In-memory key-value store

Benchmark report 2017

www.index64.com

Content

This document describes the method used to test Index64 v6.0 in different environments and it presents the results.

1. Context of the Benchmark

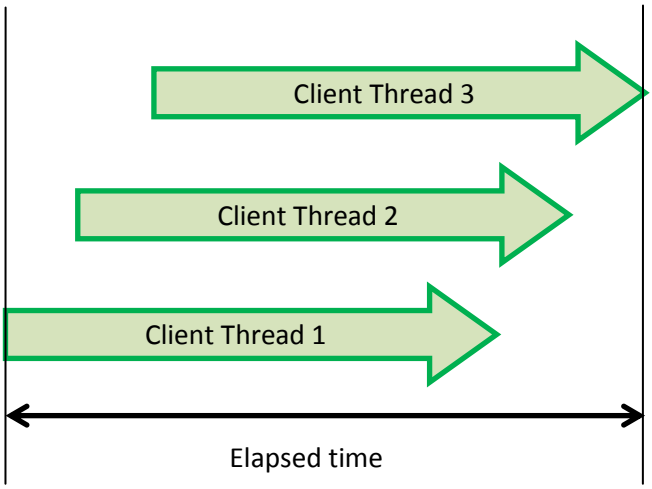
This benchmark considers the evaluation of Index64 v6.0's raw performance in write and read in comparison with Redis, the baseline offering on the in-memory key-value stores market in January 2016.

1.1. Method

Key-value stores Index64 and Redis are placed under the same conditions.

- They are tested alternately to avoid interferences.
- When the test involves Index64, a server instance is launched. The Index64 server instance activates as many threads as connected clients. When the test involves Redis, 1 server instance is launched.
- Logging of operations and pipelining are deactivated in Redis, in order to prevent interference with the interpretation of performance. Moreover, the commands are all synchronous.

The writing elapsed time starts as soon as a client thread starts its write processing. The writing elapsed time ends when all the client threads have completed their write processing. Likewise for the reading elapsed time.



1.2. General case

This performance test implements read and write commands on a set of 10 million key-value pairs. The keys used are different. The measurements are taken consecutively for a number of threads ranging from 1 to 16.

	Index64	Redis
Write command	insert	SET
Read command	select	GET

1.3. Sorted strings

This performance test implements read and write commands on a set of 10 million key-value pairs. The keys used are different. The measurements are taken consecutively for a number of threads ranging from 1 to 16.

	Index64	Redis
Write command	insert	ZADD
Read command	scanAscNext	ZRANGEBYLEX

During the sorted string test, the data are sorted on the fly by the database. For Index64, this functionality is native whereas Redis uses a specific structure which complements its main structure.

1.4. Conducting the test

Where T is the number of threads and N is the number of key-value pairs, the test is conducted as follows.

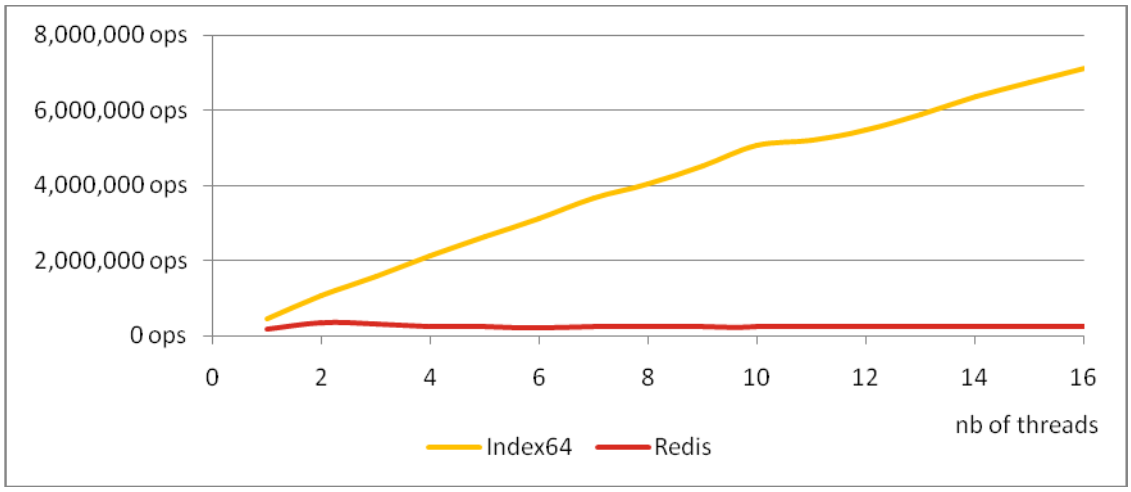
Index64	**Redis**
Launch of Index64 server instance.	Launch of Redis server instance.
For the general test, launch of the TestStrings client programme. For the sorted strings test, launch of the TestSortedSet client programme.	
The client generates N key-value pairs and splits them into T groups of the same size.	
The client launches T threads.	
The following set of steps is repeated 3 times.	
The T client threads are connected to the Index64 server instance.	The T client threads are connected to the Redis server instance.
Each client thread writes its group of key-value pairs. The client threads work simultaneously.	
Once all the writing has finished, each client thread reads its group of key-value pairs. The client threads work simultaneously.	
Once all the reading has finished, a report presents the elapsed time for the writing and the elapsed time for the reading.	

