-and-Redis
- http://stackoverflow.com/questions/5400163/when-to-redis-when-to-mongodb
- https://en.wikipedia.org/wiki/MongoDB
- https://www.g2crowd.com/press-release/best-nosql-databases-fall-2015/
- http://db-engines.com/en/ranking_trend/key-value+store http://www.oracle.com/technetwork/database/in-memory/overview/index.html
- http://searchdatamanagement.techtarget.com/feature/In-memory-processing-helps-databases-meet-need-for-IT-speed
- http://dupress.com/articles/2014-tech-trends-in-memory-revolution/
- Shiva S. G. Computer Organization, Design, and Architecture. – CRC Press, 2013.
- Carlson, Josiah L. *Redis in Action*. Manning Publications Co., 2013.
- Freedman, Craig, Erik Ismert, and Per-Åke Larson. "Compilation in the Microsoft SQL Server Hekaton Engine." *IEEE Data Eng. Bull.* 37.1 (2014): 22-30.