The number of operations per second is calculated from the elapsed time using the formula:

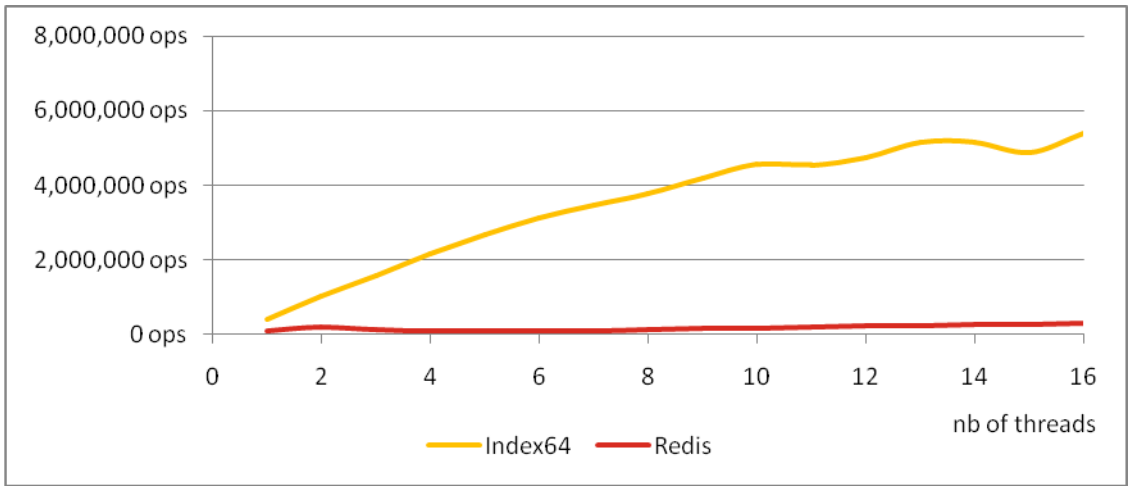
operations per second = 10 million / elapsed time

2. General Case Results

2.1. Centos 6.7 virtual machine



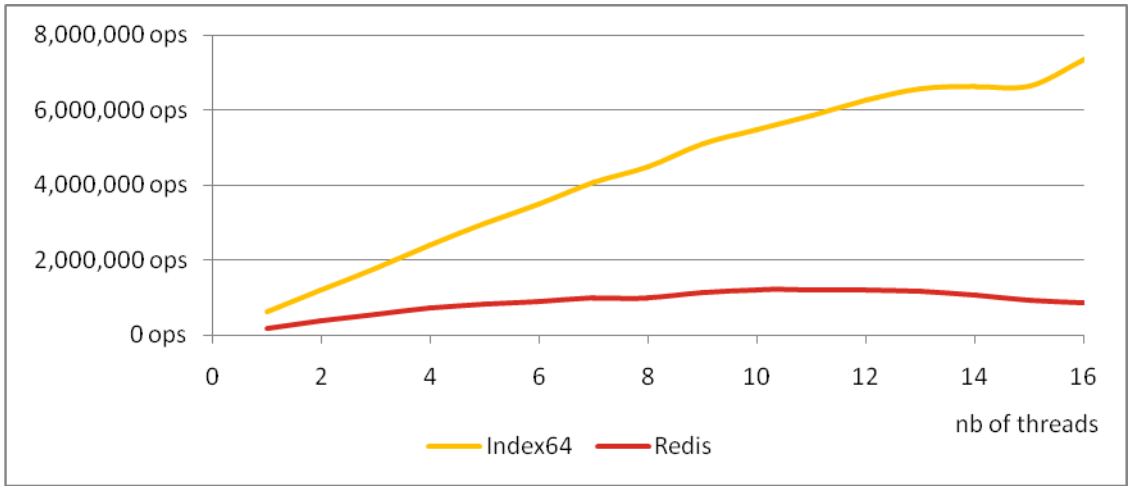
READ OPERATIONS PER SECOND
select for Index64
GET for Redis



WRITE OPERATIONS PER SECOND
insert for index64
SET for Redis

Physical: Centos Release 6.7
Machine: DELL PowerEdge T620 - 20 cores
CPU: Intel Xeon CPU E5-2690 v2 @ 3.00 GHz
Memory: 64 GB Loopback on a 1Gb network
Kvm: Centos Release 6.7 - use 20 cores
Compiler g++ 4.8.2
Index64 v6.0 - Redis 3.0.6
Asynchronous mode & Pipelining

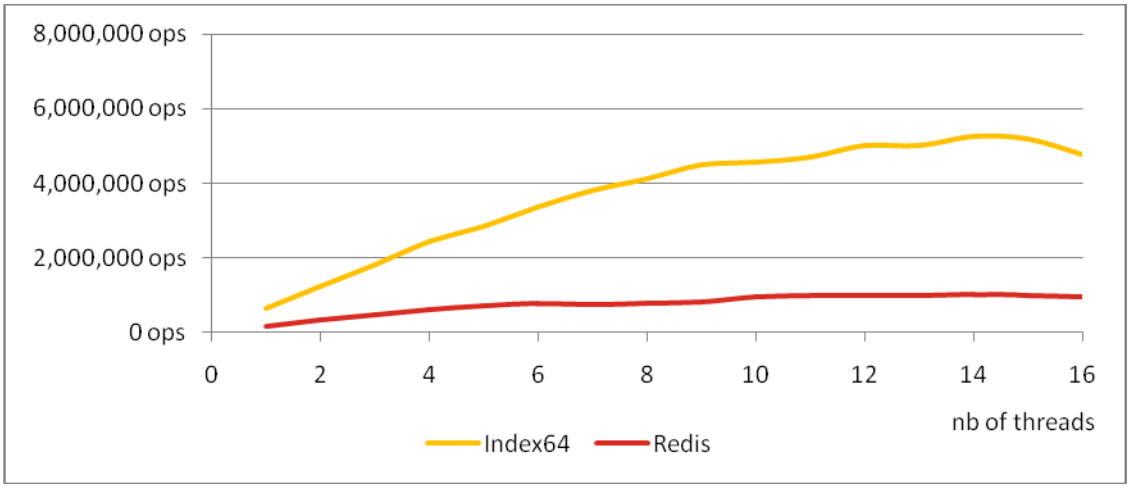
2.2. Centos 6.7 physical machine



READ OPERATIONS PER SECOND

select for Index64

GET for Redis



WRITE OPERATIONS PER SECOND

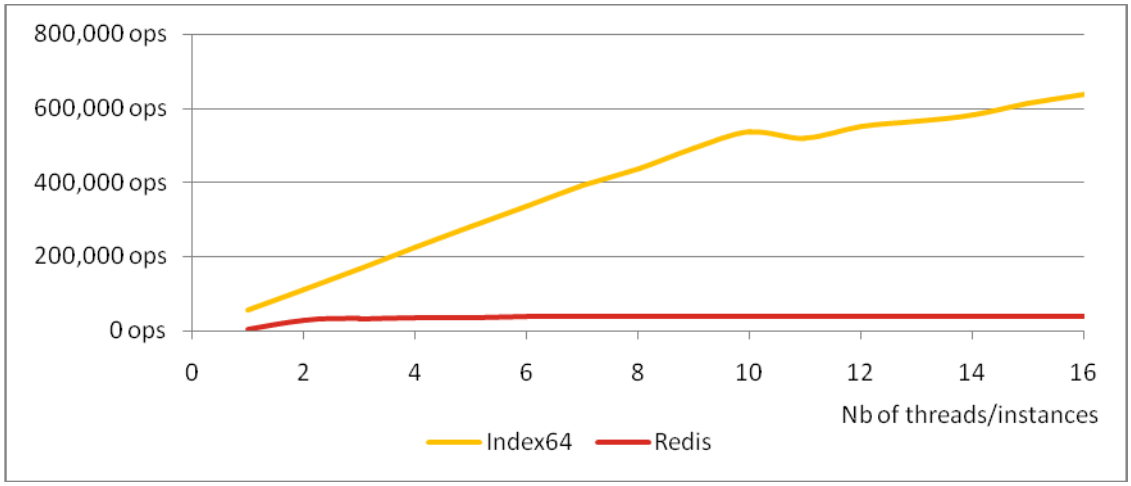
insert for index64

SET for Redis

Physical: Centos Release 6.7
Machine: DELL PowerEdge T620 - 20 cores
CPU: Intel Xeon CPU E5-2690 v2 @ 3.00 GHz
Memory: 64 GB Loopback on a 1Gb network
No kvm
Compiler g++ 4.8.2
Index64 v6.0 - Redis 3.0.6
Asynchronous mode & Pipelining

3. Sorted Strings Results

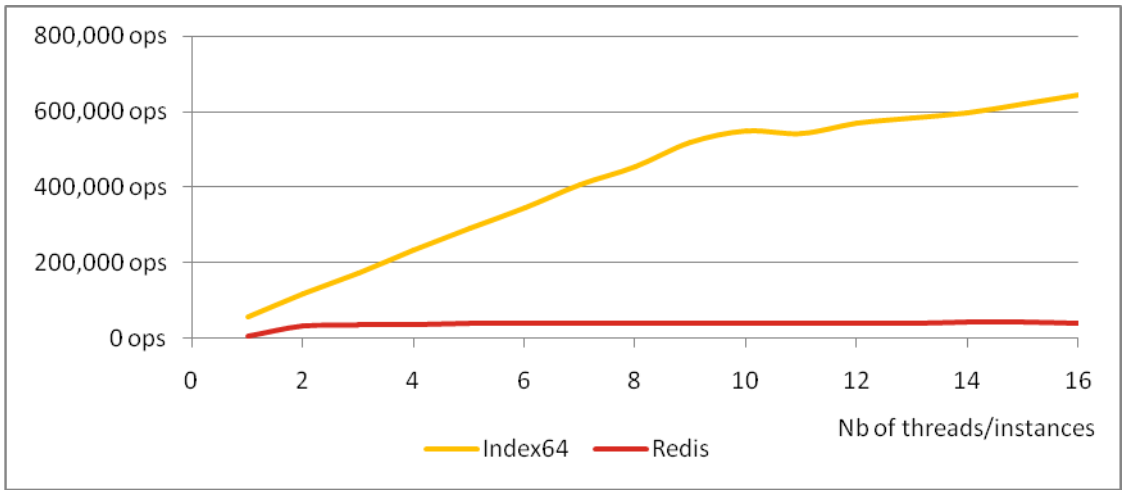
3.1. Centos 6.7 virtual machine



READ OPERATIONS PER SECOND

scanAscNext for Index64

ZRANGEBYLEX for Redis



WRITE OPERATIONS PER SECOND

insert for index64

ZADD for Redis

Physical: Centos Release 6.7
Machine: DELL PowerEdge T620 - 20 cores
CPU: Intel Xeon CPU E5-2690 v2 @ 3.00 GHz
Memory: 64 GB Loopback on a 1Gb network
Kvm: Centos Release 6.7 - use 20 cores
Compiler g++ 4.8.2
Index64 v6.0 - Redis 3.0.6
Synchronous mode

4. Appendices

4.1. Terms used

ops: Number of write or read operations per second. The read/write operations are often called GET/PUT.

system time: system time is the time shown on the machine. The difference between two system times gives duration. The maximum precision retained is the millisecond.

4.2. Test report

The graphs in paragraph 2.1 are produced from the following test report:

Steps	Keys	Threads	Index64 System Time	Redis System Time			Index64	Redis
unordered_write	10000000	1 thread	213,871 ms	1,836,224 ms			46,800 ops	5,400 ops
unordered_read	10000000	1 thread	204,611 ms	1,815,269 ms			48,900 ops	5,500 ops
unordered_write	10000000	2 threads	102,032 ms	906,945 ms			98,000 ops	11,000 ops
unordered_read	10000000	2 threads	98,541 ms	892,652 ms			101,500 ops	11,200 ops
unordered_write	10000000	3 threads	67,408 ms	603,874 ms			148,400 ops	16,600 ops
unordered_read	10000000	3 threads	71,123 ms	587,882 ms			140,600 ops	17,000 ops
unordered_write	10000000	4 threads	48,209 ms	451,893 ms			207,400 ops	22,100 ops
unordered_read	10000000	4 threads	46,440 ms	436,923 ms			215,300 ops	22,900 ops
unordered_write	10000000	5 threads	37,901 ms	370,428 ms			263,800 ops	27,000 ops
unordered_read	10000000	5 threads	36,019 ms	350,200 ms			277,600 ops	28,600 ops
unordered_write	10000000	6 threads	30,828 ms	332,777 ms			324,400 ops	30,100 ops
unordered_read	10000000	6 threads	30,013 ms	330,548 ms			333,200 ops	30,300 ops
unordered_write	10000000	7 threads	26,285 ms	287,909 ms			380,400 ops	34,700 ops
unordered_read	10000000	7 threads	25,430 ms	257,573 ms			393,200 ops	38,800 ops
unordered_write	10000000	8 threads	22,989 ms	268,818 ms			435,000 ops	37,200 ops
unordered_read	10000000	8 threads	22,494 ms	258,148 ms			444,600 ops	38,700 ops
unordered_write	10000000	9 threads	21,269 ms	217,234 ms			470,200 ops	46,000 ops
unordered_read	10000000	9 threads	20,336 ms	206,733 ms			491,700 ops	48,400 ops
unordered_write	10000000	10 threads	19,241 ms	141,834 ms			519,700 ops	70,500 ops
unordered_read	10000000	10 threads	18,077 ms	146,140 ms			553,200 ops	68,400 ops
unordered_write	10000000	11 threads	19,924 ms	96,778 ms			501,900 ops	103,300 ops
unordered_read	10000000	11 threads	18,520 ms	118,073 ms			540,000 ops	84,700 ops
unordered_write	10000000	12 threads	18,153 ms	78,282 ms			550,900 ops	127,700 ops
unordered_read	10000000	12 threads	17,829 ms	112,610 ms			560,900 ops	88,800 ops
unordered_write	10000000	13 threads	18,173 ms	72,670 ms			550,300 ops	137,600 ops
unordered_read	10000000	13 threads	17,348 ms	109,583 ms			576,400 ops	91,300 ops
unordered_write	10000000	14 threads	17,516 ms	68,382 ms			570,900 ops	146,200 ops
unordered_read	10000000	14 threads	16,523 ms	109,344 ms			605,200 ops	91,500 ops
unordered_write	10000000	15 threads	16,699 ms	61,400 ms			598,800 ops	162,900 ops
unordered_read	10000000	15 threads	15,889 ms	89,928 ms			629,400 ops	111,200 ops
unordered_write	10000000	16 threads	16,167 ms	54,457 ms			618,500 ops	183,600 ops
unordered_read	10000000	16 threads	15,408 ms	71,858 ms			649,000 ops	139,200 ops